Variable-Centered Intelligent Rule System

Building Expert Systems That Learn, from Zero to Real Applications

Irfan Subakti

Profitina

profitina.com

Variable-Centered Intelligent Rule System

Building Expert Systems That Learn, from Zero to Real Applications

Irfan Subakti

Copyright © 2026 Irfan Subakti. All rights reserved.

Published by Profitina · profitina.com

First edition, September 2026.

Every program in this book was compiled and run before its output was printed. The complete source code is in the accompanying Code folder. The programs are offered for teaching and learning; Irfan Subakti and the publisher accept no liability for their use elsewhere.

Contents

Preface ..... **xviii**
Who this book is for xviii
One idea, two languages xix
One engine: vcirs-core xix
What is in it xix
How to read it xxi
About the author xxi
Acknowledgments xxi

CHAPTER 1 Intelligent Rules in Daily Life, and Your Workbench **1**
1.1 A Morning Full of Rules 1
1.2 What Makes a Rule Intelligent 2
1.2.1 Facts, conditions and conclusions 2
1.2.2 Three things that make rules intelligent 2
1.2.3 Rules you already trust 3
1.3 Three Families of Rule Systems 3
1.3.1 Rule-based systems: strong answers, hard to change 3
1.3.2 Ripple Down Rules: easy to teach, simple answers 4
1.3.3 VCIRS: the best of both 4
1.4 VCIRS in Plain Words 4
1.4.1 Variables, nodes and rules 4
1.4.2 Building knowledge case by case 5
1.4.3 Counting what is used 5
1.4.4 Answering with explanations 6
1.4.5 One engine, two languages 6
1.5 Why Java, and Why Python 6
1.6 The Tools of Your Workbench 7
1.7 Setting Up on Windows, macOS and Linux 8
1.7.1 Windows, step by step 8
1.7.2 macOS, step by step 9
1.7.3 Linux, step by step 10
1.7.4 VS Code and its two extensions 10
1.7.5 The book's code on your computer 11
1.7.6 When something goes wrong 11
1.8 Your First Program in Two Languages 12
1.9 Your First Rules 13
1.10 A First Taste of VCIRS 15
1.11 Using It for Real 17
1.12 Common Mistakes, and How to Avoid Them 18
1.13 Chapter Summary 19
1.14 Exercises 19
1.15 Solutions to the Exercises 20
References 22

CHAPTER 2 Electricity-Bill Saver: Your First Rule Program **24**
2.1 Bu Sri's Bill 24
2.2 The Idea in Plain Words 25
2.2.1 Power, time and energy 25
2.2.2 Standby: the power of "off" 25
2.2.3 From energy to rupiah 25

2.2.4 Watts, volt-amperes and the circuit breaker ... 26
2.2.5 Saving rules ... 26
2.2.6 Why rules, and not just a spreadsheet? ... 27
2.3 Step 1: The Bill Without Rules ... 27
2.4 Step 2: The First Rules, as `if` Statements ... 28
2.5 Step 3: Rules as Values ... 30
2.5.1 The design ... 30
2.5.2 The household file ... 30
2.5.3 Whole numbers only ... 31
2.5.4 An appliance ... 31
2.5.5 The household and the bill ... 33
2.5.6 A rule is a value ... 38
2.6 Running the Saver ... 48
2.7 Making It Better ... 52
2.7.1 All the rules on one page ... 52
2.7.2 Adding a rule ... 53
2.7.3 Testing the rules ... 53
2.7.4 One appliance through the engine ... 53
2.7.5 From a list of rules to a knowledge base ... 54
2.7.6 What the rules cannot see ... 54
2.7.7 Checking the saver against the meter ... 54
2.8 Using It for Real ... 55
2.9 Common Mistakes, and How to Avoid Them ... 56
2.10 Chapter Summary ... 57
2.11 Exercises ... 58
2.12 Solutions to the Exercises ... 58
References ... 60

CHAPTER 3 Phone and Wi-Fi Troubleshooter That Learns from Each Fix ... 61
3.1 Mas Anto's Counter ... 61
3.2 The Idea in Plain Words ... 61
3.2.1 Why a flat list of rules breaks ... 61
3.2.2 Knowledge comes with a context ... 64
3.2.3 Cornerstone cases and the difference list ... 64
3.2.4 The shape of the knowledge: except and else ... 65
3.2.5 What RDR is good at, and what it is not ... 66
3.2.6 Ripple Down Rules at work ... 66
3.3 Ripple Down Rules from Scratch ... 67
3.4 Mas Anto Teaches Five Customers ... 73
3.5 From Our Engine to vcirs-core ... 77
3.6 The Wi-Fi Doctor ... 78
3.6.1 The first week's knowledge ... 78
3.6.2 The program ... 78
3.7 Four Customers at the Counter ... 87
3.7.1 The knowledge base after four customers ... 89
3.7.2 Fajar's Friday ... 91
3.8 Making It Better ... 91
3.9 Using It for Real ... 92
3.10 Common Mistakes, and How to Avoid Them ... 93
3.11 Chapter Summary ... 94
3.12 Exercises ... 94
3.13 Solutions to the Exercises ... 95
References ... 97

CHAPTER 4 Study-Program Advisor: Thinking in Variables 98
4.1 Laras and Bu Wati 98
4.2 The Idea in Plain Words 99
4.2.1 From advice to variables 99
4.2.2 Values and comparisons 100
4.2.3 A node: a small sequence of conditions 101
4.2.4 A rule: a sequence of nodes 101
4.2.5 The Variable-Centered Rule Structure 101
4.2.6 How a suggestion is found 103
4.3 Step 1: The Advice as a Flat List 103
4.4 Step 2: Variables, Nodes and Rules by Hand 108
4.5 From Our Structure to vcirs-core 118
4.6 The Study-Program Advisor 119
4.6.1 What Bu Wati taught it first 119
4.6.2 The program 120
4.7 Four Students at Bu Wati's Desk 130
4.8 Reading the Structure 133
4.9 Making It Better 135
4.10 Using It for Real 136
4.11 Common Mistakes, and How to Avoid Them 137
4.12 Chapter Summary 137
4.13 Exercises 138
4.14 Solutions to the Exercises 139
References 141

CHAPTER 5 Health Self-Check Advisor: Case-by-Case Knowledge Refinement 143
5.1 Pak Darmo and Ners Ratna 143
5.2 The Idea in Plain Words 144
5.2.1 Safety first, and not learned 144
5.2.2 Knowledge that grows, case by case 145
5.2.3 The cornerstone case and the list of difference 145
5.2.4 The six options 146
5.2.5 Choosing the conditions yourself 146
5.2.6 Checking what a new node would change 147
5.2.7 Never delete, sometimes stop 147
5.3 Step 1: The Six Options by Hand 148
5.4 Step 2: Ners Ratna Teaches the Knowledge Base 154
5.5 The Health Self-Check 162
5.5.1 What the app starts from 162
5.5.2 The program 163
5.6 Four People at the Puskesmas 178
5.7 Making It Better 182
5.8 Using It for Real 182
5.9 Common Mistakes, and How to Avoid Them 183
5.10 Chapter Summary 184
5.11 Exercises 184
5.12 Solutions to the Exercises 185
References 187

CHAPTER 6 Crop and Fruit-Tree Problem Advisor for Farmers: Rule Generation 189
6.1 Pak Slamet's Orchard 189
6.2 The Idea in Plain Words 190
6.2.1 Knowledge that is used 190
6.2.2 Credit: how often a condition was agreed with 191
6.2.3 NS, CD and Weight: how shared, how close 191

6.2.4 VUR, NUR and RUR 191
6.2.5 From usage to generation 192
6.2.6 The relative order 193
6.2.7 The expert has the last word 193
6.3 Step 1: The Usage Rates by Hand 194
6.4 Step 2: Generation on the Farm Doctor 201
6.5 The Crop and Fruit-Tree Doctor 207
6.5.1 What the doctor starts from 207
6.5.2 The program 209
6.6 Five Visits 222
6.7 Making It Better 226
6.8 Using It for Real 227
6.9 Common Mistakes, and How to Avoid Them 228
6.10 Chapter Summary 228
6.11 Exercises 229
6.12 Solutions to the Exercises 230
References 232

CHAPTER 7 Small-Shop Customer-Message Router: Usage and Credit Statistics 233
7.1 Bu Rina's Phone 233
7.2 The Idea in Plain Words 234
7.2.1 Signals, not understanding 234
7.2.2 Routing is classification 235
7.2.3 Credit: counting agreement 235
7.2.4 Weight, closeness and the three usage rates 236
7.2.5 What the numbers are for 237
7.2.6 Two kinds of counting 237
7.3 Usage Rates by Hand 237
7.4 The Signal Detector 243
7.5 The Router on vcirs-core 246
7.6 The App 252
7.7 Bu Rina's First Week 263
7.8 Reading the Numbers 265
7.9 Monday Morning 267
7.10 Making It Better 270
7.10.1 Teaching Monday's three mistakes 270
7.10.2 The voice mode 271
7.10.3 Maintaining the word lists 271
7.10.4 Tests 272
7.11 Using It for Real 272
7.12 Common Mistakes, and How to Avoid Them 273
7.13 Chapter Summary 273
7.14 Exercises 274
7.15 Solutions to the Exercises 275
References 276

CHAPTER 8 Motorcycle Fault Finder for Ojek Drivers: Forward and Backward Inference with Explanations 277
8.1 Mas Yudi's Morning 277
8.2 The Idea in Plain Words 278
8.2.1 Two directions of reasoning 278
8.2.2 Rules that feed rules 278
8.2.3 What, How and Why 279
8.2.4 How sure is the answer? 279
8.2.5 From cases to rules 280

8.2.6 Advice, not repair 280
8.2.7 Which direction, when? 281
8.3 Chaining from Scratch 281
8.4 Cak Man's Knowledge 285
8.5 The Fault Finder 287
8.6 Mornings at the Base Camp 298
8.6.1 Mas Yudi's BeAT will not start 298
8.6.2 The same morning, forward 299
8.6.3 Pak Heru's Vario 160 has no power 300
8.6.4 A warning light on an NMAX 302
8.6.5 Soft brakes 302
8.6.6 Counting the questions 303
8.7 Making It Better 303
8.7.1 A fault it cannot name 303
8.7.2 Seeing every step 305
8.7.3 Combining evidence 306
8.7.4 The order of rules 306
8.7.5 Tests 306
8.8 Using It for Real 306
8.9 Common Mistakes, and How to Avoid Them 307
8.10 Chapter Summary 308
8.11 Exercises 308
8.12 Solutions to the Exercises 309
References 310

CHAPTER 9 Cooperative-Loan Eligibility Advisor: Fuzzy VCIRS 311
9.1 The Committee of Koperasi Sumber Makmur 311
9.2 The Idea in Plain Words 312
9.2.1 Words with soft edges 312
9.2.2 Very, somewhat, not 313
9.2.3 Fuzzy conditions in VCIRS 313
9.2.4 Two kinds of imprecision 314
9.2.5 The repayment ratio and the 30 percent line 315
9.2.6 Fair and transparent by construction 315
9.2.7 One member, from start to finish 316
9.2.8 Why not a credit score? 317
9.3 Membership by Hand 317
9.4 The Cooperative's Arithmetic 321
9.5 The Advisor 325
9.6 The Committee Meeting 335
9.7 Teaching the Committee's Decision 338
9.8 Making It Better 339
9.8.1 Tuning the words with the committee 339
9.8.2 The threshold 340
9.8.3 Hedges in the knowledge 340
9.8.4 Keeping a record of the overrides 340
9.8.5 Tests 340
9.9 Using It for Real 340
9.10 Common Mistakes, and How to Avoid Them 341
9.11 Chapter Summary 342
9.12 Exercises 343
9.13 Solutions to the Exercises 343
References 344

CHAPTER 10 Laundry Stain-Removal Advisor as a Web App with a Database 346
10.1 Bu Rini's Laundry 346
10.2 The Idea in Plain Words 347
10.2.1 From one computer to several: a web app 347
10.2.2 Why a database, and why SQLite 347
10.2.3 The knowledge base as tables 348
10.2.4 Asking questions over the web: the re-run pattern 349
10.2.5 What the advisor knows, and what it must never do 349
10.3 Step 1: The Knowledge Base in a Database 350
10.4 Step 2: The Advisor, Without Any Web Code 356
10.5 Step 3: A Day at the Laundry, Without a Browser 372
10.6 Step 4: The Web Server 378
10.7 Step 5: The Pages 385
10.8 Running the Laundry Advisor 390
10.8.1 Talking to the server without a browser 393
10.8.2 The phone at the counter 394
10.8.3 Tests 394
10.9 Making It Better 394
10.10 Using It for Real 395
10.11 Common Mistakes, and How to Avoid Them 396
10.12 Chapter Summary 397
10.13 Exercises 397
10.14 Solutions to the Exercises 398
References 400

CHAPTER 11 Clinic Triage Knowledge Base for a Team: Many Experts, Versions and Audit 402
11.1 The Front Desk of Klinik Sehat Bersama 402
11.2 The Idea in Plain Words 403
11.2.1 Triage: sorting by urgency, not naming illnesses 403
11.2.2 Many experts, one knowledge base 403
11.2.3 Versions: a photograph after every change 404
11.2.4 Tests: cornerstone cases and golden cases 404
11.2.5 Four eyes: review by another expert 404
11.2.6 The audit trail, and why it is chained 405
11.2.7 When experts disagree 406
11.3 Step 1: Two Experts, One File 407
11.4 Step 2: The Team Knowledge Base 409
11.5 Step 3: The Command Runner 433
11.6 A Week at the Clinic 440
11.6.1 Monday: the first version, and a patient the knowledge misjudged 440
11.6.2 Wednesday: a proposal the tests refuse 441
11.6.3 Thursday: the front desk at work 442
11.6.4 Friday: a mistake, a rollback and a lesson 443
11.6.5 What the files remember 443
11.7 Looking Back 444
11.7.1 Tests 446
11.8 Making It Better 446
11.9 Using It for Real 447
11.10 Common Mistakes, and How to Avoid Them 448
11.11 Chapter Summary 449
11.12 Exercises 449
11.13 Solutions to the Exercises 450
References 452

CHAPTER 12 Learning Rules from Data: From a Decision Tree to a VCIRS Knowledge Base 453
12.1 Bu Lestari's Nature Club 453
12.2 The Idea in Plain Words 454
12.2.1 Knowledge from experts, knowledge from data 454
12.2.2 A decision tree is a game of questions 454
12.2.3 How the tree chooses its questions: purity 455
12.2.4 Exact arithmetic, so that Java and Python agree 457
12.2.5 A tree that is right on everything it has seen 457
12.2.6 From paths to VCIRS nodes 457
12.2.7 What the data cannot know 458
12.3 Step 1: CART from Scratch 458
12.4 Step 2: From the Tree to a Knowledge Base 472
12.5 Step 3: The Teacher's Lessons 477
12.5.1 Six animals the data never saw 477
12.5.2 The program 477
12.5.3 What happens 482
12.5.4 The game at the club 485
12.6 Step 4: A Check Against scikit-learn 485
12.6.1 Tests 488
12.7 Making It Better 488
12.8 Using It for Real 489
12.9 Common Mistakes, and How to Avoid Them 490
12.10 Chapter Summary 491
12.11 Exercises 491
12.12 Solutions to the Exercises 492
References 493

CHAPTER 13 Talking to Your Expert System: a Local Language Model with Human-Checked Knowledge 494
13.1 Messages at Nine in the Evening 494
13.2 The Idea in Plain Words 495
13.2.1 What a language model is, and what it is not 495
13.2.2 Two jobs for the model, one for the knowledge base 495
13.2.3 A model on your own computer: Ollama 497
13.2.4 Asking for JSON, not for prose 497
13.2.5 The guard 498
13.3 Step 1: Talking to Ollama 498
13.4 Step 2: A Stand-in That Runs Offline 505
13.5 Step 3: The Chat Doctor 513
13.6 Five Conversations 528
13.7 Running It with a Real Model 530
13.8 Making It Better 531
13.9 Using It for Real 532
13.10 Common Mistakes, and How to Avoid Them 532
13.11 Chapter Summary 533
13.12 Exercises 534
13.13 Solutions to the Exercises 534
References 536

CHAPTER 14 Capstone: a VCIRS Expert-System Builder Anyone Can Use 537
14.1 Bu Wati's Notebook 537
14.2 The Idea in Plain Words 538
14.2.1 What the builder must let a person do 538
14.2.2 Usage statistics in plain words 538

14.2.3 Proposals: the machine suggests, the person decides 540
14.2.4 One application, two servers, the same pages 540
14.3 Step 1: The Builder, Without the Web 541
14.4 Step 2: Bu Wati's Session, as a Script 551
14.5 Step 3: The Pages 560
14.6 Step 4: The Web Servers 576
14.7 Using the Builder 587
14.8 Making It Better 591
14.9 Using It for Real 592
14.10 Common Mistakes, and How to Avoid Them 593
14.11 Chapter Summary 593
14.12 Exercises 594
14.13 Solutions to the Exercises 594
References 596

CHAPTER 15 Deploying, Maintaining and Growing Your Expert Systems 597
15.1 The Morning the Laptop Would Not Start 597
15.2 The Idea in Plain Words 597
15.2.1 From a program you run to a service that runs 597
15.2.2 One file to copy 598
15.2.3 A knowledge base is data: backups you can trust 598
15.2.4 Watching how the knowledge is used 599
15.2.5 Personal data and the law 599
15.3 Step 1: The Advice Service 600
15.4 Step 2: Packaging and Deploying 609
15.5 Step 3: Checking a Deployment 613
15.6 Step 4: Backups 619
15.7 Step 5: Monitoring Rule Usage 627
15.8 Maintaining and Growing the Knowledge 636
15.9 Privacy, Ethics and Safety 638
15.9.1 Personal data, in practice 638
15.9.2 The ethics of advice by machine 639
15.10 Where to Go Next 639
15.11 Common Mistakes, and How to Avoid Them 640
15.12 Chapter Summary 641
15.13 Exercises 641
15.14 Solutions to the Exercises 642
References 644

List of Figures

Figure 1.1 Two families of rule systems and the hybrid that joins them. VCIRS takes the inference and explanations of rule-based systems and the case-by-case knowledge building of Ripple Down Rules, and adds usage statistics and rule generation. 3

Figure 1.2 The Variable-Centered Rule Structure. Variables are used in nodes; a node is a short sequence of conditions with a conclusion; a rule is a sequence of nodes, top level first. 5

Figure 1.3 The workbench. Java source is compiled by javac into bytecode, which java runs; Python source is run directly by the python interpreter; VS Code holds both. 7

Figure 2.1 The electricity-bill saver. The appliances and the tariff give the energy and the bill; the saving rules turn them into advice, each with a reason and a saving. 30

Figure 2.2 Bu Sri's monthly energy per appliance, now and if all the computable advice is followed (our example household). 51

Figure 3.1 The list of difference between a rule's cornerstone case and a new case. Section A holds what only the old case has; section B what only the new case has. The expert's new rule is built from these differences. 65

Figure 3.2 A single-classification Ripple Down Rules tree. Rule 0 is always true; the except branch holds exceptions, tried when a rule is true; the else branch holds the next rule, tried when a rule is false. The answer is the conclusion of the last true rule. 65

Figure 4.1 SNBT 2026 in numbers: fewer than three in ten registrants were accepted, and most of those who were accepted got in through their first or second choice. Data from the SNPMB press release of 25 May 2026 [4]. 99

Figure 4.2 Part of the Variable-Centered Rule Structure of Bu Wati's advice. Each variable is kept once and linked to the nodes that use it; each node holds only its own conditions; each rule is a sequence of nodes from the top level down. 102

Figure 4.3 The fields of a node and of a rule in the Variable-Centered Rule Structure, filled in for `medicine#1`. The rule lists its nodes in order; each node lists its conditions in order. 102

Figure 5.1 Safety first. The fixed safety questions come before the knowledge base; a single *yes* sends the person to help at once. Only then is the knowledge base, taught case by case by the nurse, consulted for self-care advice. 145

Figure 5.2 What each option would build for one case at the puskesmas. Options 1 to 3 hang an exception under the old node; option 4 puts the whole case beside it and option 5 at the top level; the nurse's own choice is one condition that says what she means. 147

Figure 6.1 How the usage rates of one node are computed. Each condition's Variable Usage Rate is its Credit times its Weight, and the Weight is the number of nodes sharing the variable times the condition's closeness to the conclusion. The node's rate is the mean of its conditions' rates. 190

Figure 6.2 Rule generation in VCIRS. The usage rates give a relative order of the knowledge; the order and the most important nodes and variables give candidates; the expert keeps what is true, and the new node answers cases that no taught node could. 193

Figure 6.3 The rule usage rates of Mbak Nur's first knowledge base. The most used rules, in amber, are those whose first node has exceptions: much of the knowledge passes through them. 206

Figure 7.1 The path of one message through the router. The detector turns words into yes/no signals, but only when the knowledge base asks for them; the knowledge base picks the route; when Bu Rina corrects a wrong route, a new exception is added and its parent's conditions earn Credit. 234

Figure 7.2 The largest Variable Usage Rate of each signal after the first week (grey) and after Monday's three corrections (teal). Payment, order, late and hours gained; the others did not change. The signal `proof` has no bar because no node uses it yet. 271

Figure 8.1 The same two rules used in both directions. Forward chaining (left) climbs from the facts through the rules to a conclusion; backward chaining (right) starts from a goal, turns the rule's conditions into subgoals and asks the driver only what no rule can work out. 278

Figure 8.2 The four levels of urgency of the fault finder's advice, from the most urgent at the top. Brake and fuel faults stop the ride; most other faults go to a bengkel, today or this week; only a few things are for the driver himself. 281

Figure 9.1 The three fuzzy words for the repayment ratio, with the hedge *very high* (dotted). The horizontal dashed line is the threshold 0.5 at which a fuzzy condition counts as true; the triangles mark the ratios of five members of Section 9.6. 313

Figure 9.2 The path of an application through the advisor. Only facts about repayment reach the engine; a fairness guard refuses any knowledge that uses a protected attribute; the advice comes with its CF, its explanation, a borderline warning and a what-if amount; the committee decides, and its decisions are taught back. 316

Figure 10.1 The tables of the stain advisor's database. The teal tables hold the VCIRS knowledge base as in Appendix E of the thesis; the amber tables belong to the laundry: the steps of each treatment and the log of consultations. 348

Figure 10.2 The re-run pattern. The page sends every answer it has; the server runs the engine from the beginning and returns the first question the engine cannot answer, or, when there is none, the advice. 349

Figure 10.3 The stain advisor as a web app. The browser and the server speak JSON over HTTP; the Advisor consults and teaches the vcirs-core knowledge base, which is loaded from and saved to the VCIRS tables, and records consultations in the laundry's own tables. 378

Figure 10.4 The advisor asks its second question. The answers so far are shown as chips; the reason for the question comes from the engine. 391

Figure 10.5 The advice for a dried tea stain on a white item: the steps, the caution and, unfolded, the explanation of the node that fired. 391

Figure 10.6 Teaching the advisor a treatment for a stain it has never seen. The extra-facts box is empty because the stain itself is what is new. 392

Figure 10.7 The knowledge page: the week's numbers, the most given advice, the knowledge as a tree of exceptions and the latest changes. 393

Figure 11.1 The life of a change. A proposal is tried on a copy of the current version and tested; if every case still passes, another expert reviews it; only an approved change becomes a new version. Every step, including every rejection, is written to the audit trail. 405

Figure 11.2 The audit trail as a chain. Each entry holds the fingerprint of the version file it created and of the entry before it, and its own fingerprint covers both, so changing any old entry or version file breaks every link after it. 406

Figure 11.3 The seven versions of the week. P2 was refused by the tests and never became a version; version 5 stopped the wrong node, and version 6 is a copy of version 4. 443

Figure 12.1 From data to a checked knowledge base. A tree is learned from the table, each leaf becomes a VCIRS node, the teacher adds exceptions, and every lesson is checked against all the animals the knowledge base already answers. 454

Figure 12.2 The decision tree learned from the 101 animals of the Zoo data. Each question has a yes branch and a no branch; each leaf gives a class and, in parentheses, how many animals of the data reached it. 455

Figure 12.3 The teacher's exceptions. Under the learned fish node, the sea turtle's lesson and, under it, the mudskipper's; under the learned amphibian node, the refused tail lesson and the kept lesson on scales. Each node names its cornerstone animal. 484

Figure 13.1 The chat front end. The language model (amber) only turns words into facts and advice into sentences; everything it produces is checked. The VCIRS knowledge base (navy) decides, and asks for the facts it still needs. The guard lets a reply through only if it gives the knowledge base's advice and nothing else. 496

Figure 13.2 The same client talks to a real Ollama with a small model, to the offline stand-in, or, when neither answers, to nobody: then it asks its own questions and writes its own reply. 513

Figure 14.1 The builder. The browser talks to a small web server (Flask in Python, the JDK's HttpServer in Java). The server only connects the web to two parts that know nothing about the web: the pages, which build HTML from the knowledge base and from the words in `ui-text.json`, and the builder, which changes the knowledge base through vcirs-core and saves it after every change. 540

Figure 14.2 The Home page after importing the cake cases: the project's name and description, its size, and the export, import and new-project sections. "Saved." confirms the last change. 587

Figure 14.3 The Consult page: the advice in large letters, how it was found, a link to teach the right answer if it is wrong, and the questions in the order that helps most. 588

Figure 14.4 The result of teaching: what the knowledge base said before, what happened, and why the new case is different from the case that created the old node. 589
Figure 14.5 The Statistics page: the questions from most to least useful with the number of nodes that use each, the most important questions and nodes, and the Node Usage Rates, largest first. 589
Figure 14.6 A proposed node on the Proposals page. Of the conditions VCIRS collected in a good order, Bu Wati keeps one, `problem = dry`, to make a shorter and more general node. 590
Figure 14.7 The Knowledge page: the tree of nodes and exceptions, marked when a node was generated, and a table with who taught each node and a Stop button. 591
Figure 15.1 The cake doctor deployed. The service runs from one file on the shop computer and answers the counter's tablet or phone; it reloads the knowledge base when a checked copy arrives from Bu Wati's laptop, writes a usage log, and is backed up every evening and reported on every month. 612
Figure 15.2 How often each node gave advice in 120 consultations, July to September 2026 (from the replayed log), with the node that never did and the consultations that got no answer. 635
Figure 15.3 The maintenance cycle: the service is used, every consultation is logged, the monthly report shows what is unanswered and unused, the expert teaches the missing cases in the builder, the new knowledge base is checked and backed up, and the checked copy goes to the service. 637

List of Tables

Table 1.1 The road map of the book: each chapter builds a real application and adds one idea of VCIRS. 6
Table 1.2 The workbench of this book, as checked in September 2026. Newer update numbers are always fine. 8
Table 1.3 The workbench on three operating systems, from an empty computer to a working editor. On Fedora use `sudo dnf install java-25-openjdk-devel python3.14`; on Arch Linux, `sudo pacman -S jdk-openjdk python`. 8
Table 1.4 The folders of the book's code. Every chapter folder has the same shape. 11
Table 1.5 The usual setup problems and their fixes. 12
Table 2.1 Household electricity tariffs of PLN for July–September 2026, left unchanged for that quarter [2], [3]. 26
Table 2.2 The nine rules of the electricity-bill saver: six about one appliance and three about the house. 53
Table 3.1 How the ideas of the hand-made engine appear in vcirs-core. 77
Table 3.2 The six options for adding knowledge in VCIRS, following the 2006 revision [8], [9]. 78
Table 4.1 The variables of Bu Wati's advice, with the question behind each one and its usual values. 100
Table 4.2 How the hand-made structure of Step 2 appears in vcirs-core. 119
Table 5.1 The six options of VCIRS knowledge building, with the question each answers. Options 1 to 3 make an exception of the old node; options 4 and 5 store the whole case as a new node elsewhere. 146
Table 8.1 What Cak Man taught the fault finder: symptoms, causes and urgency. Rules without a CF in brackets have CF 100. 286
Table 8.2 Questions asked in the sessions of this chapter, including the safety check. 303
Table 10.1 The twelve seed cases of the stain advisor, with the caution each treatment carries. 350
Table 10.2 The three calls of the stain advisor and what travels in each. 356
Table 11.1 The three urgency levels of the front-desk knowledge base, and the safe default when no rule matches. 403
Table 11.2 Who may do what in the clinic's team knowledge base. 405
Table 12.1 The five best first questions for the 101 animals, by the weighted Gini impurity of the two groups they make. Milk wins. 456

Table 13.1 Small open models in Ollama's library, September 2026, with the download size of each tag's default version. 497
Table 14.1 The pages of the builder, what a person does on each, and the VCIRS idea behind it. 538

List of Listings

Listing 1.1 `Hello.java`: the first Java program; it checks that Java is new enough. 12
Listing 1.2 `hello.py`: the first Python program; it checks that Python is new enough. 13
Listing 1.3 `MorningRules.java`: three everyday IF-THEN rules, each with its reason. 13
Listing 1.4 `morning_rules.py`: the same three rules in Python. 14
Listing 1.5 `FirstTaste.java`: a knowledge base learns a rule and then an exception to it. 15
Listing 1.6 `first_taste.py`: the same first taste in Python. 17
Listing 2.1 `BillCalculator.java`: the bill of three appliances, with no rules yet. 27
Listing 2.2 `bill_calculator.py`: the same bill in Python. 28
Listing 2.3 `FirstAdvice.java`: two saving rules written directly as `if` statements. 28
Listing 2.4 `first_advice.py`: the same two rules in Python. 29
Listing 2.5 `Appliance.java`: one appliance, its monthly energy, and copies that describe advice. 31
Listing 2.6 `Household.java`: reading the two files, and the whole-number bill arithmetic. 33
Listing 2.7 `household.py`: the appliance, the household file and the bill arithmetic in Python. 35
Listing 2.8 `SavingRule.java`: a saving rule as a value with five parts. 38
Listing 2.9 `BillSaver.java`: the electricity-bill saver, with its rules as values and a small engine that applies them. 39
Listing 2.10 `bill_saver.py`: the electricity-bill saver in Python. 44
Listing 3.1 `FlatRules.java`: a troubleshooter as a flat list of rules, tried in three versions. 62
Listing 3.2 `flat_rules.py`: the same three versions in Python. 63
Listing 3.3 `Rdr.java`: a complete single-classification Ripple Down Rules engine. 67
Listing 3.4 `rdr.py`: the same engine in Python. 70
Listing 3.5 `RdrDemo.java`: five customers teach an empty troubleshooter. 73
Listing 3.6 `rdr_demo.py`: the same five customers in Python. 74
Listing 3.7 `WifiDoctor.java`: the Wi-Fi doctor, a troubleshooter that learns from each fix, on vcirs-core. 78
Listing 3.8 `wifi_doctor.py`: the Wi-Fi doctor in Python. 83
Listing 4.1 `FlatAdvice.java`: Bu Wati's advice as a flat list of rules, indexed by variable. 103
Listing 4.2 `flat_advice.py`: the same flat list and index in Python. 105
Listing 4.3 `MiniStructure.java`: a Variable table, a Node Structure and a Rule Structure, built by hand. 108
Listing 4.4 `mini_structure.py`: the same three tables in Python. 113
Listing 4.5 `StudyAdvisor.java`: the study-program advisor on vcirs-core. 120
Listing 4.6 `study_advisor.py`: the study-program advisor in Python. 126
Listing 5.1 `SixOptions.java`: the list of difference and the six options, worked by hand. 148
Listing 5.2 `six_options.py`: the same computation in Python. 151
Listing 5.3 `NurseTeaches.java`: a nurse teaches the knowledge base case by case, on vcirs-core. 154
Listing 5.4 `nurse_teaches.py`: the same teaching session in Python. 157
Listing 5.5 `HealthCheck.java`: the health self-check, safety first, on vcirs-core. 163
Listing 5.6 `health_check.py`: the health self-check in Python. 173
Listing 6.1 `UsageByHand.java`: Credit, NS, CD, Weight, VUR, NUR and RUR by hand, checked against vcirs-core. 194
Listing 6.2 `usage_by_hand.py`: the same computation in Python. 198

Listing 6.3 `GenerateDemo.java`: rule and node generation on the farm knowledge base, with the penyuluh's review. 201
Listing 6.4 `generate_demo.py`: the same generation and review in Python. 203
Listing 6.5 `FarmDoctor.java`: the crop and fruit-tree doctor on vcirs-core, with usage rates and generation. 209
Listing 6.6 `farm_doctor.py`: the crop and fruit-tree doctor in Python. 217
Listing 7.1 `UsageByHand.java`: the usage rates of five nodes computed by hand, then by vcirs-core. 237
Listing 7.2 `usage_by_hand.py`: the same computation in Python. 240
Listing 7.3 `Signals.java`: the signal detector reads the word lists and spots whole words and phrases in a message. 243
Listing 7.4 `signals.py`: the same detector in Python. 245
Listing 7.5 `Router.java`: building the knowledge base, routing a message and teaching a correction. 246
Listing 7.6 `router.py`: the same three operations in Python. 250
Listing 7.7 `ShopRouter.java`: the message router of Toko Rina, with its five commands. 252
Listing 7.8 `shop_router.py`: the same app in Python. 259
Listing 8.1 `Chain.java`: forward and backward chaining from scratch, on four rules of the fault finder. 281
Listing 8.2 `chain.py`: the same two directions in Python. 284
Listing 8.3 `FaultFinder.java`: the fault finder with its safety check and three ways of reasoning. 287
Listing 8.4 `fault_finder.py`: the same app in Python. 294
Listing 9.1 `FuzzyTerms.java`: membership degrees and hedges by hand, checked against vcirs-core. 317
Listing 9.2 `fuzzy_terms.py`: the same table in Python. 319
Listing 9.3 `Loans.java`: installments, the repayment ratio and the what-if amount, in whole rupiah. 321
Listing 9.4 `loans.py`: the same arithmetic in Python. 323
Listing 9.5 `LoanAdvisor.java`: the fair, explained, fuzzy loan advisor. 325
Listing 9.6 `loan_advisor.py`: the same advisor in Python. 332
Listing 10.1 `db_tour.py`: the knowledge base saved in SQLite, looked at with SQL, and loaded back. 350
Listing 10.2 `DbTour.java`: the same tour of the database in Java, through JDBC. 352
Listing 10.3 `advisor.py`: the stain advisor's logic, with its knowledge in SQLite. 356
Listing 10.4 `Advisor.java`: the same logic in Java, through JDBC. 362
Listing 10.5 `advisor_demo.py`: the fifteen requests of the tour, replayed on the Advisor. 372
Listing 10.6 `AdvisorDemo.java`: the tour in Java. 372
Listing 10.7 `LaundryServer.java`: the web server in Java, on the JDK's own `HttpServer`. 378
Listing 10.8 `laundry_app.py`: the same web server in Python, on Flask. 382
Listing 10.9 `index.html`: the advisor page, which keeps the answers and talks to the server. 385
Listing 10.10 `kb.html`: the knowledge page, with the week's numbers, the tree, the latest changes and the treatments. 388
Listing 10.11 `style.css`: the look of the two pages. 390
Listing 11.1 `lost_update.py`: two experts change one shared file, and one lesson disappears. 407
Listing 11.2 `LostUpdate.java`: the same lost lesson in Java. 407
Listing 11.3 `team_kb.py`: the team knowledge base, with versions, tests, review, stopping, rollback and a chained audit trail. 409
Listing 11.4 `TeamKb.java`: the team knowledge base in Java. 419
Listing 11.5 `triage_team.py`: the command runner, for scripts, typed commands and question-by-question consultations. 433
Listing 11.6 `TriageTeam.java`: the same command runner in Java. 436
Listing 12.1 `cart.py`: a CART decision tree with exact Gini arithmetic and leave-one-out testing. 459
Listing 12.2 `Cart.java`: the same CART learner in Java. 464
Listing 12.3 `tree_to_kb.py`: every leaf of the tree becomes a top-level node, with its path as conditions and its first animal as cornerstone. 472

Listing 12.4 `TreeToKb.java`: the same conversion in Java. 473
Listing 12.5 `refine_zoo.py`: the teacher's lessons, each tried on a copy and checked against every known animal before it is kept. 477
Listing 12.6 `RefineZoo.java`: the teacher's lessons in Java. 479
Listing 12.7 `sklearn_check.py`: our tree compared with scikit-learn's `DecisionTreeClassifier`. 485
Listing 13.1 `OllamaClient.java`: a minimal client for Ollama's chat API. 498
Listing 13.2 `ollama_client.py`: the same client in Python, standard library only. 501
Listing 13.3 `AskOllama.java`: step 1, one question to the model. 502
Listing 13.4 `ask_ollama.py`: step 1 in Python. 503
Listing 13.5 `stub_ollama.py`: a stand-in for Ollama that answers from a phrase book. 505
Listing 13.6 `StubOllama.java`: the same stand-in in Java. 508
Listing 13.7 `chat_doctor.py`: the chat doctor. The model understands and phrases; the knowledge base decides; the guard checks. 513
Listing 13.8 `ChatDoctor.java`: the chat doctor in Java. 520
Listing 14.1 `builder.py`: everything a person can do with a knowledge base, without the web. 541
Listing 14.2 `Builder.java`: the same builder in Java. 545
Listing 14.3 `builder_demo.py`: Bu Wati's session, played through the builder. 551
Listing 14.4 `BuilderDemo.java`: the same session in Java. 553
Listing 14.5 `pages.py`: the HTML pages, built from the knowledge base and the words of `ui-text.json`. 560
Listing 14.6 `Pages.java`: the same pages in Java, byte for byte. 568
Listing 14.7 `builder_app.py`: the builder's web server in Flask. 576
Listing 14.8 `BuilderServer.java`: the same server with the JDK's own HttpServer. 581
Listing 15.1 `advice_service.py`: the cake doctor as a small advice service with a privacy-minded usage log. 600
Listing 15.2 `AdviceService.java`: the same service in Java. 604
Listing 15.3 `cake_doctor.py`: one program with four commands, the entry point of the `.pyz` file. 609
Listing 15.4 `CakeDoctor.java`: the same four commands in Java. 609
Listing 15.5 `build_pyz.py`: packs the service, its tools and vcirs-core into one `.pyz` file. 610
Listing 15.6 `service_check.py`: the check after every deployment, and the replay of recorded consultations. 613
Listing 15.7 `ServiceCheck.java`: the same check in Java. 615
Listing 15.8 `backup.py`: save, check and restore backups of a knowledge base, with SHA-256 fingerprints. 619
Listing 15.9 `Backup.java`: the same backup tool in Java. 622
Listing 15.10 `usage_report.py`: the monthly usage report, and the removal of old log lines. 627
Listing 15.11 `UsageReport.java`: the same report in Java. 630

List of Equations

(2.1) 25
(2.2) 25
(2.3) 26
(2.4) 31
(6.1) 191
(6.2) 191
(6.3) 191
(6.4) 191

(6.5) 192
(6.6) 192
(7.1) 236
(7.2) 236
(7.3) 236
(7.4) 236
(7.5) 236
(8.1) 279
(8.2) 280
(9.1) 312
(9.2) 313
(9.3) 314
(9.4) 314
(9.5) 315
(12.1)....456
(12.2)....456
(12.3)....457
(12.4)....458
(14.1)....539
(14.2)....539

Preface

A phone-counter owner who knows which question to ask first, a nurse at the puskesmas who knows which fever cannot wait, a mechanic at the ojek base camp who hears a weak horn and already suspects the battery: each of them carries knowledge that can be written as rules. *If this and this, then that — unless that other thing is true.* Nobody learned those rules from a book. They grew one case at a time, every time an old rule met a customer, a patient or a motorcycle it did not fit. This book is about computer programs that hold such knowledge, explain every answer they give, and keep growing the same way: case by case, from the person who knows better, without breaking what already worked.

The method at the center of it is the **Variable-Centered Intelligent Rule System**, VCIRS for short. It was proposed in a master's thesis in computer science and information engineering completed in June 2005, first published at the Information and Communication Technology Seminar (ICTS) at ITS, Surabaya, in August 2005, and revised for ICTS 2006, the same year an Indonesian edition was written for the Department of Informatics at ITS. Students at ITS then used it to build real expert systems: a fuzzy VCIRS that helps school leavers choose a study program, and a health expert system for a first, independent check of common complaints. The idea is simple to state. Classic rule-based systems answer well and explain themselves, but only an expert with a knowledge engineer can change their rules. Ripple Down Rules let an expert teach a system one case at a time, but answer only in one simple way. VCIRS joins the two: it keeps its knowledge around the **variables** the expert talks about, grouped into **nodes** and **rules**; it learns case by case; it counts how much every variable, node and rule is used; it uses those counts to rank what matters and to propose new, more general rules; and it can reason forward and backward and say *what*, *how* and *why*. This book tells that research again from the beginning, rewritten from the ground up for 2026, and turns it into applications people can use every day.

Who this book is for

You are assumed to know nothing. Not "nothing about expert systems" — nothing. If you have never written a line of code, never installed a programming language and never heard the words "knowledge base", you are exactly the reader this book was written for. A becak driver with a fever who wonders whether he can keep working, a family whose electricity bill has crept over four hundred thousand rupiah, a durian grower looking at a dark, wet patch on a trunk, an ojek driver whose scooter will not start at six in the morning, a siomay seller wondering with his daughter what she should study after school: each of them appears in these pages, and each of them should be able to follow the chapter written about their problem. Every term is introduced in plain words before it is used, every method is first worked by hand on a small example, and every program in the book was run before its output was printed here.

Why does an expert system that learns case by case matter to such readers? Because the knowledge that helps them most is local, and it changes. The advice that fits a scooter in Surabaya, a chili bed on a village slope, a cilok cart at a school gate or a clinic queue on a Monday morning is known by someone nearby, and it is never finished. A program that must be rewritten by a programmer every time its advice is wrong will soon be out of date and then abandoned. A program that the mechanic, the nurse or the shopkeeper can correct in one minute, with the case in front of them, stays useful for years, and it keeps a written record of why it says what it says.

Readers who already program will find the other half of the book: complete, tested applications, with their data structures, algorithms, measurements and tests laid out in full, ready to be adapted to a real shop, farm or clinic. That principle makes the book long. A shorter book would assume you already knew the hard parts, and such a book is no use to the person who does not.

One idea, two languages

Every application in this book is built twice, in **Java** and in **Python**, and the two versions do the same thing, read the same data files and print the same output. Java is the language of large, long-lived systems, of Android phones and of many banks and government services; its compiler catches many mistakes before the program runs. Python is the language of quick experiments, of data work and of most research in artificial intelligence; it is short and easy to read. Seeing one idea written both ways is the quickest route to understanding the idea itself rather than the habits of one language. If you know neither, start with Python and read the Java version after it; the comparison will teach you Java quickly.

Both are written and run in **Visual Studio Code**, a free editor that works the same way on Windows, macOS and Linux. Chapter 1 installs it, together with JDK 25 and Python 3.14, step by step on all three systems. Almost everything else comes with the two languages themselves: SQLite keeps the knowledge base when a chapter needs a database, each language's own small web server (or Flask for Python) serves the pages when a chapter needs a browser, and in Chapter 13 a language model runs on your own computer through Ollama, without sending anybody's messages anywhere.

One engine: vcirs-core

From Chapter 3 onwards every application stands on one shared engine, **vcirs-core**, which the book provides twice: as a Java package and as a Python package, with the same classes, the same behavior, the same printed text and the same saved files. A knowledge base taught through the Java version of an app can be opened by the Python version and gives the same answers. The engine keeps the knowledge in the three structures of VCIRS — the variables, the nodes and the rules — learns from cases with the options of the 2006 revision, computes the Variable, Node and Rule Usage Rates, generates candidate rules, and reasons forward and backward with explanations. It is small enough to read, and it is faithful to the thesis: run on the Zookeeper cases the thesis uses as its demonstration, it reproduces the thesis's tables number for number. Chapters 2 and 3 first build simpler engines by hand, so that nothing in vcirs-core is magic when we switch to it, and each later chapter opens exactly the part of the engine it needs.

What is in it

Chapter 1 looks at the rules that already run an ordinary morning, explains what makes a set of rules intelligent and how the three families — classic rule-based systems, Ripple Down Rules and VCIRS — fit together, and sets up the workbench: VS Code, JDK 25 and Python 3.14 on Windows, macOS and Linux. It ends with a first taste of VCIRS, a tiny knowledge base that learns an exception to its own rule.

Chapters 2 to 5 lay the foundations. Chapter 2 helps Bu Sri keep her family's electricity bill under four hundred thousand rupiah with the first rule program of the book: IF–THEN advice kept apart from the program, each piece with its reason and its saving in rupiah. Chapter 3 builds a phone

and Wi-Fi troubleshooter for Mas Anto's counter in the market with Ripple Down Rules, first by hand and then on vcirs-core, so that it learns the right fix every time its advice fails. Chapter 4 helps Laras and her school counselor, Bu Wati, think about study programs, and in doing so introduces the heart of VCIRS: advice turned into variables, values, conditions, nodes and rules, kept in the Variable-Centered Rule Structure. Chapter 5 builds a health self-check that asks the safety questions first and says plainly when to see a doctor, and lets the nurse at the puskesmas refine its knowledge case by case with the list of difference, the cornerstone cases and the six options of VCIRS.

Chapters 6 to 9 are about what makes VCIRS intelligent. Chapter 6 helps Pak Slamet with the troubles of his durian, mango, chili and rice, computes the Variable, Node and Rule Usage Rates, and uses them to generate new, more general rules that the village extension worker reviews before keeping. Chapter 7 routes the WhatsApp messages of a neighborhood shop to the right person in the family and reads the Credit and usage statistics the way a shopkeeper would. Chapter 8 gives ojek drivers a motorcycle fault finder that rules out danger first, reasons forward and backward over IF–THEN rules drawn from the knowledge base, and answers *what*, *how* and *why*. Chapter 9 prepares the monthly meeting of a savings-and-loan cooperative with a fuzzy VCIRS that understands *low*, *medium* and *high*, warns when a case is borderline, and refuses to learn from protected attributes.

Chapters 10 to 13 take the knowledge out into the world. Chapter 10 puts a laundry's stain-removal advisor on the web, with its knowledge base in a SQLite database, served by a Java and a Python server that answer with the same bytes. Chapter 11 lets the team of a small clinic keep a triage knowledge base together, with proposals, tests on golden cases, a second expert's review, versions and an audit trail chained with SHA-256. Chapter 12 learns rules from data, growing a decision tree from a table of animals and turning it into a VCIRS knowledge base that a biology teacher then refines. Chapter 13 lets customers talk to the Wi-Fi doctor in their own words through a local language model, while the knowledge base, checked by a person, makes every decision and a guard rejects any reply it does not back.

Chapter 14 is the capstone: a web application with which someone who has never programmed — Bu Wati the cake maker, with eleven years of notes in an exercise book — builds an expert system of her own, teaches it case by case, reads its usage statistics and accepts or ignores the rules it proposes. Chapter 15 keeps such systems alive: an advice service packaged as one file, automatic checks, backups that can be trusted, a usage log that records the advice and not the people, a monthly report that says what to teach next, and Indonesia's personal data protection law.

Every chapter follows the path a real project takes: the problem and who it helps, the idea in plain words, the application built step by step, running it, making it better, using it for real in your own shop, farm, clinic or house, common mistakes, a summary, exercises with complete solutions, and references. Where an application gives advice about health, money or safety, it says plainly when to see a doctor, a midwife, a mechanic or the people who decide.

Every program in this book was run before its output was printed. The programs are collected, chapter by chapter, in the accompanying `Code` folder, from `Chapter01` to `Chapter15`, each with a `java` and a `python` folder, the data it needs, and a README that explains how to run it in a terminal and in VS Code on Windows, macOS and Linux; the shared engine sits beside them in `vcirs-core`.

How to read it

With VS Code open and the programs running. Chapter 1 comes first for everyone, because it builds the workbench, and Chapters 3 and 4 should come before the rest, because they introduce the engine and the structure every later chapter uses; after that the chapters climb from very easy to hard, and each one names the earlier chapters it builds on, so you may also go straight to the application closest to your own life. Type the examples in rather than copying them; the typing is where the errors happen, and the errors are where the learning happens. When a program works, teach it — one new case, one correction, one condition changed in its data file — and watch how its advice changes: there is no quicker way to feel how an expert system that learns thinks. The exercises at the end of each chapter all have complete solutions — use them after your own attempt, not instead of it.

Numbered figures, tables, listings, pseudocode and equations are all live links: in the electronic edition, click "Listing 5.3" or "[4]" anywhere in the text and you are taken straight to it. The contents and the lists at the front of the book work the same way.

About the author

Irfan Subakti describes himself in one line — *Father | Educator | Entrepreneur | Bridging Tech & Nature: CS Lecturer × Crypto/Digital Business × AI/ML Specialist × Fruit Farmer | Storyteller* — and every part of that line has found its way into this book. He has taught in the Department of Informatics at ITS since 1998, and VCIRS is his own research: he proposed it, published it and revised it, and supervised the student projects that first put it to work. He supervises and examines undergraduate theses, teaches MLOps for the IBM SkillsBuild program, and builds products: he founded Profitina, an AI-visibility platform. Away from the keyboard he tends the family fruit and durian orchard, where he learned that good software, like a good orchard, is grown patiently, one well-tended piece at a time. The storyteller in him is the reason every chapter of this book begins with a person and a problem rather than with a definition.

Acknowledgments

To the students who built the first expert systems on VCIRS and showed where it was strong and where it was still thin, and to the students who worked through the earlier forms of this material and asked the questions that showed where an explanation needed more care. To the colleagues at ITS with whom intelligent systems have been taught and refined over the years. To the counter keepers, nurses, farmers, shopkeepers, drivers, mechanics, teachers and cooks whose everyday problems gave each chapter its story. The book owes its standard, too, to the official documentation of Java, Python, SQLite, Flask, VS Code and Ollama, and to the research literature on rule-based systems, Ripple Down Rules, fuzzy logic and machine learning, which is cited wherever it is used.

Surabaya, September 2026

Irfan Subakti · yifana@gmail.com

Profitina

profitina.com

Chapter 1

Intelligent Rules in Daily Life, and Your Workbench

By the end of this chapter you will be able to recognize the rules that already run your day, say in plain words what makes a set of rules *intelligent*, and explain the three ideas this whole book rests on: rules that answer questions and explain themselves, rules that are taught one case at a time, and the Variable-Centered Intelligent Rule System (VCIRS) that joins the two. You will also build your workbench: install the Java Development Kit 25, Python 3.14 and Visual Studio Code on Windows, macOS or Linux, check that each works, and run your first three programs in both languages, the last of which is a tiny knowledge base that learns an exception to its own rule.

1.1 A Morning Full of Rules

Pak Darto drives an *ojek*, a motorcycle taxi, in Sidoarjo. Before he leaves the house at half past five he looks at the sky, feels the air, glances at his phone and presses his thumb on the back tyre. He does not think of what he is doing as "running rules", but that is exactly what it is. *If the sky is dark and it is the rainy season, take the raincoat. If the phone is below thirty percent, charge it before leaving, because the passengers find him through an app. If the tyre feels soft, stop at the first workshop.* Every one of these sentences has the same shape: a condition he can check, and an action or a conclusion that follows from it.

Two streets away, Bu Sri stands in front of her electricity meter with the month's receipt in her hand. The bill is higher than last month. She knows a few rules too: *if the air conditioner runs all night, the bill goes up; if the old kitchen bulb is still there, replace it.* But she is not sure which rule matters most for her own house, and nobody has told her how much each change would save.

At the corner of the market, Mas Anto runs a phone counter, a *konter HP*. All day long people come to him saying "the internet is dead". He has learned, customer after customer, what usually fixes it. *If the Wi-Fi symbol is off, turn Wi-Fi on. But if the little airplane is on, turn off airplane mode first. If the phone is fine but no other phone in the house gets internet either, restart the router — unless the router's light is red, because then it is the provider's cable, and restarting will not help.* Notice the word *unless*. Mas Anto's knowledge did not arrive as a finished list; it grew, one exception at a time, each time a customer proved his earlier rule incomplete.

These three people are the heroes of the first chapters of this book. None of them is a programmer, and none of them needs to become one to benefit from what we will build. What they share is that their know-how can be written as rules, and that written rules can be given to a computer that will then apply them faithfully, every time, to every case, and say *why* it reached its answer. When those rules can also be *corrected* by the person who knows better, case by case, without breaking what already worked, we have the kind of system this book is about.

> **KEY INSIGHT.** A great deal of everyday expertise is already a set of rules in someone's head. Writing the rules down turns private know-how into something that can be checked, shared, explained and improved.

1.2 What Makes a Rule Intelligent

1.2.1 Facts, conditions and conclusions

Let us name the parts of Pak Darto's first rule, because we will use these names on almost every page that follows.

- A **fact** is something that is known about the situation right now: *the sky is dark*, *it is the rainy season*, *the battery is at 25 percent*. We will write facts as a **variable** and a **value**: `sky = dark`, `battery = 25`.
- A **condition** is a test on a fact: `sky = dark` is true this morning; `battery < 30` is true too. A condition uses an **operator**, one of `=`, `<>` (is not), `<`, `<=`, `>` and `>=`.
- A **conclusion** is what the rule says when all its conditions are true: `bring = raincoat`.
- A **rule** joins them: *IF* the conditions hold, *THEN* the conclusion follows. In the form used throughout this book,

```
IF   sky = dark
AND  season = rainy
THEN bring = raincoat
```

A collection of such rules about one subject is a **knowledge base**, and the program that takes the facts of a case, checks the rules and draws the conclusions is an **inference engine**. The place where the facts of the current case are kept while the engine works is called **working memory**. Together these make up a **rule-based system**, also called an **expert system**: a program that gives advice in a narrow field in the way an expert would [1].

1.2.2 Three things that make rules intelligent

A list of `if` statements in a program is not yet intelligent. Three abilities make the difference, and each will get a full chapter later in the book.

It explains itself. Ask Pak Darto why he took the raincoat and he will tell you: "the sky was dark, and it is December". A good expert system does the same. It can answer three questions about every conclusion [1], [2]: *What* did you conclude? *How* did you get there — which rules fired, using which facts? And *Why* — why do you ask me this question, or why did this rule apply? The classic medical expert system MYCIN made this famous in the 1970s: a doctor could type WHY at any question and the program would show the rule it was trying to prove [2]. An explanation is not a luxury. It is what lets a person trust the advice, or spot quickly where it went wrong.

It learns from the person who knows. Knowledge changes. Mas Anto meets a new kind of problem every week. In a traditional rule-based system, adding a rule means asking a *knowledge engineer* — a specialist who interviews the expert and edits the rule base — and hoping the new rule does not break an old one. A system that lets Mas Anto himself say "you were wrong here; in this case the right fix is this, and this is what makes the case different" learns where the learning is needed, in the context of a real case [3], [4]. This is the idea of **Ripple Down Rules**, which Chapter 3 builds from nothing.

It improves itself over time. A system that counts which facts and rules are used most can ask the most useful questions first, point to the knowledge that matters most, and even propose new rules for the expert to confirm. This is the part that the Variable-Centered Intelligent Rule System adds [5], [6], and it is why the word *intelligent* appears in its name: the system "learns" from its users while they build and use it.

> **DEFINITION.** An *intelligent rule system* is a rule-based system that (1) draws conclusions from facts with IF-THEN rules, (2) explains what it concluded, how and why, and (3) lets its users improve its knowledge, case by case, while it keeps track of which knowledge is used and useful.

1.2.3 Rules you already trust

Rules of this kind are all around you. The phone that dims its screen when the battery is low follows a rule. The motorcycle's side stand that cuts the engine if it is down while you shift into gear follows a rule. A bank's system that holds a card payment made in two cities within ten minutes follows a rule. The pharmacist who will not sell a certain medicine without a prescription follows a rule. What changes from one to the next is how many rules there are, who is allowed to change them, and whether anyone can ask *why*.

A good test of whether a problem suits a rule system is to ask: *could a patient expert explain their decision to a beginner in "if … then …" sentences?* Troubleshooting a phone, choosing what to plant, deciding which customer message is urgent, finding which part of a motorcycle is broken: all of these pass the test, and all of them appear as applications in this book. Recognizing faces in photos or understanding spoken words do not pass it — the knowledge there is not in sentences — and for those we use other tools, some of which Chapter 13 connects to a rule system.

1.3 Three Families of Rule Systems

The Variable-Centered Intelligent Rule System was proposed as a way of combining two older families of rule systems, each strong where the other is weak [5], [6]. **Figure 1.1** shows the idea; the rest of this section explains it in plain words.

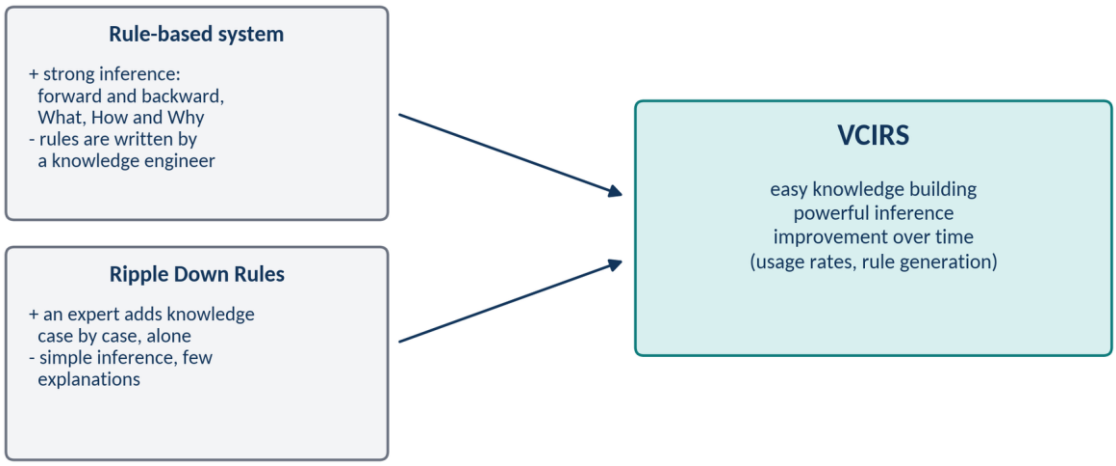

Figure 1.1 Two families of rule systems and the hybrid that joins them. VCIRS takes the inference and explanations of rule-based systems and the case-by-case knowledge building of Ripple Down Rules, and adds usage statistics and rule generation.

1.3.1 Rule-based systems: strong answers, hard to change

A classic **rule-based system** (RBS) keeps a flat collection of IF-THEN rules. Its inference engine can work in two directions. In **forward chaining** it starts from the facts it knows and fires every rule whose conditions are satisfied, adding each new conclusion to working memory, until nothing new can be concluded; this is how Pak Darto thinks in the morning, from what he sees to what he should do. In **backward chaining** it starts from a question — *is it a tiger?* — and looks for rules that would conclude it, then tries to establish their conditions, asking the user only for

what no rule can provide; this is how a doctor works from a suspected illness back to the tests that would confirm it. Both give rich explanations: What, How and Why [1].

The weakness of an RBS is maintenance. Rules interact. A new rule placed at the top of a list can hide an old one; placed at the bottom it may never be reached. Keeping a large rule base consistent is skilled, slow work, which is why a knowledge engineer is usually needed [5]. Chapter 3 shows this problem happening in a tiny troubleshooter of just three rules.

1.3.2 Ripple Down Rules: easy to teach, simple answers

Ripple Down Rules (RDR) were invented in Sydney at the end of the 1980s to let medical experts maintain a pathology expert system themselves [7], [4]. The central idea is that an expert never has to see or rearrange the rule base. The system is used on real cases. When it gives a wrong answer, the expert says what the right answer is and picks, from a list of differences, the features that make this case different from the case that originally created the rule that answered wrongly — the **cornerstone case**. The new rule is attached as an **exception** exactly where the wrong answer came from, so it can only affect cases like this one. Rules are never deleted, only refined. A system built this way at St Vincent's Hospital in Sydney, PEIRS, grew to more than two thousand rules added entirely by pathologists [5], [4]; the same method is used today in commercial systems that interpret laboratory results [4]. A later variant, Multiple Classification RDR, lets one case receive several conclusions at once, for example a patient with two independent problems [8].

The weakness of RDR is the other side of its strength. Its inference is a simple walk down a tree of exceptions, and it offers little of the question answering that an RBS gives: no backward chaining from a goal, few explanations beyond "this rule fired" [5].

1.3.3 VCIRS: the best of both

The **Variable-Centered Intelligent Rule System** keeps its knowledge in a special structure — the *Variable-Centered Rule Structure* — that serves both purposes at once [5], [6]. Knowledge is added the RDR way, case by case, with cornerstone cases and exceptions, and with six clearly defined choices for the expert [3]. The same knowledge can be read the RBS way, turned into ordinary IF-THEN rules and used for forward and backward chaining with full explanations. And because the structure is built around *variables*, the system can count how often each variable, each node and each rule is used, rank what is important, guide the user to the most useful questions and generate candidate rules for the expert to confirm.

The rest of this book builds real applications on that idea, from the very easy to the very hard. Before we build anything, let us look inside the structure once, so that its words become familiar.

1.4 VCIRS in Plain Words

1.4.1 Variables, nodes and rules

Figure 1.2 shows the three levels of the Variable-Centered Rule Structure on a two-case example that we will run at the end of this chapter.

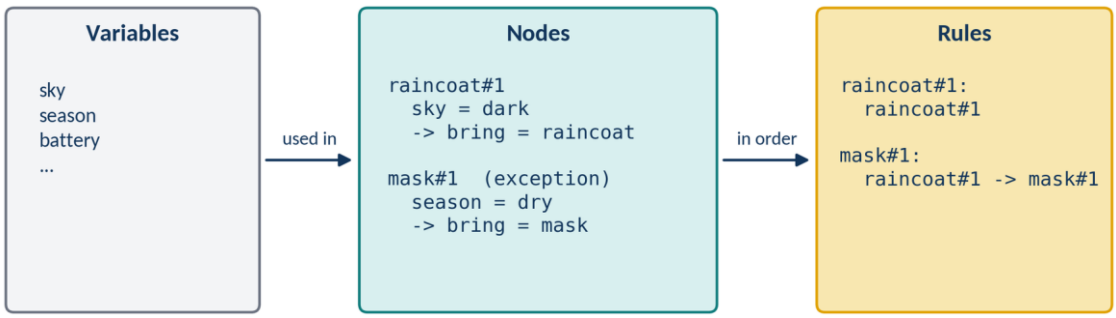

Figure 1.2 The Variable-Centered Rule Structure. Variables are used in nodes; a node is a short sequence of conditions with a conclusion; a rule is a sequence of nodes, top level first.

- A **variable** is one kind of fact: `sky`, `season`, `battery`. The system keeps one entry for each variable, however many rules use it, and it remembers where each one is used.
- A **node** is a small sequence of conditions with one or more conclusions. The node `raincoat#1` says: `sky = dark -> bring = raincoat`. The name comes from its first conclusion, with a serial number so that names stay unique.
- A **rule** is a sequence of nodes. The first node of a rule sits at the top level; each following node is an exception of the one before. The rule `mask#1` is the sequence `raincoat#1 -> mask#1`: *if the sky is dark, bring a raincoat — except when the season is dry, then bring a mask*, because a dark sky in the dry season is usually smoke or haze. A rule is named after its last node, and its conclusion is the conclusion of its last node.

This is exactly how the thesis that proposed VCIRS defines the terms: a rule is a sequence of one or more nodes, a node is a sequence of one or more variables with their values, and a node or a rule may have one or more conclusions [5].

1.4.2 Building knowledge case by case

A **case** is what a user posts: the facts of a real situation and, when building knowledge, the right conclusion. When the system gets a case, it first looks for **proper nodes**, nodes that share variables or conclusions with the case, ranked from the *perfect* node (same variables, same values, same conclusion) down to nodes that merely share a variable [5]. It then compares the new case with the cornerstone case of the chosen node and prints a **list of difference** in two sections: *A*, the conditions of the old case that the new case lacks, and *B*, the conditions of the new case that the old case lacks [3]. The expert chooses among six options: accept the existing knowledge (option 0); make an exception from the negation of section A (1), from section B (2), or from both (3); or store the whole case as a new node beside the old one (4) or at the top level (5) [3]. Chapter 3 walks through the list of difference slowly, and Chapter 5 uses every option on a real application.

1.4.3 Counting what is used

Every time a case agrees with a variable of an existing node, that variable earns a **Credit**. From the credits, from how many nodes share each variable, and from how close each variable stands to the conclusion of its node, the system computes three **usage rates**: the Variable Usage Rate (VUR), the Node Usage Rate (NUR) and the Rule Usage Rate (RUR) [5]. The larger a rate, the more that piece of knowledge is used. The rates tell the user which questions to ask first, and they drive **rule generation**: the system combines the most important nodes, in an order that respects the existing rules, into candidate rules that the expert confirms or rejects. Chapter 6 generates rules for farmers and Chapter 7 puts the statistics to work in a small shop.

1.4.4 Answering with explanations

Finally, VCIRS can answer in three ways [5]. The **RDR way** starts from the proper nodes of the facts, confirms their conditions — asking the user for anything unknown — and follows the exceptions down to the most specific conclusion. The **forward-chaining** and **backward-chaining** ways first turn the structure into ordinary IF-THEN rules, a step called **knowledge base transformation**, and then chain through them. All three record every step, so they can say what they concluded, how, and why. Chapter 8 builds a motorcycle fault finder on forward and backward chaining.

Table 1.1 shows where each of these ideas becomes an application.

Table 1.1 The road map of the book: each chapter builds a real application and adds one idea of VCIRS.

Ch	Application	The VCIRS idea it brings
1	Morning rules and a first taste (this chapter)	facts, conditions, conclusions; your workbench
2	Electricity-bill saver	plain IF-THEN rules with reasons and savings
3	Phone and Wi-Fi troubleshooter that learns	Ripple Down Rules, cornerstone cases, exceptions
4	Study-program advisor	thinking in variables: the VCIRS core
5	Health self-check advisor	the six options of case-by-case refinement
6	Crop and fruit-tree problem advisor	rule generation
7	Small-shop customer-message router	usage and credit statistics
8	Motorcycle fault finder	forward and backward inference with explanations
9	Cooperative-loan eligibility advisor	fuzzy VCIRS
10	Laundry stain-removal web app	a VCIRS knowledge base in a database
11	Clinic triage knowledge base for a team	many experts, versions and audit
12	Learning rules from data	from a decision tree to a knowledge base
13	Talking to your expert system	a local language model in front of checked knowledge
14	A VCIRS expert-system builder	everything together, for anyone
15	Deploying and maintaining	keeping expert systems alive

1.4.5 One engine, two languages

Every application from Chapter 3 onwards uses one shared engine, **vcirs-core**, which this book provides twice: as a Java package and as a Python package, with the same classes, the same behavior, the same printed text and the same saved files. A knowledge base taught in the Java version of an app can be opened by the Python version and give the same answers. The engine is small enough to read, and it is faithful to the thesis that defined VCIRS: run on the fifteen Zookeeper cases — Winston's classic little rule base for recognizing animals [9], which the thesis uses as its demonstration — it reproduces the thesis's tables of usage rates and of generated rules number for number [5]. Chapters 2 and 3 first build simpler versions by hand, so that nothing in the engine is magic when we switch to it.

1.5 Why Java, and Why Python

This book writes every program twice, and it does so on purpose.

Java is the language of large, long-lived systems: banking, government services, Android apps, the servers of many online shops. It is *compiled*: a tool called `javac` checks your whole program and

translates it into *bytecode*, which the Java Virtual Machine then runs on Windows, macOS or Linux alike. Java insists that you say what type each variable has, which feels strict at first and saves you from whole families of mistakes later. Java 25, released in September 2025, is a **long-term support** (LTS) version: it receives security and bug-fix updates for years, which is why this book uses it [10], [11]. A newer non-LTS version, Java 27, appeared on 15 September 2026 [12]; every program in this book also runs on it, and on any version from 17 onwards.

Python is the language of quick tools, data work and artificial intelligence. It is *interpreted*: the `python` program reads your file and runs it line by line, so there is no separate compile step. Its code is short and reads almost like English, which makes it ideal for trying an idea in a few minutes. Python 3.14 is the current version; its newest update in September 2026 was 3.14.7, released in August 2026 [13].

Seeing the same program in both languages is the fastest way to understand what is *idea* and what is *spelling*. A rule is a rule, whether it is written `if (battery < 30)` or `if battery < 30:`. When you can read both, you can join a team that uses either, and you can move a knowledge base from a quick Python prototype to a Java service without changing a single rule.

1.6 The Tools of Your Workbench

You need four things, all free.

1. **The Java Development Kit (JDK) 25.** The JDK contains the compiler `javac`, the runner `java`, and the standard library. We use **Eclipse Temurin**, a free, open-source build of the JDK maintained by the Adoptium project. Its newest update in September 2026 was 25.0.4, released in August 2026, followed on 2 September 2026 by a security-only patch, 25.0.4.1 [10], [11].
2. **Python 3.14**, from python.org or from your system's package manager [13].
3. **Visual Studio Code** (VS Code), a free editor from Microsoft that works on all three systems. It updates itself often; the version released on 23 September 2026 was 1.139 [14].
4. **Two VS Code extensions**: the **Extension Pack for Java** and the **Python** extension, both published by Microsoft [15], [16].

Figure 1.3 shows how these tools fit together. A Java source file such as `Hello.java` is translated by `javac` into `Hello.class`, which the `java` command runs; a Python file such as `hello.py` is run directly by `python`. VS Code is simply the window in which you write the files, press the buttons that run these tools, and read what they print.

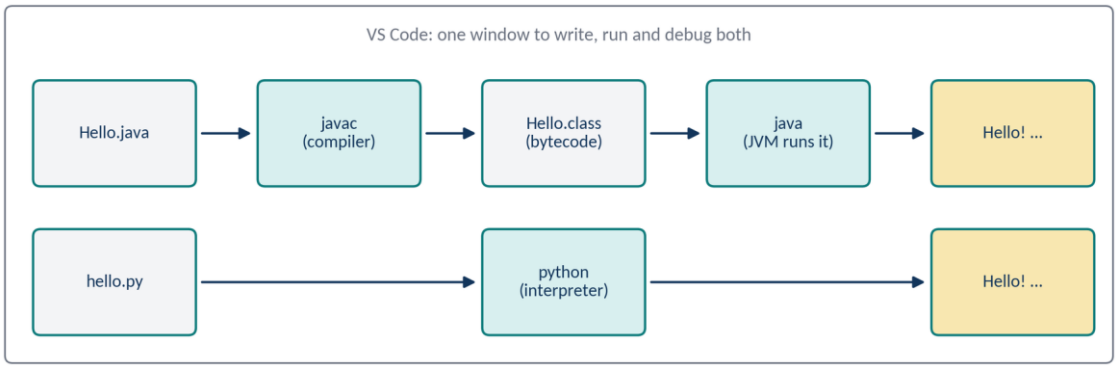

Figure 1.3 The workbench. Java source is compiled by javac into bytecode, which java runs; Python source is run directly by the python interpreter; VS Code holds both.

Table 1.2 collects the versions used to test every program in this book and the smallest versions that will still run them.

Table 1.2 The workbench of this book, as checked in September 2026. Newer update numbers are always fine.

Tool	Version in this book	Also works with	Where it comes from
JDK (Eclipse Temurin)	25 LTS (25.0.4)	17 and later, including 27	adoptium.net
Python	3.14 (3.14.7)	3.10 and later	python.org
VS Code	1.139 (September 2026)	any recent version	code.visualstudio.com
Extension Pack for Java	current	—	VS Code Extensions view
Python extension	current	—	VS Code Extensions view

A word on version numbers. Java 25 and Python 3.14 receive regular updates that fix bugs and security problems: 25.0.1, 25.0.2 … and 3.14.1, 3.14.2 … Always take the newest update of the version the book names. The numbers after the second dot will be larger on your screen than in this chapter, and that is exactly as it should be.

1.7 Setting Up on Windows, macOS and Linux

Install the language tools first, check them in a terminal, then install the editor, then connect the two. Checking in a terminal *before* opening VS Code is important: if something is wrong with Java or Python, a terminal gives a short, clear message, whereas an editor may give a confusing one. A **terminal** is a window in which you type commands instead of clicking: *PowerShell* or *Command Prompt* on Windows, *Terminal* on macOS and Linux.

Table 1.3 puts the whole procedure for the three systems on one page, so that you can come back to it later; the subsections after it explain each step.

Table 1.3 The workbench on three operating systems, from an empty computer to a working editor. On Fedora use `sudo dnf install java-25-openjdk-devel python3.14`; on Arch Linux, `sudo pacman -S jdk-openjdk python`.

Step	Windows 10/11	MacOS	Linux (Ubuntu/Debian)
1. JDK	Temurin 25 `.msi` from adoptium.net, or `winget install EclipseAdoptium.Temurin.25.JDK`	Temurin 25 `.pkg` from adoptium.net, or `brew install --cask temurin@25`	add the Adoptium repository, then `sudo apt install temurin-25-jdk`
2. Python	installer from python.org (tick **Add python.exe to PATH**), or `winget install Python.Python.3.14`	installer from python.org, or `brew install python@3.14`	`sudo apt install python3.14 python3.14-venv` (deadsnakes archive on Ubuntu)
3. Check Java	`java -version`, `javac -version`	the same	the same
4. Check Python	`python --version` (or `py --version`)	`python3 --version`	`python3.14 --version`
5. VS Code	installer from code.visualstudio.com	download, drag into *Applications*	`.deb` from code.visualstudio.com, or `sudo snap install code --classic`
6. Extensions	Extensions view (`Ctrl+Shift+X`): **Extension Pack for Java**, **Python**	the same with `Cmd+Shift+X`	the same with `Ctrl+Shift+X`

1.7.1 Windows, step by step

Install the JDK.

1. Open a web browser and go to **adoptium.net**. The front page offers the latest LTS release of Temurin for your computer; make sure it says **25** and **Windows x64** (or **aarch64** on a laptop with an ARM processor), and download the **.msi** installer of the **JDK**.
2. Run the installer. On the page called *Custom Setup*, make sure that **Add to PATH** and **Set JAVA_HOME variable** are both set to be installed: click each one and choose *Will be installed on local hard drive*. The *PATH* is the list of folders in which Windows looks for commands; adding the JDK to it is what lets you type `java` in any terminal. Accept the rest and click **Install**.
3. If you prefer typing to clicking, Windows 10 and 11 include a package manager called **winget**. Open PowerShell and type `winget install EclipseAdoptium.Temurin.25.JDK`; it downloads and runs the same installer.

Install Python.

4. Go to **python.org**, open **Downloads → Windows**, and download Python **3.14**. The page offers the *Python install manager*, which also installs a `py` command that can keep several versions side by side, as well as a classic installer. Either is fine. With the classic installer, **tick the box "Add python.exe to PATH"** on its first page before clicking *Install Now*; this is the single most common thing beginners miss. With the install manager, follow its prompts, and type `py install 3.14` in a new terminal if it asks which version to install.
5. Alternatively, in PowerShell: `winget install Python.Python.3.14`.

Check both.

6. Close every terminal you have open and open a *new* PowerShell window; old windows keep the old PATH and will not find the new tools. Type these three commands, pressing Enter after each:

```
java -version
javac -version
python --version
```

You should see the version of the runtime (three lines that begin `openjdk version "25.0.4"` or a later update), the version of the compiler (`javac 25.0.4`), and the version of Python (`Python 3.14.7`).

If `python --version` opens the Microsoft Store instead of printing a version, Windows is running a small placeholder called an *app execution alias* instead of the real Python. Try `py --version`; if that works, type `py` wherever this book says `python`. To make `python` itself work, open **Settings → Apps → Advanced app settings → App execution aliases**, switch off the two entries named *python.exe* and *python3.exe*, and open a new terminal.

1.7.2 macOS, step by step

Install the JDK. Go to **adoptium.net**, choose **Temurin 25**, **macOS**, and the right processor: **aarch64** for a Mac with Apple silicon (M1 and later), **x64** for an older Intel Mac. If you are not sure, open the Apple menu → **About This Mac**: it says *Chip Apple M…* or *Processor Intel…*. Download the **.pkg** installer and run it. It puts the JDK in `/Library/Java/JavaVirtualMachines/`, where macOS finds it by itself; there is no PATH to set. With the **Homebrew** package manager, `brew install --cask temurin@25` does the same.

Install Python. macOS may already have a `python3` command, but it often belongs to Apple's developer tools and is older; do not rely on it. Go to **python.org → Downloads → macOS**, download the **Python 3.14** installer and run it. At the end, the installer opens a folder; double-click **Install Certificates.command** in it, so that Python can make secure web connections. With Homebrew: `brew install python@3.14`.

Check both. Open **Terminal** (press `Cmd+Space`, type *Terminal*, press Enter) and type:

```
java -version
javac -version
python3 --version
```

On macOS type `python3`, not `python`: the name `python` is either missing or points somewhere unhelpful. Everywhere this book writes `python`, a Mac user types `python3`.

1.7.3 Linux, step by step

Linux distributions install software from **repositories** through a package manager. The commands below are for Ubuntu and Debian; **Table 1.3** gives the equivalents for Fedora and Arch.

Install the JDK. Your distribution may already offer `openjdk-25-jdk`; try `sudo apt install openjdk-25-jdk` first. If it is not found, add the Adoptium repository, which always carries the current Temurin releases. Type these lines one at a time; the long ones are single lines:

```
sudo apt install -y wget apt-transport-https gpg
wget -qO - https://packages.adoptium.net/artifactory/api/gpg/key/public | gpg --dearmor |
sudo tee /etc/apt/trusted.gpg.d/adoptium.gpg > /dev/null
echo "deb https://packages.adoptium.net/artifactory/deb $(awk -F=
'/^VERSION_CODENAME/{print$2}' /etc/os-release) main" | sudo tee
/etc/apt/sources.list.d/adoptium.list
sudo apt update
sudo apt install temurin-25-jdk
```

The first line installs three small helpers; the second downloads the key with which Adoptium signs its packages, so that your system can check they are genuine; the third tells the package manager where the repository is; the last two refresh the list of packages and install the JDK. Type your password when `sudo` asks for it — nothing appears on the screen as you type, which is normal.

Install Python. Ubuntu ships Python 3 as part of the system, usually not the newest version, and you must never remove or replace the system's own Python, because the operating system itself uses it. Install 3.14 *beside* it. On Ubuntu the usual source is the *deadsnakes* archive:

```
sudo add-apt-repository ppa:deadsnakes/ppa
sudo apt update
sudo apt install python3.14 python3.14-venv
```

Then `python3.14` runs the new version and `python3` still runs the system's. If your distribution already has Python 3.10 or later as `python3` — check with `python3 --version` — you may simply use that for this book.

Check both with `java -version`, `javac -version` and `python3.14 --version`. If `java -version` reports an older Java that was already installed, choose the new one with `sudo update-alternatives --config java`, and again for `javac`.

1.7.4 VS Code and its two extensions

Download VS Code from **code.visualstudio.com** and install it: on Windows run the installer and tick **Add to PATH**; on macOS open the downloaded file and drag *Visual Studio Code* into *Applications*; on Ubuntu install the `.deb` file (double-click it, or `sudo apt install ./code_*.deb`) or use `sudo snap install code --classic`.

Open VS Code and click the **Extensions** icon on the left (four small squares), or press `Ctrl+Shift+X` (`Cmd+Shift+X` on a Mac). Search for **Extension Pack for Java** and click **Install**; then search for **Python** and install the extension published by Microsoft [15], [16]. The Java pack

brings code completion, error checking as you type, a **Run** link above every `main` method and a debugger; the Python extension does the same for Python and adds a ▷ **Run Python File** button at the top right of the editor.

The status bar at the bottom of the window tells you which Python VS Code uses. Click the Python version shown there (or press `Ctrl+Shift+P` and type *Python: Select Interpreter*) and choose Python 3.14. For Java, VS Code finds the JDK through `JAVA_HOME` or the PATH; if it complains that no JDK is found, run the command *Java: Configure Java Runtime* and point it at the JDK 25 you installed.

1.7.5 The book's code on your computer

Every program in this book is in a folder called `Code_new`, arranged as **Table 1.4** shows. Put the folder somewhere simple, such as `Documents\VCIRS\Code_new` on Windows or `~/VCIRS/Code_new` on macOS and Linux, and open it in VS Code with **File → Open Folder…**. Keep the folders side by side as they are: the chapter programs find the shared engine by going two folders up and then into `vcirs-core`.

Table 1.4 The folders of the book's code. Every chapter folder has the same shape.

Folder	What is in it
`Code_new/vcirs-core/java/src/vcirs`	the shared engine in Java (package `vcirs`)
`Code_new/vcirs-core/python/vcirs`	the same engine in Python (package `vcirs`)
`Code_new/vcirs-core/data`	the Zookeeper cases of the thesis and their knowledge base
`Code_new/ChapterNN/java`	the Java programs of chapter NN
`Code_new/ChapterNN/python`	the Python programs of chapter NN
`Code_new/ChapterNN/data`	the data files of chapter NN
`Code_new/ChapterNN/README.md`	how to run everything in that chapter

1.7.6 When something goes wrong

Almost every setup problem is one of a handful, listed in **Table 1.5** with its fix. The rule behind most of them is the same: after installing anything, open a *new* terminal.

Table 1.5 The usual setup problems and their fixes.

What you see	Why	What to do
`'java' is not recognized…` or `command not found: java`	the JDK is not on the PATH	reinstall with **Add to PATH** ticked (Windows), or open a new terminal
`java -version` shows 8, 11 or 17	an older JDK is found first	Windows: move the JDK 25 entry up in *Environment Variables* → *Path*; Linux: `update-alternatives`; macOS: remove the old JDK or set `JAVA_HOME`
`javac` not found but `java` works	a runtime (JRE) was installed, not the JDK	install the **JDK**
`python` opens the Microsoft Store	an app execution alias	use `py`, or switch the aliases off (Section 1.7.1)
`python3.14: command not found` on Ubuntu	the deadsnakes archive was not added	repeat the three commands of Section 1.7.3
VS Code: *No Java runtime found*	VS Code cannot see the JDK	*Java: Configure Java Runtime*, then restart VS Code
VS Code runs the wrong Python	another interpreter is selected	*Python: Select Interpreter* → 3.14
`error: class Hello is public, should be declared in a file named Hello.java`	the file name and the class name differ	name the file exactly like the class, with capitals

1.8 Your First Program in Two Languages

The first program of any book checks that the workbench works. **Listing 1.1** is ours in Java. Besides saying hello, it asks the Java runtime which version it is and says whether that is new enough for this book.

Listing 1.1 `Hello.java`: the first Java program; it checks that Java is new enough.

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello! Java is working on this computer.");
        int version = Runtime.version().feature();   // 25 for JDK 25
        if (version >= 17) {
            System.out.println("It is new enough for every program in this book.");
        } else {
            System.out.println("It is too old: please install JDK 25.");
        }
    }
}
```

Read it line by line. `public class Hello` declares a **class**, the unit in which Java keeps code; its name must match the file name exactly, capital letter included. `public static void main(String[] args)` is the **main method**, where every Java program starts running. `System.out.println(...)` prints one line. `Runtime.version().feature()` asks the running Java for its main version number, such as 25, and stores it in the variable `version`, whose type is `int` (a whole number). Then comes our first rule of the book, written in Java: *if* the version is at least 17, say it is new enough, *else* ask for JDK 25.

To run it, open a terminal in the chapter's Java folder — in VS Code, **Terminal → New Terminal**, then `cd Chapter01/java` (`cd Chapter01\java` on Windows) — and type two commands. The first compiles the file; the second runs the class, named without `.java`:

```
javac Hello.java
java Hello
```

```
Hello! Java is working on this computer.
It is new enough for every program in this book.
```

`javac` prints nothing when all is well, and leaves a file `Hello.class` beside the source. You can also click **Run** above `main` in VS Code, which does both steps for you, or run a single-file program in one step with `java Hello.java`.

Listing 1.2 is the same program in Python.

Listing 1.2 `hello.py`: the first Python program; it checks that Python is new enough.

```
import sys

print("Hello! Python is working on this computer.")
if sys.version_info >= (3, 10):        # (3, 14) for Python 3.14
    print("It is new enough for every program in this book.")
else:
    print("It is too old: please install Python 3.14.")
```

There is no class and no `main`: Python simply runs the file from top to bottom. `import sys` makes the `sys` module available, and `sys.version_info` is Python's version as a tuple such as `(3, 14, 7, ...)`. Comparing it with `(3, 10)` asks "is this 3.10 or later?". The indented lines belong to the `if` or to the `else`; in Python the **indentation** is part of the language, not decoration. Run it with `python hello.py` (`python3 hello.py` on macOS and Linux) or with the ▷ button:

```
Hello! Python is working on this computer.
It is new enough for every program in this book.
```

If both programs print these lines, your workbench is ready.

1.9 Your First Rules

Now let us write Pak Darto's morning as a program. **Listing 1.3** keeps the facts of one morning in four variables and checks three rules. Each rule that fires prints what to do and, on the next line, *because* — its reason.

Listing 1.3 `MorningRules.java`: three everyday IF-THEN rules, each with its reason.

```java
public class MorningRules {
    public static void main(String[] args) {
        // The facts of this morning. Change them and run the program again.
        String sky = "dark";          // "clear", "cloudy" or "dark"
        boolean rainySeason = true;   // November to March in most of Java
        int batteryPercent = 25;      // the phone's battery
        int tyrePressure = 26;        // rear tyre, in psi

        int fired = 0;
        // Rule 1
        if (sky.equals("dark") && rainySeason) {
            System.out.println("Rule 1: bring the raincoat.");
            System.out.println("  because the sky is dark and it is the "
                    + "rainy season");
            fired++;
        }
        // Rule 2
        if (batteryPercent < 30) {
            System.out.println("Rule 2: charge the phone before leaving.");
            System.out.println("  because the battery is at " + batteryPercent
                    + "%, below 30%");
            fired++;
        }
        // Rule 3
        if (tyrePressure < 29) {
            System.out.println("Rule 3: pump the rear tyre at the first workshop.");
            System.out.println("  because the pressure is " + tyrePressure
                    + " psi, below 29 psi");
            fired++;
        }
        if (fired == 0) {
            System.out.println("No rule fired: have a good ride!");
        }
    }
}
```

The facts come first: a `String` (text) for the sky, a `boolean` (true or false) for the season, and two `int`s. Each rule is an `if` statement whose condition joins tests with `&&`, Java's *and*. When a rule fires, the program prints the conclusion and the reason and adds one to `fired`, which counts the rules that fired; if none did, it wishes a good ride. Running it prints:

```
Rule 1: bring the raincoat.
  because the sky is dark and it is the rainy season
Rule 2: charge the phone before leaving.
  because the battery is at 25%, below 30%
Rule 3: pump the rear tyre at the first workshop.
  because the pressure is 26 psi, below 29 psi
```

Listing 1.4 is the same in Python, where *and* is written `and`, and the counter goes up with `fired += 1`.

Listing 1.4 `morning_rules.py`: the same three rules in Python.

```
# The facts of this morning. Change them and run the program again.
sky = "dark"              # "clear", "cloudy" or "dark"
rainy_season = True       # November to March in most of Java
battery_percent = 25      # the phone's battery
tyre_pressure = 26        # rear tyre, in psi

fired = 0
# Rule 1
if sky == "dark" and rainy_season:
    print("Rule 1: bring the raincoat.")
    print("  because the sky is dark and it is the rainy season")
    fired += 1
# Rule 2
if battery_percent < 30:
    print("Rule 2: charge the phone before leaving.")
    print("  because the battery is at " + str(battery_percent)
          + "%, below 30%")
    fired += 1
# Rule 3
if tyre_pressure < 29:
    print("Rule 3: pump the rear tyre at the first workshop.")
    print("  because the pressure is " + str(tyre_pressure)
          + " psi, below 29 psi")
    fired += 1
if fired == 0:
    print("No rule fired: have a good ride!")
```

Its output is identical, line for line:

```
Rule 1: bring the raincoat.
  because the sky is dark and it is the rainy season
Rule 2: charge the phone before leaving.
  because the battery is at 25%, below 30%
Rule 3: pump the rear tyre at the first workshop.
  because the pressure is 26 psi, below 29 psi
```

Small as it is, this program already has the three parts of every rule system in this book: **facts** (the four variables at the top), **rules** (the three `if` statements) and an **explanation** (the *because* lines). What it lacks is just as instructive. The rules are mixed with the code, so changing a rule means changing the program. The program cannot answer a question such as "what would make you tell me to take the raincoat?", which is backward chaining. And it cannot learn: if Pak Darto finds out that in August a dark sky means haze, not rain, someone must edit the source. Chapter 2 separates the rules from the program; the next section gives a first taste of a knowledge base that learns.

> **PITFALL.** In Java, text is compared with `.equals`, as in `sky.equals("dark")`, never with `==`. The operator `==` asks whether two variables point at the *same object* in memory, which may be false even when the words are the same. Python's `==` compares the words themselves, so `sky == "dark"` is correct there.

1.10 A First Taste of VCIRS

The last program of this chapter uses the shared engine. It creates an empty knowledge base, teaches it two cases, and then asks it about a new morning. **Listing 1.5** is the Java version.

Listing 1.5 `FirstTaste.java`: a knowledge base learns a rule and then an exception to it.

```java
import java.util.List;

import vcirs.Case;
import vcirs.Facts;
import vcirs.InferenceResult;
import vcirs.KnowledgeBase;
import vcirs.LearnResult;

public class FirstTaste {
    public static void main(String[] args) {
        KnowledgeBase kb = new KnowledgeBase("morning");
        // Case 1: a dark sky means rain is coming.
        LearnResult r1 = kb.learn(Case.of(
                List.of("sky = dark"),
                List.of("bring = raincoat")));
        System.out.println("Case 1: " + r1);
        // Case 2: but in the dry season a dark sky is only smoke or haze.
        LearnResult r2 = kb.learn(Case.of(
                List.of("sky = dark", "season = dry"),
                List.of("bring = mask")));
        System.out.println("Case 2: " + r2);
        // Ask the knowledge base about a new morning.
        InferenceResult answer = kb.infer(Facts.of("sky = dark", "season = dry"));
        for (String line : answer.what()) {
            System.out.println(line);
        }
        for (String line : answer.how()) {
            System.out.println(line);
        }
    }
}
```

The `import` lines bring in five classes of the `vcirs` package. `Case.of` builds a case from two lists: the facts, written as `"variable = value"`, and the right conclusion. `kb.learn(...)` is the whole Ripple-Down-Rules loop in one call: the knowledge base tries the case; if its answer is already right it accepts the case; if some node gave a wrong answer it adds an exception under that node; if no node answered it adds a new node at the top level. The first case lands at the top level, because the knowledge base is empty; the second case — a dark sky in the dry season — is answered *raincoat* by the first node, which is wrong, so it becomes an exception. Finally `kb.infer(...)` asks about a morning with a dark sky in the dry season; `what()` gives the answer and `how()` the explanation.

Because this program uses the engine, the compiler must be told where the engine's source files are. From the `Chapter01/java` folder:

```
javac -d out -sourcepath ../../vcirs-core/java/src *.java
java -cp out FirstTaste
```

`-d out` puts all compiled classes in a folder called `out`; `-sourcepath` says where to find the `vcirs` classes that the program imports, and `javac` compiles exactly the ones it needs; `-cp out` (the *class path*) tells `java` where the compiled classes are. The output is:

```
Case 1: added: raincoat#1 added at the top level
Case 2: refined: mask#1 added as an exception of raincoat#1
Conclusion: bring = mask  (rule mask#1, CF 100)
  on the way: bring = raincoat
mask#1 fired because:
  sky = dark  (given, CF 100)
  season = dry  (given, CF 100)
  so bring = mask  [rule CF 100, conclusion CF 100]
```

Read it from the top. Case 1 created node `raincoat#1` at the top level. Case 2 created `mask#1` as an exception of `raincoat#1` — the knowledge base noticed on its own that its answer was wrong and put the new knowledge exactly where the wrong answer came from. The answer for the new morning is *bring = mask*, found through the rule `mask#1`; "on the way" it passed the conclusion of the more general node, *raincoat*, which the exception overrode. The explanation lists the two facts that made the rule fire and the certainty of the conclusion: 100, the highest, because we gave no uncertain facts. **Figure 1.2** is a picture of exactly this knowledge base.

Listing 1.6 is the Python version. Its first lines tell Python where the engine is, two folders up and then into `vcirs-core/python`; after that, it reads like the Java version.

Listing 1.6 `first_taste.py`: the same first taste in Python.

```
import pathlib
import sys

# Tell Python where the vcirs-core engine is (two folders up, then across).
sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import Case, KnowledgeBase

kb = KnowledgeBase("morning")
# Case 1: a dark sky means rain is coming.
r1 = kb.learn(Case.of(["sky = dark"], ["bring = raincoat"]))
print("Case 1: " + str(r1))
# Case 2: but in the dry season a dark sky is only smoke or haze.
r2 = kb.learn(Case.of(["sky = dark", "season = dry"], ["bring = mask"]))
print("Case 2: " + str(r2))
# Ask the knowledge base about a new morning.
answer = kb.infer({"sky": "dark", "season": "dry"})
for line in answer.what():
    print(line)
for line in answer.how():
    print(line)
```

Run it from the `Chapter01/python` folder with `python first_taste.py`. The output is the same as the Java output, character for character:

```
Case 1: added: raincoat#1 added at the top level
Case 2: refined: mask#1 added as an exception of raincoat#1
Conclusion: bring = mask  (rule mask#1, CF 100)
  on the way: bring = raincoat
mask#1 fired because:
  sky = dark  (given, CF 100)
  season = dry  (given, CF 100)
  so bring = mask  [rule CF 100, conclusion CF 100]
```

> **KEY INSIGHT.** In **Listing 1.5** nobody wrote an `if` statement about masks or haze. The knowledge came from a case, and the engine decided where it belongs. That is the difference between programming rules and teaching them.

1.11 Using It for Real

The morning program is small, but the habit it teaches is the one that matters: write your know-how as rules, and give every rule a reason. Here are three ways to put it to work today.

Your own morning. Replace Pak Darto's facts and rules with yours. A siomay seller might write: *if it is Friday, prepare more portions, because the mosque crowd buys after prayers; if the gas cylinder feels light, bring the spare; if it rained last night, bring the tarpaulin.* A parking attendant might write rules about which rows fill first on a market day. Start with three rules, run them for

a week, and add a fourth whenever reality surprises you. You will notice that the surprises tend to be *exceptions* to an existing rule, not new rules — which is exactly why Ripple Down Rules are built around exceptions.

A shop's restocking rules. A small shop can write *if fewer than five bottles of cooking oil are left and tomorrow is market day, order two cartons.* The facts come from counting the shelf; the conclusion is an order. When the rules live in a program, the owner's nephew who minds the shop on Sundays applies the same knowledge as the owner.

A health self-check. VCIRS has already been used to build a self-diagnosis assistant from a home medical guide, which asks about symptoms and suggests first steps [17]. Chapter 5 builds such an advisor with a rule that matters more than any other: when a symptom is serious, the only advice is to see a doctor or a midwife (*bidan*) at once.

A family orchard. On a fruit and durian orchard, rules such as *if the durian leaves turn yellow from the tip and the soil has been wet for a week, check the roots for rot* are exactly the kind of knowledge that Chapter 6 turns into an advisor, with rules that the system itself proposes for the grower to confirm.

In every case, begin with the rules you are sure of, keep the reasons, and let the cases you meet tell you where the rules are incomplete. The applications in the following chapters do precisely this, each time with a little more help from the engine.

1.12 Common Mistakes, and How to Avoid Them

Using an old terminal after installing. A terminal opened before the installation does not see the new PATH. *Avoid it:* close all terminals and open a new one after every installation.

Installing a JRE instead of a JDK. A Java *runtime* can run programs but has no `javac`. *Avoid it:* download the **JDK** of Temurin 25 and check with `javac -version`.

Forgetting "Add python.exe to PATH" on Windows. Python is installed but no terminal can find it. *Avoid it:* run the installer again, choose *Modify*, and tick the option; or use the `py` command.

Typing `python` on macOS or Linux. The name may be missing or point to an old Python. *Avoid it:* type `python3` (or `python3.14`).

Naming the file differently from the class. Java requires `public class Hello` to be in `Hello.java`, with the same capital letters. *Avoid it:* rename the file or the class so that they match.

Running `java Hello.java` for a program that needs other classes. The one-step way compiles a single file only, so `FirstTaste` fails because it cannot find `vcirs`. *Avoid it:* compile with `javac -d out -sourcepath ../../vcirs-core/java/src *.java` and run with `java -cp out FirstTaste`.

Moving a chapter folder away from `vcirs-core`. The programs look for the engine two folders up. *Avoid it:* keep the `Code_new` folder in one piece.

Comparing text with `==` in Java. It compares objects, not words. *Avoid it:* use `.equals`.

Mixing tabs and spaces in Python. Python decides which lines belong to an `if` by their indentation, and a mixture confuses it. *Avoid it:* let VS Code indent for you; it uses four spaces.

Writing a rule without a reason. A rule without a *because* cannot be checked by anyone else, and cannot explain itself later. *Avoid it:* for every rule, write down the reason at the same time as the rule.

1.13 Chapter Summary

- Much everyday expertise is already a set of **rules**: IF conditions on **facts** hold, THEN a **conclusion** follows. A condition is a **variable**, an **operator** (`= <> < <= > >=`) and a **value**.
- A **rule-based system** (expert system) has a **knowledge base** of rules, an **inference engine** that applies them, and a **working memory** for the facts of the current case. It is *intelligent* when it explains **what** it concluded, **how** and **why**, when its users can improve its knowledge case by case, and when it learns which knowledge is used.
- Classic **rule-based systems** infer powerfully, by **forward chaining** (from facts to conclusions) and **backward chaining** (from a goal to the facts it needs), but are hard to maintain. **Ripple Down Rules** let an expert add knowledge case by case as **exceptions**, checked against **cornerstone cases**, but infer simply (**Figure 1.1**).
- **VCIRS** joins the two in the **Variable-Centered Rule Structure** of **variables**, **nodes** and **rules** (**Figure 1.2**). It builds knowledge with the **list of difference** and six options, counts usage with **Credit**, **VUR**, **NUR** and **RUR**, **generates** candidate rules, and infers the RDR way or by forward and backward chaining after **knowledge base transformation**.
- Every application of this book uses the shared engine **vcirs-core**, written in Java and in Python with identical behavior; **Table 1.1** shows which idea each chapter adds.
- The workbench is **JDK 25** (Temurin), **Python 3.14** and **VS Code** with two extensions (**Table 1.2**, **Table 1.3**); **Table 1.5** lists the usual problems. Java is compiled with `javac` and run with `java`; Python is run with `python` (**Figure 1.3**).
- The first programs checked the workbench (**Listing 1.1**, **Listing 1.2**), wrote three rules with reasons (**Listing 1.3**, **Listing 1.4**), and taught a knowledge base a rule and an exception (**Listing 1.5**, **Listing 1.6**).

1.14 Exercises

1.1 Write down five rules that you or someone in your family uses every day, each in the form *IF … AND … THEN …, because …*. For each rule, name its variables, operators, values and conclusion.

1.2 For each problem, say whether it suits a rule system and why: (a) choosing a fertilizer for a durian tree whose leaves are yellowing; (b) recognizing a friend's voice on the phone; (c) deciding whether a laundry order can be ready tomorrow; (d) guessing tomorrow's rupiah exchange rate; (e) deciding which of three buses reaches the terminal first, given their routes.

1.3 Pak Darto's rules are applied from what he sees to what he does. Rewrite Rule 1 of **Listing 1.3** as a question for **backward chaining**: what goal would the engine start from, which rule would it try, and which facts would it ask about?

1.4 Add a fourth rule to **Listing 1.3** and **Listing 1.4**: *if the fuel gauge shows fewer than two bars, fill up at the first petrol station, because a passenger may want a long trip.* Keep the facts at the top and the reason on its own line. Run both programs with the fuel at one bar.

1.5 Change the facts of **Listing 1.4** so that no rule fires, and check that the program prints the good-ride message. Which facts did you change, and to what?

1.6 Using **Figure 1.2**, explain in your own words why the rule `mask#1` contains two nodes while the rule `raincoat#1` contains one. Which conclusion does each rule give?

1.7 Extend **Listing 1.6** (or **Listing 1.5**) with a third case: in the dry season a dark sky *and* the smell of rain means rain after all (`sky = dark`, `season = dry`, `smell = rain`, conclusion `bring =`

`raincoat`). Where does the engine put the new node, and what does it answer for a dark, dry morning that smells of rain?

1.8 Your friend installed Python on Windows, but `python --version` opens the Microsoft Store and `py --version` prints `Python 3.12.4`. Explain what is happening and what she should do. Is 3.12 good enough for this book?

1.15 Solutions to the Exercises

Solution 1.1. Answers vary. A good rule names checkable facts and a clear conclusion, for example: *IF the rice jar holds less than two cups AND tomorrow is Sunday THEN buy five kilograms today, because the shop is closed on Sunday.* Its variables are `rice left` and `day`; its operators are `<` and `=`; its values are `2 cups` and `Sunday`; its conclusion is `buy = 5 kg`. A rule such as "IF I feel like it THEN cook" is not usable, because nobody else can check its condition.

Solution 1.2. (a) Yes: an experienced grower can explain the checks — soil moisture, the pattern of yellowing, recent fertilizer — as IF-THEN sentences; Chapter 6 builds exactly this. (b) No: recognizing a voice is not knowledge that people can put into sentences; it is learned from examples and belongs to machine learning. (c) Yes: it depends on the weight, the service, the machines free and the time, and a shop owner can state the rules. (d) No: exchange rates depend on too many unknown factors, and no set of rules is reliable; no rule system should pretend otherwise. (e) Partly: the answer comes from calculation on timetables, which an ordinary program does better than rules, although rules can express exceptions such as "on market day route B is slow".

Solution 1.3. The goal is `bring = raincoat`. The engine looks for rules whose conclusion is `bring = raincoat` and finds Rule 1. To prove it, it must establish its conditions in order: first `sky = dark` — no rule concludes `sky`, so it asks the user "what does the sky look like?" — and then `season = rainy`, which it asks too (or computes from the month, if a rule for it exists). If both hold, the goal is proved; if the user says the sky is clear, Rule 1 fails and, with no other rule for the goal, the answer is "not proved". Chapter 8 builds this kind of reasoning.

Solution 1.4. Add a fact and a rule. The complete programs are `MorningRulesFuel.java` and `morning_rules_fuel.py` in the `Chapter01` folders; the Java version is:

```
public class MorningRulesFuel {
    public static void main(String[] args) {
        // The facts of this morning. Change them and run the program again.
        String sky = "dark";          // "clear", "cloudy" or "dark"
        boolean rainySeason = true;   // November to March in most of Java
        int batteryPercent = 25;      // the phone's battery
        int tyrePressure = 26;        // rear tyre, in psi
        int fuelBars = 1;             // bars on the fuel gauge

        int fired = 0;
        // Rule 1
        if (sky.equals("dark") && rainySeason) {
            System.out.println("Rule 1: bring the raincoat.");
            System.out.println("  because the sky is dark and it is the "
                    + "rainy season");
            fired++;
        }
        // Rule 2
        if (batteryPercent < 30) {
            System.out.println("Rule 2: charge the phone before leaving.");
            System.out.println("  because the battery is at " + batteryPercent
                    + "%, below 30%");
            fired++;
        }
        // Rule 3
        if (tyrePressure < 29) {
            System.out.println("Rule 3: pump the rear tyre at the first workshop.");
            System.out.println("  because the pressure is " + tyrePressure
                    + " psi, below 29 psi");
            fired++;
        }
        // Rule 4
        if (fuelBars < 2) {
            System.out.println("Rule 4: fill up at the first petrol station.");
            System.out.println("  because the gauge shows " + fuelBars
                    + " bar, and a passenger may want a long trip");
            fired++;
        }
        if (fired == 0) {
            System.out.println("No rule fired: have a good ride!");
        }
    }
}
```

With `fuelBars = 1` it prints all four rules; the last two lines are:

```
Rule 4: fill up at the first petrol station.
  because the gauge shows 1 bar, and a passenger may want a long trip
```

The Python version adds `fuel_bars = 1` to the facts and the same rule with `if fuel_bars < 2:`; its output is identical.

Solution 1.5. For example `sky = "clear"`, `battery_percent = 80` and `tyre_pressure = 30`: Rule 1 needs a dark sky, Rule 2 a battery below 30 and Rule 3 a pressure below 29, so none fires and the program prints `No rule fired: have a good ride!`. Changing only the season to `False` is not enough, because Rules 2 and 3 would still fire.

Solution 1.6. A rule is the sequence of nodes from the top level down to its last node. `raincoat#1` is a top-level node, so its rule is just that node and concludes `bring = raincoat`. `mask#1` was created as an exception of `raincoat#1`, so its rule is `raincoat#1 -> mask#1`: both nodes must hold — a dark sky and a dry season — and the conclusion is that of the last node, `bring = mask`.

Solution 1.7. The solution programs are `SecondTaste.java` and `second_taste.py`. The third case is answered *mask* by `mask#1`, which is wrong, so `learn` adds `raincoat#2` as an exception of `mask#1` with the one condition that tells it apart, `smell = rain`. The output of the Python version is:

```
Case 1: added: raincoat#1 added at the top level
Case 2: refined: mask#1 added as an exception of raincoat#1
Case 3: refined: raincoat#2 added as an exception of mask#1
Conclusion: bring = raincoat  (rule raincoat#2, CF 100)
  on the way: bring = raincoat, bring = mask
raincoat#2 fired because:
  sky = dark  (given, CF 100)
  season = dry  (given, CF 100)
  smell = rain  (given, CF 100)
  so bring = raincoat  [rule CF 100, conclusion CF 100]
```

The rule `raincoat#2` is now the sequence `raincoat#1 -> mask#1 -> raincoat#2`: dark sky, dry season, smell of rain; the more general conclusions *raincoat* and *mask* appear "on the way".

Solution 1.8. The `python` command is the Microsoft Store placeholder (an app execution alias), while `py` finds a real Python, version 3.12. She can either use `py` everywhere, or switch off the aliases (Section 1.7.1) and make sure Python's folder is on the PATH. Python 3.12 is newer than 3.10, so every program in this book runs on it; installing 3.14 is still recommended, and `py install 3.14` (or the python.org installer) does it without removing 3.12.

References

[1] G. R. Baur and D. V. Pigford, *Expert Systems for Business: Concepts and Applications*. Boston, MA, USA: Boyd & Fraser, 1990.

[2] B. G. Buchanan and E. H. Shortliffe, Eds., *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Reading, MA, USA: Addison-Wesley, 1984.

[3] I. Subakti, "Some revisions in VCIRS and cases reconstructing perspectives," in *Proc. Information and Communication Technology Seminar (ICTS 2006)*, Surabaya, Indonesia, 2006.

[4] P. Compton and B. H. Kang, *Ripple-Down Rules: The Alternative to Machine Learning*. Boca Raton, FL, USA: CRC Press, 2021.

[5] M. M. I. Subakti, "A Variable-Centered Intelligent Rule System," M.Sc.Eng. thesis, Dept. of Computer Science and Information Engineering, Taipei, Taiwan, Jun. 2005.

[6] I. Subakti, "A variable-centered intelligent rule system," in *Proc. 1st Annual Int. Conf. Information and Communication Technology Seminar (ICTS 2005)*, vol. 1, no. 1, Surabaya, Indonesia, Aug. 2005, pp. 167–174.

[7] P. Compton, G. Edwards, B. Kang, L. Lazarus, R. Malor, T. Menzies, P. Preston, A. Srinivasan, and C. Sammut, "Ripple down rules: possibilities and limitations," in *Proc. 6th AAAI Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada, 1991, pp. 6.1–6.20.

[8] B. H. Kang, P. Compton, and P. Preston, "Multiple classification ripple down rules: evaluation and possibilities," in *Proc. 9th AAAI-Sponsored Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada, 1995, pp. 17.1–17.20.

[9] P. H. Winston, *Artificial Intelligence*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1992.

[10] Adoptium, "Eclipse Temurin 8u502, 11.0.32, 17.0.20, 21.0.12, 25.0.4 and 26.0.2 available," Aug. 2026. [Online]. Available: https://adoptium.net/news/2026/08/eclipse-temurin-8u502-11032-17020-21012-2504-2602-available

[11] Adoptium, "Eclipse Temurin 8u504, 11.0.32.1, 17.0.20.1, 21.0.12.1, 25.0.4.1 and 26.0.2.1 available," Sep. 2, 2026. [Online]. Available: https://adoptium.net/news/2026/09/eclipse-temurin-8u504-110321-170201-210121-25041-26021-available

[12] Oracle, “The arrival of Java 27!,” *Inside.java*, Sep. 15, 2026. [Online]. Available: https://inside.java/2026/09/15/jdk-27-available/

[13] Python Software Foundation, “Python 3.14.7 and 3.13.15 are now available,” *Python Insider*, Aug. 2026. [Online]. Available: https://blog.python.org/2026/08/python-3147-31315/

[14] Microsoft, “Visual Studio Code 1.139,” release notes, Sep. 23, 2026. [Online]. Available: https://code.visualstudio.com/updates/v1_139

[15] Microsoft, “Extension Pack for Java,” Visual Studio Marketplace. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=vscjava.vscode-java-pack

[16] Microsoft, “Python,” Visual Studio Marketplace. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=ms-python.python

[17] I. Subakti and R. Hidayatullah, “Aplikasi sistem pakar untuk diagnosis awal gangguan kesehatan secara mandiri menggunakan Variable-Centered Intelligent Rule System,” vol. 6, no. 1, pp. 11–16, Jan. 2007.

Chapter 2

Electricity-Bill Saver: Your First Rule Program

By the end of this chapter you will be able to work out a household's monthly electricity bill by hand and by program, from the power of each appliance, its hours of use and the PLN tariff; write saving advice as IF-THEN rules that give a reason and a saving for every piece of advice; keep the rules apart from the program, as values in a list that a small loop applies to every appliance; do money arithmetic with whole numbers so that Java and Python agree to the last rupiah; and adapt the app to your own home, a rented room or a small shop. The app we build, the **electricity-bill saver**, reads a list of appliances, prints the bill appliance by appliance, fires thirteen pieces of advice for Bu Sri's house, and shows that following them would take her bill from Rp 407,600 to Rp 317,603 a month.

2.1 Bu Sri's Bill

Bu Sri lives with her husband and two teenage children in a small house with a 1300 VA electricity connection. Last month the bill was a little over four hundred thousand rupiah, and the family had agreed to keep it under four hundred. The neighbors have plenty of advice: switch off the air conditioner, buy LED lamps, do not leave the rice cooker on "warm" all day, unplug the television. All of it is probably true. What nobody can tell her is *how much* each piece of advice is worth in her house, which one to do first, and whether doing them all would be enough.

That is a job for rules. An electrician or an energy adviser who visits a house thinks exactly like this: *if there is an air conditioner set to 20 degrees, suggest 25; if the old kitchen bulb burns fourteen hours a day, suggest an LED and switching it off in daylight; if the rice cooker keeps rice warm all day, suggest cooking closer to mealtime.* Each rule has the shape that rule-based systems have always used — *IF* clauses joined by *AND*, *THEN* a conclusion [1] — with a condition that can be checked on one appliance, advice, a reason, and — because electricity is measured — a saving that can be calculated. A program that holds these rules can visit every house in the kampung, and it never forgets a rule or gets the arithmetic wrong.

This chapter builds that program in three steps. First we compute the bill with no rules at all, to be sure the numbers are right. Then we write two pieces of advice as ordinary `if` statements, the way Chapter 1 wrote Pak Darto's morning. Finally we take the rules out of the program's logic and make them **values**: a list of rules, each with its condition, advice, reason and effect, applied by one small loop. That last step — separating the knowledge from the machinery that uses it — is the most important idea of the chapter, and every later chapter of this book depends on it.

Keep in mind what the app is and is not. It is an adviser that does arithmetic on the numbers you give it, and it can only be as good as those numbers. It gives advice, not guarantees, and it never touches wiring: anything to do with the meter, the circuit breaker or the cables is a job for PLN or a certified electrician, and one of our rules says so.

2.2 The Idea in Plain Words

2.2.1 Power, time and energy

Every appliance has a **power**, written on its label in **watts** (W): a small LED lamp uses 9 W, a fan 45 W, a rice cooker 350 W while it cooks. Power says how fast the appliance uses electricity while it is on. How much it uses in a month depends also on **how long** it is on. Multiply the power by the hours and you get **energy** in **watt-hours** (Wh); a thousand watt-hours are one **kilowatt-hour** (kWh), the unit on the electricity meter and on the bill.

For one kind of appliance, used h hours a day, with n of them, each of power P watts, the energy in a month of 30 days is

$$E = \frac{P \times n \times h \times 30}{1000} \text{ kWh} \tag{2.1}$$

Let us try it by hand on Bu Sri's air conditioner. Its label says 400 W; there is one of it; it runs 9 hours a night. By Equation (2.1), $E = 400 \times 1 \times 9 \times 30/1000 = 108{,}000/1000 = 108$ kWh a month. Her six LED lamps of 9 W each, on 7 hours a day, use $9 \times 6 \times 7 \times 30/1000 = 11.34$ kWh — less than a tenth of one air conditioner.

2.2.2 Standby: the power of "off"

Many appliances never switch off completely while they are plugged in. The television waits for its remote control, the washing machine keeps its little display alive, a phone charger left in the socket stays warm. This **standby** power is small — around 1 W in our examples, though some older appliances use more — but it runs for all the hours the appliance is *not* in use. For an appliance with standby power S watts, the energy of a month is

$$E = \frac{(P \times h + S \times (24 - h)) \times n \times 30}{1000} \text{ kWh} \tag{2.2}$$

The air conditioner has a standby power of 1 W for the 15 hours it is off, which adds $1 \times 15 \times 30/1000 = 0.45$ kWh a month, so its true total is 108.45 kWh.

2.2.3 From energy to rupiah

The price of one kWh depends on the customer's **tariff**, which depends on the connection's power in **volt-amperes** (VA) — 450, 900, 1300, 2200 VA and so on — and on whether the household receives a subsidy. The government reviews the household tariffs every three months. For July to September 2026 it kept them unchanged [2], [3]; **Table 2.1** shows them. Postpaid customers pay these rates on the kWh the meter counts; prepaid customers buy tokens at the same rates.

Table 2.1 Household electricity tariffs of PLN for July–September 2026, left unchanged for that quarter [2], [3].

Code in our file	Tariff	VA	Rp per kWh
`R1-450S`	R-1/TR 450 VA, subsidized	450	415.00
`R1-900S`	R-1/TR 900 VA, subsidized	900	605.00
`R1-900`	R-1/TR 900 VA, non-subsidized	900	1,352.00
`R1-1300`	R-1/TR 1300 VA	1300	1,444.70
`R1-2200`	R-1/TR 2200 VA	2200	1,444.70
`R2-3500`	R-2/TR 3500–5500 VA	3500	1,699.53
`R3-6600`	R-3/TR 6600 VA and above	6600	1,699.53

The bill for energy E at a rate r rupiah per kWh is simply

$$B = E \times r \tag{2.3}$$

rounded to whole rupiah. Bu Sri is on `R1-1300`, so her air conditioner costs $108.45 \times 1444.70 = 156{,}677.7$, about Rp 156,678 a month. A real bill can also carry a street-lighting tax (*Pajak Penerangan Jalan*), set by each regency or city, and a small stamp duty; postpaid bills have a minimum charge for houses that use very little. Section 2.8 shows how to add them. Everything else in this chapter follows from Equation (2.1), Equation (2.2) and Equation (2.3).

2.2.4 Watts, volt-amperes and the circuit breaker

The number on the connection, 1300 VA for Bu Sri, is not an amount of energy; it is a limit on *power*: how much the house may draw at one moment. A **circuit breaker** in the meter box, the MCB, switches the house off when the limit is exceeded, which is what people mean when they say “the electricity went down” after switching on the iron and the pump together. Volt-amperes and watts measure almost the same thing for a lamp or a kettle, and a little less than the same thing for appliances with motors, such as a refrigerator or a pump, whose watts are somewhat below their volt-amperes. For a household warning, treating one watt as one volt-ampere is close enough, and on the safe side for most appliances.

The limit matters to the bill only indirectly: a higher VA tariff can cost more per kWh, and a connection that trips often tempts people to ask for a larger one. Often the cheaper answer is simply not to run the heavy appliances at the same time, which is exactly what one of our rules will say.

2.2.5 Saving rules

A saving rule looks at one appliance and asks a question about it. If the answer is yes, it gives advice, the reason for the advice, and — whenever it can — the appliance *as it would be if the advice were followed*: fewer hours, lower power, no standby. The saving is then just the energy before minus the energy after, both by Equation (2.2). Some advice cannot be computed honestly without measuring the room: setting the air conditioner to 25 °C instead of 20 °C certainly saves energy, and it is the Ministry of Energy’s own advice [4], but how much depends on the room, the insulation and the weather, so our rule gives the advice and says plainly that the saving “depends on the room and the weather”. A rule that invented a number would be worse than no rule.

2.2.6 Why rules, and not just a spreadsheet?

A spreadsheet could compute Equation (2.1) and Equation (2.3) for every appliance, and many families already keep one. What a spreadsheet does not do is *decide*. It will show that the kitchen lamp costs Rp 36,406 a month, but it will not say "this is an old bulb; an LED would save most of that", and it will not say why. The decision is where the expertise lives: knowing which appliances deserve attention, what can realistically be changed, and what the change is worth. Rules capture exactly that part, and they capture it in a form that can be read, argued about and corrected by the person who knows. An energy adviser who disagrees with our 25 °C rule can see it, change the number, and see the effect on every house at once.

There is a second, quieter reason. A spreadsheet's formulas are hidden in its cells, each one a small island. A rule program has one place where all the knowledge sits — here, one list — and one engine that applies it the same way to everyone. When a family asks "why did you tell us to switch off the terrace lamp?", the answer is a rule with a name, a condition and a reason, not a cell reference. That property, *knowledge you can point at*, is what the rest of the book builds on.

> **EXAMPLE.** Bu Sri's kitchen bulb: 60 W, 14 hours a day. By Equation (2.1) it uses $60 \times 14 \times 30/1000 = 25.2$ kWh a month, which at Rp 1,444.70 costs about Rp 36,406. A 9 W LED for the same 14 hours would use $9 \times 14 \times 30/1000 = 3.78$ kWh, costing about Rp 5,461. The saving is $25.2 - 3.78 = 21.42$ kWh, or about Rp 30,945 a month. If the lamp is also switched off in daylight, so that it burns 12 hours instead of 14, the LED uses $9 \times 12 \times 30/1000 = 3.24$ kWh, a further 0.54 kWh. These are exactly the numbers the app will print, and working them out once by hand is the best way to trust it.

2.3 Step 1: The Bill Without Rules

Before we write a single rule we make sure the arithmetic is right. **Listing 2.1** computes the energy and the cost of three of Bu Sri's appliances with Equation (2.1) and Equation (2.3).

Listing 2.1 `BillCalculator.java`: the bill of three appliances, with no rules yet.

```
import java.util.Locale;

public class BillCalculator {
    public static void main(String[] args) {
        double rate = 1444.70;                  // Rp per kWh, R-1/TR 1300 VA
        String[] names = {"Air conditioner", "Refrigerator", "LED lamps"};
        double[] watts = {400, 90, 9};          // power of one appliance
        int[] counts = {1, 1, 6};               // how many of them
        double[] hours = {9, 10, 7};            // hours of use per day

        double totalKwh = 0;
        for (int i = 0; i < names.length; i++) {
            // watts x hours = watt-hours a day; x 30 days; / 1000 = kWh
            double kwh = watts[i] * counts[i] * hours[i] * 30 / 1000;
            totalKwh += kwh;
            System.out.printf(Locale.US, "%-16s %6.1f kWh  Rp %,9.0f%n",
                    names[i], kwh, kwh * rate);
        }
        System.out.printf(Locale.US, "%-16s %6.1f kWh  Rp %,9.0f%n",
                "Total", totalKwh, totalKwh * rate);
    }
}
```

The appliances are described by four **arrays**, one per property, where position `i` of every array belongs to appliance `i`. The loop computes each appliance's kWh with Equation (2.1), adds it to the total, and prints a line. `printf` prints with a **format**: `%-16s` is text padded to 16 characters on

the left, `%6.1f` a number in 6 characters with one decimal, `%,9.0f` a number in 9 characters with no decimals and a comma between thousands, and `%n` a line break [5]. `Locale.US` asks for a comma as the thousands separator whatever the language of your computer; without it, a computer set to Indonesian would print `156.028`, the Indonesian way. Compile and run with `javac BillCalculator.java` and `java BillCalculator`:

```
Air conditioner   108.0 kWh  Rp   156,028
Refrigerator       27.0 kWh  Rp    39,007
LED lamps          11.3 kWh  Rp    16,383
Total             146.3 kWh  Rp   211,417
```

Check the first line against our hand computation: 108.0 kWh for the air conditioner (without its standby, which this simple version ignores) and $108 \times 1444.70 = 156{,}027.6$, printed as Rp 156,028. The LED line shows 11.3 kWh, our 11.34 rounded to one decimal.

Listing 2.2 is the same computation in Python. Its **f-strings** (`f"..."`) do what `printf` does: `{names[i]:<16}` pads to 16 characters on the left, `{kwh:6.1f}` prints one decimal in 6 characters, and `{kwh * rate:9,.0f}` prints no decimals with thousands separated by commas [6].

Listing 2.2 `bill_calculator.py`: the same bill in Python.

```
rate = 1444.70                          # Rp per kWh, R-1/TR 1300 VA
names = ["Air conditioner", "Refrigerator", "LED lamps"]
watts = [400, 90, 9]                    # power of one appliance
counts = [1, 1, 6]                      # how many of them
hours = [9, 10, 7]                      # hours of use per day

total_kwh = 0
for i in range(len(names)):
    # watts x hours = watt-hours a day; x 30 days; / 1000 = kWh
    kwh = watts[i] * counts[i] * hours[i] * 30 / 1000
    total_kwh += kwh
    print(f"{names[i]:<16} {kwh:6.1f} kWh  Rp {kwh * rate:9,.0f}")
print(f"{'Total':<16} {total_kwh:6.1f} kWh  Rp {total_kwh * rate:9,.0f}")
```

The output is identical:

```
Air conditioner   108.0 kWh  Rp   156,028
Refrigerator       27.0 kWh  Rp    39,007
LED lamps          11.3 kWh  Rp    16,383
Total             146.3 kWh  Rp   211,417
```

2.4 Step 2: The First Rules, as `if` Statements

Now let us add advice, the simplest way, for one appliance. **Listing 2.3** describes the air conditioner by its facts and writes two rules as `if` statements, exactly as Chapter 1 wrote Pak Darto's morning.

Listing 2.3 `FirstAdvice.java`: two saving rules written directly as `if` statements.

```java
public class FirstAdvice {
    public static void main(String[] args) {
        // One appliance, described by its facts.
        String name = "Bedroom air conditioner";
        String kind = "ac";
        int watts = 400;
        double hours = 9;
        int setting = 20;                       // degrees Celsius

        // Rule 1: IF it is an air conditioner AND it is set below 25 C
        //         THEN advise 25 C.
        if (kind.equals("ac") && setting < 25) {
            System.out.println(name + ": set it to 25 C instead of "
                    + setting + " C.");
        }
        // Rule 2: IF it is an air conditioner AND it runs more than 8 hours
        //         THEN advise the timer, and say how much that saves.
        if (kind.equals("ac") && hours > 8) {
            double savedKwh = watts * (hours - 8) * 30 / 1000;
            System.out.println(name + ": switch it off with the timer after"
                    + " 8 hours; that saves " + savedKwh + " kWh a month.");
        }
    }
}
```

Rule 1 has two conditions joined by *and*: the appliance is an air conditioner, and it is set below 25 °C. Rule 2 also computes a saving: the hours beyond eight, times the power, times 30 days, divided by 1000 — Equation (2.1) applied to the hours saved. The output:

```
Bedroom air conditioner: set it to 25 C instead of 20 C.
Bedroom air conditioner: switch it off with the timer after 8 hours; that saves 12.0 kWh a
month.
```

Listing 2.4 is the Python version, with the same output.

Listing 2.4 `first_advice.py`: the same two rules in Python.

```python
# One appliance, described by its facts.
name = "Bedroom air conditioner"
kind = "ac"
watts = 400
hours = 9
setting = 20                            # degrees Celsius

# Rule 1: IF it is an air conditioner AND it is set below 25 C
#         THEN advise 25 C.
if kind == "ac" and setting < 25:
    print(name + ": set it to 25 C instead of " + str(setting) + " C.")
# Rule 2: IF it is an air conditioner AND it runs more than 8 hours
#         THEN advise the timer, and say how much that saves.
if kind == "ac" and hours > 8:
    saved_kwh = watts * (hours - 8) * 30 / 1000
    print(name + ": switch it off with the timer after 8 hours; "
          + "that saves " + str(saved_kwh) + " kWh a month.")
```

```
Bedroom air conditioner: set it to 25 C instead of 20 C.
Bedroom air conditioner: switch it off with the timer after 8 hours; that saves 12.0 kWh a
month.
```

This works, and for two rules and one appliance it is the right amount of program. But imagine the full adviser written this way. Bu Sri has fourteen appliances, and a good adviser knows at least a dozen rules. Every rule would have to be repeated for every appliance, or wrapped in a loop with a growing tangle of `if` statements inside it. To add a rule you would edit the middle of the program; to see all the rules at once you would read the whole program; and to compute a *total*

saving you would have to make sure that two rules about the same lamp — "replace it with an LED" and "switch it off in daylight" — do not count the same kilowatt-hours twice. The knowledge and the machinery are mixed together.

> **KEY INSIGHT.** A rule program becomes easy to grow when the rules are *data*: a list of values, each holding its condition, its advice, its reason and its effect. The program that applies them — the inference engine — is then a small loop that never changes when a rule is added.

2.5 Step 3: Rules as Values

2.5.1 The design

Figure 2.1 shows the design of the finished app. Two files come in: a **household file** that lists the appliances and how they are used, and a **tariff file** with the price per kWh of each tariff. The program computes the energy of each appliance and the bill. Then it runs every **saving rule** on every appliance, and three **house rules** on the house as a whole, and prints each piece of advice with its reason and its saving.

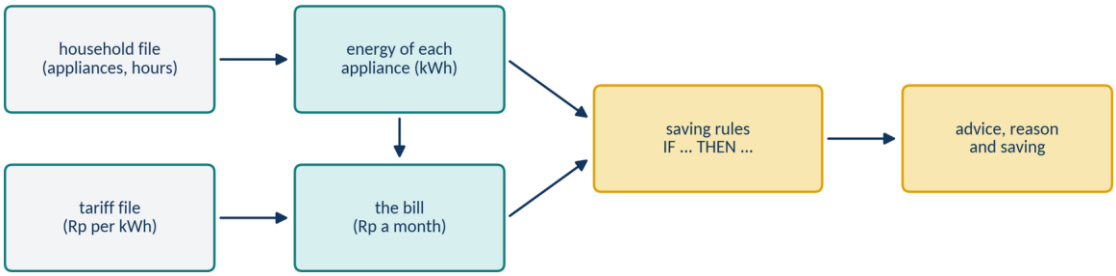

Figure 2.1 The electricity-bill saver. The appliances and the tariff give the energy and the bill; the saving rules turn them into advice, each with a reason and a saving.

2.5.2 The household file

The household file is a plain text file in the **CSV** format (comma-separated values), which any spreadsheet can save and any text editor can change. Bu Sri's file is:

```
# Bu Sri's house: a family of four, 1300 VA, postpaid
house,Bu Sri's house
tariff,R1-1300
budget,400000
name,kind,watts,count,hours,standby,setting
Bedroom air conditioner,ac,400,1,9,1,20
Refrigerator,fridge,90,1,10,0,
LED lamps,lamp-led,9,6,7,0,
Kitchen lamp (old bulb),lamp-bulb,60,1,14,0,
Terrace lamp (old bulb),lamp-bulb,40,1,12,0,
Television,tv,60,1,7,1,
Rice cooker (cooking),rice-cook,350,1,1,0,
Rice cooker (keep warm),rice-warm,45,1,14,0,
Water pump,pump,250,1,1.5,0,
Standing fan,fan,45,2,6,0,
Phone chargers,charger,10,3,3,0.5,
Iron,iron,350,1,0.5,0,
Washing machine,washer,300,1,1,1,
Wi-Fi router,router,10,1,24,0,
```

Lines that begin with `#` are comments. The first three lines give the name of the house, the tariff code of **Table 2.1** and a monthly budget in rupiah. Then comes the table: the header line `name,kind,watts,count,hours,standby,setting`, and one line per appliance. The **kind** tells the rules what the appliance is — `ac`, `fridge`, `lamp-led`, `lamp-bulb` (an old incandescent bulb), `rice-`

`cook`, `rice-warm` (the same rice cooker keeping rice warm, entered as its own line because its power and hours are different) and so on. `hours` is hours of use a day; `standby` is the standby power in watts; `setting` is the temperature of the air conditioner, and empty for everything else.

Where do the numbers come from? The watts are on the label or in the manual of each appliance. The hours are the family's own estimate; for the refrigerator, whose motor switches on and off by itself, we wrote 10 hours as an estimate of how long the motor actually runs, and Section 2.8 explains how to measure it. The tariff file, `tariff-2026-q3.csv`, holds **Table 2.1** in the same format.

2.5.3 Whole numbers only

Our first program used decimal numbers (`double` in Java, `float` in Python), and for a quick estimate that is fine. For the real app we want something stronger: the same answer, to the last rupiah, in Java and in Python, on every computer, every time. Decimal numbers in a computer are stored in binary, and a number such as 1444.70 cannot be stored exactly; the tiny error is harmless, but when a result falls exactly halfway between two rupiah the two languages may round it differently. So the app does all its arithmetic with **whole numbers**, in units small enough to need no decimals:

- power in **tenths of a watt**: 400 W is `4000`, 0.5 W is `5`;
- time in **minutes a day**: 1.5 hours is `90`;
- energy in **deci-watt-minutes**: tenths of a watt times minutes. One kWh is 1000 W × 60 min = 60,000 watt-minutes, which is 600,000 deci-watt-minutes;
- the rate in **sen**, hundredths of a rupiah: Rp 1,444.70 is `144470`.

The price of an energy e (in deci-watt-minutes) at a rate s (in sen per kWh) is then, in whole sen,

$$\text{sen} = \left\lfloor \frac{e \times s + 300{,}000}{600{,}000} \right\rfloor \tag{2.4}$$

where $\lfloor x \rfloor$ means "x rounded down to a whole number". Adding half of the divisor before dividing makes the division round to the *nearest* whole number, with halves going up. The same trick turns sen into rupiah (add 50, divide by 100) and energy into tenths of a kWh for printing (add 30,000, divide by 60,000). Whole-number division is exact in both languages, so the results are identical. Check it on the air conditioner: $e = 4000 \times 540 \times 30 + 10 \times 900 \times 30 = 65{,}070{,}000$, and by Equation (2.4) the cost is $(65{,}070{,}000 \times 144{,}470 + 300{,}000)/600{,}000$, rounded down: 15,667,771 sen, which is Rp 156,678 — the same as our hand computation with Equation (2.3).

2.5.4 An appliance

Listing 2.5 is the class that holds one line of the household file. Its method `energy()` is Equation (2.2) in these units, and three small methods make *changed copies* of the appliance, which is how a rule describes the effect of its advice: `withMinutes` for "use it less", `withDeciWatts` for "replace it with something that uses less power", and `withoutStandby` for "unplug it".

Listing 2.5 `Appliance.java`: one appliance, its monthly energy, and copies that describe advice.

```
/**
 * One line of the household file: an appliance and how it is used.
 * Power is kept in tenths of a watt and time in minutes, so that every
 * calculation is done with whole numbers and gives the same result in Java
 * and in Python.
 */
public class Appliance {
    public static final int DAYS = 30;          // a month of 30 days

    public final String name;
    public final String kind;         // ac, fridge, lamp-led, lamp-bulb, ...
    public final long deciWatts;      // power when on, in 0.1 W
    public final int count;           // how many of them
    public final long minutes;        // minutes of use per day
    public final long standbyDeciWatts;   // power when "off" but plugged in
    public final String setting;      // e.g. the AC temperature, or ""

    public Appliance(String name, String kind, long deciWatts, int count,
                     long minutes, long standbyDeciWatts, String setting) {
        this.name = name;
        this.kind = kind;
        this.deciWatts = deciWatts;
        this.count = count;
        this.minutes = minutes;
        this.standbyDeciWatts = standbyDeciWatts;
        this.setting = setting;
    }

    /** Energy in a month, in deci-watt-minutes (600,000 of them = 1 kWh). */
    public long energy() {
        long on = deciWatts * count * minutes * DAYS;
        long standby = standbyDeciWatts * count * (24 * 60 - minutes) * DAYS;
        return on + standby;
    }

    /** A copy used for more minutes or fewer (advice "use it less"). */
    public Appliance withMinutes(long newMinutes) {
        return new Appliance(name, kind, deciWatts, count, newMinutes,
                standbyDeciWatts, setting);
    }
```

Listing 2.5 (Continued) *Appliance.java: one appliance, its monthly energy, and copies that describe advice.*

```
    /** A copy with another power (advice "replace it"). */
    public Appliance withDeciWatts(long newDeciWatts) {
        return new Appliance(name, kind, newDeciWatts, count, minutes,
                standbyDeciWatts, setting);
    }

    /** A copy that uses nothing on standby (advice "unplug it"). */
    public Appliance withoutStandby() {
        return new Appliance(name, kind, deciWatts, count, minutes, 0, setting);
    }

    /** The setting as a number, or -1 when there is none. */
    public int settingNumber() {
        return setting.isEmpty() ? -1 : Integer.parseInt(setting);
    }
}
```

The fields are declared `final`: once an appliance is made, it never changes. A rule never *modifies* Bu Sri's air conditioner; it makes a new `Appliance` that describes the air conditioner as it would be with the timer set to eight hours, and the saving is the difference of the two energies. Keeping

the original untouched is what lets several rules look at the same appliance without disturbing each other.

2.5.5 The household and the bill

Listing 2.6 reads the household file and the tariff file, and holds the bill arithmetic of Section 2.5.3.

Listing 2.6 `Household.java`: reading the two files, and the whole-number bill arithmetic.

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;
import java.util.Locale;

/** A household file, its tariff, and the arithmetic of the bill. */
public class Household {
    public String name = "";
    public String tariffCode = "";
    public String tariffName = "";
    public long va;                  // the contracted power, e.g. 1300 VA
    public long rateSen;             // Rp per kWh, in sen (1/100 rupiah)
    public long budget;              // rupiah a month, 0 = no budget
    public final List<Appliance> appliances = new ArrayList<>();

    /** "1.5" with places 2 gives 150; "400" gives 40000. */
    public static long scaled(String text, int places) {
        String t = text.strip();
        String whole = t;
        String part = "";
        int dot = t.indexOf('.');
        if (dot >= 0) {
            whole = t.substring(0, dot);
            part = t.substring(dot + 1);
        }
        if (part.length() > places) {
            throw new IllegalArgumentException("too many decimals: " + text);
        }
        while (part.length() < places) {
            part = part + "0";
        }
        return Long.parseLong(whole + part);
    }
```

Listing 2.6 (Continued) *Household.java: reading the two files, and the whole-number bill arithmetic.*

```
    public static Household load(String path, String tariffPath)
            throws IOException {
        Household h = new Household();
        boolean inTable = false;
        for (String line : Files.readAllLines(Path.of(path),
                StandardCharsets.UTF_8)) {
            if (line.isBlank() || line.startsWith("#")) {
                continue;
            }
            String[] f = line.split(",", -1);
            if (f[0].equals("name")) {
                inTable = true;               // the header of the table
            } else if (inTable) {
                long hundredths = scaled(f[4], 2);         // hours x 100
                if (hundredths * 60 % 100 != 0) {
                    throw new IllegalArgumentException("hours must be whole "
                            + "minutes: " + f[4]);
                }
                h.appliances.add(new Appliance(f[0], f[1], scaled(f[2], 1),
                        Integer.parseInt(f[3]), hundredths * 60 / 100,
                        scaled(f[5], 1), f[6]));
            } else if (f[0].equals("house")) {
                h.name = f[1];
            } else if (f[0].equals("tariff")) {
                h.tariffCode = f[1];
            } else if (f[0].equals("budget")) {
                h.budget = Long.parseLong(f[1]);
            }
        }
        for (String line : Files.readAllLines(Path.of(tariffPath),
                StandardCharsets.UTF_8)) {
            String[] f = line.split(",", -1);
            if (f[0].equals(h.tariffCode)) {
                h.va = Long.parseLong(f[1]);
                h.rateSen = scaled(f[2], 2);
                h.tariffName = f[3];
            }
        }
        if (h.rateSen == 0) {
            throw new IllegalArgumentException("unknown tariff " + h.tariffCode);
        }
        return h;
    }
```

Listing 2.6 (Continued) *Household.java: reading the two files, and the whole-number bill arithmetic.*

```
    /** The price of an amount of energy, in whole rupiah (rounded). */
    public long rupiah(long energy) {
        long sen = (energy * rateSen + 300_000) / 600_000;
        return (sen + 50) / 100;
    }

    public long totalEnergy() {
        long sum = 0;
        for (Appliance a : appliances) {
            sum += a.energy();
        }
        return sum;
    }

    /** Energy as kWh with one decimal, e.g. "108.0". */
    public static String kwh(long energy) {
        long tenths = (energy + 30_000) / 60_000;
        return (tenths / 10) + "." + (tenths % 10);
    }

    /** Rupiah with thousands separators, e.g. "Rp 156,028". */
    public static String rp(long rupiah) {
        return "Rp " + String.format(Locale.US, "%,d", rupiah);
    }

    public String rate() {
        return "Rp " + String.format(Locale.US, "%,d", rateSen / 100) + "."
                + String.format("%02d", rateSen % 100);
    }
}
```

`scaled` turns a decimal written in the file into a whole number of the right unit without ever passing through a `double`: `"1.5"` with two decimal places becomes `150`, which is hours times 100; multiplying by 60 and dividing by 100 gives 90 minutes. If the hours are not a whole number of minutes (such as 0.33), the program refuses the line with a clear message rather than rounding silently. `load` reads the household file line by line: before the header it looks for the `house`, `tariff` and `budget` lines; after it, every line is an appliance. `split(",", -1)` splits a line at commas and keeps empty fields at the end, such as the empty `setting` of a lamp. Then the tariff file is searched for the household's tariff code. `rupiah` is Equation (2.4) followed by the rounding from sen to rupiah, and `kwh`, `rp` and `rate` format numbers the same way on every computer.

In Python the appliance and the household live in one module, **Listing 2.7**.

Listing 2.7 `household.py`: the appliance, the household file and the bill arithmetic in Python.

```
"""A household file, its appliances, its tariff and the bill arithmetic.

Power is kept in tenths of a watt and time in minutes, so that every
calculation is done with whole numbers and gives the same result in Python
and in Java."""

DAYS = 30                                  # a month of 30 days


class Appliance:
    def __init__(self, name, kind, deci_watts, count, minutes,
                 standby_deci_watts, setting):
        self.name = name
        self.kind = kind                   # ac, fridge, lamp-led, lamp-bulb, ...
        self.deci_watts = deci_watts       # power when on, in 0.1 W
        self.count = count                 # how many of them
        self.minutes = minutes             # minutes of use per day
        self.standby_deci_watts = standby_deci_watts  # off but plugged in
        self.setting = setting             # e.g. the AC temperature, or ""

    def energy(self):
        """Energy in a month, in deci-watt-minutes (600,000 = 1 kWh)."""
        on = self.deci_watts * self.count * self.minutes * DAYS
        standby = (self.standby_deci_watts * self.count
                   * (24 * 60 - self.minutes) * DAYS)
        return on + standby

    def with_minutes(self, new_minutes):
        return Appliance(self.name, self.kind, self.deci_watts, self.count,
                         new_minutes, self.standby_deci_watts, self.setting)

    def with_deci_watts(self, new_deci_watts):
        return Appliance(self.name, self.kind, new_deci_watts, self.count,
                         self.minutes, self.standby_deci_watts, self.setting)

    def without_standby(self):
        return Appliance(self.name, self.kind, self.deci_watts, self.count,
                         self.minutes, 0, self.setting)

    def setting_number(self):
        """The setting as a number, or -1 when there is none."""
        return int(self.setting) if self.setting else -1
```

Listing 2.7 (Continued) *household.py: the appliance, the household file and the bill arithmetic in Python.*

```
def scaled(text, places):
    """'1.5' with places 2 gives 150; '400' with places 1 gives 4000."""
    text = text.strip()
    whole, _, part = text.partition(".")
    if len(part) > places:
        raise ValueError("too many decimals: " + text)
    return int(whole + part.ljust(places, "0"))


class Household:
    def __init__(self):
        self.name = ""
        self.tariff_code = ""
        self.tariff_name = ""
        self.va = 0                     # the contracted power, e.g. 1300 VA
        self.rate_sen = 0               # Rp per kWh, in sen (1/100 rupiah)
        self.budget = 0                 # rupiah a month, 0 = no budget
        self.appliances = []

    @staticmethod
    def load(path, tariff_path):
        h = Household()
        in_table = False
        with open(path, encoding="utf-8") as f:
            for line in f.read().splitlines():
                if not line.strip() or line.startswith("#"):
                    continue
                f = line.split(",")
                if f[0] == "name":
                    in_table = True      # the header of the table
                elif in_table:
                    hundredths = scaled(f[4], 2)           # hours x 100
                    if hundredths * 60 % 100 != 0:
                        raise ValueError("hours must be whole minutes: " + f[4])
                    h.appliances.append(Appliance(
                        f[0], f[1], scaled(f[2], 1), int(f[3]),
                        hundredths * 60 // 100, scaled(f[5], 1), f[6]))
                elif f[0] == "house":
                    h.name = f[1]
                elif f[0] == "tariff":
                    h.tariff_code = f[1]
                elif f[0] == "budget":
                    h.budget = int(f[1])
        with open(tariff_path, encoding="utf-8") as f:
            for line in f.read().splitlines():
                f = line.split(",")
```

Listing 2.7 (Continued) *household.py: the appliance, the household file and the bill arithmetic in Python.*

```
                if f[0] == h.tariff_code:
                    h.va = int(f[1])
                    h.rate_sen = scaled(f[2], 2)
                    h.tariff_name = f[3]
        if h.rate_sen == 0:
            raise ValueError("unknown tariff " + h.tariff_code)
        return h

    def rupiah(self, energy):
        """The price of an amount of energy, in whole rupiah (rounded)."""
        sen = (energy * self.rate_sen + 300_000) // 600_000
        return (sen + 50) // 100

    def total_energy(self):
        return sum(a.energy() for a in self.appliances)

    def rate(self):
        return f"Rp {self.rate_sen // 100:,d}.{self.rate_sen % 100:02d}"


def kwh(energy):
    """Energy as kWh with one decimal, e.g. '108.5'."""
    tenths = (energy + 30_000) // 60_000
    return str(tenths // 10) + "." + str(tenths % 10)


def rp(rupiah):
    """Rupiah with thousands separators, e.g. 'Rp 156,678'."""
    return f"Rp {rupiah:,d}"
```

Two differences are worth noticing. Python's whole numbers have no size limit and `//` is whole-number division, so the formulas look exactly like Equation (2.4). And `text.partition(".")` splits `"1.5"` into `"1"`, `"."` and `"5"` in one step, where Java needed `indexOf` and `substring`.

2.5.6 A rule is a value

Now the heart of the chapter. **Listing 2.8** defines what a saving rule *is*: an identifier, a condition, a way to produce the advice, a way to produce the reason, and a way to produce the changed appliance.

Listing 2.8 `SavingRule.java`: a saving rule as a value with five parts.

```
import java.util.function.Function;
import java.util.function.Predicate;

/**
 * An IF-THEN rule about one appliance. "when" is the IF part; "advice"
 * and "why" are what the rule says; "change" is how the appliance would be
 * used if the advice were followed (null when we cannot compute it).
 */
public class SavingRule {
    public final String id;
    public final Predicate<Appliance> when;
    public final Function<Appliance, String> advice;
    public final Function<Appliance, String> why;
    public final Function<Appliance, Appliance> change;

    public SavingRule(String id, Predicate<Appliance> when,
                      Function<Appliance, String> advice,
                      Function<Appliance, String> why,
                      Function<Appliance, Appliance> change) {
        this.id = id;
        this.when = when;
        this.advice = advice;
        this.why = why;
        this.change = change;
    }
}
```

The parts are **functions stored in fields**. `Predicate<Appliance>` is a function that takes an appliance and answers true or false: the IF part. `Function<Appliance, String>` takes an appliance and returns text: the advice and the reason. `Function<Appliance, Appliance>` takes an appliance and returns the changed copy: the effect of the advice. In Java such a function can be written as a **lambda expression**, `a -> a.kind.equals("ac")`, which you can read aloud as "given an appliance `a`, answer whether its kind is `ac`".

Listing 2.9 is the whole app. The method `rules()` builds the list of six appliance rules; `main` reads the files, prints the bill, runs the rules, runs the three house rules, and prints the total.

Listing 2.9 `BillSaver.java`: the electricity-bill saver, with its rules as values and a small engine that applies them.

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

/**
 * Electricity-bill saver: reads a household file, works out the monthly
 * bill, runs every saving rule on every appliance, then three rules about
 * the whole house, and says what each piece of advice would save.
 * Run: java BillSaver ../data/home-sri.csv ../data/tariff-2026-q3.csv
 */
public class BillSaver {
    static final int AC_BEST = 25;            // degrees C (ESDM's "3M")
    static final long AC_HOURS = 8 * 60;      // minutes a day
    static final long LIGHT_HOURS = 12 * 60;
    static final long WARM_HOURS = 4 * 60;
    static final int LED_PERCENT = 15;        // a 9 W LED ~ a 60 W bulb
    static final long HEAVY = 2500;           // 250 W, in tenths of a watt

    static List<SavingRule> rules() {
        List<SavingRule> r = new ArrayList<>();
        r.add(new SavingRule("ac-setpoint",
                a -> a.kind.equals("ac") && a.settingNumber() >= 0
                        && a.settingNumber() < AC_BEST,
                a -> "set it to " + AC_BEST + " C instead of " + a.setting + " C",
                a -> "it is an air conditioner set below " + AC_BEST + " C",
                null));
        r.add(new SavingRule("ac-timer",
                a -> a.kind.equals("ac") && a.minutes > AC_HOURS,
                a -> "let the timer switch it off after 8 hours",
                a -> "it is an air conditioner used " + a.minutes / 60.0
                        + " hours a day",
                a -> a.withMinutes(AC_HOURS)));
        r.add(new SavingRule("old-bulb",
                a -> a.kind.equals("lamp-bulb"),
                a -> "replace it with an LED lamp of about "
                        + ledDeciWatts(a) / 10 + " W",
                a -> "it is an old bulb of " + a.deciWatts / 10 + " W",
                a -> a.withDeciWatts(ledDeciWatts(a))));
        r.add(new SavingRule("lights-daytime",
                a -> a.kind.startsWith("lamp") && a.minutes > LIGHT_HOURS,
                a -> "switch it off in daylight; 12 hours a day is enough",
                a -> "it is a lamp on for " + a.minutes / 60.0 + " hours a day",
                a -> a.withMinutes(LIGHT_HOURS)));
        r.add(new SavingRule("keep-warm",
                a -> a.kind.equals("rice-warm") && a.minutes > WARM_HOURS,
                a -> "cook closer to mealtime; keep rice warm 4 hours at most",
```

Listing 2.9 (Continued) *BillSaver.java: the electricity-bill saver, with its rules as values and a small engine that applies them.*

```
                a -> "the rice cooker keeps warm " + a.minutes / 60.0
                        + " hours a day",
                a -> a.withMinutes(WARM_HOURS)));
        r.add(new SavingRule("standby",
                a -> a.standbyDeciWatts > 0 && a.minutes < 24 * 60,
                a -> "switch it off at the socket when you are not using it",
                a -> "it uses " + a.standbyDeciWatts / 10.0
                        + " W even when it is off",
                a -> a.withoutStandby()));
        return r;
    }

    static long ledDeciWatts(Appliance a) {
        return (a.deciWatts * LED_PERCENT + 50) / 100;
    }

    public static void main(String[] args) throws IOException {
        String file = args.length > 0 ? args[0] : "../data/home-sri.csv";
        String tariff = args.length > 1 ? args[1] : "../data/tariff-2026-q3.csv";
        Household h = Household.load(file, tariff);
        long total = h.totalEnergy();
        long bill = h.rupiah(total);

        System.out.println("Electricity-bill saver: " + h.name);
        System.out.println("Tariff: " + h.tariffName);
        System.out.println("Price:  " + h.rate()
                + " per kWh (July-September 2026)");
        System.out.println();
        System.out.printf("%-26s %9s %10s %6s%n", "Appliance", "kWh/month",
                "Rp/month", "Share");
        System.out.println("-".repeat(54));
        for (Appliance a : h.appliances) {
            long e = a.energy();
            long share = (e * 100 + total / 2) / total;
            System.out.printf("%-26s %9s %10s %5d%%%n", a.name, Household.kwh(e),
                    Household.rp(h.rupiah(e)).substring(3), share);
        }
        System.out.println("-".repeat(54));
        System.out.printf("%-26s %9s %10s%n", "Total", Household.kwh(total),
                Household.rp(bill).substring(3));
        System.out.println();
```

Listing 2.9 (Continued) *BillSaver.java: the electricity-bill saver, with its rules as values and a small engine that applies them.*

```
        // Run every appliance rule on every appliance.
        List<SavingRule> rules = rules();
        List<Appliance> after = new ArrayList<>();
        int number = 0;
        System.out.println("Advice (" + rules.size() + " appliance rules and "
                + "3 house rules checked):");
        for (Appliance a : h.appliances) {
            Appliance changed = a;
            for (SavingRule rule : rules) {
                if (!rule.when.test(a)) {
                    continue;                // the IF part is false
                }
                number++;
                System.out.println(String.format("%2d. [%s] %s", number,
                        rule.id, a.name));
                System.out.println("    advice: " + rule.advice.apply(a));
                System.out.println("    why: " + rule.why.apply(a));
                if (rule.change == null) {
                    System.out.println("    saving: depends on the room and "
                            + "the weather");
                } else {
                    long saved = a.energy() - rule.change.apply(a).energy();
                    System.out.println("    saving: " + Household.kwh(saved)
                            + " kWh = " + Household.rp(h.rupiah(saved))
                            + " a month");
                    changed = rule.change.apply(changed);
                }
            }
            after.add(changed);
        }

        // House rule 1: the biggest eater.
        for (Appliance a : h.appliances) {
            long share = (a.energy() * 100 + total / 2) / total;
            if (share >= 30) {
                number++;
                System.out.println(String.format("%2d. [biggest-eater] %s",
                        number, a.name));
                System.out.println("    advice: look at it first; it uses "
                        + share + "% of your electricity");
            }
        }
        // House rule 2: everything switched on at once.
        long peak = 0;
        for (Appliance a : h.appliances) {
            peak += a.deciWatts * a.count / 10;
```

Listing 2.9 (Continued) *BillSaver.java: the electricity-bill saver, with its rules as values and a small engine that applies them.*

```
        }
        if (peak > h.va) {
            number++;
            System.out.println(String.format("%2d. [overload] the whole house",
                    number));
            System.out.println("    advice: do not run these heavy appliances "
                    + "at the same time:");
            for (Appliance a : h.appliances) {
                if (a.deciWatts >= HEAVY) {
                    System.out.println("      - " + a.name);
                }
            }
            System.out.println("    why: everything on at once needs about "
                    + peak + " W, more than your " + h.va + " VA");
            System.out.println("    safety: never change the MCB yourself; "
                    + "ask PLN if you need more power");
        }
        // House rule 3: the budget.
        long newTotal = 0;
        for (Appliance a : after) {
            newTotal += a.energy();
        }
        long newBill = h.rupiah(newTotal);
        if (h.budget > 0 && bill > h.budget) {
            number++;
            System.out.println(String.format("%2d. [budget] the whole house",
                    number));
            System.out.println("    why: the bill is " + Household.rp(bill
                    - h.budget) + " over your budget of " + Household.rp(h.budget));
            System.out.println("    with the advice above it becomes "
                    + Household.rp(newBill) + (newBill <= h.budget
                    ? ", within the budget" : ", still over the budget"));
        }
        System.out.println();
        System.out.println("If you follow the advice: " + Household.kwh(
                total - newTotal) + " kWh and " + Household.rp(bill - newBill)
                + " less each month,");
        System.out.println("that is " + Household.rp(newBill) + " instead of "
                + Household.rp(bill) + ".");
    }
}
```

Read the rules first, because they are the knowledge. `ac-setpoint` fires for an air conditioner whose setting is below 25; it gives advice and a reason but no change, because its saving cannot be computed honestly. `ac-timer` fires for an air conditioner used more than eight hours and changes the minutes to 480. `old-bulb` fires for every old bulb and changes its power to the LED power: 15 percent of the bulb's, rounded, which turns 60 W into 9 W — the usual equivalence printed on LED boxes. `lights-daytime` fires for any lamp on more than twelve hours. `keep-warm` fires for a rice cooker keeping warm more than four hours. `standby` fires for anything with standby power that is not in use all day. The constants at the top — `AC_BEST`, `AC_HOURS`, `LED_PERCENT` and the others — are the numbers the rules depend on, gathered in one place where an adviser can change them.

Then read the engine: the loop *for each appliance, for each rule*. When a rule's condition is false, `continue` moves on to the next rule. When it is true, the loop prints the rule's identifier, the appliance, the advice and the reason; if the rule has a change, it prints the saving — the energy of the appliance minus the energy of the changed copy — and applies the change to a running copy called `changed`. That running copy is the answer to the double-counting worry of Section 2.4:

when two rules fire on the kitchen lamp, the first changes it to a 9 W LED, and the second shortens the *LED's* hours, so the total saving counts each kilowatt-hour once. After the loop, the list `after` holds every appliance as it would be if all the advice were followed.

The three **house rules** look at the house as a whole. `biggest-eater` points at any appliance that uses 30 percent or more of the electricity. `overload` adds up the power of everything, as if all were switched on at once, and compares it with the connection's VA; watts and volt-amperes are not quite the same thing, but for household appliances they are close enough for a warning. When the sum is larger, it lists the heavy appliances that should not run together — and adds a safety line, because a circuit breaker (MCB) that trips is doing its job, and replacing it with a bigger one is dangerous and against PLN's rules. `budget` compares the bill with the family's budget and says whether the advice would bring it within.

Listing 2.10 is the Python version. The rules are the same five-part values, built with Python's `lambda` and a small `SavingRule` class, and the engine is the same double loop.

Listing 2.10 `bill_saver.py`: the electricity-bill saver in Python.

```
"""Electricity-bill saver: reads a household file, works out the monthly
bill, runs every saving rule on every appliance, then three rules about the
whole house, and says what each piece of advice would save.
Run: python bill_saver.py ../data/home-sri.csv ../data/tariff-2026-q3.csv
"""

import sys

from household import Household, kwh, rp

AC_BEST = 25                            # degrees C (ESDM's "3M")
AC_HOURS = 8 * 60                       # minutes a day
LIGHT_HOURS = 12 * 60
WARM_HOURS = 4 * 60
LED_PERCENT = 15                        # a 9 W LED ~ a 60 W bulb
HEAVY = 2500                            # 250 W, in tenths of a watt


class SavingRule:
    """An IF-THEN rule about one appliance. `when` is the IF part; `advice`
    and `why` are what the rule says; `change` is how the appliance would be
    used if the advice were followed (None when we cannot compute it)."""

    def __init__(self, rule_id, when, advice, why, change):
        self.id = rule_id
        self.when = when
        self.advice = advice
        self.why = why
        self.change = change


def led_deci_watts(a):
    return (a.deci_watts * LED_PERCENT + 50) // 100


def rules():
    return [
        SavingRule(
            "ac-setpoint",
            lambda a: a.kind == "ac" and 0 <= a.setting_number() < AC_BEST,
            lambda a: f"set it to {AC_BEST} C instead of {a.setting} C",
            lambda a: f"it is an air conditioner set below {AC_BEST} C",
            None),
        SavingRule(
            "ac-timer",
            lambda a: a.kind == "ac" and a.minutes > AC_HOURS,
```

Listing 2.10 (Continued) *bill_saver.py: the electricity-bill saver in Python.*

```
            lambda a: "let the timer switch it off after 8 hours",
            lambda a: "it is an air conditioner used "
                      + str(a.minutes / 60) + " hours a day",
            lambda a: a.with_minutes(AC_HOURS)),
        SavingRule(
            "old-bulb",
            lambda a: a.kind == "lamp-bulb",
            lambda a: "replace it with an LED lamp of about "
                      + str(led_deci_watts(a) // 10) + " W",
            lambda a: "it is an old bulb of " + str(a.deci_watts // 10) + " W",
            lambda a: a.with_deci_watts(led_deci_watts(a))),
        SavingRule(
            "lights-daytime",
            lambda a: a.kind.startswith("lamp") and a.minutes > LIGHT_HOURS,
            lambda a: "switch it off in daylight; 12 hours a day is enough",
            lambda a: "it is a lamp on for " + str(a.minutes / 60)
                      + " hours a day",
            lambda a: a.with_minutes(LIGHT_HOURS)),
        SavingRule(
            "keep-warm",
            lambda a: a.kind == "rice-warm" and a.minutes > WARM_HOURS,
            lambda a: "cook closer to mealtime; keep rice warm 4 hours at most",
            lambda a: "the rice cooker keeps warm " + str(a.minutes / 60)
                      + " hours a day",
            lambda a: a.with_minutes(WARM_HOURS)),
        SavingRule(
            "standby",
            lambda a: a.standby_deci_watts > 0 and a.minutes < 24 * 60,
            lambda a: "switch it off at the socket when you are not using it",
            lambda a: "it uses " + str(a.standby_deci_watts / 10)
                      + " W even when it is off",
            lambda a: a.without_standby()),
    ]


def main(args):
    path = args[0] if args else "../data/home-sri.csv"
    tariff = args[1] if len(args) > 1 else "../data/tariff-2026-q3.csv"
    h = Household.load(path, tariff)
    total = h.total_energy()
    bill = h.rupiah(total)
```

Listing 2.10 (Continued) *bill_saver.py: the electricity-bill saver in Python.*

```
    print("Electricity-bill saver: " + h.name)
    print("Tariff: " + h.tariff_name)
    print("Price:  " + h.rate() + " per kWh (July-September 2026)")
    print()
    print(f"{'Appliance':<26} {'kWh/month':>9} {'Rp/month':>10} {'Share':>6}")
    print("-" * 54)
    for a in h.appliances:
        e = a.energy()
        share = (e * 100 + total // 2) // total
        print(f"{a.name:<26} {kwh(e):>9} {rp(h.rupiah(e))[3:]:>10} "
              f"{share:>5}%")
    print("-" * 54)
    print(f"{'Total':<26} {kwh(total):>9} {rp(bill)[3:]:>10}")
    print()

    # Run every appliance rule on every appliance.
    all_rules = rules()
    after = []
    number = 0
    print("Advice (" + str(len(all_rules)) + " appliance rules and "
          + "3 house rules checked):")
    for a in h.appliances:
        changed = a
        for rule in all_rules:
            if not rule.when(a):
                continue                # the IF part is false
            number += 1
            print(f"{number:2d}. [{rule.id}] {a.name}")
            print("    advice: " + rule.advice(a))
            print("    why: " + rule.why(a))
            if rule.change is None:
                print("    saving: depends on the room and the weather")
            else:
                saved = a.energy() - rule.change(a).energy()
                print("    saving: " + kwh(saved) + " kWh = "
                      + rp(h.rupiah(saved)) + " a month")
                changed = rule.change(changed)
        after.append(changed)
```

Listing 2.10 (Continued) *bill_saver.py: the electricity-bill saver in Python.*

```
    # House rule 1: the biggest eater.
    for a in h.appliances:
        share = (a.energy() * 100 + total // 2) // total
        if share >= 30:
            number += 1
            print(f"{number:2d}. [biggest-eater] {a.name}")
            print("    advice: look at it first; it uses " + str(share)
                  + "% of your electricity")
    # House rule 2: everything switched on at once.
    peak = sum(a.deci_watts * a.count // 10 for a in h.appliances)
    if peak > h.va:
        number += 1
        print(f"{number:2d}. [overload] the whole house")
        print("    advice: do not run these heavy appliances at the same time:")
        for a in h.appliances:
            if a.deci_watts >= HEAVY:
                print("      - " + a.name)
        print("    why: everything on at once needs about " + str(peak)
              + " W, more than your " + str(h.va) + " VA")
        print("    safety: never change the MCB yourself; ask PLN if you "
              "need more power")
    # House rule 3: the budget.
    new_total = sum(a.energy() for a in after)
    new_bill = h.rupiah(new_total)
    if h.budget > 0 and bill > h.budget:
        number += 1
        print(f"{number:2d}. [budget] the whole house")
        print("    why: the bill is " + rp(bill - h.budget)
              + " over your budget of " + rp(h.budget))
        print("    with the advice above it becomes " + rp(new_bill)
              + (", within the budget" if new_bill <= h.budget
                 else ", still over the budget"))
    print()
    print("If you follow the advice: " + kwh(total - new_total) + " kWh and "
          + rp(bill - new_bill) + " less each month,")
    print("that is " + rp(new_bill) + " instead of " + rp(bill) + ".")


if __name__ == "__main__":
    main(sys.argv[1:])
```

2.6 Running the Saver

From the `Chapter02/java` folder, compile everything and run the saver on Bu Sri's house:

```
javac -d out *.java
java -cp out BillSaver ../data/home-sri.csv ../data/tariff-2026-q3.csv
```

In Python, from `Chapter02/python`: `python bill_saver.py ../data/home-sri.csv ../data/tariff-2026-q3.csv`. Both print exactly the same report:

```
Electricity-bill saver: Bu Sri's house
Tariff: R-1/TR 1300 VA
Price:  Rp 1,444.70 per kWh (July-September 2026)

Appliance                  kWh/month   Rp/month  Share
-----------------------------------------------------
Bedroom air conditioner        108.5    156,678    38%
Refrigerator                    27.0     39,007    10%
LED lamps                       11.3     16,383     4%
Kitchen lamp (old bulb)         25.2     36,406     9%
Terrace lamp (old bulb)         14.4     20,804     5%
Television                      13.1     18,940     5%
Rice cooker (cooking)           10.5     15,169     4%
Rice cooker (keep warm)         18.9     27,305     7%
Water pump                      11.3     16,253     4%
Standing fan                    16.2     23,404     6%
Phone chargers                   3.6      5,266     1%
Iron                             5.3      7,585     2%
Washing machine                  9.7     13,999     3%
Wi-Fi router                     7.2     10,402     3%
-----------------------------------------------------
Total                          282.1    407,600

Advice (6 appliance rules and 3 house rules checked):
 1. [ac-setpoint] Bedroom air conditioner
    advice: set it to 25 C instead of 20 C
    why: it is an air conditioner set below 25 C
    saving: depends on the room and the weather
 2. [ac-timer] Bedroom air conditioner
    advice: let the timer switch it off after 8 hours
    why: it is an air conditioner used 9.0 hours a day
    saving: 12.0 kWh = Rp 17,293 a month
 3. [standby] Bedroom air conditioner
    advice: switch it off at the socket when you are not using it
    why: it uses 1.0 W even when it is off
    saving: 0.5 kWh = Rp 650 a month
 4. [old-bulb] Kitchen lamp (old bulb)
    advice: replace it with an LED lamp of about 9 W
    why: it is an old bulb of 60 W
    saving: 21.4 kWh = Rp 30,945 a month
 5. [lights-daytime] Kitchen lamp (old bulb)
    advice: switch it off in daylight; 12 hours a day is enough
    why: it is a lamp on for 14.0 hours a day
    saving: 3.6 kWh = Rp 5,201 a month
 6. [old-bulb] Terrace lamp (old bulb)
    advice: replace it with an LED lamp of about 6 W
    why: it is an old bulb of 40 W
    saving: 12.2 kWh = Rp 17,683 a month
 7. [standby] Television
    advice: switch it off at the socket when you are not using it
    why: it uses 1.0 W even when it is off
    saving: 0.5 kWh = Rp 737 a month
```

```
 8. [keep-warm] Rice cooker (keep warm)
    advice: cook closer to mealtime; keep rice warm 4 hours at most
    why: the rice cooker keeps warm 14.0 hours a day
    saving: 13.5 kWh = Rp 19,503 a month
 9. [standby] Phone chargers
    advice: switch it off at the socket when you are not using it
    why: it uses 0.5 W even when it is off
    saving: 0.9 kWh = Rp 1,365 a month
10. [standby] Washing machine
    advice: switch it off at the socket when you are not using it
    why: it uses 1.0 W even when it is off
    saving: 0.7 kWh = Rp 997 a month
11. [biggest-eater] Bedroom air conditioner
    advice: look at it first; it uses 38% of your electricity
12. [overload] the whole house
    advice: do not run these heavy appliances at the same time:
      - Bedroom air conditioner
      - Rice cooker (cooking)
      - Water pump
      - Iron
      - Washing machine
    why: everything on at once needs about 2129 W, more than your 1300 VA
    safety: never change the MCB yourself; ask PLN if you need more power
13. [budget] the whole house
    why: the bill is Rp 7,600 over your budget of Rp 400,000
    with the advice above it becomes Rp 317,603, within the budget

If you follow the advice: 62.3 kWh and Rp 89,997 less each month,
that is Rp 317,603 instead of Rp 407,600.
```

Read the report the way Bu Sri would. The table says that the air conditioner alone is 38 percent of the bill, Rp 156,678 of Rp 407,600; that the old kitchen bulb costs more than the refrigerator's motor would suggest, Rp 36,406, because it burns fourteen hours a day at 60 W; and that keeping rice warm for fourteen hours costs Rp 27,305 a month, nearly twice as much as cooking it.

Then the advice, thirteen items from nine rules. The largest single saving is replacing the kitchen bulb with a 9 W LED, Rp 30,945 a month for a lamp that costs a small fraction of that. Keeping rice warm for four hours instead of fourteen saves Rp 19,503; the air-conditioner timer saves Rp 17,293. The standby items are small, a few hundred rupiah each; they are worth doing because they cost nothing, but they will not rescue a budget. The `overload` rule warns that everything together would need about 2129 W on a 1300 VA connection, and lists the five heavy appliances not to run at the same time. And the `budget` rule gives the verdict: the bill is Rp 7,600 over budget, and with the advice it would become Rp 317,603, comfortably within it — a saving of Rp 89,997 a month, or more than a million rupiah a year.

Figure 2.2 shows the same result as a picture: each appliance's monthly energy now, and with the advice.

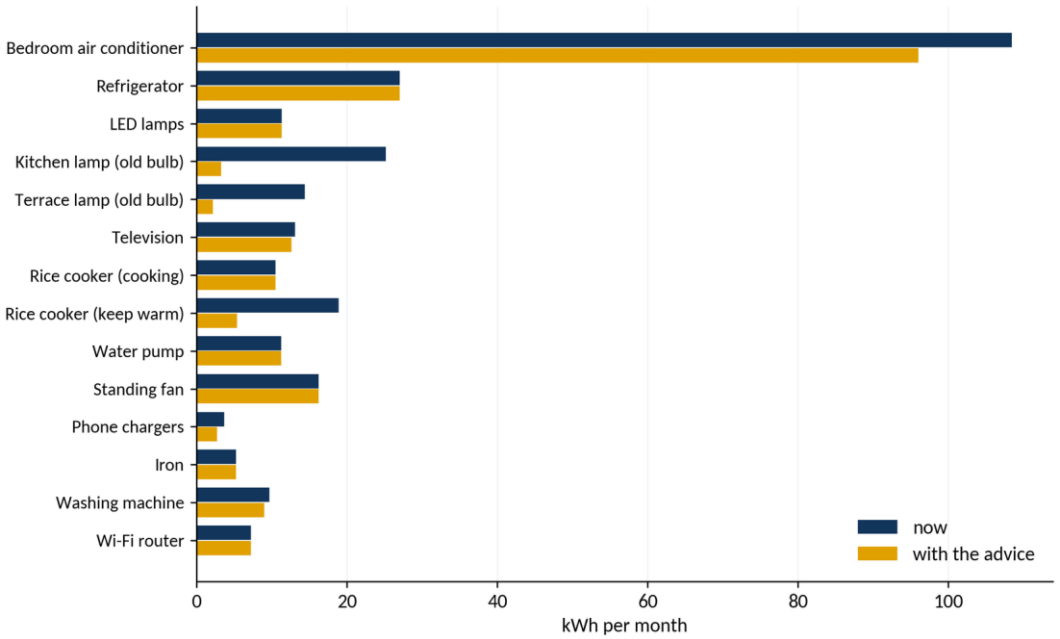

Figure 2.2 Bu Sri's monthly energy per appliance, now and if all the computable advice is followed (our example household).

The same program, with no change at all, advises Dina, a student in a rented room with its own 900 VA prepaid meter (`room-dina.csv`):

```
Electricity-bill saver: Dina's room
Tariff: R-1/TR 900 VA (non-subsidized)
Price:  Rp 1,352.00 per kWh (July-September 2026)

Appliance                  kWh/month   Rp/month  Share
-----------------------------------------------------
Ceiling lamp                     2.2      2,920     3%
Desk lamp (old bulb)             3.8      5,070     6%
Laptop                          12.2     16,548    18%
Phone charger                    1.2      1,643     2%
Small fridge                    14.4     19,469    22%
Rice cooker (cooking)            4.5      6,084     7%
Rice cooker (keep warm)         12.0     16,224    18%
Fan                             12.0     16,224    18%
Electric kettle                  4.5      6,084     7%
-----------------------------------------------------
Total                           66.8     90,266

Advice (6 appliance rules and 3 house rules checked):
 1. [old-bulb] Desk lamp (old bulb)
    advice: replace it with an LED lamp of about 3 W
    why: it is an old bulb of 25 W
    saving: 3.2 kWh = Rp 4,299 a month
 2. [standby] Laptop
    advice: switch it off at the socket when you are not using it
    why: it uses 1.0 W even when it is off
    saving: 0.5 kWh = Rp 730 a month
 3. [standby] Phone charger
    advice: switch it off at the socket when you are not using it
    why: it uses 0.5 W even when it is off
    saving: 0.3 kWh = Rp 426 a month
 4. [keep-warm] Rice cooker (keep warm)
    advice: cook closer to mealtime; keep rice warm 4 hours at most
    why: the rice cooker keeps warm 10.0 hours a day
    saving: 7.2 kWh = Rp 9,734 a month
 5. [overload] the whole house
    advice: do not run these heavy appliances at the same time:
      - Rice cooker (cooking)
      - Electric kettle
    why: everything on at once needs about 1152 W, more than your 900 VA
    safety: never change the MCB yourself; ask PLN if you need more power

If you follow the advice: 11.2 kWh and Rp 15,189 less each month,
that is Rp 75,077 instead of Rp 90,266.
```

Only four appliance rules fire, and the biggest advice is again the rice cooker's keep-warm. The overload warning appears because a 600 W kettle and a 300 W rice cooker together already fill a 900 VA connection; Dina has no budget problem, so the budget rule stays silent. Notice that the rules did not change between the two houses. Only the facts did. That is what it means for rules to be separate from the program.

2.7 Making It Better

2.7.1 All the rules on one page

Because the rules are values, we can describe them as a table, which is how an adviser would review them. **Table 2.2** lists the nine rules of the app.

Table 2.2 The nine rules of the electricity-bill saver: six about one appliance and three about the house.

Rule	IF	THEN (advice)	Effect used for the saving
`ac-setpoint`	kind is `ac` and setting below 25 °C	set it to 25 °C	none (depends on the room)
`ac-timer`	kind is `ac` and used more than 8 h	timer off after 8 h	hours become 8
`old-bulb`	kind is `lamp-bulb`	replace with an LED	power becomes 15 %
`lights-daytime`	any lamp on more than 12 h	switch off in daylight	hours become 12
`keep-warm`	kind is `rice-warm`, more than 4 h	cook closer to mealtime	hours become 4
`standby`	standby power and not used all day	switch off at the socket	standby becomes 0
`biggest-eater`	an appliance uses 30 % or more	look at it first	none
`overload`	all power together exceeds the VA	stagger the heavy appliances	none (safety)
`budget`	bill above the budget	the new bill with the advice	the sum of all effects

2.7.2 Adding a rule

Suppose an adviser wants a rule for water heaters: *if an electric water heater is on more than two hours a day, advise switching it on only half an hour before bathing.* In our design this is one new value in the list returned by `rules()`, with the kind `heater`, and not a single line of the engine changes. The same is true of removing a rule, or of changing a number such as the 25 °C. Exercise 2.5 asks you to add this rule; its solution shows the six lines it takes.

2.7.3 Testing the rules

A rule program that people rely on needs **tests**: small programs that check the arithmetic and the rules automatically, so that a change that breaks something is noticed at once. The chapter's folder has `BillSaverTest.java` and `test_bill_saver.py`. They check that `"1.5"` becomes 150, that the air conditioner uses exactly 65,070,000 deci-watt-minutes, that Bu Sri's bill is Rp 407,600, and that the rules fire in exactly the order of the report. Run them with `java -cp out BillSaverTest` and `python -m unittest test_bill_saver -v`; the Java one ends with:

```
ok    LED for 60 W
all tests passed
```

2.7.4 One appliance through the engine

It helps to follow one appliance through the double loop by hand, the way the computer does it. Take the kitchen lamp, the fourth line of the file: kind `lamp-bulb`, 600 tenths of a watt, one lamp, 840 minutes a day, no standby. At the start of its turn, `changed` is the lamp itself.

1. `ac-setpoint`: is the kind `ac`? No. The condition is false; `continue`.
2. `ac-timer`: kind `ac`? No; `continue`.
3. `old-bulb`: kind `lamp-bulb`? Yes. The rule fires: item 4 of the report. The LED power is $(600 \mathit{times} 15 + 50)/100 = 90$ tenths of a watt, 9 W. The saving printed is the lamp's energy minus that of a 9 W copy: $600 \mathit{times} 840 \mathit{times} 30 - 90 \mathit{times} 840 \mathit{times} 30 = 15{,}120{,}000 - 2{,}268{,}000 = 12{,}852{,}000$ deci-watt-minutes, which is 21.42 kWh, printed 21.4. Now `changed` becomes the 9 W copy.

4. `lights-daytime`: does the kind start with `lamp`, and is it on more than 720 minutes? Yes, 840. The rule fires: item 5. The saving *printed* is computed on the original lamp, $600 times 120 times 30 = 2{,}160{,}000$, or 3.6 kWh — what this advice alone would save. But `changed` now becomes the 9 W copy with 720 minutes, so in the *total*, this rule counts only the 0.54 kWh that the LED would still use in those two hours.
5. `keep-warm`: kind `rice-warm`? No.
6. `standby`: standby power above zero? No.

The lamp's entry in `after` is the 9 W LED on for 12 hours. The two printed savings, 21.4 and 3.6 kWh, tell Bu Sri what each piece of advice is worth *if she does only that one*; the total at the end tells her what they are worth *together*. Both are honest answers to different questions, and the report is careful to give both.

2.7.5 From a list of rules to a knowledge base

Look at **Table 2.2** once more with the vocabulary of Chapter 1. Every rule is a sequence of **conditions** on **variables** of an appliance — `kind`, `minutes`, `setting`, `standby` — with an **operator** and a **value**, and a **conclusion**, the advice. In the language of VCIRS, each row is close to a **node**. What our list does not have is the structure between the rules. The two lamp rules both fire on the kitchen lamp, and nothing records that the second only matters after the first; if an adviser wanted to say "except when the lamp is a security light, never advise switching it off at night", she would have to rewrite the rule, not add an exception. And the rules cannot be taught: every change is a change to the program. The next chapter turns exactly this around. It keeps the separation of knowledge and engine that we built here, and adds a way for the person who knows to correct the knowledge one case at a time.

2.7.6 What the rules cannot see

It is worth being honest about the limits. The rules only know what the household file tells them. They do not know that the terrace lamp is on all night for safety, or that the refrigerator is fifteen years old and its door seal is torn; a person reading the advice must still judge it. The rules do not rank advice by effort, though a family would rather replace a bulb than rearrange its cooking. And they cannot learn: if an adviser finds a new situation, someone must write a new value in the program. Chapter 3 removes that last limit with a knowledge base that learns from each case; Chapter 7 shows how counting which rules are actually used can rank them.

2.7.7 Checking the saver against the meter

A saver that nobody checks is only a guess with decimals. Fortunately the house has an honest witness: the meter. For a postpaid customer the bill itself shows the kWh used in the month, the difference between two meter readings; for a prepaid customer, the kWh bought with each token is printed on the receipt or shown in the payment app, and the meter's display counts down as they are used. Either way, the family can put the saver's total next to a number that the saver did not produce.

Suppose Bu Sri's bill for last month shows 305 kWh, while the saver says 282.1 kWh. The difference is $305 - 282.1 = 22.9$ kWh, about 7.5 percent of the bill. That is a good match for a file built from labels and guesses: a month is not exactly thirty days, the refrigerator works harder in hot weather, and nobody counts the evenings when the television stayed on longer. If the difference were 30 or 40 percent, the file would be missing something, and there is an order in which to look for it.

1. **Forgotten appliances.** A water dispenser that heats and cools all day, an aquarium pump, a second refrigerator in the shop at the front, an electric motorbike charged at night. Walk through the house once more and add every plug that is used.
2. **The biggest items first.** An error of one hour a day on the air conditioner, 400 W, is $400 \times 1 \times 30/1000 = 12$ kWh a month; the same error on a 9 W lamp is 0.27 kWh. Check the hours of the top three lines of the report before anything else.
3. **The refrigerator.** Its hours are the hardest to guess, because its motor switches itself on and off. Measure it with a plug-in energy meter for a full day, as Section 2.8 describes, instead of guessing.
4. **Labels that are ranges or maxima.** Some labels give the power at full load, and some appliances, such as a fan on its lowest speed, use less. A saver built from maxima usually comes out slightly *above* the meter, not below.

When the file and the meter agree within about a tenth, the advice can be trusted in the way that matters: the savings it prints are savings on a bill that looks like the real one. When they do not, the right move is to correct the facts, not the rules. This division of labour is the same one that the whole book relies on: the rules hold what an adviser knows in general, the facts hold what is true of this particular house, and each is corrected in its own place.

The meter also tells the family whether the advice was followed and worked. Read it on the same date each month, or note the kWh of the tokens bought, and write the number beside the month. If the kitchen bulb was replaced and the rice cooker now warms for four hours, the next months should show roughly 60 kWh less than before; if they do not, something in the house did not change as planned, and the family will know where to look.

2.8 Using It for Real

Your own house. Copy `home-sri.csv`, keep its first lines and the header, and write one line per appliance. Walk through the house with a notebook: read the label of each appliance (the power is written in W, sometimes as a range such as 350–400 W; take the larger number), and ask the family how many hours a day each is on. For the kinds, use those in **Table 2.2** where they fit; other kinds, such as `laptop` or `kettle`, are counted in the bill even when no rule is about them.

Measure instead of guessing. A plug-in energy meter, sold cheaply in electronics shops, sits between the socket and an appliance and shows the power and the kWh used. Plug the refrigerator into it for one full day: the kWh it shows, divided by the power on the label, is the real number of hours its motor runs. The same meter shows the standby power of a television or a charger when they are “off”. The Ministry of Energy also offers a household energy calculator app that estimates consumption from the same kind of list [7].

A new tariff quarter. When the government announces the tariff for the next quarter, copy `tariff-2026-q3.csv` to a new name, change the rates, and pass the new file as the second argument. Nothing else changes. The same goes for the tax lines of a real bill: to include a regional street-lighting tax of, say, 3 percent, multiply the bill by 103 and divide by 100 (with the rounding of Section 2.5.3) — Exercise 2.6 does exactly this.

Buying a new appliance. When an appliance must be replaced anyway, the energy-saving label on air conditioners and other appliances, with its stars, shows how efficient each model is; the government requires such labels and minimum efficiency standards for air conditioners [8]. Put the new model’s power in the file and the saver shows the difference.

Prepaid meters. A prepaid (*token*) customer pays the same price per kWh, but in advance. The saver's monthly kWh is then the number of kWh of tokens needed a month, and the saving tells how many fewer tokens to buy.

A warung or a small laundry. The same file format works for a shop: a display refrigerator, lights, a fan, a blender; or for a laundry: washing machines, dryers and irons, whose power is large and whose hours are long. Add rules that fit the business — *if the display fridge is half empty, turn it to a warmer setting at night* — as new values.

An energy evening in the neighborhood. A community group, a mosque committee or the monthly meeting of a neighborhood association (RT) can run the saver for several houses in one evening: one volunteer reads labels, another asks about hours, a third types the file on a laptop and prints the report. Because the rules are the same for everyone, families can compare their reports, and the most common advice — usually the old bulbs and the rice cooker — can be acted on together, for example by buying LED lamps in bulk. Keep each family's file private to that family; the numbers show daily habits, and a report is for the household it describes.

Keeping the rules up to date. The numbers inside the rules age too. LED lamps keep getting more efficient, so the 15 percent of `LED_PERCENT` may become 12; a new rice cooker may warm with less power; an adviser may decide that nine hours of air conditioning, not eight, is a fair night. Each such change is one constant at the top of `BillSaver.java` or `bill_saver.py`, and after changing it, run the tests: they will fail exactly where the old numbers were written into them, which is a reminder to check the new advice by hand before trusting it. Keep a short note in the chapter folder of what was changed, when and why, so that whoever looks after the saver next year knows where each number came from.

What the advice is and is not. The saver estimates; it does not promise. Its numbers are only as good as the watts and hours you give it, and a real bill also depends on the voltage, the age of the appliances and the weather. It is not financial advice about what to buy, either: whether an inverter air conditioner pays for itself is a decision for the family, with prices from a shop. And it never replaces an electrician: if lights flicker, a socket is warm, or the MCB trips often, call PLN or a certified electrician.

> **PITFALL.** Never "fix" a circuit breaker that trips by replacing it with a larger one or by bridging it. The breaker protects the cables in the walls from overheating. If a house really needs more power, the connection can be upgraded officially through PLN.

2.9 Common Mistakes, and How to Avoid Them

Mixing up watts and watt-hours. Watts are how fast; watt-hours are how much. A 60 W bulb does not "use 60 W a month". *Avoid it:* always multiply by hours, with Equation (2.1).

Forgetting the division by 1000. Watt-hours are a thousand times larger than kilowatt-hours; a bill a thousand times too large is the sign. *Avoid it:* check the result by hand on one appliance, as in Section 2.2.1. (The first version of the app's `kwh` method made this very mistake with the deci-watt-minutes and printed 1084.5 kWh for the air conditioner; comparing with the hand computation found it at once.)

Counting the same saving twice. Two rules about one lamp can each claim the kilowatt-hours the other already saved. *Avoid it:* apply the changes one after another to a running copy, as the engine does with `changed`.

Letting the computer's language change the numbers. Without `Locale.US`, Java prints `156.028` on an Indonesian computer. *Avoid it:* give the locale explicitly when the output must be the same everywhere.

Rounding decimals differently in two languages. Binary fractions can make Java and Python round a half differently. *Avoid it:* for money, compute with whole numbers in the smallest unit (sen), as in Equation (2.4).

A rule without a reason. Advice such as "switch it off" with no reason is ignored. *Avoid it:* every rule in the list has a `why`.

A rule that invents a saving. Guessing the saving of a temperature change would mislead. *Avoid it:* when a saving cannot be computed from the facts, say so, as `ac-setpoint` does.

Putting a rule inside the engine. A special `if` for one appliance hidden in the loop is invisible in the list of rules and breaks the design. *Avoid it:* every piece of advice is a value in `rules()`.

Comparing text with `==` in Java. The kinds are text; compare them with `.equals`, as the rules do.

Trusting the estimate over the meter. If the saver and the real bill disagree by a lot, the household file is wrong, not the meter. *Avoid it:* measure the biggest appliances with a plug-in meter and correct the file.

Comparing a thirty-day estimate with a thirty-one-day bill. The saver always counts 30 days; a bill between two meter readings may cover 28 to 31. *Avoid it:* scale the saver's total by the number of days on the bill divided by 30 before comparing, as in Exercise 2.9.

2.10 Chapter Summary

- The monthly **energy** of an appliance is power × count × hours × 30 / 1000 kWh (Equation (2.1)); **standby** adds the standby power over the hours not in use (Equation (2.2)); the **bill** is energy × rate (Equation (2.3)), with the PLN tariffs of July–September 2026 in **Table 2.1**.
- A **saving rule** checks one appliance, gives **advice** and a **reason**, and — when it honestly can — describes the appliance as it would be after the advice; the **saving** is the energy before minus the energy after.
- Rules written directly as `if` statements (**Listing 2.3**, **Listing 2.4**) mix knowledge with machinery. Written as **values** — condition, advice, reason, effect — they sit in one list and a small loop applies them (**Listing 2.8**, **Listing 2.9**, **Listing 2.10**); adding a rule changes the list, never the engine (**Figure 2.1**, **Table 2.2**).
- **Changed copies** of appliances let several rules look at the same appliance, and applying the changes one after another to a running copy counts each saving once.
- Money arithmetic in **whole numbers** — tenths of a watt, minutes, sen, and rounding by adding half the divisor (Equation (2.4)) — makes Java and Python agree to the last rupiah.
- For Bu Sri's house the saver fired thirteen pieces of advice and showed that the bill would fall from Rp 407,600 to Rp 317,603 a month (**Figure 2.2**); the same rules advised Dina's room with no change.
- The app advises; it does not guarantee, give financial advice, or replace an electrician.

2.11 Exercises

2.1 By hand, with Equation (2.1) and Equation (2.3), compute the monthly energy and cost of Bu Sri's water pump (250 W, 1.5 hours a day) on the `R1-1300` tariff. Compare with the report.

2.2 A family on the subsidized 900 VA tariff (Rp 605 per kWh) replaces four 40 W bulbs, each on 6 hours a day, with 6 W LEDs. How many kWh and how many rupiah does it save a month? And on the non-subsidized 900 VA tariff?

2.3 Using Equation (2.4), compute by hand in whole numbers the cost in sen and in rupiah of the Wi-Fi router (10 W, 24 hours a day) on `R1-1300`. Check your answer against the report.

2.4 Why does the `standby` rule require `minutes < 24 * 60`? Give an appliance in Bu Sri's file for which the condition matters.

2.5 Add the rule of Section 2.7.2 to both versions: *IF the kind is `heater` AND it is on more than 2 hours a day THEN switch it on only 30 minutes before bathing*, with the reason and the effect. Add a 350 W water heater used 3 hours a day to a copy of Bu Sri's file and run the saver.

2.6 Many regions add a street-lighting tax to the bill. Change the program so that the household file may contain a line `tax,3`, meaning 3 percent, and the report prints the bill with the tax. Use whole-number arithmetic.

2.7 Write the household file of your own home (or of a house you know), run the saver, and write three sentences for the family: the biggest eater, the best piece of advice, and whether the advice is worth the effort.

2.8 The `overload` rule adds the power of *everything*, but a real house never runs everything at once. Suggest a better rule, in words: which facts would it need, and what would it compute?

2.9 Bu Sri's next bill covers 31 days and shows 300 kWh. (a) Is the saver's estimate of 282.1 kWh, which counts 30 days, a good match? (b) After the family says it has followed all the advice, a 30-day bill shows 233.3 kWh. The saver predicted $282.1 - 62.3 = 219.8$ kWh. Which piece of advice was most likely not followed, and what should the family do to be sure?

2.12 Solutions to the Exercises

Solution 2.1. $E = 250 \times 1 \times 1.5 \times 30/1000 = 11.25$ kWh, and $11.25 \times 1444.70 = 16{,}252.9$, so about Rp 16,253. The report prints 11.3 kWh (11.25 rounded to one decimal, halves up) and Rp 16,253.

Solution 2.2. Before: $40 \times 4 \times 6 \times 30/1000 = 28.8$ kWh; after: $6 \times 4 \times 6 \times 30/1000 = 4.32$ kWh; the saving is 24.48 kWh a month. At Rp 605 per kWh that is $24.48 \times 605 = 14{,}810.4$, about Rp 14,810; at Rp 1,352 it is 33,097 rupiah (rounded from 33,096.96). The same LEDs save more than twice as much money for a non-subsidized household.

Solution 2.3. In the app's units the router uses $e = 100 \times 1440 \times 30 = 4{,}320{,}000$ deci-watt-minutes (10 W is 100 tenths; 24 hours are 1440 minutes), which is 7.2 kWh. By Equation (2.4): $(4{,}320{,}000 \times 144{,}470 + 300{,}000)/600{,}000 = 624{,}110{,}700{,}000/600{,}000 = 1{,}040{,}184.5$, rounded down to 1,040,184 sen. In rupiah: $(1{,}040{,}184 + 50)/100 = 10{,}402.34$, rounded down to Rp 10,402 — the report's number.

Solution 2.4. An appliance in use all 24 hours is never "off": it has no standby hours, so by Equation (2.2) its standby energy is zero and unplugging it saves nothing — and it would stop an

appliance that is meant to run all day. In Bu Sri's file the Wi-Fi router runs 24 hours; its standby field is 0, so the rule does not fire either way, but a router whose label lists, say, 0.5 W of standby power would get the absurd advice "switch it off at the socket when you are not using it" without the condition `minutes < 24 * 60`.

Solution 2.5. In Java, add to `rules()`:

```
/** Exercise 2.5: a saving rule for electric water heaters. */
public class HeaterRule {
    static SavingRule rule() {
        return new SavingRule("heater",
                a -> a.kind.equals("heater") && a.minutes > 2 * 60,
                a -> "switch it on only 30 minutes before bathing",
                a -> "it is a water heater on " + a.minutes / 60.0
                        + " hours a day",
                a -> a.withMinutes(30));
    }
}
```

and add `r.add(HeaterRule.rule());` before `return r;` (the solution folder keeps the rule in its own class, `HeaterRule.java`, so that the book's `BillSaver.java` stays as printed). The Python solution adds, in `rules()`:

```
"""Exercise 2.5: a saving rule for electric water heaters."""

from bill_saver import SavingRule


def heater_rule():
    return SavingRule(
        "heater",
        lambda a: a.kind == "heater" and a.minutes > 2 * 60,
        lambda a: "switch it on only 30 minutes before bathing",
        lambda a: "it is a water heater on " + str(a.minutes / 60)
                  + " hours a day",
        lambda a: a.with_minutes(30))
```

With the line `Water heater,heater,350,1,3,0,` added to a copy of the file (`home-sri-heater.csv`), the saver running with the extra rule (`BillSaverHeater`, `bill_saver_heater.py`) prints, among its advice:

```
11. [heater] Water heater
    advice: switch it on only 30 minutes before bathing
    why: it is a water heater on 3.0 hours a day
    saving: 26.3 kWh = Rp 37,923 a month
```

Solution 2.6. Read the line in `Household.load` (`tax` → a field `taxPercent`), and after computing `bill` compute the tax in whole numbers: `tax = (bill * taxPercent + 50) / 100` in Java, `(bill * tax_percent + 50) // 100` in Python, rounding to the nearest rupiah. Print `bill`, `tax` and `bill + tax`. For Bu Sri at 3 percent, the tax is $(407{,}600 \times 3 + 50)/100 = 12{,}228.5$, rounded down to Rp 12,228, and the total Rp 419,828. The savings also shrink the tax, by 3 percent of the saving.

Solution 2.7. Answers vary. A good answer names the appliance with the largest share, the advice with the largest saving per unit of effort (usually replacing old bulbs, or reducing keep-warm or air-conditioner hours), and compares the total saving with the effort and any cost, such as the price of LED lamps, which usually pays for itself within a few months.

Solution 2.8. A better rule needs, for each heavy appliance, the times of day it is used (for example "iron 19:00–19:30", "pump 05:00–06:30"), and would compute, for every moment of the day, the sum of the power of the appliances on at that moment; it warns only when the largest of

these sums exceeds the VA, and names the appliances that overlap then. With hours only, as in our file, the honest thing is the present rule: a warning, not a certainty.

Solution 2.9. (a) For 31 days the saver's estimate becomes $282.1 \times 31/30 \approx 291.5$ kWh. The meter shows $300 - 291.5 = 8.5$ kWh more, under 3 percent: a very good match, better than the 7.5 percent of Section 2.7.7 once the extra day is counted. (b) The meter shows $233.3 - 219.8 = 13.5$ kWh more than predicted. That is exactly the saving of item 8, keeping rice warm for four hours instead of fourteen, and close to the 12.0 kWh of the air-conditioner timer, item 2; the small standby items cannot explain it. The likely answer is the rice cooker, but the numbers cannot tell the two apart. The family should ask who does what at home, or plug the rice cooker into a plug-in energy meter for a day and read how many kWh it used; a day of fourteen hours of warming shows about 0.63 kWh, a day of four hours about 0.18.

References

[1] M. M. I. Subakti, "A Variable-Centered Intelligent Rule System," M.Sc.Eng. thesis, Dept. of Computer Science and Information Engineering, Taipei, Taiwan, Jun. 2005.

[2] Fortune Indonesia, "Daftar tarif listrik terbaru Agustus 2026, tidak naik," Aug. 2026. [Online]. Available: https://www.fortuneidn.com/news/tarif-listrik-terbaru-agustus-2026-d7w02-00-lflgw-prph33

[3] CNBC Indonesia, "Tarif listrik terbaru PLN per kWh, berlaku hingga September 2026," Aug. 10, 2026. [Online]. Available: https://www.cnbcindonesia.com/news/20260810101545-4-757861/tarif-listrik-terbaru-pln-per-kwh-berlaku-hingga-september-2026

[4] Kementerian Energi dan Sumber Daya Mineral, "3M, resep hemat energi dari Kementerian ESDM," Ditjen EBTKE, Nov. 8, 2017. [Online]. Available: https://www.esdm.go.id/id/berita-unit/direktorat-jenderal-ebtke/3m-resep-hemat-energi-dari-kementerian-esdm

[5] Oracle, "Class Formatter," *Java SE 25 & JDK 25 API Specification*, 2025. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Formatter.html

[6] Python Software Foundation, "Format specification mini-language," *The Python Standard Library*, 3.14. [Online]. Available: https://docs.python.org/3/library/string.html#formatspec

[7] Kementerian Energi dan Sumber Daya Mineral, "Ayo unduh aplikasi kalkulator hemat energi rumah tangga," Ditjen EBTKE. [Online]. Available: https://www.esdm.go.id/id/berita-unit/direktorat-jenderal-ebtke/ayo-unduh-aplikasi-kalkulator-hemat-energi-rumah-tangga

[8] Kementerian Energi dan Sumber Daya Mineral, "Pemerintah terapkan SKEM dan pencantuman label hemat energi untuk peranti pengkondisi udara," Ditjen EBTKE. [Online]. Available: https://www.esdm.go.id/id/berita-unit/directorate-general-ebtke/pemerintah-terapkan-skem-dan-pencantuman-label-hemat-energi-untuk-peranti-pengkondisi-udara

Chapter 3

Phone and Wi-Fi Troubleshooter That Learns from Each Fix

By the end of this chapter you will be able to explain why a flat list of rules becomes harder to maintain with every rule you add; describe the answer of Ripple Down Rules — exceptions attached in the context where a mistake happened, checked against cornerstone cases, chosen from a list of differences; build a complete single-classification Ripple Down Rules engine from scratch in Java and in Python, in about a hundred lines; and use the shared vcirs-core engine to run the **Wi-Fi doctor**, a troubleshooter for phone and Wi-Fi problems that asks only the questions it needs, explains its advice, and learns the right fix from the person at the counter every time its advice fails. You will run four real conversations with it, watch its knowledge grow, and see the saved knowledge base that both the Java and the Python versions read.

3.1 Mas Anto's Counter

Mas Anto's phone counter is a two-meter glass case at the corner of the market: phone cases, chargers, SIM cards, and a small stool for customers who need help. By his own count, half of the people who sit on that stool say the same thing: *the internet is dead*. Indonesia had about 235 million internet users in 2026, more than eight in ten people [1], and for most of them the internet *is* the phone. When it stops, they come to people like Mas Anto.

Over the years he has learned what to ask. *Wi-Fi or mobile data? Is the Wi-Fi symbol on? Is the little airplane on? Do the other phones at home have internet? What color are the router's lights?* And he has learned what to do with the answers. His knowledge did not come from a book. It came from customers, and in a very particular order: first a simple rule that works most of the time, then a customer for whom it fails, then a correction for *that kind* of customer. "Turn on Wi-Fi" — except when airplane mode is on. "Restart the router" — except when the router's light is red, because then the problem is the provider's cable, and restarting only wastes time.

Mas Anto would like two things. He would like his nephew Fajar, who minds the counter on Fridays, to give the same advice he would give. And he would like his knowledge to keep growing without anyone having to rewrite it, because every week brings a new kind of problem. This chapter builds the program that does both. It is the first application in this book that *learns*: every time its advice fails, the person at the counter tells it what fixed the problem and what was different about this case, and the program puts that knowledge exactly where it belongs.

3.2 The Idea in Plain Words

3.2.1 Why a flat list of rules breaks

Let us start with the obvious design and watch it go wrong. A troubleshooter could be a list of rules tried from the top; the first rule whose conditions match the customer's facts gives the answer. **Listing 3.1** tries three versions of such a list on four customers whose right answers we know.

Listing 3.1 `FlatRules.java`: a troubleshooter as a flat list of rules, tried in three versions.

```
import java.util.List;
import java.util.Map;

/** A troubleshooter as a flat list of rules: the first rule that matches wins. */
public class FlatRules {
    record Rule(Map<String, String> when, String fix) {
        boolean matches(Map<String, String> facts) {
            for (Map.Entry<String, String> c : when.entrySet()) {
                if (!c.getValue().equals(facts.get(c.getKey()))) {
                    return false;
                }
            }
            return true;
        }
    }

    static String advise(List<Rule> rules, Map<String, String> facts) {
        for (Rule r : rules) {
            if (r.matches(facts)) {
                return r.fix();
            }
        }
        return "no advice";
    }

    public static void main(String[] args) {
        Rule wifiOff = new Rule(Map.of("wifi icon", "off"), "turn on Wi-Fi");
        Rule router = new Rule(Map.of("other devices online", "no"),
                "restart the router");
        Rule airplane = new Rule(Map.of("airplane mode", "on"),
                "turn off airplane mode");

        List<Map<String, String>> cases = List.of(
                Map.of("wifi icon", "off", "airplane mode", "off"),
                Map.of("wifi icon", "on", "other devices online", "no"),
                Map.of("wifi icon", "off", "airplane mode", "on"),
                Map.of("wifi icon", "on", "airplane mode", "on",
                        "other devices online", "no"));
        List<String> right = List.of("turn on Wi-Fi", "restart the router",
                "turn off airplane mode", "restart the router");
```

Listing 3.1 (Continued) *FlatRules.java: a troubleshooter as a flat list of rules, tried in three versions.*

```
        List<List<Rule>> versions = List.of(
                List.of(wifiOff, router),
                List.of(wifiOff, router, airplane),
                List.of(airplane, wifiOff, router));
        List<String> names = List.of("Two rules", "New rule at the end",
                "New rule at the front");
        for (int v = 0; v < versions.size(); v++) {
            System.out.println(names.get(v) + ":");
            for (int i = 0; i < cases.size(); i++) {
                String got = advise(versions.get(v), cases.get(i));
                String mark = got.equals(right.get(i)) ? "right" : "WRONG";
                System.out.printf("  case %d: %-24s %s%n", i + 1, got, mark);
            }
        }
    }
}
```

A `record` is a short way to write a Java class that only holds data: `Rule` holds the conditions (`when`, a map from variable to value) and the fix. `matches` checks that every condition's value equals the customer's fact. `advise` returns the fix of the first matching rule. The customers are maps of facts,

and `right` holds the fix that actually worked for each of them. **Listing 3.2** is the same in Python, where a rule is simply a pair of a dictionary and a fix.

Listing 3.2 `flat_rules.py`: the same three versions in Python.

```
"""A troubleshooter as a flat list of rules: the first rule that matches wins."""


def matches(when, facts):
    return all(facts.get(var) == value for var, value in when.items())


def advise(rules, facts):
    for when, fix in rules:
        if matches(when, facts):
            return fix
    return "no advice"


wifi_off = ({"wifi icon": "off"}, "turn on Wi-Fi")
router = ({"other devices online": "no"}, "restart the router")
airplane = ({"airplane mode": "on"}, "turn off airplane mode")

cases = [
    {"wifi icon": "off", "airplane mode": "off"},
    {"wifi icon": "on", "other devices online": "no"},
    {"wifi icon": "off", "airplane mode": "on"},
    {"wifi icon": "on", "airplane mode": "on", "other devices online": "no"},
]
right = ["turn on Wi-Fi", "restart the router", "turn off airplane mode",
         "restart the router"]

versions = [[wifi_off, router], [wifi_off, router, airplane],
            [airplane, wifi_off, router]]
names = ["Two rules", "New rule at the end", "New rule at the front"]
for name, rules in zip(names, versions):
    print(name + ":")
    for i, facts in enumerate(cases):
        got = advise(rules, facts)
        mark = "right" if got == right[i] else "WRONG"
        print(f"  case {i + 1}: {got:<24} {mark}")
```

Both print:

```
Two rules:
  case 1: turn on Wi-Fi            right
  case 2: restart the router       right
  case 3: turn on Wi-Fi            WRONG
  case 4: restart the router       right
New rule at the end:
  case 1: turn on Wi-Fi            right
  case 2: restart the router       right
  case 3: turn on Wi-Fi            WRONG
  case 4: restart the router       right
New rule at the front:
  case 1: turn on Wi-Fi            right
  case 2: restart the router       right
  case 3: turn off airplane mode   right
  case 4: turn off airplane mode   WRONG
```

With two rules, customer 3 — whose Wi-Fi symbol is off *because airplane mode is on* — gets the wrong advice. Mas Anto adds the obvious rule, “if airplane mode is on, turn it off”. Added at the **end** of the list it never fires for customer 3, because the Wi-Fi rule above it matches first. Added at the **front** it fixes customer 3 — and breaks customer 4, whose phone happens to be in airplane mode with Wi-Fi on (modern phones allow that) while the real problem is the router. A rule that

is right for one customer, placed in a list, changes the answers for customers it was never meant for.

This is the maintenance problem of rule-based systems in miniature [2], [3]. With three rules we can see it; with three hundred, nobody can. Every new rule must be checked against every old one, and the order of the rules becomes knowledge in its own right, invisible and fragile. In practice this is why large rule bases needed knowledge engineers, and why experts found them so hard to keep up to date [4].

3.2.2 Knowledge comes with a context

Now look at how Mas Anto actually thinks. He never says "if airplane mode is on, turn it off" in the abstract. He says: *when I told a customer to turn on Wi-Fi and it did not help, the difference was that airplane mode was on.* His new rule is an **exception** to an old rule, and it only makes sense in the context of that old rule: among the customers whose Wi-Fi symbol is off. Customer 4, whose Wi-Fi is on, never reaches it.

This observation is the heart of **Ripple Down Rules** (RDR), proposed at the end of the 1980s by Paul Compton and his colleagues for a pathology expert system [5], [2], and later used to build PEIRS, which interpreted chemical pathology reports at St Vincent's Hospital in Sydney with knowledge added entirely by pathologists [6]. An expert's knowledge, they argued, is always a justification given in a context: *this* conclusion rather than *that* one, for *this* case. So the system should store knowledge in exactly that form. When the system gives a wrong answer for a case, the expert adds a new rule that applies only where the wrong answer came from — as an exception of the rule that gave it. Rules are never edited or deleted; they are only refined, and the refinements "ripple down" below them [4], [7].

> **DEFINITION.** In *Ripple Down Rules*, every rule may have *exceptions*: rules that are tried only when it is true, and whose conclusion replaces its conclusion when they are true as well. A new rule is always added as an exception at the place where the wrong conclusion was reached, so it can change the answer only for cases that reach that place.

3.2.3 Cornerstone cases and the difference list

How does the expert choose the conditions of the exception? RDR keeps, with every rule, the case that caused it to be created: its **cornerstone case** [2], [4]. When a new case is answered wrongly by some rule, the system compares the new case with that rule's cornerstone case and shows the **differences**. The expert only has to pick, among the differences, the features that explain why the new case needs another answer. **Figure 3.1** shows this for customer 3: the cornerstone case of the rule "turn on Wi-Fi" had airplane mode *off*; the new case has it *on*. The difference is the reason, and it becomes the condition of the exception.

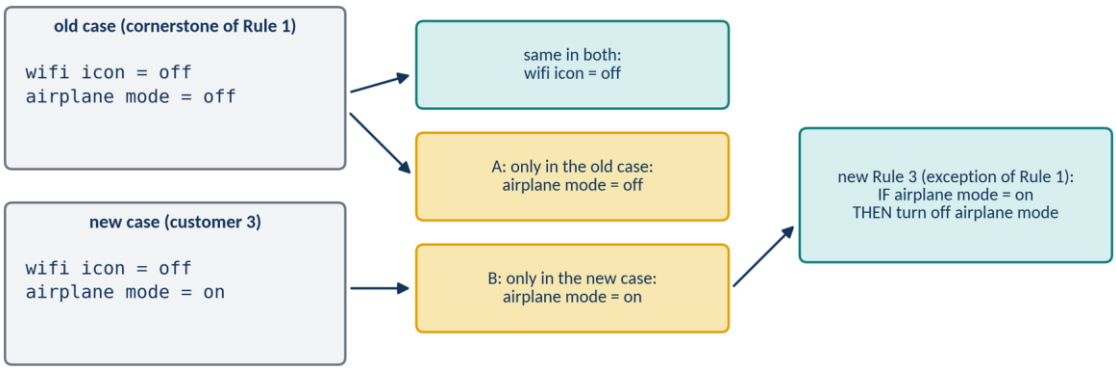

Figure 3.1 The list of difference between a rule's cornerstone case and a new case. Section A holds what only the old case has; section B what only the new case has. The expert's new rule is built from these differences.

The 2006 revision of VCIRS names the two sides of this comparison precisely [8], [9]. Section **A** holds the conditions found only in the old (cornerstone) case; section **B** holds those found only in the new case. A new exception can be made from the negation of section A (*airplane mode is not off*), from section B (*airplane mode is on*), or from both. Our hand-made engine in this chapter uses section B, the simplest and most common choice; the vcirs-core engine offers all of them.

Why does the cornerstone case guarantee that nothing breaks? Because the new condition is, by construction, *false* for the old case: the old case did not have airplane mode on. So the rule that answered the old case correctly still answers it correctly. Every rule in an RDR knowledge base carries the case that justifies it, and every new rule is chosen to leave that case alone. That is what lets an expert who has never seen the rule base add knowledge safely [4].

3.2.4 The shape of the knowledge: except and else

In the original, single-classification form of RDR, the knowledge base is a **binary tree** [2]. Every rule has two branches. The **except** branch is tried when the rule is true: it holds the exceptions of the rule. The **else** branch is tried when the rule is false: it holds the next rule to try. **Figure 3.2** shows the tree that Mas Anto builds in Section 3.4, with his five customers.

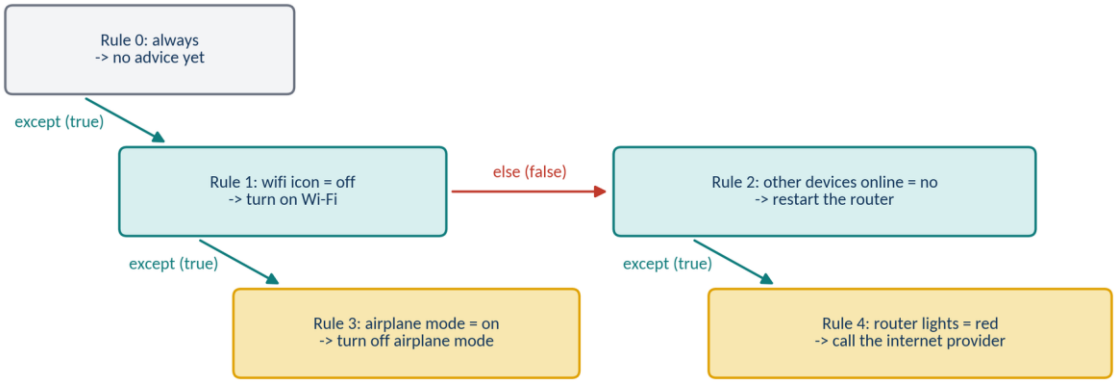

Figure 3.2 A single-classification Ripple Down Rules tree. Rule 0 is always true; the except branch holds exceptions, tried when a rule is true; the else branch holds the next rule, tried when a rule is false. The answer is the conclusion of the last true rule.

To answer a case, start at Rule 0, which is always true and gives the default answer *no advice yet*. Go down its except branch. At each rule, if it is true, remember its conclusion and follow its except branch; if it is false, follow its else branch. When there is no branch to follow, stop. The answer is the conclusion of the **last rule that was true**.

Try it on customer 4 of **Listing 3.1**: Wi-Fi on, airplane mode on, other devices offline. Rule 0 is true (answer so far: *no advice yet*); Rule 1, *wifi icon = off*, is false, so we take its else branch; Rule 2, *other devices online = no*, is true (answer: *restart the router*); we take its except branch to Rule 4, *router lights = red*, which is false, and it has no else branch, so we stop. The answer is *restart the router* — right, because the airplane rule, Rule 3, lives under Rule 1, and customer 4 never went there.

And where does a *new* rule go? Exactly where the search stopped. If the last rule tried was true, the new rule becomes its except branch; if it was false, its else branch. Either way, only cases that follow the same path as the case being corrected will ever reach it.

3.2.5 What RDR is good at, and what it is not

Ripple Down Rules have a particular character, and it helps to know it before trusting a troubleshooter built on them [3], [4]. Their answers are **precise**: because every rule was made to fix a real case, and made so as not to disturb the cases before it, the knowledge base tends to answer the cases it has seen, and cases like them, correctly. The price is **coverage**: a young RDR knowledge base is cautious. It answers what it has been taught and, for a truly new situation, says "no advice yet" rather than guessing. Compared with a machine-learning method trained on thousands of examples, an RDR system typically wins on precision and loses on coverage, especially at the beginning [3]. For a phone counter this is exactly the right character. A wrong answer wastes the customer's time and Mas Anto's reputation; "I do not know yet — let me look" costs nothing, and the next customer with the same problem gets the answer.

Second, rules tend to be **over-general** at first: Mas Anto's first rule, *if the Wi-Fi symbol is off, turn Wi-Fi on*, is true most of the time and wrong some of the time. That is not a flaw; it is how knowledge is built. Each exception narrows a rule exactly where it is too general, and only there.

Third, single-classification RDR gives **one answer** per case. A customer can have two independent problems — no quota *and* a weak signal — and a single answer can only name one. The variant called Multiple Classification RDR evaluates every branch that applies and gives all the conclusions it finds [10]; the vcirs-core engine can do the same when asked (`multiple = true`), and Chapter 5 uses this for a health advisor, where two findings may both matter.

Finally, RDR by itself **reasons simply**. It walks the tree; it cannot start from a goal, such as "is it the router?", and ask only the questions needed to prove it. That is backward chaining, the strength of classic rule-based systems, and it is exactly what the variable-centered structure of VCIRS adds back [3], [11]. Chapter 8 uses it.

3.2.6 Ripple Down Rules at work

None of this is only a tidy idea on paper. After PEIRS, Ripple Down Rules were used in commercial systems in pathology laboratories, where the pathologists themselves, without programmers, added rules one case at a time while doing their daily work, and knowledge bases grown this way reached thousands of rules [4]. What stands out in those reports is the cost: adding a rule takes a few minutes, and that time hardly grows as the knowledge base gets bigger, because the expert never needs to look at the whole rule base — only at the case in front of her, the cornerstone case beside it, and the list of differences [4], [7].

The same pattern appears in the other fields gathered in the review of two decades of RDR research: help desks, the classification of email and documents, and configuration [7]. What they have in common is what Mas Anto's counter has. There is a person who knows the answer when

she sees a case, but cannot write down all her knowledge in advance; there is a stream of cases that keeps coming; and a wrong answer can be recognized, so it can be corrected. Wherever those three things exist, knowledge can grow out of the work itself.

There is also a lesson about what is *not* needed. Those systems did not wait until their knowledge base was "finished" before using it. They started with a few rules, answered what they could, and left the rest to the expert, who then taught it. Our Wi-Fi doctor will work in exactly that way.

3.3 Ripple Down Rules from Scratch

We now build this engine ourselves. **Listing 3.3** is the whole of it in Java.

Listing 3.3 `Rdr.java`: a complete single-classification Ripple Down Rules engine.

```
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

/**
 * Single-classification Ripple Down Rules, built from scratch. Every rule
 * has an "except" branch (tried when the rule is true) and an "else" branch
 * (tried when it is false); the answer is the conclusion of the last rule
 * that was true. A new rule is hung exactly where the search stopped.
 */
public class Rdr {
    public static class Rule {
        public final int number;
        public final Map<String, String> conditions;
        public final String conclusion;
        public final Map<String, String> cornerstone;   // the case that made it
        public Rule ifTrue;                             // the "except" branch
        public Rule ifFalse;                            // the "else" branch

        Rule(int number, Map<String, String> conditions, String conclusion,
             Map<String, String> cornerstone) {
            this.number = number;
            this.conditions = conditions;
            this.conclusion = conclusion;
            this.cornerstone = cornerstone;
        }

        boolean matches(Map<String, String> facts) {
            for (Map.Entry<String, String> c : conditions.entrySet()) {
                if (!c.getValue().equals(facts.get(c.getKey()))) {
                    return false;
                }
            }
            return true;
        }
```

Listing 3.3 (Continued) *Rdr.java: a complete single-classification Ripple Down Rules engine.*

```
        public String text() {
            List<String> parts = new ArrayList<>();
            for (Map.Entry<String, String> c : conditions.entrySet()) {
                parts.add(c.getKey() + " = " + c.getValue());
            }
            String when = parts.isEmpty() ? "(always)"
                    : String.join(" AND ", parts);
            return "Rule " + number + ": IF " + when + " THEN " + conclusion;
        }
    }

    /** The answer, the rule that gave it, and where the search stopped. */
    public static class Answer {
        public String conclusion;
        public Rule lastTrue;
        public Rule lastTried;
        public boolean lastResult;
        public final List<String> trace = new ArrayList<>();
    }

    public final Rule root = new Rule(0, new LinkedHashMap<>(), "no advice yet",
            new LinkedHashMap<>());
    private int count = 0;

    public Answer classify(Map<String, String> facts) {
        Answer a = new Answer();
        a.lastTrue = root;
        a.lastTried = root;
        a.lastResult = true;
        Rule rule = root.ifTrue;
        while (rule != null) {
            boolean ok = rule.matches(facts);
            a.trace.add("Rule " + rule.number + (ok ? " is true" : " is false"));
            a.lastTried = rule;
            a.lastResult = ok;
            if (ok) {
                a.lastTrue = rule;
                rule = rule.ifTrue;
            } else {
                rule = rule.ifFalse;
            }
        }
        a.conclusion = a.lastTrue.conclusion;
        return a;
    }
```

Listing 3.3 (Continued) *Rdr.java: a complete single-classification Ripple Down Rules engine.*

```
    /** The facts of the new case that its cornerstone case does not have. */
    public static Map<String, String> differences(Map<String, String> facts,
                                                  Map<String, String> cornerstone) {
        Map<String, String> out = new LinkedHashMap<>();
        for (Map.Entry<String, String> f : facts.entrySet()) {
            if (!f.getValue().equals(cornerstone.get(f.getKey()))) {
                out.put(f.getKey(), f.getValue());
            }
        }
        return out;
    }

    /** Hang a new rule where the search for this case stopped. */
    public Rule learn(Map<String, String> facts, String conclusion,
                      Map<String, String> conditions) {
        Answer a = classify(facts);
        Rule rule = new Rule(++count, conditions, conclusion, facts);
        if (a.lastResult) {
            a.lastTried.ifTrue = rule;
        } else {
            a.lastTried.ifFalse = rule;
        }
        return rule;
    }

    /** "wifi icon = off, airplane mode = on" */
    public static String show(Map<String, String> facts) {
        List<String> parts = new ArrayList<>();
        for (Map.Entry<String, String> f : facts.entrySet()) {
            parts.add(f.getKey() + " = " + f.getValue());
        }
        return parts.isEmpty() ? "(nothing)" : String.join(", ", parts);
    }

    public List<String> tree() {
        List<String> lines = new ArrayList<>();
        draw(root, "", "", lines);
        return lines;
    }
```

Listing 3.3 (Continued) *Rdr.java: a complete single-classification Ripple Down Rules engine.*

```
    private void draw(Rule rule, String indent, String label, List<String> out) {
        out.add(indent + label + rule.text());
        if (rule.ifTrue != null) {
            draw(rule.ifTrue, indent + "  ", "except: ", out);
        }
        if (rule.ifFalse != null) {
            draw(rule.ifFalse, indent, "else:   ", out);
        }
    }
}
```

Read it in four parts.

The rule. A `Rule` has a number, its conditions (a map from variable to value, all of which must match), its conclusion, its cornerstone case, and the two branches `ifTrue` (except) and `ifFalse` (else), which are other rules or `null`. `matches` checks the conditions against the facts of a case; `text` writes the rule the way this book writes rules.

The answer. `Answer` records the conclusion, `lastTrue` (the rule that gave it), `lastTried` and `lastResult` (the rule where the search stopped, and whether it was true there), and a `trace` of every rule tried, which is the engine's explanation of *how* it reached its answer.

Classifying. `classify` is the walk of Section 3.2.4 in a dozen lines. It starts below `root`, which stands for Rule 0, and repeats: test the rule, record it in the trace, remember where we are; if the rule is true, remember it as `lastTrue` and go to `ifTrue`, otherwise go to `ifFalse`; stop at `null`. The answer is `lastTrue.conclusion`.

Learning. `differences` lists the facts of a new case whose value differs from (or is missing in) a cornerstone case: section B of the difference list. `learn` classifies the case again to find where the search stops, creates the new rule with the case as its cornerstone, and hangs it on the `ifTrue` branch of the last rule tried if that rule was true, or on its `ifFalse` branch if it was false. That is all RDR needs. `tree` prints the knowledge base, indenting exceptions under their rule and writing else branches at the same depth.

Listing 3.4 is the same engine in Python.

Listing 3.4 `rdr.py`: the same engine in Python.

```
"""Single-classification Ripple Down Rules, built from scratch. Every rule
has an "except" branch (tried when the rule is true) and an "else" branch
(tried when it is false); the answer is the conclusion of the last rule that
was true. A new rule is hung exactly where the search stopped."""


class Rule:
    def __init__(self, number, conditions, conclusion, cornerstone):
        self.number = number
        self.conditions = conditions
        self.conclusion = conclusion
        self.cornerstone = cornerstone   # the case that made it
        self.if_true = None              # the "except" branch
        self.if_false = None             # the "else" branch

    def matches(self, facts):
        return all(facts.get(var) == value
                   for var, value in self.conditions.items())

    def text(self):
        parts = [var + " = " + value for var, value in self.conditions.items()]
        when = " AND ".join(parts) if parts else "(always)"
        return "Rule " + str(self.number) + ": IF " + when + " THEN " \
            + self.conclusion


class Answer:
    """The answer, the rule that gave it, and where the search stopped."""

    def __init__(self, root):
        self.conclusion = None
        self.last_true = root
        self.last_tried = root
        self.last_result = True
        self.trace = []


def show(facts):
    """'wifi icon = off, airplane mode = on'"""
    parts = [var + " = " + value for var, value in facts.items()]
    return ", ".join(parts) if parts else "(nothing)"
```

Listing 3.4 (Continued) *rdr.py: the same engine in Python.*

```
def differences(facts, cornerstone):
    """The facts of the new case that its cornerstone case does not have."""
    return {var: value for var, value in facts.items()
            if cornerstone.get(var) != value}


class Rdr:
    def __init__(self):
        self.root = Rule(0, {}, "no advice yet", {})
        self.count = 0

    def classify(self, facts):
        a = Answer(self.root)
        rule = self.root.if_true
        while rule is not None:
            ok = rule.matches(facts)
            a.trace.append("Rule " + str(rule.number)
                           + (" is true" if ok else " is false"))
            a.last_tried = rule
            a.last_result = ok
            if ok:
                a.last_true = rule
                rule = rule.if_true
            else:
                rule = rule.if_false
        a.conclusion = a.last_true.conclusion
        return a

    def learn(self, facts, conclusion, conditions):
        """Hang a new rule where the search for this case stopped."""
        a = self.classify(facts)
        self.count += 1
        rule = Rule(self.count, conditions, conclusion, facts)
        if a.last_result:
            a.last_tried.if_true = rule
        else:
            a.last_tried.if_false = rule
        return rule

    def tree(self):
        lines = []
```

Listing 3.4 (Continued) *rdr.py: the same engine in Python.*

```
        def draw(rule, indent, label):
            lines.append(indent + label + rule.text())
            if rule.if_true is not None:
                draw(rule.if_true, indent + "  ", "except: ")
            if rule.if_false is not None:
                draw(rule.if_false, indent, "else:   ")

        draw(self.root, "", "")
        return lines
```

The Python version is a little shorter: a dictionary comprehension builds the differences in one line, and `all(...)` checks every condition in one expression. The logic is identical, line for line, and so is the output of the program that uses it.

> **KEY INSIGHT.** The engine never looks at the meaning of a rule. It only follows true and false branches and hangs new rules where the search stopped. All the troubleshooting knowledge is in the rules, and all of it came from cases.

3.4 Mas Anto Teaches Five Customers

Listing 3.5 replays Mas Anto's first five customers of a morning. For each, it shows the search, the answer, and — if the answer is wrong — the right fix, the differences from the cornerstone case of the rule that answered, and the rule Mas Anto adds, built from the differences he chooses. At the end it prints the knowledge base and tries every earlier customer again.

Listing 3.5 `RdrDemo.java`: five customers teach an empty troubleshooter.

```
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

/** Mas Anto teaches the troubleshooter, one customer at a time. */
public class RdrDemo {
    /** facts("wifi icon=off", "airplane mode=on") keeps the order given. */
    static Map<String, String> facts(String... pairs) {
        Map<String, String> m = new LinkedHashMap<>();
        for (String p : pairs) {
            String[] kv = p.split("=");
            m.put(kv[0], kv[1]);
        }
        return m;
    }

    static void customer(Rdr kb, int n, Map<String, String> facts, String right,
                         String... chosen) {
        Rdr.Answer a = kb.classify(facts);
        System.out.println("Customer " + n + ": " + Rdr.show(facts));
        System.out.println("  search: " + (a.trace.isEmpty() ? "(no rules yet)"
                : String.join(", ", a.trace)));
        System.out.println("  answer: " + a.conclusion);
        if (a.conclusion.equals(right)) {
            System.out.println("  right!");
            return;
        }
        Map<String, String> diff = Rdr.differences(facts,
                a.lastTrue.cornerstone);
        System.out.println("  wrong; the right fix is: " + right);
        System.out.println("  differences from the case of Rule "
                + a.lastTrue.number + ": " + Rdr.show(diff));
        Rdr.Rule rule = kb.learn(facts, right, facts(chosen));
        System.out.println("  learned " + rule.text());
    }
```

Listing 3.5 (Continued) *RdrDemo.java: five customers teach an empty troubleshooter.*

```
    public static void main(String[] args) {
        Rdr kb = new Rdr();
        List<Map<String, String>> seen = List.of(
                facts("wifi icon=off", "airplane mode=off"),
                facts("wifi icon=on", "other devices online=no"),
                facts("wifi icon=off", "airplane mode=on"),
                facts("wifi icon=on", "airplane mode=on",
                        "other devices online=no"),
                facts("wifi icon=on", "other devices online=no",
                        "router lights=red"));
        List<String> right = List.of("turn on Wi-Fi", "restart the router",
                "turn off airplane mode", "restart the router",
                "call the internet provider");
        customer(kb, 1, seen.get(0), right.get(0), "wifi icon=off");
        customer(kb, 2, seen.get(1), right.get(1), "other devices online=no");
        customer(kb, 3, seen.get(2), right.get(2), "airplane mode=on");
        customer(kb, 4, seen.get(3), right.get(3));
        customer(kb, 5, seen.get(4), right.get(4), "router lights=red");

        System.out.println();
        System.out.println("The knowledge base now:");
        for (String line : kb.tree()) {
            System.out.println("  " + line);
        }
        System.out.println();
        System.out.println("Every earlier customer again:");
        for (int i = 0; i < seen.size(); i++) {
            String got = kb.classify(seen.get(i)).conclusion;
            System.out.printf("  customer %d: %-28s %s%n", i + 1, got,
                    got.equals(right.get(i)) ? "right" : "WRONG");
        }
    }
}
```

`facts` builds a map that keeps the order in which the facts are written, so that the differences are printed in a predictable order. `customer` does one step of teaching: classify, print, and if the answer is wrong, compute the differences against `a.lastTrue.cornerstone` and call `learn` with the conditions Mas Anto chose (passed as the last arguments). **Listing 3.6** is the Python version.

Listing 3.6 `rdr_demo.py`: the same five customers in Python.

```
"""Mas Anto teaches the troubleshooter, one customer at a time."""

from rdr import Rdr, differences, show


def facts(*pairs):
    """facts("wifi icon=off", "airplane mode=on") keeps the order given."""
    return dict(p.split("=") for p in pairs)


def customer(kb, n, case, right, *chosen):
    a = kb.classify(case)
    print("Customer " + str(n) + ": " + show(case))
    print("  search: " + (", ".join(a.trace) if a.trace else "(no rules yet)"))
    print("  answer: " + a.conclusion)
    if a.conclusion == right:
        print("  right!")
        return
    diff = differences(case, a.last_true.cornerstone)
    print("  wrong; the right fix is: " + right)
    print("  differences from the case of Rule " + str(a.last_true.number)
          + ": " + show(diff))
    rule = kb.learn(case, right, facts(*chosen))
    print("  learned " + rule.text())


kb = Rdr()
seen = [facts("wifi icon=off", "airplane mode=off"),
        facts("wifi icon=on", "other devices online=no"),
        facts("wifi icon=off", "airplane mode=on"),
        facts("wifi icon=on", "airplane mode=on", "other devices online=no"),
        facts("wifi icon=on", "other devices online=no", "router lights=red")]
right = ["turn on Wi-Fi", "restart the router", "turn off airplane mode",
         "restart the router", "call the internet provider"]
customer(kb, 1, seen[0], right[0], "wifi icon=off")
customer(kb, 2, seen[1], right[1], "other devices online=no")
customer(kb, 3, seen[2], right[2], "airplane mode=on")
customer(kb, 4, seen[3], right[3])
customer(kb, 5, seen[4], right[4], "router lights=red")
```

Listing 3.6 (Continued) *rdr_demo.py: the same five customers in Python.*

```
print()
print("The knowledge base now:")
for line in kb.tree():
    print("  " + line)
print()
print("Every earlier customer again:")
for i, case in enumerate(seen):
    got = kb.classify(case).conclusion
    print(f"  customer {i + 1}: {got:<28} "
          + ("right" if got == right[i] else "WRONG"))
```

Run `java -cp out RdrDemo` (after `javac -d out Rdr.java RdrDemo.java`) or `python rdr_demo.py`:

```
Customer 1: wifi icon = off, airplane mode = off
  search: (no rules yet)
  answer: no advice yet
  wrong; the right fix is: turn on Wi-Fi
  differences from the case of Rule 0: wifi icon = off, airplane mode = off
  learned Rule 1: IF wifi icon = off THEN turn on Wi-Fi
Customer 2: wifi icon = on, other devices online = no
  search: Rule 1 is false
  answer: no advice yet
  wrong; the right fix is: restart the router
  differences from the case of Rule 0: wifi icon = on, other devices online = no
  learned Rule 2: IF other devices online = no THEN restart the router
Customer 3: wifi icon = off, airplane mode = on
  search: Rule 1 is true
  answer: turn on Wi-Fi
  wrong; the right fix is: turn off airplane mode
  differences from the case of Rule 1: airplane mode = on
  learned Rule 3: IF airplane mode = on THEN turn off airplane mode
Customer 4: wifi icon = on, airplane mode = on, other devices online = no
  search: Rule 1 is false, Rule 2 is true
  answer: restart the router
  right!
Customer 5: wifi icon = on, other devices online = no, router lights = red
  search: Rule 1 is false, Rule 2 is true
  answer: restart the router
  wrong; the right fix is: call the internet provider
  differences from the case of Rule 2: router lights = red
  learned Rule 4: IF router lights = red THEN call the internet provider

The knowledge base now:
  Rule 0: IF (always) THEN no advice yet
    except: Rule 1: IF wifi icon = off THEN turn on Wi-Fi
      except: Rule 3: IF airplane mode = on THEN turn off airplane mode
    else:   Rule 2: IF other devices online = no THEN restart the router
      except: Rule 4: IF router lights = red THEN call the internet provider

Every earlier customer again:
  customer 1: turn on Wi-Fi                right
  customer 2: restart the router           right
  customer 3: turn off airplane mode       right
  customer 4: restart the router           right
  customer 5: call the internet provider   right
```

Follow the knowledge as it grows.

Customer 1 arrives at an empty knowledge base. The search tries no rule; the answer is Rule 0's *no advice yet*. The difference from Rule 0's cornerstone (which is empty) is everything the customer said. Mas Anto chooses only `wifi icon = off` — airplane mode being off is not the reason — and Rule 1 is hung on Rule 0's except branch.

Customer 2 has Wi-Fi on. Rule 1 is false; the search stops there, so the new Rule 2, *other devices online = no → restart the router*, is hung on Rule 1's **else** branch. It will be tried for every customer whose Wi-Fi is on.

Customer 3 is the airplane-mode customer who broke the flat list. Rule 1 is true, so the answer is *turn on Wi-Fi*, which is wrong. The differences from Rule 1's cornerstone case (customer 1) come down to one fact, `airplane mode = on`, and Rule 3 is hung on Rule 1's **except** branch: an exception, in the context where it belongs.

Customer 4 is the customer that the flat list got wrong after the "fix". Here nothing needs to be taught: Rule 1 is false, Rule 2 is true, and the answer is right. The airplane rule is never consulted, because customer 4 never enters Rule 1's context.

Customer 5 has a red light on the router. Rule 2 answers *restart the router*, which will not help: the provider's line is down. The only difference from Rule 2's cornerstone (customer 2) is `router lights = red`, and Rule 4 becomes Rule 2's exception.

The knowledge base printed at the end is the tree of **Figure 3.2**. And the last lines are the proof that matters: every earlier customer, tried again, still gets the right answer. No rule was reordered, no rule was edited, and none of the old answers changed. Each new rule was chosen, from the differences, to be false for the cornerstone case beside it.

3.5 From Our Engine to vcirs-core

The hand-made engine is complete, but it is deliberately small. It keeps its knowledge only in memory, it cannot ask questions, it offers only one way of building an exception, and its explanation is a list of true and false rules. The shared engine, vcirs-core, keeps the same Ripple-Down-Rules heart and adds what a real app needs. **Table 3.1** shows how the ideas of our engine appear in it.

Table 3.1 How the ideas of the hand-made engine appear in vcirs-core.

Our RDR engine (Section 3.3)	Vcirs-core
a rule with conditions and a conclusion	a **node**: a sequence of conditions with one or more conclusions
the except branch of a rule	the **children** of a node (its exceptions)
the else branch	the order of the top-level nodes and of the children
the path from Rule 0 to a rule	a **rule** of the Rule Structure: a sequence of nodes
the cornerstone case	`node.cornerstone`, the case that created the node
`differences` (section B only)	`differences(node, case)`: sections **A** and **B**
`learn(case, conclusion, conditions)`	`learn(case)` (automatic) or `refine(case, option, node, chosen)`
`classify` and its trace	`infer(facts, asker)`: asks unknown facts, explains How and Why
the tree in memory	a knowledge base saved as JSON (or SQLite)

The biggest additions are three. First, **questions**. Our engine needs all the facts in advance; vcirs-core asks for a fact only when a node needs it, so a customer answers three or four questions instead of a whole form. Second, the **six options** of knowledge building from the VCIRS revision [8], [9], summarized in **Table 3.2**. The one-call `learn(case)` chooses among them the way an RDR system would: accept the case if the answer is already right (option 0), add an exception from both sections of the difference list if a node gave the wrong answer (option 3), and add a new top-level node if no node answered at all (option 5). Third, the knowledge is **variable-centered** [3], [11]: every condition is stored with its variable, every use of a variable is counted, and the same knowledge can later be read as ordinary IF-THEN rules for forward and backward chaining. This chapter uses only the first two; the later chapters use the rest.

Table 3.2 The six options for adding knowledge in VCIRS, following the 2006 revision [8], [9].

Option	The new node is made of	It is placed
0	nothing: the existing answer is accepted	— (the check is recorded)
1	the negations of section A of the difference list	as an exception of the old node
2	section B of the difference list	as an exception of the old node
3	options 1 and 2 together	as an exception of the old node
4	all the facts of the new case	beside the old node (same parent)
5	all the facts of the new case	at the top level

One more addition matters for safety. Before adding an exception, vcirs-core can list the **affected cornerstones**: the cornerstone cases of other nodes that the new node would also catch, and whose answer would therefore change [4]. For a well-chosen exception the list is empty. When it is not, the expert sees exactly which earlier cases to look at, and can add another distinguishing fact — this is the "iterate until the expert accepts the rule and its consequences" loop of RDR, done by the machine.

3.6 The Wi-Fi Doctor

3.6.1 The first week's knowledge

The app starts from what Mas Anto taught it in his first week, six cases stored in a case file, `wifi-seed-cases.json`:

```
{
  "name": "Phone and Wi-Fi doctor",
  "description": "What Mas Anto taught the troubleshooter in his first week at the phone
counter.",
  "cases": [
    {"facts": ["connection = wifi", "wifi icon = off"], "conclusions": ["fix = turn on
Wi-Fi"], "option": -1, "author": "Anto"},
    {"facts": ["connection = wifi", "wifi icon = off", "airplane mode = on"],
"conclusions": ["fix = turn off airplane mode"], "option": -1, "author": "Anto"},
    {"facts": ["connection = wifi", "wifi icon = on", "other devices online = no"],
"conclusions": ["fix = restart the router"], "option": -1, "author": "Anto"},
    {"facts": ["connection = wifi", "wifi icon = on", "other devices online = no",
"router lights = red"], "conclusions": ["fix = call the internet provider"], "option": -
1, "author": "Anto"},
    {"facts": ["connection = mobile data", "mobile data = off"], "conclusions": ["fix =
turn on mobile data"], "option": -1, "author": "Anto"},
    {"facts": ["connection = mobile data", "mobile data = on", "quota left = no"],
"conclusions": ["fix = buy a data package"], "option": -1, "author": "Anto"}
  ]
}
```

Each case lists the facts and the right fix; `"option": -1` means "let `learn` decide", so the knowledge base is built exactly as if Mas Anto had taught the six cases one after another at the counter. The first time the app runs, it builds its knowledge base from this file and saves it as `wifi-kb.json`; after that it always loads and updates the saved file.

3.6.2 The program

Listing 3.7 is the Wi-Fi doctor in Java.

Listing 3.7 `WifiDoctor.java`: the Wi-Fi doctor, a troubleshooter that learns from each fix, on vcirs-core.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;

import vcirs.Case;
import vcirs.CaseFile;
import vcirs.Conclusion;
import vcirs.Condition;
import vcirs.DifferenceList;
import vcirs.Fact;
import vcirs.Facts;
import vcirs.InferenceResult;
import vcirs.KnowledgeBase;
import vcirs.LearnResult;
import vcirs.Node;
import vcirs.Question;
import vcirs.Report;
import vcirs.VcirsException;

/**
 * Phone and Wi-Fi doctor: a troubleshooter that learns from each fix.
 * java WifiDoctor [--echo] [--tree] [knowledge-base.json]
 * It asks about the symptoms, gives the fix the knowledge base knows, asks
 * whether it worked, and, when it did not, learns the right fix as a new
 * Ripple-Down-Rules exception. --echo repeats the answers it reads, so that
 * a session typed from a file reads like a conversation.
 */
public class WifiDoctor {
    static final String[][] QUESTIONS = {
        {"connection", "Are you using Wi-Fi or mobile data?",
            "wifi", "mobile data"},
        {"wifi icon", "Is the Wi-Fi symbol at the top of the screen on or off?",
            "on", "off"},
        {"airplane mode", "Is airplane mode (the little plane) on?", "on", "off"},
        {"other devices online",
            "Do other phones get internet on the same Wi-Fi?", "yes", "no"},
        {"router lights", "Are the router's lights normal, red or off?",
            "normal", "red", "off"},
        {"mobile data", "Is mobile data switched on?", "on", "off"},
        {"quota left", "Is there still internet quota on the SIM card?",
            "yes", "no"},
```

Listing 3.7 (Continued) *WifiDoctor.java: the Wi-Fi doctor, a troubleshooter that learns from each fix, on vcirs-core.*

```
    };

    static BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in, StandardCharsets.UTF_8));
    static boolean echo = false;
    static KnowledgeBase kb;

    /** Reads the user's answer; with echo on, prints it too. */
    static String read(String prompt) {
        System.out.print(prompt);
        String line;
        try {
            line = in.readLine();
        } catch (IOException e) {
            line = null;
        }
        String text = line == null ? "" : line.strip();
        if (echo) {
            System.out.println(text);
        }
        return text;
    }

    static Fact ask(Question q) {
        String info = kb.variables.containsKey(q.variable)
                ? kb.variables.get(q.variable).question : "";
        String text = read((info.isEmpty() ? q.variable : info) + " ("
                + String.join("/", q.choices) + ") > ");
        return text.isEmpty() ? null : new Fact(q.variable, text);  // Enter = ?
    }

    /** Load the knowledge base, or build it from the seed cases. */
    static KnowledgeBase openKb(String path) {
        if (Files.exists(Path.of(path))) {
            return KnowledgeBase.load(path);
        }
        Path parent = Path.of(path).toAbsolutePath().getParent();
        CaseFile seed = CaseFile.read(parent.resolve("wifi-seed-cases.json")
                .toString());
        KnowledgeBase k = new KnowledgeBase(seed.name, seed.description);
        for (String[] q : QUESTIONS) {
            List<String> choices = new ArrayList<>();
            for (int i = 2; i < q.length; i++) {
                choices.add(q[i]);
            }
            k.declare(q[0], q[1], choices);
```

Listing 3.7 (Continued) *WifiDoctor.java: the Wi-Fi doctor, a troubleshooter that learns from each fix, on vcirs-core.*

```
        }
        seed.build(k);
        k.save(path);
        return k;
    }

    public static void main(String[] args) {
        String path = "../data/wifi-kb.json";
        boolean tree = false;
        for (String a : args) {
            if (a.equals("--echo")) {
                echo = true;
            } else if (a.equals("--tree")) {
                tree = true;
            } else {
                path = a;
            }
        }
        kb = openKb(path);
        if (tree) {
            for (String line : Report.tree(kb)) {
                System.out.println(line);
            }
            return;
        }
        System.out.println("Phone and Wi-Fi doctor: " + kb.nodes.size()
                + " fixes learned so far.");
        System.out.println("Answer each question; press Enter if you do not "
                + "know.");
        InferenceResult result = kb.infer(Facts.none(), WifiDoctor::ask);
        Node fired = result.paths.isEmpty() ? null : result.paths.get(0).last();
        if (fired != null && !result.conclusions().isEmpty()) {
            System.out.println("Advice: " + result.conclusions().get(0).value);
            System.out.println("Why:");
            for (String line : result.how()) {
                System.out.println("  " + line);
            }
        } else {
            System.out.println("I have no advice for this problem yet.");
        }
```

Listing 3.7 (Continued) *WifiDoctor.java: the Wi-Fi doctor, a troubleshooter that learns from each fix, on vcirs-core.*

```
        List<Condition> facts = new ArrayList<>();
        for (Fact f : result.memory.facts.values()) {
            if (!f.variable.equals("fix")) {
                facts.add(new Condition(f.variable, "=", f.value));
            }
        }
        boolean worked = fired != null
                && read("Did it fix the problem? (yes/no) > ").equals("yes");
        if (worked) {
            Case c = new Case(facts, result.conclusions(), 100);
            kb.learn(c, "counter");              // recorded as a verified case
            kb.save(path);
            System.out.println("Good. The knowledge base was right; nothing "
                    + "new to learn.");
            return;
        }
        String fix = read("What fixed it in the end? (Enter if nothing did) > ");
        if (fix.isEmpty()) {
            System.out.println("Then there is nothing to learn this time.");
            return;
        }
        while (true) {
            String extra = read("Another fact that was different this time? "
                    + "(variable = value, Enter when done) > ");
            if (extra.isEmpty()) {
                break;
            }
            try {
                facts.add(Condition.parse(extra));
            } catch (VcirsException e) {
                System.out.println("Please write it as: variable = value");
            }
        }
        Case c = new Case(facts, List.of(new Conclusion("fix", fix)), 100);
        if (fired != null) {
            DifferenceList d = kb.differences(fired.id, c);
            for (String line : d.lines()) {
                System.out.println(line);
            }
            List<Condition> conds = new ArrayList<>();
            for (Condition x : d.a) {
                conds.add(x.negated());
            }
            conds.addAll(d.b);
            List<String> hit = kb.affectedCornerstones(fired.id, conds,
                    c.conclusions);
```

Listing 3.7 (Continued) *WifiDoctor.java: the Wi-Fi doctor, a troubleshooter that learns from each fix, on vcirs-core.*

```
            System.out.println("Earlier cases whose answer would change: "
                    + (hit.isEmpty() ? "none" : String.join(", ", hit)));
        }
        LearnResult learned;
        try {
            learned = kb.learn(c, "counter");
        } catch (VcirsException e) {
            System.out.println("Cannot learn: " + e.getMessage());
            return;
        }
        System.out.println("Learned: " + learned);
        kb.save(path);
        System.out.println("Saved " + path + ": " + kb.summary());
    }
}
```

Go through `main` step by step, because it is the whole life of one customer.

1. **Open the knowledge.** `openKb` loads `wifi-kb.json`, or, the first time, builds it: it declares the seven variables with the question to ask for each and the usual answers (`QUESTIONS`), then feeds the six seed cases to the engine with `seed.build(k)`, and saves the result.
2. **Consult.** `kb.infer(Facts.none(), WifiDoctor::ask)` starts with no facts at all. The engine walks the knowledge base from its top-level nodes, and whenever a node needs a fact it does not have, it calls `ask`, which prints the declared question with its usual answers and reads the reply. An empty reply means "I do not know", and the node that needed it simply does not fire. When a node fires, the engine follows its exceptions, asking only what they need.
3. **Advise and explain.** The answer is the conclusion of the last node that fired; `result.how()` explains it, fact by fact.
4. **Did it work?** If the customer says yes, the case is given to `learn`, which finds the answer already right and records it as a verified case (option 0) — the verification on the fly that RDR is known for [4].
5. **Learn.** If the advice failed (or there was none), the program asks what fixed the problem in the end, and then, one by one, for any other fact that was different this time. Here Mas Anto can bring in a variable the knowledge base has never heard of. The program shows the list of difference against the cornerstone case of the node that answered, and the affected cornerstones; then `kb.learn(c, "counter")` adds the knowledge and the file is saved.

Why does the doctor ask about the connection first? Because it walks the top-level nodes in the order they were taught, and checks each node's conditions in their order; the first seed case happened to begin with `connection`. A node whose first condition is false is abandoned at once, without asking about the rest, so a customer on mobile data is asked about the Wi-Fi symbol only if a node about mobile data needs it. And a fact, once known, is never asked again: the working memory of the consultation keeps every answer, and every node that needs it reads it from there. In session 2 below you will see the engine skip a whole node after one answer and move on.

The `--echo` option repeats each answer read from the input. It exists so that a session can be typed in a file and replayed, which is how the transcripts in this chapter were produced; at the counter you would run the program without it and type the answers. **Listing 3.8** is the same app in Python.

Listing 3.8 `wifi_doctor.py`: the Wi-Fi doctor in Python.

```
"""Phone and Wi-Fi doctor: a troubleshooter that learns from each fix.

    python wifi_doctor.py [--echo] [--tree] [knowledge-base.json]

It asks about the symptoms, gives the fix the knowledge base knows, asks
whether it worked, and, when it did not, learns the right fix as a new
Ripple-Down-Rules exception. --echo repeats the answers it reads, so that a
session typed from a file reads like a conversation. --tree shows the rules.
"""

import os
import pathlib
import sys

# Tell Python where the vcirs-core engine is (two folders up, then across).
sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import (BOTH, Case, Condition, Conclusion, KnowledgeBase,
                   VcirsError, casefile, report)

QUESTIONS = [
    ("connection", "Are you using Wi-Fi or mobile data?",
     ["wifi", "mobile data"]),
    ("wifi icon", "Is the Wi-Fi symbol at the top of the screen on or off?",
     ["on", "off"]),
    ("airplane mode", "Is airplane mode (the little plane) on?", ["on", "off"]),
    ("other devices online",
     "Do other phones get internet on the same Wi-Fi?", ["yes", "no"]),
    ("router lights", "Are the router's lights normal, red or off?",
     ["normal", "red", "off"]),
    ("mobile data", "Is mobile data switched on?", ["on", "off"]),
    ("quota left", "Is there still internet quota on the SIM card?",
     ["yes", "no"]),
]


class Console:
    """Reads the user's answers; with echo on, prints them too."""

    def __init__(self, echo):
        self.echo = echo
```

Listing 3.8 (Continued) *wifi_doctor.py: the Wi-Fi doctor in Python.*

```
    def read(self, prompt):
        print(prompt, end="")
        line = sys.stdin.readline()
        text = line.strip()
        if self.echo:
            print(text)
        return text

    def ask(self, question):
        var = question.variable
        info = KB.variables[var].question if var in KB.variables else ""
        choices = "/".join(question.choices)
        text = self.read((info or var) + " (" + choices + ") > ")
        return text or None                    # Enter = "I do not know"


def open_kb(path):
    """Load the knowledge base, or build it from the seed cases."""
    if os.path.exists(path):
        return KnowledgeBase.load(path)
    seed = os.path.join(os.path.dirname(path), "wifi-seed-cases.json")
    name, description, steps = casefile.read_cases(seed)
    kb = KnowledgeBase(name, description)
    for var, question, choices in QUESTIONS:
        kb.declare(var, question, choices)
    casefile.build(kb, steps)
    kb.save(path)
    return kb


def main(args):
    global KB
    echo = "--echo" in args
    files = [a for a in args if not a.startswith("--")]
    path = files[0] if files else "../data/wifi-kb.json"
    KB = kb = open_kb(path)
    if "--tree" in args:
        for line in report.tree(kb):
            print(line)
        return
    con = Console(echo)
    print("Phone and Wi-Fi doctor: " + str(len(kb.nodes))
          + " fixes learned so far.")
    print("Answer each question; press Enter if you do not know.")
    result = kb.infer({}, con.ask)
    fired = result.paths[0].last() if result.paths else None
```

Listing 3.8 (Continued) *wifi_doctor.py: the Wi-Fi doctor in Python.*

```
    if fired is not None and result.conclusions():
        print("Advice: " + result.conclusions()[0].value)
        print("Why:")
        for line in result.how():
            print("  " + line)
    else:
        print("I have no advice for this problem yet.")

    facts = [Condition(f.variable, "=", f.value)
             for f in result.memory.facts.values() if f.variable != "fix"]
    worked = fired is not None and con.read(
        "Did it fix the problem? (yes/no) > ") == "yes"
    if worked:
        case = Case(facts, list(result.conclusions()))
        kb.learn(case, "counter")               # recorded as a verified case
        kb.save(path)
        print("Good. The knowledge base was right; nothing new to learn.")
        return
    fix = con.read("What fixed it in the end? (Enter if nothing did) > ")
    if not fix:
        print("Then there is nothing to learn this time.")
        return
    while True:
        extra = con.read("Another fact that was different this time? "
                         "(variable = value, Enter when done) > ")
        if not extra:
            break
        try:
            facts.append(Condition.parse(extra))
        except VcirsError:
            print("Please write it as: variable = value")
    case = Case(facts, [Conclusion("fix", fix)])
    if fired is not None:
        d = kb.differences(fired.id, case)
        for line in d.lines():
            print(line)
        conds = [c.negated() for c in d.a] + d.b
        hit = kb.affected_cornerstones(fired.id, conds, case.conclusions)
        print("Earlier cases whose answer would change: "
              + (", ".join(hit) if hit else "none"))
    try:
        learned = kb.learn(case, "counter")
    except VcirsError as e:
        print("Cannot learn: " + str(e))
        return
    print("Learned: " + str(learned))
```

Listing 3.8 (Continued) *wifi_doctor.py: the Wi-Fi doctor in Python.*

```
    kb.save(path)
    print("Saved " + path + ": " + kb.summary())


if __name__ == "__main__":
    main(sys.argv[1:])
```

The Python asker is the method `ask` of a small `Console` object; any function that takes a question and returns an answer (or `None`) will do, which is how the engine stays independent of where the answers come from — a console today, a web page in Chapter 10, a chat window in Chapter 13.

3.7 Four Customers at the Counter

The chapter's `data` folder holds four conversations, `session-1.txt` to `session-4.txt`, one answer per line. Starting from a fresh knowledge base, run them in order from the `Chapter03/java` folder:

```
javac -d out -sourcepath ../../vcirs-core/java/src *.java
java -cp out WifiDoctor --echo < ../data/session-1.txt
```

and so on for sessions 2, 3 and 4; or, from `Chapter03/python`, `python wifi_doctor.py --echo < ../data/session-1.txt`. (In Windows PowerShell, which has no `<`, type `Get-Content ..\data\session-1.txt | java -cp out WifiDoctor --echo`.) The two languages print the same transcripts and write the same `wifi-kb.json`, byte for byte.

Session 1: the advice works. A customer on Wi-Fi whose Wi-Fi symbol is off.

```
Phone and Wi-Fi doctor: 6 fixes learned so far.
Answer each question; press Enter if you do not know.
Are you using Wi-Fi or mobile data? (wifi/mobile data) > wifi
Is the Wi-Fi symbol at the top of the screen on or off? (on/off) > off
Is airplane mode (the little plane) on? (on/off) > on
Advice: turn off airplane mode
Why:
  turn off airplane mode#1 fired because:
    connection = wifi  (answered by the user, CF 100)
    wifi icon = off  (answered by the user, CF 100)
    airplane mode = on  (answered by the user, CF 100)
    so fix = turn off airplane mode  [rule CF 100, conclusion CF 100]
Did it fix the problem? (yes/no) > yes
Good. The knowledge base was right; nothing new to learn.
```

The engine asked three questions and no more. The first top-level node, *turn on Wi-Fi*, needs `connection = wifi` and `wifi icon = off`; both hold, so it fires, and the engine goes on to its exception, which needs `airplane mode = on` — asked, confirmed, fired. The answer comes from the exception, and the explanation lists the three facts that made it. The customer confirms, and the case is recorded as verified: nothing new to learn, but the history of the knowledge base now shows that this knowledge was checked on a real customer.

Session 2: the advice fails, and the doctor learns. Wi-Fi on, the other phones offline, and the router's lights *off*.

```
Phone and Wi-Fi doctor: 6 fixes learned so far.
Answer each question; press Enter if you do not know.
Are you using Wi-Fi or mobile data? (wifi/mobile data) > wifi
Is the Wi-Fi symbol at the top of the screen on or off? (on/off) > on
Do other phones get internet on the same Wi-Fi? (yes/no) > no
Are the router's lights normal, red or off? (normal/red/off) > off
Advice: restart the router
Why:
  restart the router#1 fired because:
    connection = wifi  (answered by the user, CF 100)
    wifi icon = on  (answered by the user, CF 100)
    other devices online = no  (answered by the user, CF 100)
    so fix = restart the router  [rule CF 100, conclusion CF 100]
Did it fix the problem? (yes/no) > no
What fixed it in the end? (Enter if nothing did) > switch the router on
Another fact that was different this time? (variable = value, Enter when done) >
Differences with the cornerstone case of restart the router#1:
  A (only in the old case): (none)
  B (only in the new case): router lights = off
Earlier cases whose answer would change: none
Learned: refined: switch the router on#1 added as an exception of restart the router#1
Saved ../data/wifi-kb.json: Phone and Wi-Fi doctor: 8 variables, 7 nodes, 7 rules
```

This time the first node fails at its second condition (the Wi-Fi symbol is on), so the engine moves to the next top-level node, *restart the router*, and asks the one fact it lacks. It fires; its exception, *call the internet provider*, needs a red light, so the engine asks about the lights — off — and the exception does not fire. The advice is *restart the router*, and the explanation shows why. But the router was not broken; it was switched off at the wall. Mas Anto types the real fix. He adds no other fact, because the one that matters, the lights being off, was already asked. The difference list against the cornerstone of *restart the router* shows exactly that one fact in section B; no earlier case would change its answer; and the doctor adds `switch the router on#1` as an exception of `restart the router#1` — option 3 of **Table 3.2**, in the language of VCIRS.

Session 3: no advice at all, and a new variable. A customer on mobile data with the data switched on and quota left, but a weak signal.

```
Phone and Wi-Fi doctor: 7 fixes learned so far.
Answer each question; press Enter if you do not know.
Are you using Wi-Fi or mobile data? (wifi/mobile data) > mobile data
Is mobile data switched on? (on/off) > on
Is there still internet quota on the SIM card? (yes/no) > yes
I have no advice for this problem yet.
What fixed it in the end? (Enter if nothing did) > find a stronger signal
Another fact that was different this time? (variable = value, Enter when done) > signal
bars = 1
Another fact that was different this time? (variable = value, Enter when done) >
Learned: added: find a stronger signal#1 added at the top level
Saved ../data/wifi-kb.json: Phone and Wi-Fi doctor: 9 variables, 8 nodes, 8 rules
```

No node fired: the two mobile-data nodes need mobile data off, or no quota. The doctor says so honestly and asks what fixed it. Mas Anto moved the customer to the doorway, where the signal is better, and he adds a fact that no question has ever asked: `signal bars = 1`. The engine creates a new variable for it and adds the case as a new top-level node (option 5). From now on, the doctor knows about signal bars.

Session 4: the same problem again. The customer of session 2 comes back a week later, or one of her neighbors does.

```
Phone and Wi-Fi doctor: 8 fixes learned so far.
Answer each question; press Enter if you do not know.
Are you using Wi-Fi or mobile data? (wifi/mobile data) > wifi
Is the Wi-Fi symbol at the top of the screen on or off? (on/off) > on
Do other phones get internet on the same Wi-Fi? (yes/no) > no
Are the router's lights normal, red or off? (normal/red/off) > off
Advice: switch the router on
Why:
  switch the router on#1 fired because:
    connection = wifi  (answered by the user, CF 100)
    wifi icon = on  (answered by the user, CF 100)
    other devices online = no  (answered by the user, CF 100)
    router lights = off  (answered by the user, CF 100)
    so fix = switch the router on  [rule CF 100, conclusion CF 100]
Did it fix the problem? (yes/no) > yes
Good. The knowledge base was right; nothing new to learn.
```

The doctor now gives the right answer at once, and it can explain it: the four facts, including the one it learned in session 2. This is the whole promise of the chapter in one transcript. Nobody wrote a line of code between session 2 and session 4. The knowledge came from the person who knew, in the context of a real case, and it went exactly where it belongs.

3.7.1 The knowledge base after four customers

`--tree` prints the knowledge base as a tree of nodes and exceptions:

```
(root)
+- turn on Wi-Fi#1: connection = wifi, wifi icon = off -> fix = turn on Wi-Fi
|  +- turn off airplane mode#1: airplane mode = on -> fix = turn off airplane mode
+- restart the router#1: connection = wifi, wifi icon = on, other devices online = no ->
fix = restart the router
|  +- call the internet provider#1: router lights = red -> fix = call the internet provider
|  +- switch the router on#1: router lights = off -> fix = switch the router on
+- turn on mobile data#1: connection = mobile data, mobile data = off -> fix = turn on
mobile data
+- buy a data package#1: connection = mobile data, mobile data = on, quota left = no -> fix
= buy a data package
+- find a stronger signal#1: connection = mobile data, mobile data = on, quota left = yes,
signal bars = 1 -> fix = find a stronger signal
```

Compare it with the binary tree of **Figure 3.2**. The vcirs-core tree shows each node with *all* its conditions and puts exceptions directly under their node; the else branches of our hand-made engine have become simply the order in which nodes are tried. The node learned in session 2 sits next to the node from the first week — two exceptions of *restart the router*, one for a red light and one for no light at all.

The saved file, `wifi-kb.json`, is plain text that anyone can read. Here is the node learned in session 2, as the file stores it:

```
{
  "id": "switch the router on#1",
  "parent": "restart the router#1",
  "cf": 100,
  "conditions": [
    {
      "variable": "router lights",
      "operator": "=",
      "value": "off",
      "cf": 100,
      "credit": 1
    }
  ],
  "conclusions": [
    {
      "variable": "fix",
      "value": "switch the router on"
    }
  ],
  "cornerstone": {
    "facts": [
      {
        "variable": "connection",
        "operator": "=",
        "value": "wifi",
        "cf": 100
      },
      {
        "variable": "wifi icon",
        "operator": "=",
        "value": "on",
        "cf": 100
      },
      {
        "variable": "other devices online",
        "operator": "=",
        "value": "no",
        "cf": 100
      },
      {
        "variable": "router lights",
        "operator": "=",
        "value": "off",
        "cf": 100
      }
    ],
    "conclusions": [
      {
        "variable": "fix",
        "value": "switch the router on"
      }
    ],
    "cf": 100
  },
  "stopped": false,
  "generated": false,
  "author": "counter",
  "seq": 7
}
```


It records its parent, its one condition with the variable, operator, value, certainty factor and Credit, its conclusion, the whole cornerstone case (the four facts of the customer in session 2), and who taught it. The file also keeps a `history` of every change; its last entries say that session 2 added this node with option 3, session 3 added a node with option 5, and session 4 accepted this node's answer (option 0).

3.7.2 Fajar's Friday

On Friday Fajar minds the counter alone, with the knowledge base of the four sessions. A young man sits down: his phone is on Wi-Fi, the Wi-Fi symbol is on, and the other phones in his boarding house have internet — only his does not. The doctor asks three questions. The first node fails on the Wi-Fi symbol; the second, *restart the router*, fails on the other devices, which are online; the mobile-data nodes and the signal node fail at once on the connection, which is already known. No node fires, and the doctor says it has no advice yet.

Fajar remembers that the boarding house changed its Wi-Fi password this week. He has the phone forget the network and join it again with the new password, and the internet comes back. He types the fix, *forget the network and enter the new password*, and one more fact that no question asks about: `wifi password changed = yes`. As in session 3, nothing had answered, so `learn` adds the case as a new top-level node, with all four facts as its conditions.

On Monday Mas Anto prints the tree with `--tree` and finds the new node at the bottom, with its cornerstone case and the history entry that records when it was added. He did not have to be at the counter to gain the knowledge, and Fajar did not have to know anything about rules to give it. If Mas Anto thinks the node is too narrow — perhaps the other devices do not matter when the password has changed — he can teach a better node the next time such a customer comes, and stop this one.

3.8 Making It Better

Numbers, not only words. The node learned in session 3 says `signal bars = 1`. A customer with *zero* bars has the same problem but will not match. The `learn` call builds conditions from the facts as they were given, which is the safe default. When the expert knows better, `refine` accepts the conditions she chooses, and a condition may use any operator of the engine, such as `signal bars <= 1`. Exercise 3.6 does exactly this.

Questions in a better order. The doctor asks questions in the order of its nodes. VCIRS can do better: the usage rates of Chapter 7 rank the variables by how much they are used, and the engine's `questionOrder()` puts the most useful questions first. With a few hundred nodes, the difference between asking the right question first and the wrong one is several questions per customer.

Stopping bad advice. Suppose Mas Anto decides that advising customers to call the provider is useless because the provider never answers, and he would rather advise a visit to the provider's office. RDR never deletes a node, but it can **stop** it: `kb.stop(nodeId)` keeps the node, its cornerstone and its exceptions, but its conclusion is no longer given. A new node with the better advice can then be taught as usual.

Reviewing the affected cornerstones. In our sessions the list of affected cornerstones was always empty. When it is not, the app should show those cases and ask whether their answers should really change, and let the expert add another fact if not. This is a few more lines of the same kind as the difference list, and Exercise 3.8 asks for them.

More than one right answer. If a customer has both an empty quota and a weak signal, the doctor gives the first fix it finds. Calling `kb.infer(facts, asker, true)` (Python: `multiple=True`) makes the engine continue after the first answer and follow every branch that applies, returning one path per conclusion. For a counter, the usual practice is to try the first fix,

and come back to the doctor if the problem remains, which the doctor then learns from. For a health advisor it is different, and Chapter 5 uses the multiple mode.

> **PITFALL.** A knowledge base taught by one person reflects that person's habits, good and bad. If Mas Anto always restarts the router first because it is quick, the doctor will learn to advise it even when a question about the lights would find the cause faster. Review the tree now and then, and stop nodes whose advice has become bad practice.

Two languages, one knowledge base. Because both versions of the engine read and write the same JSON, Fajar can run the Python version on the counter's old laptop while Mas Anto runs the Java version at home, and they can copy `wifi-kb.json` between them. The file is small; one line per fact.

Tests. `RdrTest.java` and `test_rdr.py` check the hand-made engine (an exception fires, an old case keeps its answer, the differences are right) and the seed knowledge base (six nodes, the provider node under the router node, the right answer for a red light, and that `learn` refines when it should). They end with:

```
all tests passed
```

3.9 Using It for Real

At a phone counter. Run the doctor on the counter's computer and answer its questions as you talk to the customer. When its advice fails, teach it before the customer leaves, while the case is fresh; it takes less than a minute. After a month, the knowledge base holds your experience of that month, and your assistant gives your advice on Fridays. Keep a copy of `wifi-kb.json` each week: it is a small file, and it is your knowledge.

How many cases to start with? Fewer than you think. The six seed cases of this chapter took Mas Anto ten minutes to write, and they already answer the most common complaints. Write the three to ten situations you see every week as seed cases, one per line of the case file, and let everything else be taught at the counter. Resist the temptation to write every rule you can imagine in advance; rules written without a real case have no cornerstone that was ever checked, and they bring back the problems of the flat list. After a few weeks, print the tree with `--tree` and read it with a colleague: a node that nobody recognizes, or two nodes that say the same thing in different words, is worth a conversation.

Is the doctor getting smarter? The history in `wifi-kb.json` answers this with no extra effort. Every successful consultation is recorded as option 0, every new exception as option 3, and every new top-level node as option 5. Count the three each week. In the first weeks, options 3 and 5 will be frequent: the doctor is still learning a lot. Over time the share of option 0 should grow, because more and more customers arrive with problems it already knows. If one week option 5 suddenly jumps, there is a new kind of problem — a phone system update, a network outage in the area — and that is the right moment to read the new nodes together. These numbers are not a perfect measure, because a customer who does not come back never says whether the advice worked, but they are honest and cheap, and they point the right way.

Other troubleshooters. Nothing in `WifiDoctor` is about Wi-Fi except the list of questions and the seed file. A laptop service shop, a motorcycle workshop, a school's computer helper, a water-pump repairman — each can write their own questions and a few seed cases and start teaching. Chapter 8 builds a motorcycle fault finder with the engine's forward and backward chaining; Chapter 14 builds a tool that lets anyone create such a troubleshooter without programming.

Many teachers. When several people teach the same knowledge base, each node records who taught it and the history records every change, so a disagreement can be traced to a case and a person. Chapter 11 builds on this for a clinic team.

Safe advice. A troubleshooter should only advise what the customer can do safely. Do not teach it fixes that mean opening a phone, charging a swollen battery, or climbing to a router on a pole; teach it to say "bring it to a service center" instead. A swollen or hot battery is a fire risk, and the right advice is to stop using the phone.

Only the facts you need. The cases stored in the knowledge base are technical facts, such as the state of the Wi-Fi symbol or the router's lights. There is no reason to store a customer's name, number or messages in them, and good reason not to: the knowledge base is meant to be shared, and it should hold knowledge about problems, not about people.

3.10 Common Mistakes, and How to Avoid Them

Putting a new rule at the front, or at the back, of a flat list. Either may change answers for customers the rule was not meant for, as **Listing 3.1** showed. *Avoid it:* add knowledge as an exception in the context where the wrong answer came from.

Choosing too many differences. If Mas Anto had chosen *all* the differences for customer 1 (`wifi icon = off` and `airplane mode = off`), Rule 1 would not fire for later customers who did not mention airplane mode. *Avoid it:* choose the facts that *explain* the different answer, not everything that happened to be different.

Choosing too few. A condition that is also true for the cornerstone case cannot tell the cases apart; the engine refuses to learn when the difference list is empty. *Avoid it:* when the engine says it cannot tell the case apart, add the fact that really was different.

Editing or deleting old rules. It breaks the guarantee that old cases keep their answers. *Avoid it:* refine with exceptions; to retire advice, stop the node.

Teaching a fix that was not verified. If the "fix" did not really solve the problem, the knowledge base learns a mistake. *Avoid it:* teach only after the customer's problem is actually solved.

Teaching in a hurry. When the queue is long, the temptation is to type the fix without thinking about which facts were really different. *Avoid it:* if there is no time, note the case on paper and teach it when the counter is quiet; knowledge taught an hour later is just as useful.

Answering a question you are not sure about. A wrong fact makes the wrong node fire. *Avoid it:* press Enter for "I do not know"; the engine treats an unknown fact as not matching.

Different words for the same thing. `wifi`, `Wi-Fi` and `WiFi` are three different values to the engine. *Avoid it:* use the usual answers the question offers, which the app shows in parentheses; the engine collects them from the knowledge base [3].

Moving the knowledge base file. The app looks for `../data/wifi-kb.json` and the seed file beside it. *Avoid it:* keep the folder layout, or give the path as an argument.

Forgetting that the tree is built from cases. A beautiful tree typed in by hand is a flat rule base in disguise, with all its problems. *Avoid it:* let the knowledge grow from real cases.

3.11 Chapter Summary

- A **flat list** of rules, tried from the top, lets a new rule change the answers of cases it was not meant for; its order becomes hidden, fragile knowledge (**Listing 3.1**, **Listing 3.2**).
- **Ripple Down Rules** store knowledge as **exceptions** in the context where a mistake happened. Every rule keeps its **cornerstone case**; a new rule is built from the **differences** between the new case and that cornerstone, so it leaves the old case's answer unchanged (**Figure 3.1**).
- In single-classification RDR the knowledge base is a binary tree of **except** (tried when true) and **else** (tried when false) branches; the answer is the conclusion of the last true rule, and a new rule is hung where the search stopped (**Figure 3.2**).
- The whole engine fits in a hundred lines (**Listing 3.3**, **Listing 3.4**); five customers taught it four rules without changing any earlier answer (**Listing 3.5**, **Listing 3.6**).
- **vcirs-core** keeps the same heart and adds questions asked only when needed, the **list of difference** with sections A and B, the **six options** of VCIRS knowledge building, **affected cornerstones**, explanations, and a JSON knowledge base shared by Java and Python (**Table 3.1**, **Table 3.2**).
- The **Wi-Fi doctor** (**Listing 3.7**, **Listing 3.8**) asked three to four questions per customer, explained every piece of advice, recorded a verified case, learned an exception when its advice failed, learned a new top-level node with a new variable when it had no advice, and gave the right answer at once to the next customer with the same problem.

3.12 Exercises

3.1 By hand, using the tree of **Figure 3.2**, find the answer and list the rules tried for: (a) Wi-Fi off, airplane mode off; (b) Wi-Fi on, other devices online yes; (c) Wi-Fi on, other devices online no, router lights red, airplane mode on.

3.2 In **Listing 3.1**, is there *any* order of the three rules that gives all four customers the right answer? Try all six orders by hand or with a small change to the program, and explain the result.

3.3 Continue **Listing 3.6** with a sixth customer: Wi-Fi off, airplane mode off, *battery saver on*, and the right fix is *turn off battery saver* (some phones switch Wi-Fi off to save power). Which rule answers first, what are the differences, and where does the new rule go? Draw the new tree.

3.4 In the hand-made engine, why must `learn` classify the case again instead of using the answer the program already printed? What could go wrong if the tree changed in between?

3.5 For session 2 of Section 3.7, write out the list of difference by hand: the cornerstone case of `restart the router#1` and the facts of the new case. Why is section A empty?

3.6 Session 3 taught the node `signal bars = 1`. Using `refine` with chosen conditions, teach instead a node with the condition `signal bars <= 1`, starting from the seed knowledge base, and show that it answers for 0 and 1 bar but not for 3.

3.7 Explain in your own words the difference between option 2 and option 5 of **Table 3.2**, using session 2 and session 3 as examples. Why did `learn` choose option 3 in session 2 and not option 2?

3.8 Describe how you would extend `WifiDoctor` so that, when the list of affected cornerstones is not empty, it shows each affected case and lets Mas Anto add another fact before the knowledge base changes. Which vcirs-core calls would you use?

3.9 In Fajar's Friday (Section 3.7.2) the new node has four conditions. Which of them would you keep if you could choose, and why? Using the order in which the doctor walks the top-level nodes, explain which questions a later customer with the same problem would be asked, and whether `wifi password changed` would ever be asked.

3.13 Solutions to the Exercises

Solution 3.1. (a) Rule 0 true; Rule 1 (*wifi icon = off*) true; Rule 3 (*airplane mode = on*) false, no else branch: stop. Answer *turn on Wi-Fi* (Rule 1). (b) Rule 0; Rule 1 false → else; Rule 2 (*other devices online = no*) false, no else branch: stop. The last true rule is Rule 0: *no advice yet*. (c) Rule 0; Rule 1 false → else; Rule 2 true → except; Rule 4 (*router lights = red*) true, no except branch: stop. Answer *call the internet provider*; airplane mode is never looked at.

Solution 3.2. Call the rules W (*wifi icon = off*), R (*other devices online = no*) and A (*airplane mode = on*). Customer 3 matches W and A, so A must come before W; customer 4 matches R and A, so R must come before A; customers 1 and 2 each match one rule only. Of the six orders, exactly one satisfies both constraints: R, A, W — and it does give all four customers the right answer. But the lesson is in what happens next. A fifth customer with Wi-Fi off, airplane mode on and the other phones offline, whose fix is to turn off airplane mode, matches R first and gets the wrong answer; fixing that requires A before R, which contradicts customer 4. With every new customer the search for an order starts again, and sometimes no order exists. Exceptions avoid the search altogether: in the RDR tree, customer 5's rule lives in the context of Wi-Fi being off.

Solution 3.3. Rule 1 answers *turn on Wi-Fi* (Rule 3 is false, since airplane mode is off). The differences from Rule 1's cornerstone case (Wi-Fi off, airplane off) are just `battery saver = on`. The search stopped at Rule 3, which was false, so the new Rule 5, *battery saver = on → turn off battery saver*, is hung on Rule 3's **else** branch: it is still an exception of Rule 1, tried after the airplane exception. The tree becomes: Rule 0 → except Rule 1 → except Rule 3, else Rule 5; Rule 1 → else Rule 2 → except Rule 4.

Solution 3.4. The place to hang the new rule is where the search for *this* case stops, which depends on the whole tree. Classifying again guarantees that the place is computed on the current tree. If another rule had been added in between (for example by a colleague on the same knowledge base), the old answer's stopping point could be wrong, and the new rule could be hung where this case never arrives — so it would never fire.

Solution 3.5. The cornerstone of `restart the router#1` is the seed case: `connection = wifi`, `wifi icon = on`, `other devices online = no`. The new case has those three facts and `router lights = off`. Section A (only in the old case) is empty because every fact of the old case is also in the new case; section B holds `router lights = off`. So options 1 and 3 give the same node as option 2 here.

Solution 3.6. The solution programs are `SignalRule.java` and `signal_rule.py`. The Python version is:

```
"""Exercise 3.6: teach a rule with a numeric condition, chosen by hand."""

import pathlib
import sys

sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import TOP_LEVEL, Case, Condition, KnowledgeBase, casefile

name, description, steps = casefile.read_cases("../data/wifi-seed-cases.json")
kb = casefile.build(KnowledgeBase(name, description), steps)
case = Case.of(["connection = mobile data", "mobile data = on",
                "quota left = yes", "signal bars = 1"],
               ["fix = find a stronger signal"])
# The expert keeps the case's facts but widens "signal bars = 1" to "<= 1".
chosen = [Condition.parse("connection = mobile data"),
          Condition.parse("mobile data = on"),
          Condition.parse("signal bars <= 1")]
node = kb.refine(case, TOP_LEVEL, chosen=chosen, author="Anto")
print("New node: " + str(node))
for bars in ["0", "1", "3"]:
    answer = kb.infer({"connection": "mobile data", "mobile data": "on",
                       "quota left": "yes", "signal bars": bars})
    print("signal bars = " + bars + ": " + answer.what()[0])
```

It prints:

```
New node: find a stronger signal#1: connection = mobile data, mobile data = on, signal bars
<= 1 -> fix = find a stronger signal
signal bars = 0: Conclusion: fix = find a stronger signal  (rule find a stronger signal#1,
CF 100)
signal bars = 1: Conclusion: fix = find a stronger signal  (rule find a stronger signal#1,
CF 100)
signal bars = 3: No node matched the case.
```

With three bars the condition `signal bars <= 1` is false, and no node matches — as it should be, since a good signal is not this problem. The engine compares `signal bars` as numbers because both sides are numbers.

Solution 3.7. Option 2 builds the new node from section B of the list of difference and puts it *under* the old node, as an exception; option 5 builds it from *all* the facts of the case and puts it at the *top level*, with no parent. In session 2 a node (*restart the router*) gave a wrong answer, so the new knowledge is an exception of it; `learn` uses option 3, the union of the negated section A and section B, because that is the most specific safe choice — and here section A was empty, so the result is the same as option 2. In session 3 no node answered at all, so there was nothing to be an exception of, and `learn` used option 5.

Solution 3.8. After computing the conditions (`conds`), loop: call `kb.affectedCornerstones(fired.id, conds, c.conclusions)`; if the list is empty, call `learn` (or `refine(c, BOTH, fired.id, conds, "counter")`). If not, print each affected node with `kb.node(id).cornerstone` and its conclusion, and ask whether its answer should change. If Mas Anto says no, ask for another fact, add it to the case and to `conds`, and repeat. When he accepts, call `refine` with the chosen conditions. This is the "iterate until the expert accepts the rule and its consequences" procedure of RDR [4].

Solution 3.9. The condition that explains the fix is `wifi password changed = yes`, together with `connection = wifi`; `wifi icon = on` and `other devices online = yes` only describe this customer. A node with just those two conditions would also catch a customer whose other devices are offline because the router is down, but the router nodes are tried first, so they would

answer such a customer before this node is reached. With the four conditions as learned, a later customer is asked the connection, the Wi-Fi symbol and the other devices, exactly as the young man was; when those match, the engine reaches the new node and asks the new question, `wifi password changed`, whose variable has no declared question, so the doctor asks it by its name. The question is never asked of a customer on mobile data.

References

[1] ANTARA News, "Survei APJII: Penetrasi internet di Indonesia 2026 capai 81,7 persen," May 2026. [Online]. Available: https://www.antaranews.com/berita/5576225/survei-apjii-penetrasi-internet-di-indonesia-2026-capai-817-persen

[2] P. Compton, G. Edwards, B. Kang, L. Lazarus, R. Malor, T. Menzies, P. Preston, A. Srinivasan, and C. Sammut, "Ripple down rules: possibilities and limitations," in *Proc. 6th AAAI Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada, 1991, pp. 6.1–6.20.

[3] M. M. I. Subakti, "A Variable-Centered Intelligent Rule System," M.Sc.Eng. thesis, Dept. of Computer Science and Information Engineering, Taipei, Taiwan, Jun. 2005.

[4] P. Compton and B. H. Kang, *Ripple-Down Rules: The Alternative to Machine Learning*. Boca Raton, FL, USA: CRC Press, 2021.

[5] P. Compton, R. Horn, R. Quinlan, and L. Lazarus, "Maintaining an expert system," in *Applications of Expert Systems*, J. R. Quinlan, Ed. London, U.K.: Addison-Wesley, 1989, pp. 366–385.

[6] G. Edwards, P. Compton, R. Malor, A. Srinivasan, and L. Lazarus, "PEIRS: a pathologist-maintained expert system for the interpretation of chemical pathology reports," *Pathology*, vol. 25, no. 1, pp. 27–34, 1993.

[7] D. Richards, "Two decades of ripple down rules research," *The Knowledge Engineering Review*, vol. 24, no. 2, pp. 159–184, 2009.

[8] I. Subakti, "Some revisions in VCIRS and cases reconstructing perspectives," in *Proc. Information and Communication Technology Seminar (ICTS 2006)*, Surabaya, Indonesia, 2006.

[9] I. Subakti, *Variable-Centered Intelligent Rule System*, Indonesian ed. (resume). Surabaya, Indonesia: Dept. of Informatics, Institut Teknologi Sepuluh Nopember, 2006.

[10] B. H. Kang, P. Compton, and P. Preston, "Multiple classification ripple down rules: evaluation and possibilities," in *Proc. 9th AAAI-Sponsored Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada, 1995, pp. 17.1–17.20.

[11] I. Subakti, "A variable-centered intelligent rule system," in *Proc. 1st Annual Int. Conf. Information and Communication Technology Seminar (ICTS 2005)*, vol. 1, no. 1, Surabaya, Indonesia, Aug. 2005, pp. 167–174.

Chapter 4

Study-Program Advisor: Thinking in Variables

By the end of this chapter you will be able to take the advice of an experienced person, written as ordinary sentences, and turn it into **variables**, **values**, **conditions**, **nodes** and **rules**; explain what the *Variable-Centered Rule Structure* is and why keeping knowledge around variables makes it smaller, easier to check and easier to grow; build that structure by hand, in Java and in Python, as three small tables — a Variable table, a Node Structure and a Rule Structure; and run the **study-program advisor**, an app on the shared vcirs-core engine that helps a school leaver think about which study program suits her, asks only the questions it needs, explains every suggestion, shows the programs it met on the way, and learns from the school counselor when she disagrees. You will also read the structure the engine keeps and see, in its tables, which variables and nodes matter most — the first step toward the rule generation of Chapter 6.

4.1 Laras and Bu Wati

Every afternoon at half past one, Pak Joko parks his siomay cart at the gate of a state senior high school on the edge of town. By two o'clock the students are out, and by half past two the steamer is nearly empty. His daughter Laras studies at the same school, in class XII, and this year the question at the family's dinner mat is always the same: *what should Laras study after school, and where?*

The question is harder than it sounds, and the numbers show why. Indonesia's state universities admit new students mainly through two national selections run by SNPMB. The first, **SNBP**, is based on school achievement: only the best students of each school are *eligible* — the top 40 percent of a school with A accreditation, 25 percent with B, 5 percent with C — and each eligible student chooses at most two programs; in 2026 the results were announced on 31 March [1]. The second, **SNBT**, is based on a computer-based written test, the UTBK. In 2026 registration ran from 25 March to 7 April, the test itself from 21 to 30 April, and the results came out on 25 May [2], [3]. A student may choose up to four programs, with rules about how many may be academic (S1) programs and how many vocational ones, and she may sit the UTBK only once in 2026 [3]. **Figure 4.1** shows what happened. Of 871,496 people who registered for SNBT 2026, 256,369 were accepted — 29.42 percent — into 286,864 available seats; 86,118 of those accepted, a third, were KIP Kuliah applicants, whose study costs are supported by the government [4].

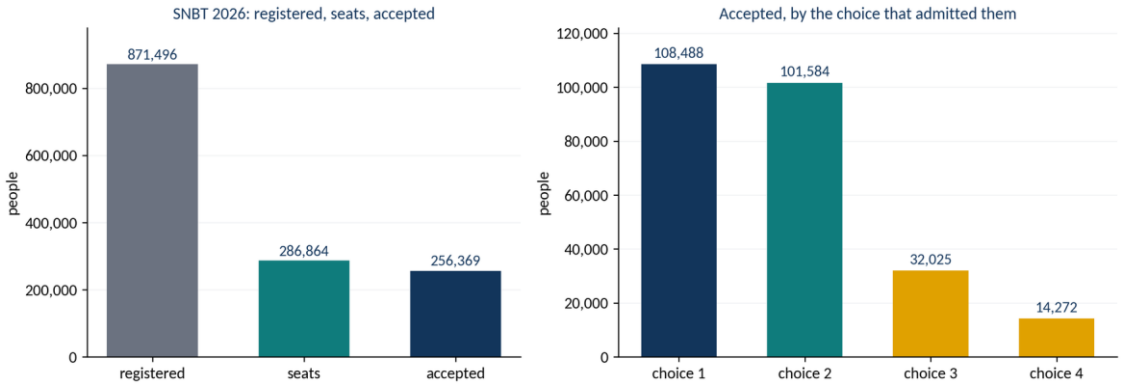

Figure 4.1 SNBT 2026 in numbers: fewer than three in ten registrants were accepted, and most of those who were accepted got in through their first or second choice. Data from the SNPMB press release of 25 May 2026 [4].

The right half of the figure holds the lesson that matters most for a family like Pak Joko's. Of the people accepted, 108,488 got in through their first choice and 101,584 through their second; only 46,297 through their third or fourth [4]. The choices at the top of the list do almost all the work. Choosing them well — programs that fit the student, that she can reach, and that the family can support — is worth far more than any trick on the day of the test.

This is where Bu Wati comes in. She has been the school's guidance counselor (*guru BK*) for nineteen years, and she has sat with hundreds of students like Laras. She does not know the future, but she knows patterns. *A girl who loves living things and is not afraid of blood often does well in nursing, and if her science scores are strong she should look at medicine too. A boy who loves computers but whose family needs him to earn soon should look at a three-year diploma (D3) in informatics, not only at the four-year degree.* She says these things one student at a time, in her small room next to the library, and each year the queue at her door is longer than the time she has.

Bu Wati would like two things. She would like every student to hear her advice, even the shy ones who never knock on her door. And she would like to see her own knowledge laid out in front of her, so that she can check it, correct it and pass it on to the young counselor who will one day take her place. This chapter builds the program that does both. On the way, it teaches the idea that gives VCIRS its name: **thinking in variables**.

> **PITFALL.** The advisor in this chapter gives a counselor's suggestions, not decisions. The official requirements of each program — minimum scores, subject prerequisites, whether a program accepts applicants with color blindness, its capacity (*daya tampung*) — are published by SNPMB and by each university, and they change from year to year. The student always checks them there before choosing [2].

4.2 The Idea in Plain Words

4.2.1 From advice to variables

Listen to one of Bu Wati's sentences again: *a girl who loves living things and is not afraid of blood often does well in nursing.* Three different things are going on in it. There is something Bu Wati asks about — what the student loves. There is the answer she listens for — living things. And there is the advice she gives when the answers fit — nursing.

VCIRS gives each of these a name. The thing she asks about is a **variable**: `likes`, `blood ok`, `science score`. The answer is a **value** of that variable: `living things`, `yes`, `88`. A variable together with a comparison and a value is a **condition**: `likes = living things`, `science score >= 85`. And the advice is a **conclusion**, which is itself a variable with a value: `program = nursing`.

This split looks almost too simple to be worth a name, but it is the whole trick of the chapter. As long as knowledge lives in sentences, a computer can only store it and print it back. Once it is split into variables and values, the computer can *ask* for the value of a variable, *compare* the answer with a condition, *count* how often a variable is used, and *find* every piece of advice that depends on it. The thesis that proposed VCIRS puts the variable at the very center for exactly this reason: "the fundamental element of Variable-Centered Rule Structure is a variable, posted by the user," and the system "maintains carefully the variable about its values, structure and occurrence" [5], [6].

Table 4.1 lists the variables that appear in Bu Wati's advice, with the question behind each one. It is worth writing such a table before writing any code. It is the vocabulary of the knowledge base, and a knowledge base whose vocabulary is muddled — `likes` in one place, `enjoys` in another, `hobby` in a third — cannot be searched or counted.

Table 4.1 The variables of Bu Wati's advice, with the question behind each one and its usual values.

Variable	The question behind it	Usual values
`likes`	What do you enjoy most?	computers, living things, numbers, people, building things, drawing
`wants`	After school, do you want to work soon or study longer?	work soon, study longer
`blood ok`	Are you all right with seeing blood and wounds?	yes, no
`science score`	Your average report score in science (0–100)?	a number
`math score`	Your average report score in mathematics (0–100)?	a number
`patient with children`	Are you patient with small children?	yes, no
`drawing`	How good are you at drawing?	good, average
`program`	(the conclusion: the suggested study program)	informatics, nursing, medicine, …

4.2.2 Values and comparisons

Most values are words: `living things`, `work soon`, `yes`. For these the only sensible comparisons are *equal* (`=`) and *not equal* (`<>`). Some values are numbers, such as a report score of 88, and for them it makes sense to ask *at least* (`>=`), *more than* (`>`), *at most* (`<=`) and *less than* (`<`). VCIRS allows exactly these six operators, `=`, `<>`, `<`, `<=`, `>` and `>=`, and it compares two values as numbers only when both of them look like numbers [5]. So `science score >= 85` is true for a student with 88 and false for one with 80, while `likes >= computers` is simply never true, because words have no order that the engine could trust.

A condition that uses a number is a small but real step up from the rules of Chapter 3. Bu Wati would never say "if the science score is 88"; she says "if the science scores are *strong*", and the number 85 is how she makes *strong* precise. Chapter 9 goes one step further and lets the engine understand *strong* itself, as a fuzzy value.

4.2.3 A node: a small sequence of conditions

A **node** is one piece of advice: a short sequence of conditions, in a fixed order, together with one or more conclusions [5]. The node for the living-things advice is simply

```
likes = living things -> program = biology
```

and the node for nursing, which is an *exception* of the biology node, holds only what is new:

```
blood ok = yes -> program = nursing
```

This is the Ripple-Down-Rules idea of Chapter 3 in a new setting. The nursing node is tried only for students who already fit the biology node, so it does not need to say `likes = living things` again. And its own exception, the medicine node, needs only `science score >= 85`. Written as a flat list, the three pieces of advice would repeat `likes = living things` three times and `blood ok = yes` twice; kept as nodes, every condition is written once, where it belongs.

The order of the conditions inside a node matters. It is the order in which the engine checks them — and asks about them — and it is also, as Chapter 6 shows, a small piece of knowledge in its own right: a condition later in a node stands *closer to the conclusion*, and VCIRS gives it more weight when it counts how the knowledge is used [5]. The position of a condition in its node is called its **variable order**.

Every node has a name, its **NodeID**, made from its first conclusion and a serial number: the node that concludes `program = nursing` is `nursing#1`; a second node with the same conclusion would be `nursing#2` [5]. The names are not decoration. The history of the knowledge base, the explanations and the reports all refer to nodes by these names, so a counselor can say "`nursing#1` was right for Laras" and everybody knows which piece of advice she means.

4.2.4 A rule: a sequence of nodes

A **rule** in VCIRS is a sequence of nodes, from the top level down: `biology#1 -> nursing#1 -> medicine#1` is one rule [5], [6]. It reads: *if the student likes living things, and she is all right with blood, and her science score is at least 85, then the program to look at is medicine.* The conclusion of a rule is the conclusion of its last node, and the rule is named after its last node too, so this rule is `medicine#1`.

Every node is the last node of exactly one rule — the path that leads down to it — so a knowledge base with thirteen nodes has thirteen rules. But a node can appear in *many* rules: `biology#1` is the first node of `biology#1`, of `nursing#1` and of `medicine#1`. The thesis calls this **sharing**: a node shared by many rules, like a variable shared by many nodes, is used in many places and is therefore likely to be important [5]. Keep this word in mind; the tables of Section 4.8 count it.

The nodes met on the way down a rule also carry conclusions: before `medicine#1` fires, `biology#1` and `nursing#1` have fired and concluded *biology* and *nursing*. The engine keeps these as **intermediate conclusions** [5]. For a doctor's diagnosis they might be noise; for a student choosing programs they are exactly the backup choices that **Figure 4.1** says she needs.

4.2.5 The Variable-Centered Rule Structure

Put the pieces together and you have the structure that gives VCIRS its name. **Figure 4.2** shows part of it for Bu Wati's advice.

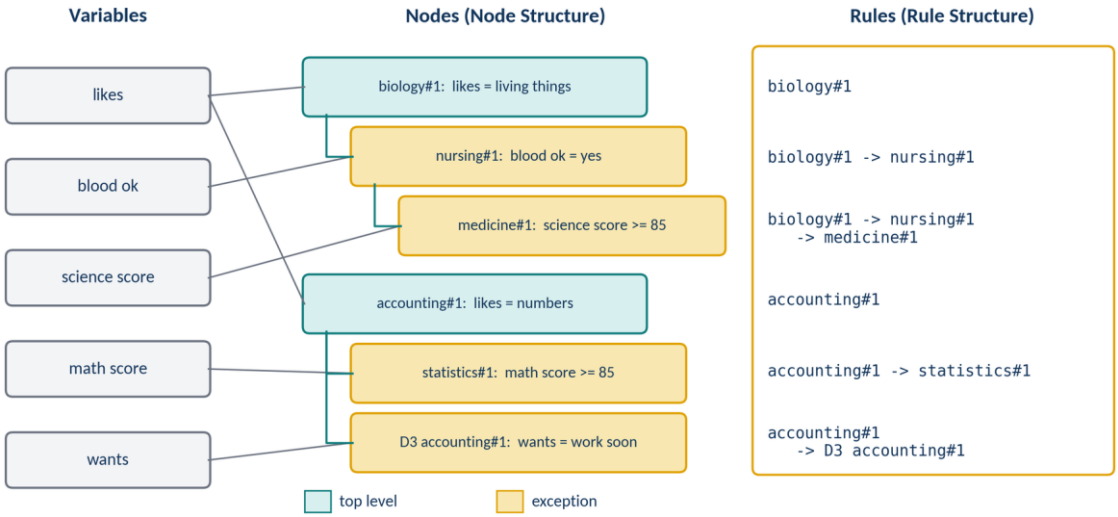

Figure 4.2 Part of the Variable-Centered Rule Structure of Bu Wati's advice. Each variable is kept once and linked to the nodes that use it; each node holds only its own conditions; each rule is a sequence of nodes from the top level down.

It has three parts [5], [7]:

- the **Variable table** keeps every variable once, however many nodes use it, with its question and its usual values, and remembers where it is used and at which position;
- the **Node Structure** keeps every node: its NodeID, its parent (none for a top-level node), its conditions in order, its conclusions, its certainty (the NodeCFValue, 100 unless the expert is less sure), and the **cornerstone case** that created it;
- the **Rule Structure** keeps every rule as a list of NodeIDs, each with its position in the rule.

Figure 4.3 shows the fields of one node and of one rule, as the thesis defines them. Two fields there have not been explained yet. The **CF value** of each condition says how certain it is, on the scale of 0 to 100 used by classic rule-based systems; all of Bu Wati's conditions are certain, so all of them are 100. The **Credit** of a condition counts how many times a later case *agreed* with it; it starts at 1, and Chapter 6 shows how it grows and what it is used for.

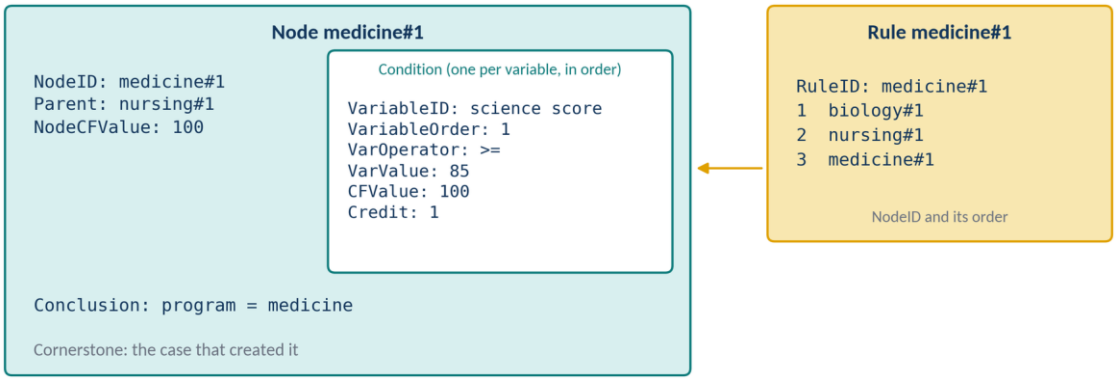

Figure 4.3 The fields of a node and of a rule in the Variable-Centered Rule Structure, filled in for `medicine#1`. The rule lists its nodes in order; each node lists its conditions in order.

Why go to this trouble, when Chapter 3's Ripple Down Rules worked well enough as a tree? Because a tree answers only one kind of question: *given these facts, what is the answer?* The three tables answer many more. *Which advice depends on the science score?* Look up `science score` in the Variable table. *Which pieces of advice would change if the biology node were wrong?* Look up the rules that share `biology#1`. *Which question should the advisor ask first?* Count how much each variable is used. *Can the same knowledge be read as ordinary IF-THEN rules, for forward and*

backward reasoning? Yes: each rule is a path of nodes, and its conditions can be collected into one IF-THEN rule, which is what Chapter 8 does. The thesis sums it up as "easy knowledge building, powerful knowledge inferencing and evolutionally system performance refining" at the same time [6], [5], and a fuzzy study-program advisor built on VCIRS, with the departments of the Chinese University of Hong Kong as its case study, used the very same structure [8].

> **DEFINITION.** The *Variable-Centered Rule Structure* keeps a knowledge base as three linked tables: the *variables* (each kept once, with where it is used), the *nodes* (short ordered sequences of conditions with conclusions, each an exception of its parent), and the *rules* (sequences of nodes from the top level down, named after their last node).

4.2.6 How a suggestion is found

Answering a student works as in Chapter 3, with one improvement. The engine tries the top-level nodes in order; for each one it checks the conditions in order and asks the student for any value it does not know yet. A node whose first condition is false is abandoned at once, without asking about the rest. When a node **fires** — all its conditions hold — the engine goes down to its exceptions and does the same there, as deep as it can. The suggestion is the conclusion of the last node that fired, and the conclusions of the nodes above it are the intermediate conclusions [5].

The improvement is that a value, once known, is never asked again. It stays in the **working memory** of the consultation, the list of facts known so far, and every node that needs it reads it from there. For Laras, the first node asks `likes`; she answers *living things*; the informatics node fails at once; the biology node fires without another question; its exception, nursing, asks `blood ok`; and so on. Laras answers three questions, not eight. That is what a good counselor does, and it is what a form with every question on it never does.

4.3 Step 1: The Advice as a Flat List

Before we build the structure, let us look at the knowledge the way most programs would store it: as a flat list of complete rules, each with all its conditions. **Listing 4.1** stores Bu Wati's thirteen pieces of advice that way and then does what VCIRS does first with any knowledge: it **indexes it by variable**.

Listing 4.1 `FlatAdvice.java`: Bu Wati's advice as a flat list of rules, indexed by variable.

```
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

/** Bu Wati's advice as a flat list of rules, then indexed by variable. */
public class FlatAdvice {
    record Rule(List<String> conditions, String program) { }

    static final List<Rule> RULES = List.of(
            new Rule(List.of("likes = computers"), "informatics"),
            new Rule(List.of("likes = computers", "wants = work soon"),
                    "D3 informatics"),
            new Rule(List.of("likes = living things"), "biology"),
            new Rule(List.of("likes = living things", "blood ok = yes"),
                    "nursing"),
            new Rule(List.of("likes = living things", "blood ok = yes",
                    "science score >= 85"), "medicine"),
            new Rule(List.of("likes = numbers"), "accounting"),
            new Rule(List.of("likes = numbers", "math score >= 85"),
                    "statistics"),
            new Rule(List.of("likes = numbers", "wants = work soon"),
                    "D3 accounting"),
            new Rule(List.of("likes = people"), "communication science"),
            new Rule(List.of("likes = people", "patient with children = yes"),
                    "primary teaching"),
            new Rule(List.of("likes = building things"), "civil engineering"),
            new Rule(List.of("likes = building things", "drawing = good",
                    "math score >= 75"), "architecture"),
            new Rule(List.of("likes = drawing"), "visual communication design"));

    static final String[] OPERATORS = {" <= ", " >= ", " <> ", " = ", " < ",
        " > "};

    /** "math score >= 85" -> {"math score", ">=", "85"} */
    static String[] split(String condition) {
        for (String op : OPERATORS) {
            int at = condition.indexOf(op);
            if (at > 0) {
                return new String[] {condition.substring(0, at), op.strip(),
                    condition.substring(at + op.length())};
            }
        }
        throw new IllegalArgumentException("no operator in " + condition);
    }
```

Listing 4.1 (Continued) *FlatAdvice.java: Bu Wati's advice as a flat list of rules, indexed by variable.*

```
    static String times(int n) {
        return n + (n == 1 ? " time" : " times");
    }

    public static void main(String[] args) {
        Map<String, List<String>> values = new LinkedHashMap<>();
        Map<String, List<Integer>> usedIn = new LinkedHashMap<>();
        Map<String, Integer> count = new LinkedHashMap<>();
        int total = 0;
        for (int i = 0; i < RULES.size(); i++) {
            for (String c : RULES.get(i).conditions()) {
                String[] p = split(c);
                String shown = p[1].equals("=") ? p[2] : p[1] + " " + p[2];
                values.putIfAbsent(p[0], new ArrayList<>());
                usedIn.putIfAbsent(p[0], new ArrayList<>());
                if (!values.get(p[0]).contains(shown)) {
                    values.get(p[0]).add(shown);
                }
                usedIn.get(p[0]).add(i + 1);
                count.merge(c, 1, Integer::sum);
                total++;
            }
        }
        System.out.println(RULES.size() + " rules, " + total + " conditions, "
                + values.size() + " variables.");
        System.out.println();
        for (String var : values.keySet()) {
            List<String> numbers = new ArrayList<>();
            for (int n : usedIn.get(var)) {
                numbers.add(String.valueOf(n));
            }
            System.out.println(var + " (used " + times(numbers.size()) + ")");
            System.out.println("  values: " + String.join(", ", values.get(var)));
            System.out.println("  rules:  " + String.join(", ", numbers));
        }
        System.out.println();
        System.out.println("Conditions written more than once:");
        for (Map.Entry<String, Integer> e : count.entrySet()) {
            if (e.getValue() > 1) {
                System.out.println("  " + e.getKey() + ": " + times(e.getValue()));
            }
        }
    }
}
```

A `record Rule` holds the conditions (as text, such as `"math score >= 85"`) and the program. `split` cuts a condition into its three parts: the variable, the operator and the value. It looks for each operator *with a space on both sides*, so that `" >= "` is found in `"math score >= 85"` and `" = "` is not mistaken for part of `>=`. `main` then walks through every condition of every rule and fills three maps: the values seen for each variable, the rules that use it, and how many times each complete condition is written. `LinkedHashMap` is a map that remembers the order in which keys were first added, so the variables come out in the order they first appear in the advice. **Listing 4.2** is the same program in Python.

Listing 4.2 `flat_advice.py`: the same flat list and index in Python.

```
"""Bu Wati's advice as a flat list of rules, then indexed by variable."""

RULES = [
    (["likes = computers"], "informatics"),
    (["likes = computers", "wants = work soon"], "D3 informatics"),
    (["likes = living things"], "biology"),
    (["likes = living things", "blood ok = yes"], "nursing"),
    (["likes = living things", "blood ok = yes", "science score >= 85"],
     "medicine"),
    (["likes = numbers"], "accounting"),
    (["likes = numbers", "math score >= 85"], "statistics"),
    (["likes = numbers", "wants = work soon"], "D3 accounting"),
    (["likes = people"], "communication science"),
    (["likes = people", "patient with children = yes"],
     "primary teaching"),
    (["likes = building things"], "civil engineering"),
    (["likes = building things", "drawing = good", "math score >= 75"],
     "architecture"),
    (["likes = drawing"], "visual communication design"),
]

OPERATORS = [" <= ", " >= ", " <> ", " = ", " < ", " > "]


def split(condition):
    """'math score >= 85' -> ['math score', '>=', '85']"""
    for op in OPERATORS:
        at = condition.find(op)
        if at > 0:
            return [condition[:at], op.strip(), condition[at + len(op):]]
    raise ValueError("no operator in " + condition)


def times(n):
    return str(n) + (" time" if n == 1 else " times")
```

Listing 4.2 (Continued) *flat_advice.py: the same flat list and index in Python.*

```
def main():
    index = {}                # variable -> {"values": [...], "rules": [...]}
    count = {}                # condition -> how many rules write it
    for number, (conditions, program) in enumerate(RULES, start=1):
        for c in conditions:
            var, op, value = split(c)
            entry = index.setdefault(var, {"values": [], "rules": []})
            shown = value if op == "=" else op + " " + value
            if shown not in entry["values"]:
                entry["values"].append(shown)
            entry["rules"].append(number)
            count[c] = count.get(c, 0) + 1

    total = sum(len(conditions) for conditions, _ in RULES)
    print(str(len(RULES)) + " rules, " + str(total) + " conditions, "
          + str(len(index)) + " variables.")
    print()
    for var, entry in index.items():
        print(var + " (used " + times(len(entry["rules"])) + ")")
        print("  values: " + ", ".join(entry["values"]))
        print("  rules:  " + ", ".join(str(n) for n in entry["rules"]))
    print()
    print("Conditions written more than once:")
    for c, n in count.items():
        if n > 1:
            print("  " + c + ": " + times(n))


if __name__ == "__main__":
    main()
```

The Python version uses a dictionary of small dictionaries for the index, and `setdefault`, which returns the entry for a variable and creates it first if it does not exist yet. The last lines, `if __name__ == "__main__": main()`, run `main` only when the file is run as a program, not when another file imports `split` from it (the tests of Section 4.9 do). Run `java -cp out FlatAdvice` (after compiling, see Section 4.7) or `python flat_advice.py`; both print:

```
13 rules, 22 conditions, 7 variables.

likes (used 13 times)
  values: computers, living things, numbers, people, building things, drawing
  rules:  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13
wants (used 2 times)
  values: work soon
  rules:  2, 8
blood ok (used 2 times)
  values: yes
  rules:  4, 5
science score (used 1 time)
  values: >= 85
  rules:  5
math score (used 2 times)
  values: >= 85, >= 75
  rules:  7, 12
patient with children (used 1 time)
  values: yes
  rules:  10
drawing (used 1 time)
  values: good
  rules:  12

Conditions written more than once:
  likes = computers: 2 times
  wants = work soon: 2 times
  likes = living things: 3 times
  blood ok = yes: 2 times
  likes = numbers: 3 times
  likes = people: 2 times
  likes = building things: 2 times
```

Read the index as a counselor would. The first line says that thirteen pieces of advice need only seven variables: the knowledge is much smaller than it looks. The variable `likes` appears in every rule, which tells us that it is the first thing to ask and the backbone of the whole knowledge base. `math score` is used twice, with two different thresholds — 85 for statistics, 75 for architecture — which a counselor might want to look at side by side. And the last block shows the price of the flat list: `likes = living things` is written three times, `likes = numbers` three times, `blood ok = yes` twice. Every repetition is a place where the knowledge can drift apart. If Bu Wati one day decides that "living things" should be "living things and nature", she must find and change it in three rules, and if she misses one, the knowledge base quietly disagrees with itself.

Note one line that is *not* a repetition in the same sense: `wants = work soon` appears twice, in the rule for D3 informatics and in the rule for D3 accounting. These are two different pieces of advice that happen to ask the same question, and they will stay two nodes in the structure. VCIRS keeps the *variable* `wants` once and links it to both nodes; the Variable table of the next step shows this.

4.4 Step 2: Variables, Nodes and Rules by Hand

Listing 4.3 keeps the same advice the VCIRS way. Each node holds only its own conditions and names its parent; the program builds the Variable table, the Node Structure and the Rule Structure as it goes, and uses them to answer four students.

Listing 4.3 `MiniStructure.java`: a Variable table, a Node Structure and a Rule Structure, built by hand.

```java
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

/**
 * Bu Wati's advice kept the VCIRS way: a Variable table, a Node Structure
 * and a Rule Structure, built by hand. A node holds only its own conditions;
 * the conditions of its parent are not written again.
 */
public class MiniStructure {
    static final String[] OPERATORS = {" <= ", " >= ", " <> ", " = ", " < ",
        " > "};

    /** "math score >= 85" -> {"math score", ">=", "85"} */
    static String[] split(String condition) {
        for (String op : OPERATORS) {
            int at = condition.indexOf(op);
            if (at > 0) {
                return new String[] {condition.substring(0, at), op.strip(),
                    condition.substring(at + op.length())};
            }
        }
        throw new IllegalArgumentException("no operator in " + condition);
    }

    static boolean holds(String condition, Map<String, String> facts) {
        String[] p = split(condition);
        String fact = facts.get(p[0]);
        if (fact == null) {
            return false;                    // an unknown fact never matches
        }
        switch (p[1]) {
            case "=": return fact.equals(p[2]);
            case "<>": return !fact.equals(p[2]);
            default: break;
        }
        double a = Double.parseDouble(fact);  // the others compare numbers
        double b = Double.parseDouble(p[2]);
        switch (p[1]) {
            case "<": return a < b;
            case "<=": return a <= b;
            case ">": return a > b;
            default: return a >= b;
        }
    }
```

Listing 4.3 (Continued) *MiniStructure.java: a Variable table, a Node Structure and a Rule Structure, built by hand.*

```
    record Node(String id, String parent, List<String> conditions,
                String program) { }

    static class Structure {
        final Map<String, Node> nodes = new LinkedHashMap<>();   // Node Str.
        final Map<String, List<String>> rules = new LinkedHashMap<>();
        final Map<String, List<String>> variables = new LinkedHashMap<>();

        String add(String parent, List<String> conditions, String program) {
            int n = 1;
            while (nodes.containsKey(program + "#" + n)) {
                n++;
            }
            Node node = new Node(program + "#" + n, parent, conditions,
                    program);
            nodes.put(node.id(), node);
            List<String> path = new ArrayList<>(List.of(node.id()));
            String p = parent;
            while (p != null) {              // a rule is the path from the top
                path.add(0, p);
                p = nodes.get(p).parent();
            }
            rules.put(node.id(), path);      // named after its last node
            for (int i = 0; i < conditions.size(); i++) {
                String var = split(conditions.get(i))[0];
                variables.putIfAbsent(var, new ArrayList<>());
                variables.get(var).add(node.id() + ", position " + (i + 1));
            }
            return node.id();
        }

        List<Node> children(String parent) {
            List<Node> out = new ArrayList<>();
            for (Node n : nodes.values()) {
                if (parent == null ? n.parent() == null
                        : parent.equals(n.parent())) {
                    out.add(n);
                }
            }
            return out;
        }
```

Listing 4.3 (Continued) *MiniStructure.java: a Variable table, a Node Structure and a Rule Structure, built by hand.*

```
        /**
         * Fire the first matching top-level node, then its first matching
         * exception, and so on down. The last node fired gives the answer.
         */
        List<String> answer(Map<String, String> facts) {
            List<String> path = new ArrayList<>();
            List<Node> level = children(null);
            while (true) {
                Node fired = null;
                for (Node node : level) {
                    boolean all = true;
                    for (String c : node.conditions()) {
                        all = all && holds(c, facts);
                    }
                    if (all) {
                        fired = node;
                        break;
                    }
                }
                if (fired == null) {
                    return path;
                }
                path.add(fired.id());
                level = children(fired.id());
            }
        }
    }

    /** Bu Wati's thirteen pieces of advice; exceptions name their parent. */
    static Structure buWati() {
        Structure s = new Structure();
        String inf = s.add(null, List.of("likes = computers"), "informatics");
        s.add(inf, List.of("wants = work soon"), "D3 informatics");
        String bio = s.add(null, List.of("likes = living things"), "biology");
        String nur = s.add(bio, List.of("blood ok = yes"), "nursing");
        s.add(nur, List.of("science score >= 85"), "medicine");
        String acc = s.add(null, List.of("likes = numbers"), "accounting");
        s.add(acc, List.of("math score >= 85"), "statistics");
        s.add(acc, List.of("wants = work soon"), "D3 accounting");
        String com = s.add(null, List.of("likes = people"),
                "communication science");
        s.add(com, List.of("patient with children = yes"), "primary teaching");
        String civ = s.add(null, List.of("likes = building things"),
                "civil engineering");
        s.add(civ, List.of("drawing = good", "math score >= 75"),
                "architecture");
```

Listing 4.3 (Continued) *MiniStructure.java: a Variable table, a Node Structure and a Rule Structure, built by hand.*

```
        s.add(null, List.of("likes = drawing"), "visual communication design");
        return s;
    }

    public static void main(String[] args) {
        Structure s = buWati();
        int stored = 0;
        for (Node n : s.nodes.values()) {
            stored += n.conditions().size();
        }
        System.out.println("Node Structure: " + s.nodes.size() + " nodes, "
                + stored + " conditions");
        for (Node n : s.nodes.values()) {
            System.out.println("  " + n.id() + (n.parent() != null
                    ? "  (exception of " + n.parent() + ")" : "  (top level)"));
            System.out.println("      " + String.join(", ", n.conditions())
                    + " -> program = " + n.program());
        }
        System.out.println("Rule Structure: " + s.rules.size() + " rules");
        for (Map.Entry<String, List<String>> r : s.rules.entrySet()) {
            System.out.println("  " + r.getKey() + ": "
                    + String.join(" -> ", r.getValue()));
        }
        System.out.println("Variable table: " + s.variables.size()
                + " variables");
        for (Map.Entry<String, List<String>> v : s.variables.entrySet()) {
            System.out.println("  " + v.getKey() + ", used in "
                    + v.getValue().size() + " node(s):");
            for (String where : v.getValue()) {
                System.out.println("      " + where);
            }
        }

        System.out.println();
        String[] names = {"Laras", "Bima", "Sari", "Tono"};
        List<Map<String, String>> students = List.of(
                Map.of("likes", "living things", "blood ok", "yes",
                        "science score", "88"),
                Map.of("likes", "computers", "wants", "work soon"),
                Map.of("likes", "numbers", "math score", "80",
                        "wants", "study longer"),
                Map.of("likes", "music"));
        for (int i = 0; i < names.length; i++) {
            List<String> path = s.answer(students.get(i));
            if (path.isEmpty()) {
                System.out.println(names[i] + ": no advice yet");
```

Listing 4.3 (Continued) *MiniStructure.java: a Variable table, a Node Structure and a Rule Structure, built by hand.*

```
            } else {
                System.out.println(names[i] + ": "
                        + s.nodes.get(path.get(path.size() - 1)).program()
                        + "  (path: " + String.join(" -> ", path) + ")");
            }
        }
    }
}
```

Read it from the top.

Comparing. `split` is the same as in Step 1. `holds` decides whether one condition is true for a student's facts. A fact the student has not given never matches. Words are compared with `=` and

`<>`; for the four ordering operators both sides are turned into numbers with `Double.parseDouble` and compared as numbers.

The three tables. A `Node` has an ID, a parent (`null` for a top-level node), its conditions and its program. `Structure` holds the three tables as maps: `nodes` (the Node Structure), `rules` (each rule ID with its list of node IDs, the Rule Structure) and `variables` (each variable with the places where it is used, the Variable table).

Adding a node. `add` does, in twenty lines, what the thesis describes for the Node and Rule Structures [5]. It makes the NodeID from the program and the first free serial number (`medicine#1`, and `medicine#2` if the first is taken). It stores the node. It builds the node's rule by walking up from the new node through its parents to the top level, putting each one at the front of the path, and stores the path under the node's own ID — a rule is named after its last node. Finally it records, for each condition, the variable, the node and the position, which is the variable order.

Answering. `answer` starts with the top-level nodes (the children of `null`), fires the first one whose conditions all hold, then looks among *its* children, and so on down, collecting the path. When no node at the current level fires, the path so far is the answer, and the last node on it gives the program. An empty path means *no advice yet*.

Bu Wati's knowledge. `buWati` adds the thirteen nodes. Look at how little each exception says: `add(nur, List.of("science score >= 85"), "medicine")` hangs medicine under nursing with one condition. `main` prints the three tables and answers four students. **Listing 4.4** is the Python version.

Listing 4.4 `mini_structure.py`: the same three tables in Python.

```
"""Bu Wati's advice kept the VCIRS way: a Variable table, a Node Structure
and a Rule Structure, built by hand. A node holds only its own conditions;
the conditions of its parent are not written again."""

OPERATORS = [" <= ", " >= ", " <> ", " = ", " < ", " > "]


def split(condition):
    """'math score >= 85' -> ['math score', '>=', '85']"""
    for op in OPERATORS:
        at = condition.find(op)
        if at > 0:
            return [condition[:at], op.strip(), condition[at + len(op):]]
    raise ValueError("no operator in " + condition)


def holds(condition, facts):
    var, op, value = split(condition)
    if var not in facts:
        return False                    # an unknown fact never matches
    fact = facts[var]
    if op == "=":
        return fact == value
    if op == "<>":
        return fact != value
    a, b = float(fact), float(value)    # the other operators compare numbers
    return {"<": a < b, "<=": a <= b, ">": a > b, ">=": a >= b}[op]


class Node:
    def __init__(self, node_id, parent, conditions, program):
        self.id = node_id
        self.parent = parent            # None for a top-level node
        self.conditions = conditions
        self.program = program


class Structure:
    def __init__(self):
        self.nodes = {}       # node id -> Node          (Node Structure)
        self.rules = {}       # rule id -> [node ids]    (Rule Structure)
        self.variables = {}   # variable -> [(node id, position)]
```

Listing 4.4 (Continued) *mini_structure.py: the same three tables in Python.*

```
    def add(self, parent, conditions, program):
        n = 1
        while program + "#" + str(n) in self.nodes:
            n += 1
        node = Node(program + "#" + str(n), parent, conditions, program)
        self.nodes[node.id] = node
        path, p = [node.id], parent
        while p is not None:             # a rule is the path from the top
            path.insert(0, p)
            p = self.nodes[p].parent
        self.rules[node.id] = path       # named after its last node
        for position, c in enumerate(conditions, start=1):
            var = split(c)[0]
            self.variables.setdefault(var, []).append((node.id, position))
        return node.id

    def children(self, parent):
        return [n for n in self.nodes.values() if n.parent == parent]

    def answer(self, facts):
        """Fire the first matching top-level node, then its first matching
        exception, and so on down. The last node fired gives the answer."""
        path, level = [], self.children(None)
        while True:
            fired = None
            for node in level:
                if all(holds(c, facts) for c in node.conditions):
                    fired = node
                    break
            if fired is None:
                return path
            path.append(fired.id)
            level = self.children(fired.id)


def bu_wati():
    """Bu Wati's thirteen pieces of advice; exceptions name their parent."""
    s = Structure()
    inf = s.add(None, ["likes = computers"], "informatics")
    s.add(inf, ["wants = work soon"], "D3 informatics")
    bio = s.add(None, ["likes = living things"], "biology")
    nur = s.add(bio, ["blood ok = yes"], "nursing")
    s.add(nur, ["science score >= 85"], "medicine")
    acc = s.add(None, ["likes = numbers"], "accounting")
    s.add(acc, ["math score >= 85"], "statistics")
    s.add(acc, ["wants = work soon"], "D3 accounting")
```

Listing 4.4 (Continued) *mini_structure.py: the same three tables in Python.*

```
    com = s.add(None, ["likes = people"], "communication science")
    s.add(com, ["patient with children = yes"], "primary teaching")
    civ = s.add(None, ["likes = building things"], "civil engineering")
    s.add(civ, ["drawing = good", "math score >= 75"], "architecture")
    s.add(None, ["likes = drawing"], "visual communication design")
    return s


def main():
    s = bu_wati()
    stored = sum(len(n.conditions) for n in s.nodes.values())
    print("Node Structure: " + str(len(s.nodes)) + " nodes, " + str(stored)
          + " conditions")
    for n in s.nodes.values():
        print("  " + n.id + ("  (exception of " + n.parent + ")" if n.parent
                             else "  (top level)"))
        print("      " + ", ".join(n.conditions) + " -> program = " + n.program)
    print("Rule Structure: " + str(len(s.rules)) + " rules")
    for rule_id, path in s.rules.items():
        print("  " + rule_id + ": " + " -> ".join(path))
    print("Variable table: " + str(len(s.variables)) + " variables")
    for var, used in s.variables.items():
        print("  " + var + ", used in " + str(len(used)) + " node(s):")
        for nid, pos in used:
            print("      " + nid + ", position " + str(pos))

    print()
    students = [
        ("Laras", {"likes": "living things", "blood ok": "yes",
                   "science score": "88"}),
        ("Bima", {"likes": "computers", "wants": "work soon"}),
        ("Sari", {"likes": "numbers", "math score": "80",
                  "wants": "study longer"}),
        ("Tono", {"likes": "music"}),
    ]
    for name, facts in students:
        path = s.answer(facts)
        if not path:
            print(name + ": no advice yet")
        else:
            print(name + ": " + s.nodes[path[-1]].program + "  (path: "
                  + " -> ".join(path) + ")")
```

Listing 4.4 (Continued) *mini_structure.py: the same three tables in Python.*

```
if __name__ == "__main__":
    main()
```

In Python, `holds` picks the right comparison from a small dictionary of the four results, and `answer` uses `all(...)` to check every condition in one expression. Both programs print:

```
Node Structure: 13 nodes, 14 conditions
  informatics#1  (top level)
      likes = computers -> program = informatics
  D3 informatics#1  (exception of informatics#1)
      wants = work soon -> program = D3 informatics
  biology#1  (top level)
      likes = living things -> program = biology
  nursing#1  (exception of biology#1)
      blood ok = yes -> program = nursing
  medicine#1  (exception of nursing#1)
      science score >= 85 -> program = medicine
  accounting#1  (top level)
      likes = numbers -> program = accounting
  statistics#1  (exception of accounting#1)
      math score >= 85 -> program = statistics
  D3 accounting#1  (exception of accounting#1)
      wants = work soon -> program = D3 accounting
  communication science#1  (top level)
      likes = people -> program = communication science
  primary teaching#1  (exception of communication science#1)
      patient with children = yes -> program = primary teaching
  civil engineering#1  (top level)
      likes = building things -> program = civil engineering
  architecture#1  (exception of civil engineering#1)
      drawing = good, math score >= 75 -> program = architecture
  visual communication design#1  (top level)
      likes = drawing -> program = visual communication design
Rule Structure: 13 rules
  informatics#1: informatics#1
  D3 informatics#1: informatics#1 -> D3 informatics#1
  biology#1: biology#1
  nursing#1: biology#1 -> nursing#1
  medicine#1: biology#1 -> nursing#1 -> medicine#1
  accounting#1: accounting#1
  statistics#1: accounting#1 -> statistics#1
  D3 accounting#1: accounting#1 -> D3 accounting#1
  communication science#1: communication science#1
  primary teaching#1: communication science#1 -> primary teaching#1
  civil engineering#1: civil engineering#1
  architecture#1: civil engineering#1 -> architecture#1
  visual communication design#1: visual communication design#1
Variable table: 7 variables
  likes, used in 6 node(s):
      informatics#1, position 1
      biology#1, position 1
      accounting#1, position 1
      communication science#1, position 1
      civil engineering#1, position 1
      visual communication design#1, position 1
  wants, used in 2 node(s):
      D3 informatics#1, position 1
      D3 accounting#1, position 1
```

```
  blood ok, used in 1 node(s):
      nursing#1, position 1
  science score, used in 1 node(s):
      medicine#1, position 1
  math score, used in 2 node(s):
      statistics#1, position 1
      architecture#1, position 2
  patient with children, used in 1 node(s):
      primary teaching#1, position 1
  drawing, used in 1 node(s):
      architecture#1, position 1

Laras: medicine  (path: biology#1 -> nursing#1 -> medicine#1)
Bima: D3 informatics  (path: informatics#1 -> D3 informatics#1)
Sari: accounting  (path: accounting#1)
Tono: no advice yet
```

Compare the first line with Step 1: the same thirteen pieces of advice now need **14 conditions instead of 22**. Nothing was lost; the eight missing conditions are the ones an exception inherits from the nodes above it. The Rule Structure shows the other side of the same coin: the rule `medicine#1` is the sequence `biology#1 -> nursing#1 -> medicine#1`, and walking it collects exactly the three conditions that the flat rule had to write out in full.

The Variable table now reads like the index of a book. `likes` is used in six nodes, always in position 1 — it is the first question of every top-level node. `wants` is used in two nodes, D3 informatics and D3 accounting: one variable, two pieces of advice. `math score` is used in two nodes too, in position 1 of `statistics#1` and in position 2 of `architecture#1`, after `drawing = good`. If Bu Wati wants to know every piece of advice that depends on a mathematics score, the answer is in one line.

And the four students get sensible answers. **Laras** loves living things, is all right with blood and has 88 in science: the path is `biology#1 -> nursing#1 -> medicine#1`, and the suggestion is medicine — with biology and nursing met on the way. **Bima** loves computers and wants to work soon: D3 informatics. **Sari** loves numbers, but her mathematics score of 80 is below the 85 of the statistics node, and she wants to study longer, so neither exception of accounting fires: accounting. **Tono** loves music, and nobody has taught the knowledge base anything about music yet: no advice. That is the honest answer, and Section 4.7 shows how the counselor fills such gaps.

> **KEY INSIGHT.** Keeping knowledge around variables does not change what the knowledge *says*; it changes what you can *do* with it. The same thirteen pieces of advice, kept as three linked tables, are smaller, cannot disagree with themselves, and can be searched by variable, by node and by rule.

4.5 From Our Structure to vcirs-core

The hand-made structure is complete as an idea, but three things are missing for a real advisor. It needs all the facts in advance, so a student would have to fill in every question; it cannot learn, so Bu Wati would have to edit the program to add advice; and it forgets everything when the program ends. The shared engine, vcirs-core, keeps exactly the same three tables and adds these missing pieces. **Table 4.2** shows how the ideas of Step 2 appear in it.

Table 4.2 How the hand-made structure of Step 2 appears in vcirs-core.

Our structure (Section 4.4)	Vcirs-core
`split` and `holds`	`Condition.parse("math score >= 85")` and `kb.evaluate(condition, facts)`
a `Node` with an ID, a parent, conditions and a program	`Node`: `id`, `parent`, `cf`, `conditions`, `conclusions`, `cornerstone`
the `nodes` map (Node Structure)	`kb.nodes`
the `rules` map (Rule Structure)	`kb.rules`, each a `Rule` with `nodes`
the `variables` map (Variable table)	`kb.variables` (with `question` and `choices`) and `kb.usage().variableSharing()`
`add(parent, conditions, program)`	`kb.refine(case, option, parent, chosen)`, or simply `kb.learn(case)`
`answer(facts)`	`kb.infer(facts, asker)`: asks unknown values, explains How and Why
the tables printed by `main`	`Report.variableSharing`, `Report.nodeSharing`, `Report.rules`, `Report.tree`
(lost when the program ends)	`kb.save(path)` and `KnowledgeBase.load(path)`, one JSON file for both languages

Two differences deserve a word. First, in vcirs-core a condition is an object, not a piece of text: a `Condition` has a variable, an operator, a value, a CF and a Credit, exactly the fields of **Figure 4.3**, so nothing has to be split again and again. Second, nodes are not added by naming a parent by hand. The counselor gives a **case** — the facts of a real student and the right program — and `learn` finds where it belongs, the way Chapter 3 did: if a node already gives the right program, the case is recorded as a check; if a node gives a wrong one, the new node becomes its exception, built from the differences with that node's cornerstone case; if no node answers, the new node goes to the top level [9], [7]. So Bu Wati never has to know which node is whose parent. She only has to know the right answer for the student in front of her.

4.6 The Study-Program Advisor

4.6.1 What Bu Wati taught it first

The advisor starts from Bu Wati's thirteen pieces of advice, written as thirteen cases in a case file, `study-seed-cases.json`:

```
{
  "name": "Study-program advisor",
  "description": "What Bu Wati, the school counselor, taught the advisor from last year's
school leavers.",
  "cases": [
    {"facts": ["likes = computers"], "conclusions": ["program = informatics"], "option":
-1, "author": "Wati"},
    {"facts": ["likes = computers", "wants = work soon"], "conclusions": ["program = D3
informatics"], "option": -1, "author": "Wati"},
    {"facts": ["likes = living things"], "conclusions": ["program = biology"], "option":
-1, "author": "Wati"},
    {"facts": ["likes = living things", "blood ok = yes"], "conclusions": ["program =
nursing"], "option": -1, "author": "Wati"},
    {"facts": ["likes = living things", "blood ok = yes", "science score >= 85"],
"conclusions": ["program = medicine"], "option": -1, "author": "Wati"},
    {"facts": ["likes = numbers"], "conclusions": ["program = accounting"], "option": -1,
"author": "Wati"},
    {"facts": ["likes = numbers", "math score >= 85"], "conclusions": ["program =
statistics"], "option": -1, "author": "Wati"},
    {"facts": ["likes = numbers", "wants = work soon"], "conclusions": ["program = D3
accounting"], "option": -1, "author": "Wati"},
    {"facts": ["likes = people"], "conclusions": ["program = communication science"],
"option": -1, "author": "Wati"},
    {"facts": ["likes = people", "patient with children = yes"], "conclusions": ["program
= primary teaching"], "option": -1, "author": "Wati"},
    {"facts": ["likes = building things"], "conclusions": ["program = civil
engineering"], "option": -1, "author": "Wati"},
    {"facts": ["likes = building things", "drawing = good", "math score >= 75"],
"conclusions": ["program = architecture"], "option": -1, "author": "Wati"},
    {"facts": ["likes = drawing"], "conclusions": ["program = visual communication
design"], "option": -1, "author": "Wati"}
  ]
}
```

Each case is a student Bu Wati remembers, reduced to the facts that mattered and the program she advised. The medicine case, for example, says: *likes living things, all right with blood, science score at least 85: medicine.* Conditions with numbers are written with their operator, `science score >= 85`, because that is what the counselor means. `"option": -1` asks the engine to decide where each case goes, by `learn`, exactly as if Bu Wati had brought the students to the advisor one by one in this order. Because of that order, the nursing case lands under biology, the medicine case under nursing, and the statistics and D3 accounting cases both under accounting — the tree of **Figure 4.2**, built without anyone naming a parent.

The first time the advisor runs, it builds its knowledge base from this file and saves it as `study-kb.json` in the same folder; after that it loads and updates the saved file.

4.6.2 The program

Listing 4.5 is the advisor in Java.

Listing 4.5 `StudyAdvisor.java`: the study-program advisor on vcirs-core.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;

import vcirs.Case;
import vcirs.CaseFile;
import vcirs.Conclusion;
import vcirs.Condition;
import vcirs.Fact;
import vcirs.Facts;
import vcirs.InferenceResult;
import vcirs.KnowledgeBase;
import vcirs.LearnResult;
import vcirs.Node;
import vcirs.Question;
import vcirs.Report;
import vcirs.Usage;
import vcirs.Variable;
import vcirs.VcirsException;

/**
 * Study-program advisor: Bu Wati's knowledge, kept as variables, nodes and
 * rules.  java StudyAdvisor [--echo] [--tree] [--tables] [knowledge-base.json]
 * It asks a school leaver only the questions it needs, suggests a study
 * program with the programs met on the way, explains why, and learns from
 * the counselor when she disagrees. --tables prints the Variable-Centered
 * Rule Structure: where each variable is used, which rules share each node,
 * and every rule as a sequence of nodes.
 */
public class StudyAdvisor {
    static final String[][] QUESTIONS = {
        {"likes", "What do you enjoy most?", "computers", "living things",
            "numbers", "people", "building things", "drawing"},
        {"wants", "After school, do you want to work soon or study longer?",
            "work soon", "study longer"},
        {"blood ok", "Are you all right with seeing blood and wounds?",
            "yes", "no"},
        {"science score", "Your average report score in science (0-100)?"},
        {"math score", "Your average report score in mathematics (0-100)?"},
        {"patient with children", "Are you patient with small children?",
            "yes", "no"},
```

Listing 4.5 (Continued) *StudyAdvisor.java: the study-program advisor on vcirs-core.*

```
        {"drawing", "How good are you at drawing?", "good", "average"},
        {"family farm", "Does your family have a farm, orchard or fish pond?",
            "yes", "no"},
    };

    static BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in, StandardCharsets.UTF_8));
    static boolean echo = false;
    static KnowledgeBase kb;

    /** Reads the user's answer; with echo on, prints it too. */
    static String read(String prompt) {
        System.out.print(prompt);
        String line;
        try {
            line = in.readLine();
        } catch (IOException e) {
            line = null;
        }
        String text = line == null ? "" : line.strip();
        if (echo) {
            System.out.println(text);
        }
        return text;
    }

    static Fact ask(Question q) {
        Variable var = kb.variables.get(q.variable);
        String text = q.variable;
        List<String> choices = q.choices;
        if (var != null && !var.question.isEmpty()) {
            text = var.question;                    // as Bu Wati wrote it
            choices = var.choices;
        }
        if (!choices.isEmpty()) {
            text += " (" + String.join("/", choices) + ")";
        }
        String answer = read(text + " > ");
        return answer.isEmpty() ? null : new Fact(q.variable, answer); // Enter
    }
```

Listing 4.5 (Continued) *StudyAdvisor.java: the study-program advisor on vcirs-core.*

```
    /** Load the knowledge base, or build it from the seed cases. */
    static KnowledgeBase openKb(String path) {
        if (Files.exists(Path.of(path))) {
            return KnowledgeBase.load(path);
        }
        Path parent = Path.of(path).toAbsolutePath().getParent();
        CaseFile seed = CaseFile.read(parent.resolve("study-seed-cases.json")
                .toString());
        KnowledgeBase k = new KnowledgeBase(seed.name, seed.description);
        for (String[] q : QUESTIONS) {
            List<String> choices = new ArrayList<>();
            for (int i = 2; i < q.length; i++) {
                choices.add(q[i]);
            }
            k.declare(q[0], q[1], choices);
        }
        seed.build(k);
        k.save(path);
        return k;
    }

    static void printTables() {
        System.out.println("Where each variable is used (the Variable table):");
        for (String line : Report.variableSharing(kb)) {
            System.out.println("  " + line);
        }
        System.out.println("Which rules share each node:");
        for (String line : Report.nodeSharing(kb)) {
            System.out.println("  " + line);
        }
        System.out.println("Every rule as a sequence of nodes "
                + "(the Rule Structure):");
        for (String line : Report.rules(kb)) {
            System.out.println("  " + line);
        }
        Usage usage = kb.usage();
        System.out.println("Most important variables: "
                + String.join(", ", usage.importantVariables()));
        System.out.println("Most important nodes: "
                + String.join(", ", usage.importantNodes()));
    }
```

Listing 4.5 (Continued) *StudyAdvisor.java: the study-program advisor on vcirs-core.*

```
    public static void main(String[] args) {
        String path = "../data/study-kb.json";
        boolean tree = false;
        boolean tables = false;
        for (String a : args) {
            if (a.equals("--echo")) {
                echo = true;
            } else if (a.equals("--tree")) {
                tree = true;
            } else if (a.equals("--tables")) {
                tables = true;
            } else {
                path = a;
            }
        }
        kb = openKb(path);
        if (tree) {
            for (String line : Report.tree(kb)) {
                System.out.println(line);
            }
            return;
        }
        if (tables) {
            printTables();
            return;
        }
        System.out.println("Study-program advisor: " + kb.nodes.size()
                + " pieces of advice so far.");
        System.out.println("Answer each question; press Enter if you do not "
                + "know.");
        InferenceResult result = kb.infer(Facts.none(), StudyAdvisor::ask);
        Node fired = result.paths.isEmpty() ? null : result.paths.get(0).last();
        if (fired != null && !result.conclusions().isEmpty()) {
            System.out.println("Suggested program: "
                    + result.conclusions().get(0).value);
            List<String> onWay = new ArrayList<>();
            for (Conclusion c : result.paths.get(0).intermediate()) {
                onWay.add(c.value);
            }
            if (!onWay.isEmpty()) {
                System.out.println("Also worth a look: "
                        + String.join(", ", onWay));
            }
            System.out.println("Why:");
            for (String line : result.how()) {
                System.out.println("  " + line);
```

Listing 4.5 (Continued) *StudyAdvisor.java: the study-program advisor on vcirs-core.*

```
            }
        } else {
            System.out.println("I have no suggestion for this student yet.");
        }

        List<Condition> facts = new ArrayList<>();
        for (Fact f : result.memory.facts.values()) {
            if (!f.variable.equals("program")) {
                facts.add(new Condition(f.variable, "=", f.value));
            }
        }
        boolean agree = fired != null
                && read("Does the counselor agree? (yes/no) > ").equals("yes");
        if (agree) {
            kb.learn(new Case(facts, result.conclusions(), 100), "counselor");
            kb.save(path);
            System.out.println("Good. The case is recorded as checked.");
            return;
        }
        String program = read("Which program does the counselor suggest? "
                + "(Enter if none) > ");
        if (program.isEmpty()) {
            System.out.println("Then there is nothing to learn this time.");
            return;
        }
        while (true) {
            String extra = read("Another fact that matters? "
                    + "(variable = value, Enter when done) > ");
            if (extra.isEmpty()) {
                break;
            }
            try {
                facts.add(Condition.parse(extra));
            } catch (VcirsException e) {
                System.out.println("Please write it as: variable = value");
            }
        }
        Case c = new Case(facts, List.of(new Conclusion("program", program)),
                100);
        if (fired != null) {
            for (String line : kb.differences(fired.id, c).lines()) {
                System.out.println(line);
            }
        }
        LearnResult learned;
        try {
```

Listing 4.5 (Continued) *StudyAdvisor.java: the study-program advisor on vcirs-core.*

```
            learned = kb.learn(c, "counselor");
        } catch (VcirsException e) {
            System.out.println("Cannot learn: " + e.getMessage());
            return;
        }
        System.out.println("Learned: " + learned);
        kb.save(path);
        System.out.println("Saved " + path + ": " + kb.summary());
    }
}
```

Here is the life of one consultation, as `main` runs it.

1. **Open the knowledge base.** `openKb` loads `study-kb.json`, or, the first time, creates it: it **declares** the eight variables of `QUESTIONS` — each with the question to ask and its usual

answers — and then feeds the seed cases to the engine with `seed.build(k)`. Declaring a variable is how the Variable table of **Figure 4.2** gets its questions. A variable can be declared before any node uses it: `family farm` is declared here, although no seed case mentions it, so that the advisor already knows how to ask about it on the day a node needs it.

2. **Consult.** `kb.infer(Facts.none(), StudyAdvisor::ask)` starts with an empty working memory. Whenever a node needs a value the engine does not have, it calls `ask` with a `Question`. `ask` looks the variable up in the Variable table (`kb.variables`) and prints the counselor's question with its usual answers; an empty reply means "I do not know", and the node that needed the value does not fire.
3. **Suggest and explain.** The first path of the result, `result.paths.get(0)`, is the rule that answered. Its last node gives the suggested program; `intermediate()` gives the conclusions of the nodes above it, printed as "Also worth a look"; and `result.how()` explains the suggestion condition by condition, saying for each value where it came from.
4. **Check with the counselor.** If Bu Wati agrees, the case is given to `learn`, which finds the answer already right and records it in the history as a check. If she does not, the program asks for the program she would suggest and for any other fact that matters, shows the list of difference against the cornerstone case of the node that answered, and lets `learn` add the new knowledge. The file is saved.

The options at the end of the command line open the knowledge base in other ways: `--tree` prints it as a tree of nodes and exceptions, and `--tables` prints the three tables of the Variable-Centered Rule Structure, together with the most important variables and nodes (Section 4.8). `--echo` repeats every answer the program reads, so that a consultation typed into a file reads like a conversation when it is replayed; that is how the transcripts of this chapter were made. **Listing 4.6** is the same app in Python.

Listing 4.6 `study_advisor.py`: the study-program advisor in Python.

```
"""Study-program advisor: Bu Wati's knowledge, kept as variables, nodes
and rules.

    python study_advisor.py [--echo] [--tree] [--tables] [knowledge-base.json]

It asks a school leaver only the questions it needs, suggests a study
program with the programs met on the way, explains why, and learns from
the counselor when she disagrees. --tables prints the Variable-Centered
Rule Structure: where each variable is used, which rules share each node,
and every rule as a sequence of nodes.
"""

import os
import pathlib
import sys

# Tell Python where the vcirs-core engine is (two folders up, then across).
sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import (Case, Condition, Conclusion, KnowledgeBase, VcirsError,
                   casefile, report)

QUESTIONS = [
    ("likes", "What do you enjoy most?",
     ["computers", "living things", "numbers", "people", "building things",
      "drawing"]),
    ("wants", "After school, do you want to work soon or study longer?",
     ["work soon", "study longer"]),
    ("blood ok", "Are you all right with seeing blood and wounds?",
     ["yes", "no"]),
    ("science score", "Your average report score in science (0-100)?", []),
    ("math score", "Your average report score in mathematics (0-100)?", []),
    ("patient with children", "Are you patient with small children?",
     ["yes", "no"]),
    ("drawing", "How good are you at drawing?", ["good", "average"]),
    ("family farm", "Does your family have a farm, orchard or fish pond?",
     ["yes", "no"]),
]


class Console:
    """Reads the user's answers; with echo on, prints them too."""
```

Listing 4.6 (Continued) *study_advisor.py: the study-program advisor in Python.*

```
    def __init__(self, kb, echo):
        self.kb = kb
        self.echo = echo

    def read(self, prompt):
        print(prompt, end="")
        text = sys.stdin.readline().strip()
        if self.echo:
            print(text)
        return text

    def ask(self, question):
        var = self.kb.variables.get(question.variable)
        text, choices = question.variable, question.choices
        if var is not None and var.question:
            text, choices = var.question, var.choices   # as Bu Wati wrote it
        if choices:
            text += " (" + "/".join(choices) + ")"
        return self.read(text + " > ") or None    # Enter = "I do not know"


def open_kb(path):
    """Load the knowledge base, or build it from the seed cases."""
    if os.path.exists(path):
        return KnowledgeBase.load(path)
    seed = os.path.join(os.path.dirname(path), "study-seed-cases.json")
    name, description, steps = casefile.read_cases(seed)
    kb = KnowledgeBase(name, description)
    for var, question, choices in QUESTIONS:
        kb.declare(var, question, choices)
    casefile.build(kb, steps)
    kb.save(path)
    return kb


def print_tables(kb):
    print("Where each variable is used (the Variable table):")
    for line in report.variable_sharing(kb):
        print("  " + line)
    print("Which rules share each node:")
    for line in report.node_sharing(kb):
        print("  " + line)
    print("Every rule as a sequence of nodes (the Rule Structure):")
    for line in report.rules(kb):
        print("  " + line)
    usage = kb.usage()
```

Listing 4.6 (Continued) *study_advisor.py: the study-program advisor in Python.*

```
    print("Most important variables: " + ", ".join(usage.important_variables()))
    print("Most important nodes: " + ", ".join(usage.important_nodes()))


def main(args):
    echo = "--echo" in args
    files = [a for a in args if not a.startswith("--")]
    path = files[0] if files else "../data/study-kb.json"
    kb = open_kb(path)
    if "--tree" in args:
        for line in report.tree(kb):
            print(line)
        return
    if "--tables" in args:
        print_tables(kb)
        return
    con = Console(kb, echo)
    print("Study-program advisor: " + str(len(kb.nodes))
          + " pieces of advice so far.")
    print("Answer each question; press Enter if you do not know.")
    result = kb.infer({}, con.ask)
    fired = result.paths[0].last() if result.paths else None
    if fired is not None and result.conclusions():
        print("Suggested program: " + result.conclusions()[0].value)
        on_way = result.paths[0].intermediate()
        if on_way:
            print("Also worth a look: "
                  + ", ".join(c.value for c in on_way))
        print("Why:")
        for line in result.how():
            print("  " + line)
    else:
        print("I have no suggestion for this student yet.")

    facts = [Condition(f.variable, "=", f.value)
             for f in result.memory.facts.values() if f.variable != "program"]
    agree = fired is not None and con.read(
        "Does the counselor agree? (yes/no) > ") == "yes"
    if agree:
        kb.learn(Case(facts, list(result.conclusions())), "counselor")
        kb.save(path)
        print("Good. The case is recorded as checked.")
        return
    program = con.read("Which program does the counselor suggest? "
                       "(Enter if none) > ")
    if not program:
```

Listing 4.6 (Continued) *study_advisor.py: the study-program advisor in Python.*

```
        print("Then there is nothing to learn this time.")
        return
    while True:
        extra = con.read("Another fact that matters? "
                         "(variable = value, Enter when done) > ")
        if not extra:
            break
        try:
            facts.append(Condition.parse(extra))
        except VcirsError:
            print("Please write it as: variable = value")
    case = Case(facts, [Conclusion("program", program)])
    if fired is not None:
        for line in kb.differences(fired.id, case).lines():
            print(line)
    try:
        learned = kb.learn(case, "counselor")
    except VcirsError as e:
        print("Cannot learn: " + str(e))
        return
    print("Learned: " + str(learned))
    kb.save(path)
    print("Saved " + path + ": " + kb.summary())


if __name__ == "__main__":
    main(sys.argv[1:])
```

The Python asker is the method `ask` of a small `Console` object, which also holds the knowledge base so that it can look up each variable's question. Any function that takes a question and returns an answer would do; Chapter 10 puts the same knowledge base behind a web page, and only the asker changes.

4.7 Four Students at Bu Wati's Desk

Compile and run the advisor from the `Chapter04/java` folder:

```
javac -d out -sourcepath ../../vcirs-core/java/src *.java
java -cp out StudyAdvisor --echo < ../data/session-1.txt
```

or, from `Chapter04/python`, `python study_advisor.py --echo < ../data/session-1.txt`. (Windows PowerShell has no `<`; type `Get-Content ..\data\session-1.txt | java -cp out StudyAdvisor --echo` instead.) The `data` folder holds the four consultations of this section, one answer per line. Run them in order on a fresh knowledge base — delete `study-kb.json` first if it exists — and you will see exactly these transcripts, from Java and from Python alike, and the same `study-kb.json`, byte for byte.

Session 1: Laras. She loves living things, she has helped her mother clean her little brother's scraped knees for years, and her science average is 88.

```
Study-program advisor: 13 pieces of advice so far.
Answer each question; press Enter if you do not know.
What do you enjoy most? (computers/living things/numbers/people/building things/drawing) >
living things
Are you all right with seeing blood and wounds? (yes/no) > yes
Your average report score in science (0-100)? > 88
Suggested program: medicine
Also worth a look: biology, nursing
Why:
  medicine#1 fired because:
    likes = living things  (answered by the user, CF 100)
    blood ok = yes  (answered by the user, CF 100)
    science score >= 85  (science score = 88, answered by the user, CF 100)
    so program = medicine  [rule CF 100, conclusion CF 100]
Does the counselor agree? (yes/no) > yes
Good. The case is recorded as checked.
```

Three questions, no more. The first top-level node, informatics, needs `likes = computers`; the engine asks about `likes`, and the answer *living things* makes it fail at once. The biology node needs the same variable, which is already in working memory, so it fires without a new question. Its first exception, nursing, needs `blood ok`, which is asked; its exception, medicine, needs `science score`, which is asked and compared as a number, 88 against 85. The explanation shows exactly that: `science score >= 85  (science score = 88, ...)`. The suggestion is medicine, and biology and nursing — the nodes met on the way — are offered as further programs to look at. This is the part of the output Laras's family should read most carefully. Medicine at a state university is among the most competitive choices of all; nursing and biology are programs she also fits, and **Figure 4.1** says that the second choice matters almost as much as the first. Bu Wati agrees, and the case is recorded as checked.

Session 2: Bima. He loves computers, and he wants to work soon: his father drives an online motorcycle taxi, and Bima is the eldest of four.

```
Study-program advisor: 13 pieces of advice so far.
Answer each question; press Enter if you do not know.
What do you enjoy most? (computers/living things/numbers/people/building things/drawing) >
computers
After school, do you want to work soon or study longer? (work soon/study longer) > work
soon
Suggested program: D3 informatics
Also worth a look: informatics
Why:
  D3 informatics#1 fired because:
    likes = computers  (answered by the user, CF 100)
    wants = work soon  (answered by the user, CF 100)
    so program = D3 informatics  [rule CF 100, conclusion CF 100]
Does the counselor agree? (yes/no) > yes
Good. The case is recorded as checked.
```

Two questions. The informatics node fires on the first answer, its exception D3 informatics fires on the second, and the suggestion is the three-year diploma, with the four-year degree as the program worth a look. For Bima this is more than a technicality. Under the SNBT 2026 rules a student who chooses three or four programs must include vocational programs — and, with four choices, at least one D3 [3] — so a D3 is not a consolation prize; it is part of a well-built plan.

Session 3: Rudi, and a gap in the knowledge. Rudi loves living things too, but he faints at the sight of blood. His family grows chili and keeps a small durian orchard behind the house, and he has helped on the land since he could walk.

```
Study-program advisor: 13 pieces of advice so far.
Answer each question; press Enter if you do not know.
What do you enjoy most? (computers/living things/numbers/people/building things/drawing) >
living things
Are you all right with seeing blood and wounds? (yes/no) > no
Suggested program: biology
Why:
  biology#1 fired because:
    likes = living things  (answered by the user, CF 100)
    so program = biology  [rule CF 100, conclusion CF 100]
Does the counselor agree? (yes/no) > no
Which program does the counselor suggest? (Enter if none) > agrotechnology
Another fact that matters? (variable = value, Enter when done) > family farm = yes
Another fact that matters? (variable = value, Enter when done) >
Differences with the cornerstone case of biology#1:
  A (only in the old case): (none)
  B (only in the new case): blood ok = no, family farm = yes
Learned: refined: agrotechnology#1 added as an exception of biology#1
Saved ../data/study-kb.json: Study-program advisor: 9 variables, 14 nodes, 14 rules
```

The advisor suggests biology: the nursing exception needs `blood ok = yes` and fails, so the biology node's own conclusion stands. Bu Wati disagrees. For a boy who has grown up on a farm and loves it, she would suggest **agrotechnology**, the program that studies how crops are grown, protected and improved. She types it, and one more fact that no question has asked yet: `family farm = yes`.

The list of difference compares Rudi with the cornerstone case of `biology#1` — the seed case with only `likes = living things`. Section A, the facts only the old case has, is empty. Section B holds the two facts only Rudi has: `blood ok = no` and `family farm = yes`. `learn` builds the new node from both sections — option 3 of the six options met in Chapter 3 — and hangs it under `biology#1`. The knowledge base now has fourteen nodes.

Look closely at the new node, because it is not quite what Bu Wati meant. She would advise agrotechnology to a farm child whether or not he is afraid of blood; the condition `blood ok = no` came along only because it happened to be different from the cornerstone case. The node is *correct* — it will never give a wrong answer to a student like Rudi — but it is *narrower* than it needs to be: a farm child who is all right with blood will be sent to nursing and never reach it. Choosing which differences really matter is the expert's job, and Chapter 5 gives the expert the tools to do it: the six options, a condition chosen by hand, and a check of the cases that a new node would affect.

Session 4: Wulan, a week later. Wulan's parents have a fish pond and a small orchard; she loves living things, and like Rudi she would rather not see blood.

```
Study-program advisor: 14 pieces of advice so far.
Answer each question; press Enter if you do not know.
What do you enjoy most? (computers/living things/numbers/people/building things/drawing) >
living things
Are you all right with seeing blood and wounds? (yes/no) > no
Does your family have a farm, orchard or fish pond? (yes/no) > yes
Suggested program: agrotechnology
Also worth a look: biology
Why:
  agrotechnology#1 fired because:
    likes = living things  (answered by the user, CF 100)
    blood ok = no  (answered by the user, CF 100)
    family farm = yes  (answered by the user, CF 100)
    so program = agrotechnology  [rule CF 100, conclusion CF 100]
Does the counselor agree? (yes/no) > yes
Good. The case is recorded as checked.
```

Now the advisor asks the new question — `Does your family have a farm, orchard or fish pond?` — worded as Bu Wati declared it, because the node learned in session 3 needs the variable `family farm`. The suggestion is agrotechnology, with biology on the way, and the explanation lists the three facts that made it. Nobody changed a line of code between Rudi and Wulan.

`--tree` prints the knowledge base after the four students:

```
(root)
+- informatics#1: likes = computers -> program = informatics
|  +- D3 informatics#1: wants = work soon -> program = D3 informatics
+- biology#1: likes = living things -> program = biology
|  +- nursing#1: blood ok = yes -> program = nursing
|  |  +- medicine#1: science score >= 85 -> program = medicine
|  +- agrotechnology#1: blood ok = no, family farm = yes -> program = agrotechnology
+- accounting#1: likes = numbers -> program = accounting
|  +- statistics#1: math score >= 85 -> program = statistics
|  +- D3 accounting#1: wants = work soon -> program = D3 accounting
+- communication science#1: likes = people -> program = communication science
|  +- primary teaching#1: patient with children = yes -> program = primary teaching
+- civil engineering#1: likes = building things -> program = civil engineering
|  +- architecture#1: drawing = good, math score >= 75 -> program = architecture
+- visual communication design#1: likes = drawing -> program = visual communication design
```

Compare it with **Figure 4.2**: the same shape, with one new exception under `biology#1` beside `nursing#1`. The top-level nodes are tried in the order they were taught, and so are the exceptions of each node.

4.8 Reading the Structure

The tree shows the knowledge the way the engine *walks* it. The three tables show it the way the engine *keeps* it, and that is where thinking in variables pays off. `--tables` prints them:

```
Where each variable is used (the Variable table):
  VariableID             NumOfNodeIDs  NodeID Used                    VariableOrder
  ---------------------  ------------  ----------------------------  -------------
  science score          1             medicine#1                     1
  patient with children  1             primary teaching#1             1
  drawing                1             architecture#1                 1
  family farm            1             agrotechnology#1               2
  wants                  2             D3 informatics#1               1
                                       D3 accounting#1                1
  blood ok               2             nursing#1                      1
                                       agrotechnology#1               1
  math score             2             statistics#1                   1
                                       architecture#1                 2
  likes                  6             informatics#1                  1
                                       biology#1                      1
                                       accounting#1                   1
                                       communication science#1        1
                                       civil engineering#1            1
                                       visual communication design#1  1
Which rules share each node:
  NodeID                         NumOfRuleIDs  RuleID Used                    NodeOrder
  -----------------------------  ------------  ----------------------------  ---------
  D3 informatics#1               1             D3 informatics#1               2
  medicine#1                     1             medicine#1                     3
  statistics#1                   1             statistics#1                   2
  D3 accounting#1                1             D3 accounting#1                2
  primary teaching#1             1             primary teaching#1             2
  architecture#1                 1             architecture#1                 2
  visual communication design#1  1             visual communication design#1  1
  agrotechnology#1               1             agrotechnology#1               2
  informatics#1                  2             informatics#1                  1
                                               D3 informatics#1               1
  nursing#1                      2             nursing#1                      2
                                               medicine#1                     2
  communication science#1        2             communication science#1        1
                                               primary teaching#1             1
  civil engineering#1            2             civil engineering#1            1
                                               architecture#1                 1
  accounting#1                   3             accounting#1                   1
                                               statistics#1                   1
                                               D3 accounting#1                1
  biology#1                      4             biology#1                      1
                                               nursing#1                      1
                                               medicine#1                     1
                                               agrotechnology#1               1
Every rule as a sequence of nodes (the Rule Structure):
  informatics#1: informatics#1
  D3 informatics#1: informatics#1 -> D3 informatics#1
  biology#1: biology#1
  nursing#1: biology#1 -> nursing#1
  medicine#1: biology#1 -> nursing#1 -> medicine#1
  accounting#1: accounting#1
  statistics#1: accounting#1 -> statistics#1
  D3 accounting#1: accounting#1 -> D3 accounting#1
  communication science#1: communication science#1
  primary teaching#1: communication science#1 -> primary teaching#1
  civil engineering#1: civil engineering#1
  architecture#1: civil engineering#1 -> architecture#1
  visual communication design#1: visual communication design#1
  agrotechnology#1: biology#1 -> agrotechnology#1
Most important variables: likes
Most important nodes: biology#1
```

The Variable table. The first table lists every variable used by some node, with the number of nodes that use it (NumOfNodeIDs), each of those nodes and the variable's position in it (VariableOrder). It is sorted from the least shared to the most shared variable, the way the thesis

prints it [5]. Four variables are used by one node each; `wants`, `math score` and — since Rudi — `blood ok` are used by two; and `likes` by six. Two things are worth noticing. `blood ok` is now shared by `nursing#1` (as `yes`) and by `agrotechnology#1` (as `no`): the same question serves two pieces of advice. And `family farm`, declared from the start, appears only now, because only now does a node use it.

Which rules share each node. The second table lists every node with the number of rules that contain it (NumOfRuleIDs), each of those rules and the node's position in it (NodeOrder). A node that is the last node of its own rule only, such as `medicine#1`, is used by one rule. `biology#1` is used by four: its own, and those of nursing, medicine and agrotechnology, where it is always node 1. The more rules share a node, the more of the knowledge depends on it: if `biology#1` were wrong, four suggestions would be wrong with it [5].

Every rule as a sequence of nodes. The third table is the Rule Structure: fourteen rules, one per node, each a sequence of nodes from the top level down. It is the same information as the tree, written the way Chapter 8's forward and backward reasoning will read it.

Most important variables and nodes. The last two lines are the first result of what the thesis calls **variable analysis**: the most important variables are those shared by the most nodes, and the most important nodes are those shared by the most rules [5]. Here the answer is `likes` and `biology#1` — no surprise to Bu Wati, who knows that what a student loves is the first thing to ask, and that most of her difficult cases are students who love living things. In a knowledge base of a few hundred nodes built by several counselors, nobody would know the answer without the tables. Chapter 6 uses these important nodes and variables as the starting point for **rule generation**: the engine proposes new rules and nodes built around them, and the expert confirms the ones that make sense. Chapter 7 adds the other half, **value analysis**, which weighs each variable by its Credit and its position and turns the counts into usage rates.

4.9 Making It Better

A better order of questions. The advisor asks `likes` first because the first seed case happened to begin with it. For this knowledge base that is also the best order, but in general it need not be. The engine can rank the variables by how much they are used — `kb.usage().questionOrder()` — and an app can ask the most useful questions first. Chapter 7 does this for a shop's customer messages.

Numbers the counselor chose. The seed cases use `science score >= 85` because Bu Wati wrote them that way. When a student is taught at the desk, however, the facts come from the consultation, and a number becomes an exact condition such as `science score = 91`, which only a student with exactly 91 will ever match. When a new node needs a number, the counselor should choose the condition herself, with `refine` and a list of chosen conditions; Chapter 3's Exercise 3.6 showed how, and Chapter 5 makes it part of the teaching screen.

Backup programs. The "Also worth a look" line lists only the programs met on the way down one rule. A student may fit two rules at once — Sari, who loves numbers, has a mathematics score of 90 and wants to work soon, fits both statistics and D3 accounting. `kb.infer(facts, asker, true)` (Python: `multiple=True`) makes the engine follow every branch that fits and return one path per suggestion, which is exactly what a list of up to four SNBT choices needs. Exercise 4.6 asks you to try it.

Checking a plan against the rules. Once a student has a list of programs, a few plain rules can check it against the SNBT 2026 regulations: at most four programs, at most two academic ones, and with three or four choices the right mix of vocational programs and at least one D3 in some combinations [3]. These rules do not come from cases; they come from a regulation, and they are best written as ordinary code, like the saving rules of Chapter 2. Exercise 4.9 builds the checker.

Tests. `StudyTest.java` and `test_study.py` check the pieces of this chapter: that `split` cuts conditions correctly, that the hand-made structure has thirteen nodes and the right rules and answers, that the seed knowledge base has thirteen nodes with `medicine#1` under `nursing#1` and `likes` as its most important variable, that `learn` hangs Rudi's case under `biology#1`, and that the plan checker of Exercise 4.9 accepts and rejects the right plans. They end with:

```
all tests passed
```

4.10 Using It for Real

In a school. Run the advisor on the counseling room's computer, or on a laptop at the school's orientation day, and let students consult it before they see the counselor. It does not replace the conversation; it prepares it. A student who arrives with "the advisor suggested nursing, with biology on the way — and I did not know nursing had a D3" asks better questions and gets more out of fifteen minutes. When the counselor disagrees with a suggestion, she teaches the advisor there and then; after one school year the knowledge base holds her experience of that year.

Start from last year's leavers. The best seed cases are real students: last year's class XII, where they applied, where they were accepted and how they are doing. Write the ten or fifteen most typical ones as cases, as in `study-seed-cases.json`, and let everything else be taught at the desk. Keep the facts to what a student can answer honestly about herself — what she enjoys, her report averages, whether she can stand the sight of blood, what her family needs — and do not store names in the cases. The knowledge base is meant to be shared with the next counselor; it should hold knowledge about kinds of students, not about particular people.

Keep the official facts outside the advisor. Programs, their capacity (*daya tampung*), their requirements and the rules of SNBP and SNBT change every year [3], [2]. They belong on the SNPMB portal and on each university's admission page, not in a knowledge base of counselor's advice. What the advisor keeps is the knowledge that does *not* change every year: which kinds of students thrive in which kinds of programs.

The money question. For many families, including Pak Joko's, the first question is not *which program* but *can we afford it*. Say it plainly to every student: a third of the students accepted through SNBT 2026 were KIP Kuliah applicants [4]. A variable such as `needs KIP Kuliah` with a node that says *apply for KIP Kuliah at registration* is one of the most useful pieces of advice a school can give, and Exercise 4.8 asks where in the advisor it belongs.

Beyond schools. Nothing in `StudyAdvisor` is about study programs except the list of questions and the seed file. A vocational training center can advise which course suits a job seeker; a cooperative can advise which savings product suits a member; a mosque's youth group can advise which volunteer role suits a newcomer. Each one writes its variables first, as in **Table 4.1**, then a dozen seed cases, and teaches the rest as it goes. Chapter 14 builds a tool that does this without programming.

4.11 Common Mistakes, and How to Avoid Them

Two names for one variable. `likes`, `enjoys` and `hobby` in different cases become three variables, and the Variable table can no longer show where the knowledge about interests lives. *Avoid it:* write the table of variables first (**Table 4.1**) and keep to it; declare every variable with its question.

Two spellings for one value. `living things`, `Living things` and `living-things` are three different values to the engine. *Avoid it:* declare the usual values with each variable; the advisor shows them in parentheses, and students choose among them.

One variable for two things. A variable `interest` with values such as `computers`, `yes` and `work soon` mixes three questions into one, and no condition on it can be read. *Avoid it:* one variable, one question.

Exact numbers where a threshold is meant. `math score = 90` matches only 90. *Avoid it:* use `>=` or `<=` in the seed cases, and choose the condition by hand when a new node needs a number.

Forgetting that the order of conditions matters. The engine checks, and asks, conditions in the order they are written. A node written as `science score >= 85, likes = living things` would ask every student for a science score before asking what she loves. *Avoid it:* put the condition that rules out the most students first.

Accepting every difference. In session 3, `blood ok = no` came into Rudi's node only because it differed from the cornerstone case, and made the node narrower than Bu Wati meant. *Avoid it:* read the list of difference before accepting a new node; Chapter 5 shows how to keep only the facts that explain the new advice.

Treating the suggestion as a decision. The advisor knows what Bu Wati knows, no more. *Avoid it:* present its output as "programs worth a look", and always check the official requirements [2].

Putting yearly rules into the knowledge base. A node that says *choose at most two S1 programs* is a regulation, not a counselor's experience, and it will be wrong the year the regulation changes. *Avoid it:* keep regulations in ordinary code (Exercise 4.9), and experience in the knowledge base.

4.12 Chapter Summary

- Advice becomes knowledge a computer can use when it is split into **variables** (what is asked), **values** (the answers), **conditions** (a variable, one of the operators `=`, `<>`, `<`, `<=`, `>`, `>=`, and a value) and **conclusions** (**Table 4.1**).
- A **node** is a short ordered sequence of conditions with conclusions, named after its first conclusion (`nursing#1`); an exception holds only its own conditions. A **rule** is a sequence of nodes from the top level down, named after its last node; its conclusion is that of its last node, and the nodes above give intermediate conclusions.
- The **Variable-Centered Rule Structure** keeps three linked tables — variables, nodes, rules (**Figure 4.2**, **Figure 4.3**). Indexing the advice by variable showed its vocabulary and its repetitions (**Listing 4.1**, **Listing 4.2**); keeping it as nodes cut 22 conditions to 14 and made it searchable by variable, node and rule (**Listing 4.3**, **Listing 4.4**).
- vcirs-core keeps the same tables and adds questions asked only when needed, learning from cases, explanations and a JSON file shared by Java and Python (**Table 4.2**).
- The **study-program advisor** (**Listing 4.5**, **Listing 4.6**) asked two or three questions per student, suggested a program with backups met on the way, explained every suggestion, learned

a new exception from the counselor with a new variable, and asked the new question of the next student.

- **Variable analysis** reads the tables: the most shared variable (`likes`) and the most shared node (`biology#1`) are the most important, and they are where Chapter 6's rule generation starts. With fewer than three in ten SNBT 2026 registrants accepted, and nearly all of them through their first two choices (**Figure 4.1**), helping students choose well is worth the effort.

4.13 Exercises

4.1 Split each of these pieces of advice into variables, conditions and a conclusion, using the variables of **Table 4.1** wherever you can. Say whether each one would be a top-level node or an exception, and of which node. (a) *A student who loves people and is patient with small children should look at primary teaching.* (b) *If a student loves building things, draws well and has a mathematics average of at least 75, architecture is worth a look.* (c) *A student who loves computers and also draws well should look at visual communication design rather than informatics.*

4.2 By hand, using the structure printed by **Listing 4.4**, find the path and the suggestion for: (a) `likes = building things`, `drawing = good`, `math score = 70`; (b) `likes = people`, `patient with children = yes`; (c) `likes = numbers`, `math score = 90`, `wants = work soon`.

4.3 Write out, as one flat IF-THEN rule, the rule `medicine#1`. Then suppose Bu Wati teaches three more exceptions under `medicine#1`, each with one condition. How many conditions does the structure gain, and how many would a flat list of complete rules gain?

4.4 After session 3, the Variable table says that `blood ok` is used by two nodes. Step 1 complained that `blood ok = yes` was written twice in the flat list. Is the new sharing the same problem? Explain.

4.5 In session 3, suppose Rudi had pressed Enter at the question about blood instead of answering *no*. Which facts would the case have held, what would the list of difference have been, and what node would `learn` have made? Would that node answer a farm child who *is* all right with blood?

4.6 Sari loves numbers, has a mathematics average of 90 and wants to work soon. Write a small program that runs the seed knowledge base on her facts twice, once normally and once with `multiple` switched on, and prints every rule that answers. Explain the difference, and why it matters for an SNBT plan.

4.7 Which node would become the most important node, in the sense of Section 4.8, if Bu Wati taught three more exceptions under `D3 informatics#1`? Count the rules that would share each candidate.

4.8 Bu Wati wants every student who needs financial support to be told to register for KIP Kuliah. Where should this advice go? Consider (a) a case with the conclusion `note = apply for KIP Kuliah` taught to the advisor, (b) a plain rule in the app's code, and (c) a second small knowledge base. Explain what `learn` would do with (a).

4.9 Write a program that checks a student's list of SNBT 2026 choices — each a program with its level, S1, D4 or D3 — against the rules of Section 4.1 and Section 4.9 [3], and prints either *allowed* or what is wrong. Try it on five plans.

4.14 Solutions to the Exercises

Solution 4.1. (a) Variables `likes` and `patient with children`; conditions `likes = people` and `patient with children = yes`; conclusion `program = primary teaching`. It is an exception of `communication science#1`, holding only `patient with children = yes` — exactly the node of Step 2. (b) Variables `likes`, `drawing` and `math score`; conditions `likes = building things`, `drawing = good`, `math score >= 75`; conclusion `program = architecture`; an exception of `civil engineering#1` holding the last two conditions, in that order. (c) Variables `likes` and `drawing`; conditions `likes = computers` and `drawing = good`; conclusion `program = visual communication design`. It is an exception of `informatics#1` holding only `drawing = good`. The variable `drawing` is then shared by two nodes, `architecture#1` and the new one — a question already declared serves a second piece of advice.

Solution 4.2. (a) `civil engineering#1` fires on `likes = building things`. Its exception `architecture#1` checks `drawing = good` (true) and then `math score >= 75`, which is false for 70, so it does not fire. The path is `civil engineering#1`, and the suggestion is civil engineering. (b) `communication science#1` fires, then `primary teaching#1`: primary teaching. (c) `accounting#1` fires; its first exception, `statistics#1`, needs `math score >= 85`, true for 90, and fires. The search goes down, not sideways, so `D3 accounting#1` is never tried: the suggestion is statistics. Exercise 4.6 shows how to see both.

Solution 4.3. IF `likes = living things` AND `blood ok = yes` AND `science score >= 85` THEN `program = medicine`. Three exceptions of one condition each add three conditions to the structure. In a flat list each new rule must repeat the three conditions of `medicine#1` and add its own: 3 × 4 = 12 conditions.

Solution 4.4. No. The flat list wrote the *same* condition, `blood ok = yes`, twice, as part of two rules, so a change to it had to be made in two places. After session 3 the two nodes use the same *variable* with different values — `blood ok = yes` for nursing and `blood ok = no` for agrotechnology — as two different pieces of advice. The variable is kept once in the Variable table, with its question; only the conditions differ. That is sharing, and the Variable table is what makes it visible.

Solution 4.5. An unknown answer is not put into working memory, so the case would hold `likes = living things` from the consultation and `family farm = yes` typed by Bu Wati. The list of difference against the cornerstone case of `biology#1` would have an empty section A and `family farm = yes` alone in section B, and `learn` would make the node `agrotechnology#1: family farm = yes -> program = agrotechnology` under `biology#1` — the node Bu Wati meant. It still would not answer a farm child who is all right with blood: `nursing#1` comes first among the exceptions of `biology#1`, and it fires for her, so the engine goes down into nursing and never tries the agrotechnology node. The order of exceptions is knowledge too; Chapter 5 shows how the expert handles such a case.

Solution 4.6. The solution programs are `BackupPrograms.java` and `backup_programs.py`. The Python version is:

```
"""Exercise 4.6: every program a student fits, with multiple=True."""

import pathlib
import sys

sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import KnowledgeBase, casefile

name, description, steps = casefile.read_cases("../data/study-seed-cases.json")
kb = casefile.build(KnowledgeBase(name, description), steps)
sari = {"likes": "numbers", "math score": "90", "wants": "work soon"}
for multiple in (False, True):
    result = kb.infer(sari, multiple=multiple)
    print("every rule that fits:" if multiple else "the first rule only:")
    for path in result.paths:
        print("  " + path.conclusions()[0].value + "  (rule "
              + " -> ".join(n.id for n in path.nodes) + ")")
```

It prints:

```
the first rule only:
  statistics  (rule accounting#1 -> statistics#1)
every rule that fits:
  statistics  (rule accounting#1 -> statistics#1)
  D3 accounting  (rule accounting#1 -> D3 accounting#1)
```

Normally the engine stops at the first rule that answers, `statistics#1`, because the first exception of accounting that fires wins. With `multiple` on, it follows every exception that fires and returns one path per rule, as Multiple Classification RDR does: statistics *and* D3 accounting. For an SNBT plan of up to four choices, which must mix academic and vocational programs, the second list is exactly what Sari needs to talk over with Bu Wati.

Solution 4.7. Today `biology#1` is shared by four rules (after session 3) and `accounting#1` by three, `informatics#1` by two. Three new exceptions under `D3 informatics#1` create three new rules, each of which starts `informatics#1 -> D3 informatics#1 -> …`. Then `informatics#1` is shared by 2 + 3 = 5 rules, `D3 informatics#1` by 1 + 3 = 4, and `biology#1` still by 4. The most important node would be `informatics#1`.

Solution 4.8. (a) is a trap. `learn` compares the knowledge base's answer (`program = ...`) with the case's conclusion (`note = apply for KIP Kuliah`); they differ, so it adds an exception under the node that answered, and from then on that node's students get the note *instead of* a program. The note has nothing to do with which program a student fits. (b) is the simplest correct answer: a plain rule in the app, like the rules of Chapter 2, printed after the suggestion — *if needs KIP Kuliah = yes, register for KIP Kuliah when you register for SNBP or SNBT*. (c) is right when the school has many such pieces of advice about costs, housing or scholarships: a second small knowledge base, consulted after the first, keeps them apart and can be taught case by case like the first.

Solution 4.9. The solution programs are `SnbtChoices.java` and `snbt_choices.py`. The Python version is:

```
"""Exercise 4.9: check a plan of SNBT 2026 program choices against the
rules SNPMB published for 2026."""


def problems(choices):
    """choices: a list of (program, level) with level "S1" (academic),
    "D4" or "D3" (vocational). An empty answer means the plan is allowed."""
    n = len(choices)
    academic = sum(1 for _, level in choices if level == "S1")
    d3 = sum(1 for _, level in choices if level == "D3")
    out = []
    if n < 1 or n > 4:
        out.append("choose one to four programs")
    if academic > 2:
        out.append("at most two academic (S1) programs")
    if n == 3 and academic == 0 and d3 == 0:
        out.append("three vocational choices need at least one D3 program")
    if n == 4 and academic == 0:
        out.append("four choices need one or two academic programs")
    if n == 4 and d3 == 0:
        out.append("four choices need at least one D3 program")
    return out


PLANS = [
    ("Laras, plan 1", [("medicine", "S1"), ("nursing", "S1")]),
    ("Laras, plan 2", [("medicine", "S1"), ("nursing", "S1"),
                       ("biology", "S1")]),
    ("Laras, plan 3", [("medicine", "S1"), ("nursing", "S1"),
                       ("D3 nursing", "D3"), ("applied midwifery", "D4")]),
    ("Bima, plan 1", [("software engineering", "D4"), ("multimedia", "D4"),
                      ("computer networks", "D4")]),
    ("Bima, plan 2", [("informatics", "S1"), ("D3 informatics", "D3"),
                      ("software engineering", "D4"), ("multimedia", "D4")]),
]

if __name__ == "__main__":
    for name, plan in PLANS:
        found = problems(plan)
        print(name + ": " + ("allowed" if not found else "; ".join(found)))
```

It prints:

```
Laras, plan 1: allowed
Laras, plan 2: at most two academic (S1) programs
Laras, plan 3: allowed
Bima, plan 1: three vocational choices need at least one D3 program
Bima, plan 2: allowed
```

The rules come straight from the SNPMB page for 2026 [3]: one or two choices are free; three choices may be two academic and one vocational, one academic and two vocational, or three vocational with at least one D3; four choices must be two academic and two vocational, or one academic and three vocational, always with at least one D3. The function writes these as five checks. Because regulations change, the checks sit in one small function, easy to update next year, and not in the knowledge base.

References

[1] Universitas Negeri Yogyakarta, “Seleksi Nasional Berdasarkan Prestasi (SNBP) 2026,” PMB UNY, 2026. [Online]. Available: https://pmb.uny.ac.id/jalur/SNBP

[2] Kementerian Pendidikan Dasar dan Menengah, “Panduan umum dan ketentuan UTBK-SNBT 2026,” Pusat Informasi Rumah Pendidikan, 2026. [Online]. Available:

https://pusatinformasi.rumahpendidikan.kemendikdasmen.go.id/hc/id/articles/55040921152537-Panduan-Umum-dan-Ketentuan-UTBK-SNBT-2026

[3] SNPMB, "Informasi umum UTBK-SNBT 2026," Seleksi Nasional Penerimaan Mahasiswa Baru, 2026. [Online]. Available: https://snpmb.id/utbk-snbt/informasi-umum

[4] SNPMB, "Siaran pers 02/sipers/snpmb/V/2026 tentang pengumuman hasil SNBT 2026," May 25, 2026. [Online]. Available: https://files.snpmb.id/web2026/02_Sipers%20Pengumuman%20SNBT%202026.pdf

[5] M. M. I. Subakti, "A Variable-Centered Intelligent Rule System," M.Sc.Eng. thesis, Dept. of Computer Science and Information Engineering, Taipei, Taiwan, Jun. 2005.

[6] I. Subakti, "A variable-centered intelligent rule system," in *Proc. 1st Annual Int. Conf. Information and Communication Technology Seminar (ICTS 2005)*, vol. 1, no. 1, Surabaya, Indonesia, Aug. 2005, pp. 167–174.

[7] I. Subakti, *Variable-Centered Intelligent Rule System*, Indonesian ed. (resume). Surabaya, Indonesia: Dept. of Informatics, Institut Teknologi Sepuluh Nopember, 2006.

[8] I. Subakti and O. Wijayanto, "Penerapan konsep fuzzy dalam Variable-Centered Intelligent Rule System (studi kasus: pemilihan jurusan di Chinese University of Hongkong)," Dept. of Informatics, Institut Teknologi Sepuluh Nopember, Surabaya, Indonesia.

[9] I. Subakti, "Some revisions in VCIRS and cases reconstructing perspectives," in *Proc. Information and Communication Technology Seminar (ICTS 2006)*, Surabaya, Indonesia, 2006.

Chapter 5

Health Self-Check Advisor: Case-by-Case Knowledge Refinement

By the end of this chapter you will be able to explain why a health self-check must put fixed **safety rules** before anything it has learned; compute, by hand and with the engine, the **list of difference** between a new case and a node's **cornerstone case**; say what each of the **six options** of VCIRS builds, and choose among them; pick the conditions of a new node yourself when you know better than the list of difference; check which **earlier cases** a new node would change before you add it; retire advice by **stopping** a node instead of deleting it; and run the **health self-check**, an app in Java and Python that asks the safety questions first, then gives every piece of self-care advice that applies with the conditions behind it, and lets the nurse at the puskesmas teach it — choosing the node, the option and the conditions — whenever its advice is missing or wrong.

5.1 Pak Darmo and Ners Ratna

Pak Darmo has pedaled a becak in the same neighborhood of Surabaya for thirty years. On Monday morning he woke up hot and aching, and by Wednesday he was still hot. His wife said *go to the puskesmas*; Pak Darmo said a becak that does not move earns nothing, and that a fever always passes. Across the street, his neighbor's son had a fever the same week and ended up in hospital with dengue.

Pak Darmo's choice is the choice millions of people make every week: *is this something I can look after at home, or do I need to see someone?* The question matters, because the common illnesses and the dangerous ones start in the same way. A fever is usually a passing infection, and sometimes dengue: in 2024 the world reported more than 14.6 million dengue cases and more than 12,000 deaths, and the dangerous warning signs of severe dengue — severe belly pain, vomiting that does not stop, bleeding gums or nose, blood in vomit or stool — often appear *after* the fever has gone down [1]. A headache is usually tiredness or too little water, and sometimes high blood pressure, which an estimated 1.4 billion adults aged 30 to 79 had in 2024, 44 percent of them without knowing it [2]. A cough is usually a cold, and sometimes tuberculosis, of which Indonesia had about one in ten of the world's new cases in 2024, second only to India [3]. Diarrhea is usually a bad meal, and it is also one of the leading killers of small children worldwide, which is why every health worker knows the value of oral rehydration salts [4].

At the puskesmas two streets away, Ners Ratna runs the monthly *posbindu*, the community post where adults have their blood pressure, weight and blood sugar checked. She has answered Pak Darmo's question a thousand times, in the same careful way: first the dangerous signs, which mean *go now*; then what you can do yourself; and always *when to come back*. She would like her careful way to reach the people who never come to the posbindu — the becak drivers, the market sellers, the parking attendants who cannot leave their spot. And she would like to keep improving it, one real patient at a time, because every week she meets a situation her advice did not foresee.

This chapter builds that self-check. It is the most careful application in this book so far, and it teaches the part of VCIRS that is about *care*: how an expert refines knowledge case by case, choosing exactly what to add and checking what it will change before she adds it.

> **PITFALL.** A self-check gives self-care advice and tells people when to see a health worker. It does not diagnose, it does not prescribe, and it never replaces a doctor, a nurse or a bidan. Every piece of advice in this chapter ends in the same direction: when in doubt, or when anything gets worse, go to the puskesmas or call 119 [5].

5.2 The Idea in Plain Words

5.2.1 Safety first, and not learned

The most important rules of a health self-check are not learned at all. They are written once, in the program, and asked before anything else:

- **Stroke signs.** A face that droops on one side, sudden weakness or numbness on one side of the body, slurred speech or sudden inability to speak, sudden trouble seeing, a sudden headache worse than any before. The Ministry of Health teaches these with the phrase **SeGeRa Ke RS** — *senyum tidak simetris, gerak tubuh melemah, bicara pelo, kebas, rabun, sakit kepala hebat* — which also says what to do: go to the hospital quickly, because treatment works best within about four and a half hours [6].
- **Chest pain or pressure, or trouble breathing.**
- **The warning signs of severe dengue** in a person with fever: severe belly pain, vomiting three or more times a day, bleeding gums or nose, blood in vomit or stool, great weakness or restlessness. A person with these signs needs the emergency room, not advice [1], [7].
- **Signs of losing too much water**: unable to keep water down, or passing very little urine [4].

If the answer to any of these is *yes*, the self-check says one thing — call **119**, the Ministry of Health's emergency number, or go to the nearest hospital emergency room (IGD) now — and asks nothing more [5]. (The general emergency number **112** also connects to help [8].) The self-check is also honest about who it is for: it is for adults who are not pregnant. For a child, or during pregnancy, it sends the person to the bidan or the puskesmas at once.

Why are these rules *fixed*, when the whole point of VCIRS is knowledge that grows? Because the thing that makes case-by-case learning powerful — any expert can change the answer for a new kind of case — is exactly what must never happen to a danger sign. A tired nurse at the end of a long day might teach an exception that, without her noticing, sends a certain kind of chest pain to "rest at home". A knowledge base that can learn can also learn a mistake. The safety rules therefore sit *before* the knowledge base, in plain code, where no teaching can reach them, and they change only when the program is changed on purpose, with the sources checked again. **Figure 5.1** shows the arrangement.

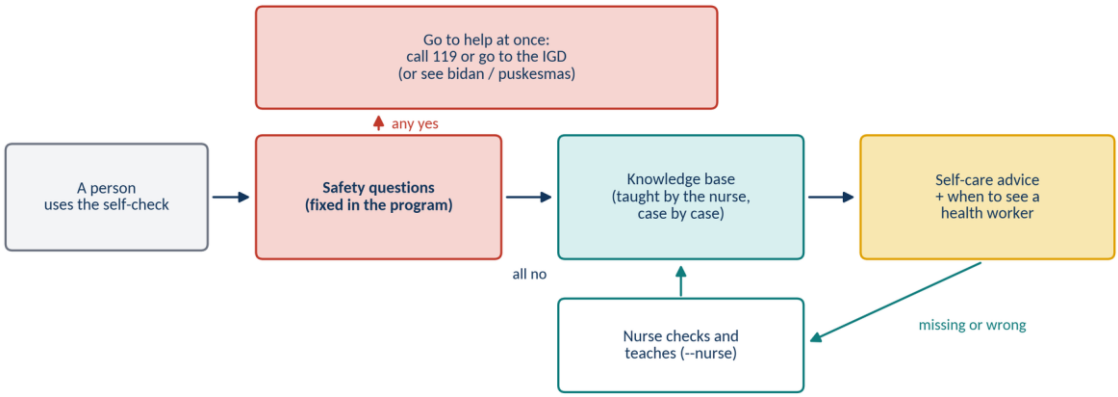

Figure 5.1 Safety first. The fixed safety questions come before the knowledge base; a single *yes* sends the person to help at once. Only then is the knowledge base, taught case by case by the nurse, consulted for self-care advice.

The original VCIRS health self-check, built from the diagrams of a home-medicine handbook, made the same kind of choice: its knowledge was grouped by who the patient was — children, adults, men, women — because the same complaint needs different handling in each group, and a question meant for an adult must not be asked about a baby [9].

5.2.2 Knowledge that grows, case by case

Behind the safety questions sits a VCIRS knowledge base, and here Ners Ratna is the expert. It starts small: *a fever: rest, drink, paracetamol as on the label; a cough: warm drinks, cover your cough*. Then, one real patient at a time, she refines it. A fever that has lasted two days or more needs a check at the puskesmas, because that is when dengue must be ruled out [7]. A cough of two weeks or more needs a TB check, and the TB tests and medicine at the puskesmas are free [10]. A headache with a blood-pressure reading of 140 or more needs another measurement on another day, because that is the line above which WHO calls blood pressure high when it is found on two separate days [2].

This is Ripple Down Rules as Chapter 3 described it: the knowledge base is used on real cases; when its answer is missing or wrong, the expert adds knowledge in the context of that answer; nothing is ever deleted [11], [12]. And every time the answer is right, the nurse can say so, and VCIRS records the check — *verification on the fly*, in the words of the thesis [13].

Two things are different from the Wi-Fi doctor, and both come from the nature of health. First, a person can have several complaints at once: Pak Darmo has a fever *and* a headache, and both need advice. So the self-check follows every branch that applies, as Multiple Classification RDR does [14], and gives one piece of advice per branch. Second, the cost of a careless new rule is higher, so the expert needs more control over what is added. That control is the subject of the next four sections.

5.2.3 The cornerstone case and the list of difference

Every node keeps the case that created it, its **cornerstone case** [11], [13]. When a new case gets a wrong or incomplete answer from a node, VCIRS lays the new case beside that node's cornerstone case and lists how they differ, in two sections, as the 2006 revision defines them [15], [16]:

- **Section A** holds the facts that only the *old* case has;
- **Section B** holds the facts that only the *new* case has.

Facts both cases share are not listed: they cannot explain why the two cases need different advice. The list of difference is the expert's worksheet. Everything she might write into a new node is either on it or, if she knows something the cases do not show, added by her.

5.2.4 The six options

With the list of difference in front of her, the expert chooses one of six options [15], [16]. **Table 5.1** lists them, with the question each one answers.

Table 5.1 The six options of VCIRS knowledge building, with the question each answers. Options 1 to 3 make an exception of the old node; options 4 and 5 store the whole case as a new node elsewhere.

Option	The new node holds	It is placed	Use it when
0	nothing: the answer is accepted	— (the check is recorded)	the advice was right
1	the negations of section A	under the old node, as an exception	what the old case *had* is the reason it differs
2	section B	under the old node, as an exception	what the new case *has* is the reason
3	options 1 and 2 together	under the old node, as an exception	both are part of the reason (what `learn` does)
4	all the facts of the new case	beside the old node, under the same parent	the new case is a different situation at the same level
5	all the facts of the new case	at the top level	nothing in the knowledge base is related

Options 1, 2 and 3 are the heart of Ripple Down Rules: the new node is an exception, tried only for cases that reached the old node, so it cannot disturb anything else. Option 3, both sections together, is the most specific safe choice, and it is what the one-call `learn` of Chapters 3 and 4 uses. Options 4 and 5 do not make an exception at all: they store the whole case as a new node, beside the old one or at the top, which is right when the new case is not a refinement of the old advice but a different situation.

5.2.5 Choosing the conditions yourself

The six options build a node from the facts of two cases, and facts are specific: *fever days = 3*, *cough weeks = 1*. An expert usually knows the rule behind the case. Ners Ratna does not mean "a fever of exactly three days"; she means "a fever of two days or more". VCIRS therefore lets the expert **choose the conditions herself**, in her own order, instead of taking the option's list as it is [13]. The thesis already allows the user to order the conditions; the engine goes one step further and accepts any conditions she writes.

This is where the art is. A node that is **too specific** — *fever days = 3, joint pain = yes* — will be right for the case that made it and silent for nearly everyone else; the knowledge base grows, but its coverage does not. A node that is **too general** — *fever days <> 1* — will catch cases it was never meant for. **Figure 5.2** shows the six options and the nurse's own choice for one case, and Section 5.3 computes, for five later patients, which of them each node would catch.

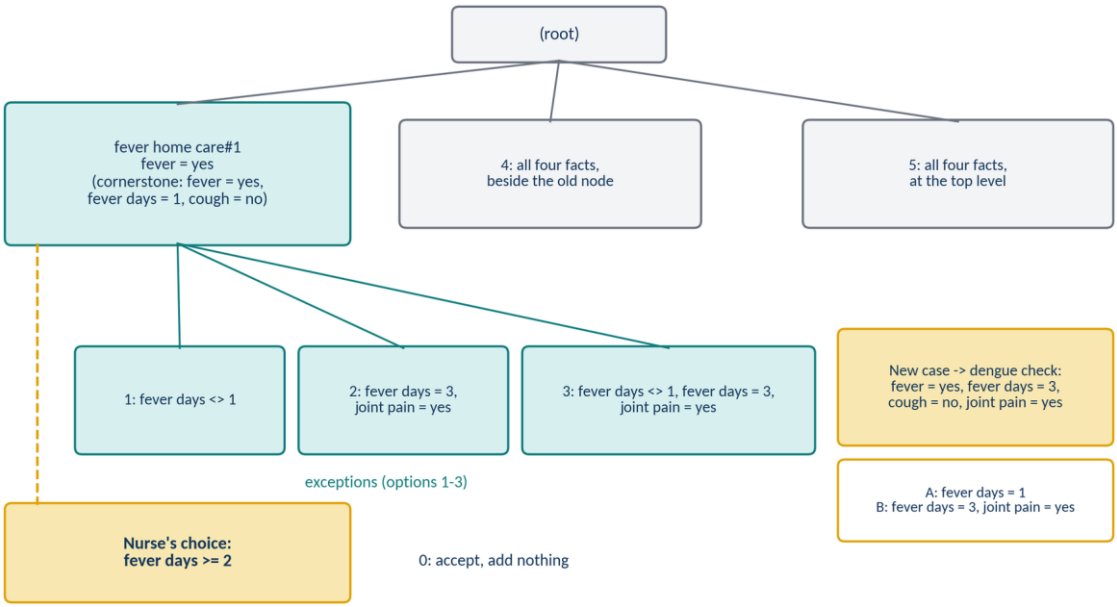

Figure 5.2 What each option would build for one case at the puskesmas. Options 1 to 3 hang an exception under the old node; option 4 puts the whole case beside it and option 5 at the top level; the nurse's own choice is one condition that says what she means.

5.2.6 Checking what a new node would change

An exception can only affect cases that reach its parent, but *among* those cases it can still catch more than the expert intended. RDR's answer is to look at the cornerstone cases: before a new node is added, the system checks every stored cornerstone case against the new node's full path of conditions, and lists those that would now get a different answer [11]. These are the **affected cornerstones**. The expert looks at each one and decides: either its answer should indeed change, or she adds another condition that tells it apart, and the check runs again. For a well-chosen node the list is empty.

This is cheap for the machine and invaluable for the expert. She never has to read the whole knowledge base; she reads only the few earlier cases that her new node would touch. It is the reason RDR knowledge bases could be maintained by the experts themselves, one case at a time, for years [11], [17].

5.2.7 Never delete, sometimes stop

Nothing in a VCIRS knowledge base is ever deleted: a node that was once added keeps its cornerstone case, its exceptions and its place in the history [13]. When a piece of advice becomes wrong — a medicine is withdrawn, a service moves, a guideline changes — the expert **stops** the node. A stopped node still takes part in the search, and its exceptions still work, but its own conclusion is no longer given. The better advice is then taught as a new node. Exercise 5.7 does exactly this.

> **DEFINITION.** *Case-by-case knowledge refinement* is the loop in which an expert, for each real case whose answer is missing or wrong, reads the list of difference against the cornerstone case of the node that answered, chooses one of the six options (and, if she knows better, the conditions herself), checks the affected cornerstones, and only then adds the new node. Right answers are recorded as checks; nothing is deleted.

5.3 Step 1: The Six Options by Hand

Before we let the engine do it, let us compute the list of difference and all six options ourselves, for the case in **Figure 5.2**. The old node is `fever home care#1`, whose cornerstone case is a fever of one day with no cough. The new case is a fever of three days, no cough, with sore joints, and its right advice is a dengue check. **Listing 5.1** builds every option, adds Ners Ratna's own choice, and tries each resulting node on five later patients.

Listing 5.1 `SixOptions.java`: the list of difference and the six options, worked by hand.

```
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

/**
 * The list of difference and the six options of VCIRS, worked by hand on
 * one case from the puskesmas: which node would each option build, and which
 * later patients would that node catch?
 */
public class SixOptions {
    static final String[] OPERATORS = {" <= ", " >= ", " <> ", " = ", " < ",
        " > "};
    static final Map<String, String> NEGATION = Map.of("=", "<>", "<>", "=",
            "<", ">=", ">=", "<", ">", "<=", "<=", ">");

    static String[] split(String condition) {
        for (String op : OPERATORS) {
            int at = condition.indexOf(op);
            if (at > 0) {
                return new String[] {condition.substring(0, at), op.strip(),
                    condition.substring(at + op.length())};
            }
        }
        throw new IllegalArgumentException("no operator in " + condition);
    }

    static String negate(String condition) {
        String[] p = split(condition);
        return p[0] + " " + NEGATION.get(p[1]) + " " + p[2];
    }

    static boolean holds(String condition, Map<String, String> facts) {
        String[] p = split(condition);
        String fact = facts.get(p[0]);
        if (fact == null) {
            return false;
        }
        if (p[1].equals("=") || p[1].equals("<>")) {
            return fact.equals(p[2]) == p[1].equals("=");
        }
        double a = Double.parseDouble(fact);
        double b = Double.parseDouble(p[2]);
        switch (p[1]) {
            case "<": return a < b;
            case "<=": return a <= b;
```

Listing 5.1 (Continued) *SixOptions.java: the list of difference and the six options, worked by hand.*

```
            case ">": return a > b;
            default: return a >= b;
        }
    }

    static String show(List<String> conditions) {
        return conditions.isEmpty() ? "(none)" : String.join(", ", conditions);
    }

    record Option(String name, String place, List<String> conditions) { }

    static Map<String, String> facts(String... pairs) {
        Map<String, String> m = new LinkedHashMap<>();
        for (int i = 0; i < pairs.length; i += 2) {
            m.put(pairs[i], pairs[i + 1]);
        }
        return m;
    }

    public static void main(String[] args) {
        String oldId = "fever home care#1";
        List<String> oldCase = List.of("fever = yes", "fever days = 1",
                "cough = no");
        List<String> newCase = List.of("fever = yes", "fever days = 3",
                "cough = no", "joint pain = yes");
        List<String> a = new ArrayList<>();          // only in the old case
        for (String c : oldCase) {
            if (!newCase.contains(c)) {
                a.add(c);
            }
        }
        List<String> b = new ArrayList<>();          // only in the new case
        for (String c : newCase) {
            if (!oldCase.contains(c)) {
                b.add(c);
            }
        }
        List<String> negA = new ArrayList<>();
        for (String c : a) {
            negA.add(negate(c));
        }
        List<String> both = new ArrayList<>(negA);
        both.addAll(b);
```

Listing 5.1 (Continued) *SixOptions.java: the list of difference and the six options, worked by hand.*

```
        String under = "under " + oldId;             // an exception
        List<Option> options = List.of(
                new Option("Option 1", under, negA),
                new Option("Option 2", under, b),
                new Option("Option 3", under, both),
                new Option("Option 4", "beside " + oldId, newCase),
                new Option("Option 5", "at the top level", newCase),
                new Option("Nurse's choice", under, List.of("fever days >= 2")));

        System.out.println("Old node " + oldId + ", made by the case:");
        System.out.println("   " + show(oldCase));
        System.out.println("New case, whose right advice is dengue check:");
        System.out.println("   " + show(newCase));
        System.out.println("List of difference:");
        System.out.println("  A (only in the old case): " + show(a));
        System.out.println("  B (only in the new case): " + show(b));
        System.out.println();
        System.out.println("Option 0: nothing new; the old advice is accepted");
        for (Option o : options) {
            System.out.println(o.name() + ": " + o.place() + ": "
                    + show(o.conditions()));
        }

        Map<String, Map<String, String>> patients = new LinkedHashMap<>();
        patients.put("fever 2 days", facts("fever", "yes", "fever days", "2"));
        patients.put("3 days + sore joints", facts("fever", "yes",
                "fever days", "3", "joint pain", "yes", "cough", "no"));
        patients.put("fever 5 days", facts("fever", "yes", "fever days", "5"));
        patients.put("1 day + sore joints", facts("fever", "yes",
                "fever days", "1", "joint pain", "yes"));
        patients.put("fever since today", facts("fever", "yes",
                "fever days", "0"));
        System.out.println();
        System.out.println("Which later patients would the new node send to "
                + "dengue check?");
        for (Option o : options) {
            List<String> caught = new ArrayList<>();
            for (Map.Entry<String, Map<String, String>> p : patients.entrySet()) {
                boolean all = true;
                for (String c : o.conditions()) {
                    all = all && holds(c, p.getValue());
                }
                if (all) {
                    caught.add(p.getKey());
                }
            }
```

Listing 5.1 (Continued) *SixOptions.java: the list of difference and the six options, worked by hand.*

```
            System.out.println("  " + String.format("%-16s", o.name() + ":")
                    + (caught.isEmpty() ? "nobody" : String.join(", ", caught)));
        }
    }
}
```

The helpers are those of Chapter 4 with one addition. `NEGATION` maps each operator to its opposite (`=` to `<>`, `>=` to `<`, and so on), and `negate` uses it to turn a condition into its negation: `fever days = 1` becomes `fever days <> 1`. `holds` treats `=` and `<>` as comparisons of words and the other four as comparisons of numbers.

`main` computes section A as the facts of the old case that the new case lacks, and section B as the facts of the new case that the old case lacks, exactly as the definition says. The six options are then built in six lines: option 1 from the negations of A, option 2 from B, option 3 from both, options 4

and 5 from the whole new case, placed beside the old node or at the top level. The nurse's choice is one condition she writes herself. Finally, each node is tried on five patients who might come later — all of them already have a fever, so all of them reach the old node — and the program prints which patients each node would send to a dengue check. **Listing 5.2** is the Python version.

Listing 5.2 `six_options.py`: the same computation in Python.

```
"""The list of difference and the six options of VCIRS, worked by hand on
one case from the puskesmas: which node would each option build, and which
later patients would that node catch?"""

OPERATORS = [" <= ", " >= ", " <> ", " = ", " < ", " > "]
NEGATION = {"=": "<>", "<>": "=", "<": ">=", ">=": "<", ">": "<=", "<=": ">"}


def split(condition):
    for op in OPERATORS:
        at = condition.find(op)
        if at > 0:
            return [condition[:at], op.strip(), condition[at + len(op):]]
    raise ValueError("no operator in " + condition)


def negate(condition):
    var, op, value = split(condition)
    return var + " " + NEGATION[op] + " " + value


def holds(condition, facts):
    var, op, value = split(condition)
    if var not in facts:
        return False
    fact = facts[var]
    if op in ("=", "<>"):
        return (fact == value) == (op == "=")
    a, b = float(fact), float(value)
    return {"<": a < b, "<=": a <= b, ">": a > b, ">=": a >= b}[op]


def show(conditions):
    return ", ".join(conditions) if conditions else "(none)"


def main():
    old_id = "fever home care#1"
    old_case = ["fever = yes", "fever days = 1", "cough = no"]
    new_case = ["fever = yes", "fever days = 3", "cough = no", "joint pain = yes"]
    a = [c for c in old_case if c not in new_case]     # only in the old case
    b = [c for c in new_case if c not in old_case]     # only in the new case
```

Listing 5.2 (Continued) *six_options.py: the same computation in Python.*

```
    under = "under " + old_id                          # an exception
    options = [
        ("Option 1", under, [negate(c) for c in a]),
        ("Option 2", under, b),
        ("Option 3", under, [negate(c) for c in a] + b),
        ("Option 4", "beside " + old_id, new_case),
        ("Option 5", "at the top level", new_case),
        ("Nurse's choice", under, ["fever days >= 2"]),
    ]

    print("Old node " + old_id + ", made by the case:")
    print("  " + show(old_case))
    print("New case, whose right advice is dengue check:")
    print("  " + show(new_case))
    print("List of difference:")
    print("  A (only in the old case): " + show(a))
    print("  B (only in the new case): " + show(b))
    print()
    print("Option 0: nothing new; the old advice is accepted")
    for name, place, conditions in options:
        print(name + ": " + place + ": " + show(conditions))

    patients = [
        ("fever 2 days", {"fever": "yes", "fever days": "2"}),
        ("3 days + sore joints", {"fever": "yes", "fever days": "3",
                                  "joint pain": "yes", "cough": "no"}),
        ("fever 5 days", {"fever": "yes", "fever days": "5"}),
        ("1 day + sore joints", {"fever": "yes", "fever days": "1",
                                 "joint pain": "yes"}),
        ("fever since today", {"fever": "yes", "fever days": "0"}),
    ]
    print()
    print("Which later patients would the new node send to dengue check?")
    for name, _, conditions in options:
        caught = [p for p, facts in patients
                  if all(holds(c, facts) for c in conditions)]
        print("  " + (name + ":").ljust(16) + (", ".join(caught) or "nobody"))


if __name__ == "__main__":
    main()
```

Both print:

```
Old node fever home care#1, made by the case:
  fever = yes, fever days = 1, cough = no
New case, whose right advice is dengue check:
  fever = yes, fever days = 3, cough = no, joint pain = yes
List of difference:
  A (only in the old case): fever days = 1
  B (only in the new case): fever days = 3, joint pain = yes

Option 0: nothing new; the old advice is accepted
Option 1: under fever home care#1: fever days <> 1
Option 2: under fever home care#1: fever days = 3, joint pain = yes
Option 3: under fever home care#1: fever days <> 1, fever days = 3, joint pain = yes
Option 4: beside fever home care#1: fever = yes, fever days = 3, cough = no, joint pain =
yes
Option 5: at the top level: fever = yes, fever days = 3, cough = no, joint pain = yes
Nurse's choice: under fever home care#1: fever days >= 2

Which later patients would the new node send to dengue check?
  Option 1:       fever 2 days, 3 days + sore joints, fever 5 days, fever since today
  Option 2:       3 days + sore joints
  Option 3:       3 days + sore joints
  Option 4:       3 days + sore joints
  Option 5:       3 days + sore joints
  Nurse's choice: fever 2 days, 3 days + sore joints, fever 5 days
```

Read the last block slowly, because it is the whole lesson of this chapter in six lines.

Options 2, 3, 4 and 5 each catch one patient: the one with three days of fever and sore joints — the new case itself. They are all *correct*: none of them sends anybody to a dengue check who should not go. But they are useless for the patient with two days of fever, and for the one with five days, who also need the check. These nodes are **too specific**: they remember one case instead of learning a rule.

Option 1, `fever days <> 1`, catches the patients with two, three and five days of fever — and also the patient whose fever started *today*, because zero is not one. It is **too general**: it learned "not one day" when the lesson was "at least two days". The mistake would not show up for weeks, until someone with a morning fever was sent for a blood test.

The nurse's choice, `fever days >= 2`, catches exactly the three patients who need a dengue check and none of the others. It is not on the list of difference; it is what the nurse knows. The list of difference told her *where* the reason lies — in the number of fever days — and she wrote *what* the reason is.

Why not let the machine choose the threshold itself? With enough cases it could: a program that has seen a hundred fevers, each labeled *needed a dengue check* or *did not*, can search for the number of days that separates the two groups best. Chapter 12 does exactly that, learning rules from data and then handing them to VCIRS. But notice what such a program needs: a hundred labeled cases before it can say anything, and the labels themselves come from people like Ners Ratna. At a puskesmas that is just starting its self-check there are no hundred cases, and there is one nurse who already knows the answer, from her training, from the Ministry of Health's guidance [7] and from years of patients. Case-by-case refinement lets her put that knowledge in on the first day, from the first case, and lets every later case check it. The two approaches are not rivals. Data can suggest a threshold; the expert confirms it, or corrects it, with a case in front of her.

There is also a question of trust. When a node says `fever days >= 2`, the nurse can read it, the doctor can check it against the guidance, and the patient's daughter can understand it. A threshold

found by a program in a table of numbers may be just as good, but until someone reads it and ties it to a reason, it is a number without a reason. In health, the reason is part of the advice.

> **KEY INSIGHT.** The list of difference shows the expert where to look; the options turn it into a node mechanically; but only the expert knows the rule behind the case. The best node is often built from one fact of the list, rewritten as the rule she actually uses.

5.4 Step 2: Ners Ratna Teaches the Knowledge Base

Now the same work on the engine, for a whole teaching session. In **Listing 5.3**, Ners Ratna builds the self-check's knowledge base from ten real cases. For each one the program shows what the knowledge base says, the list of difference, what the chosen option would build, the conditions she keeps instead (if any), and the affected cornerstones; then it adds the node.

Listing 5.3 `NurseTeaches.java`: a nurse teaches the knowledge base case by case, on vcirs-core.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Map;

import vcirs.Case;
import vcirs.Conclusion;
import vcirs.Condition;
import vcirs.DifferenceList;
import vcirs.InferenceResult;
import vcirs.KnowledgeBase;
import vcirs.Node;
import vcirs.Report;

/**
 * Ners Ratna teaches the health self-check, case by case. For every case
 * she sees what the knowledge base says, reads the list of difference,
 * chooses an option (and, when she knows better, the conditions herself),
 * and checks which earlier cases the new node would affect.
 */
public class NurseTeaches {
    static final int ACCEPT = KnowledgeBase.ACCEPT;
    static final int ADD_NEW = KnowledgeBase.ADD_NEW;
    static final int BOTH = KnowledgeBase.BOTH;
    static final int TOP_LEVEL = KnowledgeBase.TOP_LEVEL;

    static String show(List<Condition> conds) {
        List<String> parts = new ArrayList<>();
        for (Condition c : conds) {
            parts.add(c.toString());
        }
        return String.join(", ", parts);
    }

    static List<Condition> parse(List<String> texts) {
        List<Condition> out = new ArrayList<>();
        for (String t : texts) {
            out.add(Condition.parse(t));
        }
        return out;
    }
```

Listing 5.3 (Continued) *NurseTeaches.java: a nurse teaches the knowledge base case by case, on vcirs-core.*

```
    /** The conditions an option builds, before the nurse changes them. */
    static List<Condition> builtBy(KnowledgeBase kb, int option, Node fired,
                                   Case c) {
        List<Condition> out = new ArrayList<>();
        if (option >= 1 && option <= 3) {
            DifferenceList d = kb.differences(fired.id, c);
            if (option == 1 || option == 3) {
                for (Condition x : d.a) {
                    out.add(x.negated());
                }
            }
            if (option == 2 || option == 3) {
                out.addAll(d.b);
            }
            return out;
        }
        out.addAll(c.facts);
        return out;
    }

    static void teach(KnowledgeBase kb, List<String> facts, String advice,
                      int option, List<String> keep, List<String> more) {
        Case c = Case.of(facts, List.of("advice = " + advice));
        System.out.println("Case: " + String.join(", ", facts) + " -> "
                + advice);
        InferenceResult result = kb.infer(c.known());
        Node fired = result.paths.isEmpty() ? null : result.paths.get(0).last();
        System.out.println("  knowledge base says: " + (fired != null
                ? fired.conclusions.get(0).value + " (" + fired.id + ")"
                : "nothing"));
        if (option == ACCEPT) {
            kb.refine(c, ACCEPT, fired.id, null, "Ratna");
            System.out.println("  option 0: " + KnowledgeBase.OPTION_TEXT.get(0));
            return;
        }
        if (fired != null) {
            for (String line : kb.differences(fired.id, c).lines()) {
                System.out.println("   " + line);
            }
        }
        List<Condition> conds = builtBy(kb, option, fired, c);
        System.out.println("  option " + option + " builds: " + show(conds));
        if (keep != null) {
            conds = parse(keep);
            System.out.println("  Ratna keeps instead: " + String.join(", ", keep));
        }
```

Listing 5.3 (Continued) *NurseTeaches.java: a nurse teaches the knowledge base case by case, on vcirs-core.*

```
        String parent = option == TOP_LEVEL ? null : fired.id;
        List<String> hit = kb.affectedCornerstones(parent, conds, c.conclusions);
        System.out.println("  earlier cases whose advice would change: "
                + (hit.isEmpty() ? "none" : String.join(", ", hit)));
        if (!hit.isEmpty() && more != null) {
            conds.addAll(parse(more));
            System.out.println("  Ratna adds: " + String.join(", ", more));
            hit = kb.affectedCornerstones(parent, conds, c.conclusions);
            System.out.println("  earlier cases whose advice would change: "
                    + (hit.isEmpty() ? "none" : String.join(", ", hit)));
        }
        Node node = kb.refine(c, option, parent, conds, "Ratna");
        System.out.println("  new node " + node);
    }

    static void teachAll(KnowledgeBase kb) {
        teach(kb, List.of("fever = yes"), "fever home care", TOP_LEVEL, null,
                null);
        teach(kb, List.of("cough = yes", "cough weeks = 1"), "cough home care",
                TOP_LEVEL, List.of("cough = yes"), null);
        teach(kb, List.of("diarrhea = yes"), "ORS at home", TOP_LEVEL, null,
                null);
        teach(kb, List.of("headache = yes"), "headache home care", TOP_LEVEL,
                null, null);
        teach(kb, List.of("fever = yes", "fever days = 3"), "dengue check",
                ADD_NEW, List.of("fever days >= 2"), null);
        teach(kb, List.of("fever = yes", "fever days = 1"), "fever home care",
                ACCEPT, null, null);
        teach(kb, List.of("cough = yes", "cough weeks = 3", "smoker = yes"),
                "TB check", BOTH, List.of("cough weeks >= 2"), null);
        teach(kb, List.of("cough = yes", "cough weeks = 1", "smoker = yes"),
                "quit smoking help", ADD_NEW, null, List.of("cough weeks < 2"));
        teach(kb, List.of("headache = yes", "blood pressure = 150"),
                "blood pressure check", ADD_NEW,
                List.of("blood pressure >= 140"), null);
        teach(kb, List.of("diarrhea = yes", "diarrhea days = 4"),
                "diarrhea check", ADD_NEW, List.of("diarrhea days >= 3"), null);
    }
```

Listing 5.3 (Continued) *NurseTeaches.java: a nurse teaches the knowledge base case by case, on vcirs-core.*

```
    public static void main(String[] args) {
        KnowledgeBase kb = new KnowledgeBase("Health self-check");
        teachAll(kb);
        System.out.println();
        for (String line : Report.tree(kb)) {
            System.out.println(line);
        }
        int kept = 0;
        for (Node n : kb.nodes.values()) {
            for (Conclusion c : kb.infer(n.cornerstone.known()).conclusions()) {
                if (c.same(n.conclusions.get(0))) {
                    kept++;
                    break;
                }
            }
        }
        System.out.println("Cornerstone cases that still get their own advice: "
                + kept + " of " + kb.nodes.size());
        List<String> used = new ArrayList<>();
        for (Map<String, Object> h : kb.history) {
            used.add(String.valueOf(h.get("option")));
        }
        System.out.println("Options used: " + String.join(", ", used));
    }
}
```

`builtBy` is the six-options computation of Step 1 done with the engine's own pieces: `kb.differences(fired.id, c)` returns the list of difference as a `DifferenceList` with sections `a` and `b`, and `negated()` negates a condition. `teach` is one teaching step. It asks the knowledge base for its answer with `kb.infer(c.known())`, the known facts of the case; accepts it with option 0 when the answer is right; otherwise prints the list of difference and what the option would build, replaces the conditions with the nurse's own when she gives them (`keep`), and asks the engine for the affected cornerstones with `kb.affectedCornerstones(parent, conds, c.conclusions)`. If the list is not empty and she has another condition ready (`more`), it adds that condition and checks again. Finally `kb.refine(c, option, parent, conds, "Ratna")` adds the node. `refine` is the engine's general knowledge-building call: a case, one of the six options, the old node, the chosen conditions and the name of the expert [13], [15].

`teachAll` is the session itself — the ten cases in the order Ners Ratna met them — and `main` runs it, prints the tree, checks that every cornerstone case still gets its own advice, and lists the options used, as the history records them. **Listing 5.4** is the Python version.

Listing 5.4 `nurse_teaches.py`: the same teaching session in Python.

```
"""Ners Ratna teaches the health self-check, case by case. For every case
she sees what the knowledge base says, reads the list of difference,
chooses an option (and, when she knows better, the conditions herself),
and checks which earlier cases the new node would affect."""

import pathlib
import sys

# Tell Python where the vcirs-core engine is (two folders up, then across).
sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import (ACCEPT, ADD_NEW, BOTH, OPTION_TEXT, TOP_LEVEL, Case,
                   Condition, KnowledgeBase, report)


def built_by(kb, option, fired, case):
    """The conditions an option builds, before the nurse changes them."""
    if option in (1, 2, 3):
        d = kb.differences(fired.id, case)
        neg = [c.negated() for c in d.a] if option in (1, 3) else []
        return neg + (d.b if option in (2, 3) else [])
    return list(case.facts)


def teach(kb, facts, advice, option, keep=None, more=None):
    case = Case.of(facts, ["advice = " + advice])
    print("Case: " + ", ".join(facts) + " -> " + advice)
    result = kb.infer(case)
    fired = result.paths[0].last() if result.paths else None
    print("  knowledge base says: " + (fired.conclusions[0].value + " ("
          + fired.id + ")" if fired else "nothing"))
    if option == ACCEPT:
        kb.refine(case, ACCEPT, fired.id, author="Ratna")
        print("  option 0: " + OPTION_TEXT[0])
        return
    if fired is not None:
        for line in kb.differences(fired.id, case).lines():
            print("  " + line)
    conds = built_by(kb, option, fired, case)
    print("  option " + str(option) + " builds: "
          + ", ".join(str(c) for c in conds))
    if keep is not None:
        conds = [Condition.parse(c) for c in keep]
        print("  Ratna keeps instead: " + ", ".join(keep))
    parent = None if option == TOP_LEVEL else fired.id
    hit = kb.affected_cornerstones(parent, conds, case.conclusions)
```

Listing 5.4 (Continued) *nurse_teaches.py: the same teaching session in Python.*

```
    print("  earlier cases whose advice would change: "
          + (", ".join(hit) if hit else "none"))
    if hit and more:
        conds += [Condition.parse(c) for c in more]
        print("  Ratna adds: " + ", ".join(more))
        hit = kb.affected_cornerstones(parent, conds, case.conclusions)
        print("  earlier cases whose advice would change: "
              + (", ".join(hit) if hit else "none"))
    node = kb.refine(case, option, parent, chosen=conds, author="Ratna")
    print("  new node " + str(node))


def teach_all(kb):
    teach(kb, ["fever = yes"], "fever home care", TOP_LEVEL)
    teach(kb, ["cough = yes", "cough weeks = 1"], "cough home care",
          TOP_LEVEL, keep=["cough = yes"])
    teach(kb, ["diarrhea = yes"], "ORS at home", TOP_LEVEL)
    teach(kb, ["headache = yes"], "headache home care", TOP_LEVEL)
    teach(kb, ["fever = yes", "fever days = 3"], "dengue check", ADD_NEW,
          keep=["fever days >= 2"])
    teach(kb, ["fever = yes", "fever days = 1"], "fever home care", ACCEPT)
    teach(kb, ["cough = yes", "cough weeks = 3", "smoker = yes"], "TB check",
          BOTH, keep=["cough weeks >= 2"])
    teach(kb, ["cough = yes", "cough weeks = 1", "smoker = yes"],
          "quit smoking help", ADD_NEW, more=["cough weeks < 2"])
    teach(kb, ["headache = yes", "blood pressure = 150"],
          "blood pressure check", ADD_NEW, keep=["blood pressure >= 140"])
    teach(kb, ["diarrhea = yes", "diarrhea days = 4"], "diarrhea check",
          ADD_NEW, keep=["diarrhea days >= 3"])


if __name__ == "__main__":
    kb = KnowledgeBase("Health self-check")
    teach_all(kb)
    print()
    for line in report.tree(kb):
        print(line)
    kept = [n for n in kb.nodes.values()
            if any(c.same(n.conclusions[0])
                   for c in kb.infer(n.cornerstone).conclusions())]
    print("Cornerstone cases that still get their own advice: "
          + str(len(kept)) + " of " + str(len(kb.nodes)))
    used = [h["option"] for h in kb.history]
    print("Options used: " + ", ".join(str(o) for o in used))
```

Both print:

```
Case: fever = yes -> fever home care
  knowledge base says: nothing
  option 5 builds: fever = yes
  earlier cases whose advice would change: none
  new node fever home care#1: fever = yes -> advice = fever home care
Case: cough = yes, cough weeks = 1 -> cough home care
  knowledge base says: nothing
  option 5 builds: cough = yes, cough weeks = 1
  Ratna keeps instead: cough = yes
  earlier cases whose advice would change: none
  new node cough home care#1: cough = yes -> advice = cough home care
Case: diarrhea = yes -> ORS at home
  knowledge base says: nothing
  option 5 builds: diarrhea = yes
  earlier cases whose advice would change: none
  new node ORS at home#1: diarrhea = yes -> advice = ORS at home
Case: headache = yes -> headache home care
  knowledge base says: nothing
  option 5 builds: headache = yes
  earlier cases whose advice would change: none
  new node headache home care#1: headache = yes -> advice = headache home care
Case: fever = yes, fever days = 3 -> dengue check
  knowledge base says: fever home care (fever home care#1)
  Differences with the cornerstone case of fever home care#1:
    A (only in the old case): (none)
    B (only in the new case): fever days = 3
  option 2 builds: fever days = 3
  Ratna keeps instead: fever days >= 2
  earlier cases whose advice would change: none
  new node dengue check#1: fever days >= 2 -> advice = dengue check
Case: fever = yes, fever days = 1 -> fever home care
  knowledge base says: fever home care (fever home care#1)
  option 0: accept the existing knowledge (inference only)
Case: cough = yes, cough weeks = 3, smoker = yes -> TB check
  knowledge base says: cough home care (cough home care#1)
  Differences with the cornerstone case of cough home care#1:
    A (only in the old case): cough weeks = 1
    B (only in the new case): cough weeks = 3, smoker = yes
  option 3 builds: cough weeks <> 1, cough weeks = 3, smoker = yes
  Ratna keeps instead: cough weeks >= 2
  earlier cases whose advice would change: none
  new node TB check#1: cough weeks >= 2 -> advice = TB check
Case: cough = yes, cough weeks = 1, smoker = yes -> quit smoking help
  knowledge base says: cough home care (cough home care#1)
  Differences with the cornerstone case of cough home care#1:
    A (only in the old case): (none)
    B (only in the new case): smoker = yes
  option 2 builds: smoker = yes
  earlier cases whose advice would change: TB check#1
  Ratna adds: cough weeks < 2
  earlier cases whose advice would change: none
  new node quit smoking help#1: smoker = yes, cough weeks < 2 -> advice = quit smoking help
```

```
Case: headache = yes, blood pressure = 150 -> blood pressure check
  knowledge base says: headache home care (headache home care#1)
  Differences with the cornerstone case of headache home care#1:
    A (only in the old case): (none)
    B (only in the new case): blood pressure = 150
  option 2 builds: blood pressure = 150
  Ratna keeps instead: blood pressure >= 140
  earlier cases whose advice would change: none
  new node blood pressure check#1: blood pressure >= 140 -> advice = blood pressure check
Case: diarrhea = yes, diarrhea days = 4 -> diarrhea check
  knowledge base says: ORS at home (ORS at home#1)
  Differences with the cornerstone case of ORS at home#1:
    A (only in the old case): (none)
    B (only in the new case): diarrhea days = 4
  option 2 builds: diarrhea days = 4
  Ratna keeps instead: diarrhea days >= 3
  earlier cases whose advice would change: none
  new node diarrhea check#1: diarrhea days >= 3 -> advice = diarrhea check

(root)
+- fever home care#1: fever = yes -> advice = fever home care
|  +- dengue check#1: fever days >= 2 -> advice = dengue check
+- cough home care#1: cough = yes -> advice = cough home care
|  +- TB check#1: cough weeks >= 2 -> advice = TB check
|  +- quit smoking help#1: smoker = yes, cough weeks < 2 -> advice = quit smoking help
+- ORS at home#1: diarrhea = yes -> advice = ORS at home
|  +- diarrhea check#1: diarrhea days >= 3 -> advice = diarrhea check
+- headache home care#1: headache = yes -> advice = headache home care
   +- blood pressure check#1: blood pressure >= 140 -> advice = blood pressure check
Cornerstone cases that still get their own advice: 9 of 9
Options used: 5, 5, 5, 5, 2, 0, 3, 2, 2, 2
```

Walk through the session with Ners Ratna.

Cases 1 to 4 start the knowledge base. Each time the knowledge base says *nothing*, and she uses option 5, a new top-level node, one per complaint. In case 2 the option would build `cough = yes, cough weeks = 1`, but she keeps only `cough = yes`: a cough of any length deserves the home-care advice, and the length belongs in the exceptions. The cornerstone case still remembers the whole case, one week of cough, and that will matter in case 7.

Case 5 is Step 1 again, with the engine. The knowledge base says *fever home care*; the list of difference has an empty section A (the cornerstone of `fever home care#1` was only `fever = yes`) and `fever days = 3` in section B; option 2 would build exactly that; she keeps `fever days >= 2` instead. No earlier case would change its advice, and `dengue check#1` is hung under `fever home care#1`.

Case 6 is a fever of one day, and the knowledge base gives home care, which is right. She accepts it: option 0. Nothing is added, but the history now shows that this node was checked on a real patient.

Case 7 is a smoker with a three-week cough, who needs a TB check. The cornerstone case of `cough home care#1` had a one-week cough, so this time section A is not empty: `cough weeks = 1`. Option 3 would build `cough weeks <> 1, cough weeks = 3, smoker = yes` — the negation of A and all of B — which would catch only smokers with exactly three weeks of cough. Ners Ratna keeps `cough weeks >= 2`: the length is the reason, and smoking is not.

Case 8 is the one where the check of affected cornerstones earns its place. A smoker with a one-week cough should be offered help to quit, at the puskesmas or through the Ministry of Health's free quitline [18]. Option 2 builds `smoker = yes`, which looks right. But the engine finds that the cornerstone case of `TB check#1` — the smoker with a three-week cough from case 7 — would

also be caught by the new node, and would be given quit-smoking help instead of, or next to, the TB check it needs. She adds `cough weeks < 2`, the check runs again, and the list is empty. The new node now reads *smoker, and a cough of less than two weeks*: the long coughs stay with the TB check.

Cases 9 and 10 follow the pattern of case 5: a headache with a blood-pressure reading of 150 teaches `blood pressure >= 140` [2], and a diarrhea of four days teaches `diarrhea days >= 3`.

The last three parts of the output are the proof that nothing broke. The tree has four top-level nodes and five exceptions, each holding only what it adds. Every one of the nine cornerstone cases, run through the finished knowledge base, still gets its own advice — the guarantee of Ripple Down Rules, checked by the machine. And the history lists the options exactly as they were used: four times option 5, once option 0, once option 3 and four times option 2, in that order. This is a record Ners Ratna can show her head of puskesmas: who taught what, how, and when.

5.5 The Health Self-Check

5.5.1 What the app starts from

The app starts from the knowledge of Ners Ratna's first month, written as nine seed cases in `health-seed-cases.json`:

```
{
  "name": "Health self-check",
  "description": "Self-care advice for adults, taught by Ners Ratna at the puskesmas. Not
a diagnosis.",
  "cases": [
    {"facts": ["fever = yes"], "conclusions": ["advice = fever home care"], "option": 5,
"author": "Ratna"},
    {"facts": ["cough = yes"], "conclusions": ["advice = cough home care"], "option": 5,
"author": "Ratna"},
    {"facts": ["diarrhea = yes"], "conclusions": ["advice = ORS at home"], "option": 5,
"author": "Ratna"},
    {"facts": ["headache = yes"], "conclusions": ["advice = headache home care"],
"option": 5, "author": "Ratna"},
    {"facts": ["fever = yes", "fever days >= 2"], "conclusions": ["advice = dengue
check"], "option": 2, "node": "fever home care#1", "author": "Ratna"},
    {"facts": ["cough = yes", "cough weeks >= 2"], "conclusions": ["advice = TB check"],
"option": 2, "node": "cough home care#1", "author": "Ratna"},
    {"facts": ["cough = yes", "smoker = yes", "cough weeks < 2"], "conclusions": ["advice
= quit smoking help"], "option": 2, "node": "cough home care#1", "author": "Ratna"},
    {"facts": ["headache = yes", "blood pressure >= 140"], "conclusions": ["advice =
blood pressure check"], "option": 2, "node": "headache home care#1", "author": "Ratna"},
    {"facts": ["diarrhea = yes", "diarrhea days >= 3"], "conclusions": ["advice =
diarrhea check"], "option": 2, "node": "ORS at home#1", "author": "Ratna"}
  ]
}
```

It builds the same tree as the teaching session of Section 5.4. Where the nurse chose a condition, the seed case states it directly — `fever days >= 2` — and names the option (2) and the node it refines (`"node": "fever home care#1"`), so that the engine builds exactly the exception she chose.

The conclusions are short labels, such as `dengue check`, not whole sentences. That keeps node names readable (`dengue check#1`) and lets the puskesmas change the *wording* of a piece of advice without touching the *knowledge*. The wording lives in a separate file, `advice-texts.txt`, one line per label:

```
# The wording of each piece of advice, kept apart from the knowledge.
# One line per advice: label = text. Lines starting with # are notes.
fever home care = Rest, drink plenty of water, and take paracetamol as written on the
label. Measure your temperature morning and evening.
dengue check = A fever of two days or more needs a check: go to the puskesmas today and
ask whether a blood test is needed. Until then take paracetamol only, not ibuprofen or
aspirin.
cough home care = Rest, drink warm water, and cover your mouth when you cough. Wear a
mask near babies and old people.
TB check = A cough of two weeks or more should be checked at the puskesmas, including for
TB. TB tests and medicine there are free.
quit smoking help = Smoking keeps a cough going. Ask the puskesmas about its quit-smoking
service, or call the free quitline 0-800-177-6565.
ORS at home = Drink oralit (ORS) after every loose stool, keep eating small meals, and
wash your hands with soap.
diarrhea check = Diarrhea for three days or more needs a check at the puskesmas.
report food poisoning = Several people sick after the same food may be food poisoning:
tell the puskesmas today, so that others can be warned.
headache home care = Rest, drink water, and take paracetamol as written on the label.
blood pressure check = A top number of 140 or more is high. Measure again on another day
at the posbindu or puskesmas; high blood pressure needs a doctor's care.
```

When a doctor at the puskesmas asks for a phrase to be changed, the nurse edits one line of this file; the knowledge base, its cornerstones and its history stay as they are.

5.5.2 The program

Listing 5.5 is the self-check in Java.

Listing 5.5 `HealthCheck.java`: the health self-check, safety first, on vcirs-core.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

import vcirs.Case;
import vcirs.CaseFile;
import vcirs.Conclusion;
import vcirs.Condition;
import vcirs.DifferenceList;
import vcirs.Fact;
import vcirs.Facts;
import vcirs.InferenceResult;
import vcirs.KnowledgeBase;
import vcirs.Node;
import vcirs.Question;
import vcirs.RdrPath;
import vcirs.Report;
import vcirs.Variable;
import vcirs.VcirsException;

/**
 * Health self-check: safe first, then self-care advice that a nurse teaches
 * case by case. It is not a diagnosis.
 * java HealthCheck [--echo] [--nurse] [--tree] [knowledge-base.json]
 * The safety questions come first and are fixed in the program: if one of
 * them is answered yes, the program sends the person to help at once and
 * asks nothing more. Only then does it consult the knowledge base, following
 * every piece of advice that applies. With --nurse, the nurse checks the
 * advice and teaches the knowledge base: she chooses the node, the option
 * and the conditions, and sees which earlier cases the new node would affect.
 */
public class HealthCheck {
    // Safety first: fixed rules that no teaching can change or remove.
    static final String ADULT = "Is this self-check for an adult (18 or older) "
            + "who is not pregnant?";
    static final String NOT_ADULT = "This self-check is for adults who are not "
            + "pregnant. For a child, or during pregnancy, see the bidan or the "
            + "puskesmas.";
    static final String[][] RED_FLAGS = {
```

Listing 5.5 (Continued) *HealthCheck.java: the health self-check, safety first, on vcirs-core.*

```
        {"Right now: sudden weakness or numbness on one side, a drooping face, "
            + "slurred speech, sudden trouble seeing, or the worst headache ever?",
            "EMERGENCY: these can be signs of a stroke. Call 119 or go to the "
            + "nearest hospital emergency room (IGD) now."},
        {"Chest pain or pressure, or trouble breathing?",
            "EMERGENCY: call 119 or go to the IGD now."},
        {"Fever with any of: severe belly pain, vomiting 3 or more times a day, "
            + "bleeding gums or nose, blood in vomit or stool, great weakness?",
            "URGENT: these are warning signs of severe dengue. Go to the IGD now."},
        {"Unable to keep water down, or passing very little urine?",
            "URGENT: the body may be losing too much water. Go to the puskesmas "
            + "or the IGD now."},
    };
    static final String[][] QUESTIONS = {
        {"fever", "Do you have a fever?", "yes", "no"},
        {"fever days", "For how many days have you had the fever?"},
        {"cough", "Do you have a cough?", "yes", "no"},
        {"cough weeks", "For how many weeks have you been coughing?"},
        {"smoker", "Do you smoke?", "yes", "no"},
        {"diarrhea", "Diarrhea (three or more loose stools a day)?", "yes", "no"},
        {"diarrhea days", "For how many days have you had diarrhea?"},
        {"others ill", "Are others who ate the same food sick too?", "yes", "no"},
        {"headache", "Do you have a headache?", "yes", "no"},
        {"blood pressure", "Top number of your blood pressure, if measured today?"},
    };

    static BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in, StandardCharsets.UTF_8));
    static boolean echo = false;
    static KnowledgeBase kb;

    /** Break text into lines of at most width characters, at spaces. */
    static List<String> wrap(String text, int width) {
        List<String> lines = new ArrayList<>();
        String line = "";
        for (String word : text.split(" ", -1)) {
            if (!line.isEmpty() && line.length() + 1 + word.length() > width) {
                lines.add(line);
                line = word;
            } else {
                line = line.isEmpty() ? word : line + " " + word;
            }
        }
        lines.add(line);
        return lines;
    }
```

Listing 5.5 (Continued) *HealthCheck.java: the health self-check, safety first, on vcirs-core.*

```
    /** Print text wrapped, the first line after first, the others after rest. */
    static void show(String first, String rest, String text) {
        String body = text.stripLeading();
        String lead = " ".repeat(text.length() - body.length());  // the indent
        List<String> lines = wrap(body, 72);
        for (int i = 0; i < lines.size(); i++) {
            System.out.println((i == 0 ? first + lead : rest) + lines.get(i));
        }
    }

    /** Reads the user's answer; with echo on, prints it too. */
    static String read(String prompt) {
        String bare = prompt.replaceAll("[ >]+$", "");
        List<String> lines = wrap(bare, 74);              // long questions wrap
        for (int i = 0; i < lines.size() - 1; i++) {
            System.out.println(lines.get(i));
        }
        System.out.print(lines.get(lines.size() - 1) + " > ");
        String line;
        try {
            line = in.readLine();
        } catch (IOException e) {
            line = null;
        }
        String text = line == null ? "" : line.strip();
        if (echo) {
            System.out.println(text);
        }
        return text;
    }

    static boolean yes(String question) {
        return read(question + " (yes/no) > ").equals("yes");
    }
```

Listing 5.5 (Continued) *HealthCheck.java: the health self-check, safety first, on vcirs-core.*

```
    static Fact ask(Question q) {
        Variable var = kb.variables.get(q.variable);
        String text = q.variable;
        List<String> choices = q.choices;
        if (var != null && !var.question.isEmpty()) {
            text = var.question;
            choices = var.choices;
        }
        if (!choices.isEmpty()) {
            text += " (" + String.join("/", choices) + ")";
        }
        String answer = read(text + " > ");
        return answer.isEmpty() ? null : new Fact(q.variable, answer); // Enter
    }

    /** advice-texts.txt: one "label = text" per line. */
    static Map<String, String> readTexts(Path path) {
        Map<String, String> texts = new LinkedHashMap<>();
        try {
            for (String line : Files.readAllLines(path, StandardCharsets.UTF_8)) {
                int at = line.indexOf('=');
                if (at >= 0 && !line.startsWith("#")) {
                    texts.put(line.substring(0, at).strip(),
                            line.substring(at + 1).strip());
                }
            }
        } catch (IOException e) {
            throw new VcirsException("cannot read " + path);
        }
        return texts;
    }

    /** Load the knowledge base, or build it from the seed cases. */
    static KnowledgeBase openKb(String path) {
        if (Files.exists(Path.of(path))) {
            return KnowledgeBase.load(path);
        }
        Path parent = Path.of(path).toAbsolutePath().getParent();
        CaseFile seed = CaseFile.read(parent.resolve("health-seed-cases.json")
                .toString());
        KnowledgeBase k = new KnowledgeBase(seed.name, seed.description);
        for (String[] q : QUESTIONS) {
            List<String> choices = new ArrayList<>();
            for (int i = 2; i < q.length; i++) {
                choices.add(q[i]);
            }
```

Listing 5.5 (Continued) *HealthCheck.java: the health self-check, safety first, on vcirs-core.*

```
            k.declare(q[0], q[1], choices);
        }
        seed.build(k);
        k.save(path);
        return k;
    }

    /** The fixed safety questions. True when self-care advice may follow. */
    static boolean safe() {
        if (!yes(ADULT)) {
            show("", "", NOT_ADULT);
            return false;
        }
        for (String[] flag : RED_FLAGS) {
            if (yes(flag[0])) {
                show("", "", flag[1]);
                return false;
            }
        }
        return true;
    }

    static String join(List<Condition> conds) {
        List<String> parts = new ArrayList<>();
        for (Condition c : conds) {
            parts.add(c.toString());
        }
        return String.join(", ", parts);
    }

    /** The conditions an option builds from the list of difference. */
    static List<Condition> builtBy(int option, Node node, Case c) {
        List<Condition> out = new ArrayList<>();
        if (option >= 1 && option <= 3) {
            DifferenceList d = kb.differences(node.id, c);
            if (option == 1 || option == 3) {
                for (Condition x : d.a) {
                    out.add(x.negated());
                }
            }
            if (option == 2 || option == 3) {
                out.addAll(d.b);
            }
            return out;
        }
        out.addAll(c.facts);                          // options 4 and 5
```

Listing 5.5 (Continued) *HealthCheck.java: the health self-check, safety first, on vcirs-core.*

```
        return out;
    }

    static List<Condition> factsOf(InferenceResult result) {
        List<Condition> facts = new ArrayList<>();
        for (Fact f : result.memory.facts.values()) {
            if (!f.variable.equals("advice")) {
                facts.add(new Condition(f.variable, "=", f.value));
            }
        }
        return facts;
    }

    /** The nurse corrects or completes the advice, choosing everything. */
    static void teach(InferenceResult result, String path) {
        String advice = read("Which advice is right or missing? (label) > ");
        if (advice.isEmpty()) {
            return;
        }
        List<Node> fired = new ArrayList<>();
        for (RdrPath p : result.paths) {
            fired.add(p.last());
        }
        for (int i = 0; i < fired.size(); i++) {
            System.out.println("  " + (i + 1) + ") " + fired.get(i).id);
        }
        String pick = read("Which node gave wrong or incomplete advice? "
                + "(number, Enter for none) > ");
        Node node = pick.isEmpty() ? null : fired.get(Integer.parseInt(pick) - 1);
        List<Condition> facts = factsOf(result);
        while (true) {
            String extra = read("Another fact that matters? "
                    + "(variable = value, Enter when done) > ");
            if (extra.isEmpty()) {
                break;
            }
            try {
                facts.add(Condition.parse(extra));
            } catch (VcirsException e) {
                System.out.println("Please write it as: variable = value");
            }
        }
        Case c = new Case(facts, List.of(new Conclusion("advice", advice)), 100);
        if (node != null) {
            for (String line : kb.differences(node.id, c).lines()) {
                show("", "    ", line);
```

Listing 5.5 (Continued) *HealthCheck.java: the health self-check, safety first, on vcirs-core.*

```
            }
        }
        int option = Integer.parseInt(read("Option (0 accept, 1-3 exception, "
                + "4 beside, 5 top level) > "));
        if (option != 5 && node == null) {
            System.out.println("Options 0 to 4 need a node; nothing was changed.");
            return;
        }
        if (option == KnowledgeBase.ACCEPT) {
            kb.refine(c, KnowledgeBase.ACCEPT, node.id, null, "nurse");
            kb.save(path);
            System.out.println("Accepted; the check is recorded.");
            return;
        }
        List<Condition> conds = builtBy(option, node, c);
        show("", "  ", "Option " + option + " builds: " + join(conds));
        String keep = read("Conditions to keep, separated by ';' "
                + "(Enter = as built) > ");
        if (!keep.isEmpty()) {
            conds = new ArrayList<>();
            for (String k : keep.split(";")) {
                conds.add(Condition.parse(k));
            }
        }
        String parent = option == 5 ? null
                : option == 4 ? node.parent : node.id;
        while (true) {
            List<String> hit = kb.affectedCornerstones(parent, conds,
                    c.conclusions);
            System.out.println("Earlier cases whose advice would change: "
                    + (hit.isEmpty() ? "none" : String.join(", ", hit)));
            String more = hit.isEmpty() ? "" : read("Add a condition to tell "
                    + "them apart? (Enter to go ahead) > ");
            if (more.isEmpty()) {
                break;
            }
            conds = new ArrayList<>(conds);
            conds.add(Condition.parse(more));
        }
        Node learned;
        try {
            learned = kb.refine(c, option, node == null ? null : node.id, conds,
                    "nurse");
        } catch (VcirsException e) {
            System.out.println("Cannot learn: " + e.getMessage());
            return;
```

Listing 5.5 (Continued) *HealthCheck.java: the health self-check, safety first, on vcirs-core.*

```
        }
        System.out.println("Learned: " + learned);
        kb.save(path);
        System.out.println("Saved " + path + ": " + kb.summary());
    }

    public static void main(String[] args) {
        String path = "../data/health-kb.json";
        boolean tree = false;
        boolean nurse = false;
        for (String a : args) {
            if (a.equals("--echo")) {
                echo = true;
            } else if (a.equals("--tree")) {
                tree = true;
            } else if (a.equals("--nurse")) {
                nurse = true;
            } else {
                path = a;
            }
        }
        kb = openKb(path);
        if (tree) {
            for (String line : Report.tree(kb)) {
                System.out.println(line);
            }
            return;
        }
        Map<String, String> texts = readTexts(Path.of(path).toAbsolutePath()
                .getParent().resolve("advice-texts.txt"));
        System.out.println("Health self-check for adults. It is not a "
                + "diagnosis: it tells you");
        System.out.println("what you can do yourself, and when to see a health "
                + "worker.");
        System.out.println("First, the safety questions.");
        if (!safe()) {
            return;
        }
        System.out.println("Now about how you feel. Press Enter if you do not "
                + "know.");
        InferenceResult result = kb.infer(Facts.none(), HealthCheck::ask, true);
        if (result.paths.isEmpty()) {
            System.out.println("The self-check has no advice for this. If you "
                    + "are worried, see the puskesmas.");
        }
        for (RdrPath p : result.paths) {
```

Listing 5.5 (Continued) *HealthCheck.java: the health self-check, safety first, on vcirs-core.*

```
            for (Conclusion c : p.conclusions()) {
                show("* ", "  ", texts.getOrDefault(c.value, c.value));
            }
            for (Conclusion c : p.intermediate()) {
                show("  and: ", "  ", texts.getOrDefault(c.value, c.value));
            }
            List<Condition> used = new ArrayList<>();
            for (Node n : p.nodes) {
                used.addAll(n.conditions);
            }
            show("  (", "  ", p.ruleId() + ": " + join(used) + ")");
        }
        show("", "", "If anything gets worse or a warning sign appears, see the "
                + "puskesmas at once or call 119.");
        if (!nurse) {
            return;
        }
        if (yes("Nurse: is the advice right and complete?")) {
            Case c = new Case(factsOf(result), result.conclusions(), 100);
            for (RdrPath p : result.paths) {
                kb.refine(c, KnowledgeBase.ACCEPT, p.last().id, null, "nurse");
            }
            kb.save(path);
            System.out.println("Good. The check is recorded.");
        } else {
            teach(result, path);
        }
    }
}
```

It is the longest program so far, so take it in five parts.

Safety. `ADULT`, `NOT_ADULT` and `RED_FLAGS` are the fixed safety rules of Section 5.2.1, each a question and the advice to give on a *yes*. `safe` asks them in order and stops at the first *yes*. They are constants in the program, not nodes in the knowledge base, and no teaching can change them.

Words on a page. `wrap` breaks a long text into lines of at most a given width, at spaces, and `show` prints a wrapped text with a prefix on the first line (for example `*`) and an indent on the others, keeping any indent the text already has. `read` wraps long questions the same way. This is all about people: a question or a piece of advice that runs off the edge of a phone screen is not read.

The knowledge base. `openKb` declares the ten variables of `QUESTIONS` and builds the knowledge base from the seed cases the first time, exactly as in Chapter 4. `readTexts` reads the wording of the advice. `main` then consults the knowledge base with `kb.infer(Facts.none(), HealthCheck::ask, true)` — the last argument, `true`, asks for **every** branch that applies. For each path it prints the advice of the last node after `*`, the advice of the nodes above it after `and:`, and, in parentheses, the rule and the conditions that made it, so that the person — and the nurse — can see why.

Checking. With `--nurse`, the program asks the nurse whether the advice is right and complete. If it is, every node that answered is recorded as checked with option 0.

Teaching. If it is not, `teach` walks the nurse through the refinement loop of Section 5.2: the advice that is right or missing; the node that gave wrong or incomplete advice, chosen from those that answered (or none, for a new top-level node); any other fact that matters; the list of difference; the option; what that option would build (`builtBy`); the conditions to keep, which she may type herself, separated by semicolons; and then the affected cornerstones, again and again, until the list is empty or she decides to go ahead. Only then does `refine` add the node, and the file is saved.

Listing 5.6 is the same app in Python.

Listing 5.6 `health_check.py`: the health self-check in Python.

```
"""Health self-check: safe first, then self-care advice that a nurse
teaches case by case. It is not a diagnosis.

    python health_check.py [--echo] [--nurse] [--tree] [knowledge-base.json]

The safety questions come first and are fixed in the program: if one of
them is answered yes, the program sends the person to help at once and asks
nothing more. Only then does it consult the knowledge base, following every
piece of advice that applies. With --nurse, the nurse checks the advice and
teaches the knowledge base: she chooses the node, the option and the
conditions, and sees which earlier cases the new node would affect.
"""

import os
import pathlib
import sys

# Tell Python where the vcirs-core engine is (two folders up, then across).
sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import (ACCEPT, Case, Condition, Conclusion, KnowledgeBase,
                   VcirsError, casefile, report)

# Safety first: fixed rules that no teaching can change or remove.
ADULT = "Is this self-check for an adult (18 or older) who is not pregnant?"
NOT_ADULT = ("This self-check is for adults who are not pregnant. For a "
             "child, or during pregnancy, see the bidan or the puskesmas.")
RED_FLAGS = [
    ("Right now: sudden weakness or numbness on one side, a drooping face, "
     "slurred speech, sudden trouble seeing, or the worst headache ever?",
     "EMERGENCY: these can be signs of a stroke. Call 119 or go to the "
     "nearest hospital emergency room (IGD) now."),
    ("Chest pain or pressure, or trouble breathing?",
     "EMERGENCY: call 119 or go to the IGD now."),
    ("Fever with any of: severe belly pain, vomiting 3 or more times a day, "
     "bleeding gums or nose, blood in vomit or stool, great weakness?",
     "URGENT: these are warning signs of severe dengue. Go to the IGD now."),
    ("Unable to keep water down, or passing very little urine?",
     "URGENT: the body may be losing too much water. Go to the puskesmas or "
     "the IGD now."),
]
QUESTIONS = [
    ("fever", "Do you have a fever?", ["yes", "no"]),
    ("fever days", "For how many days have you had the fever?", []),
    ("cough", "Do you have a cough?", ["yes", "no"]),
    ("cough weeks", "For how many weeks have you been coughing?", []),
```

Listing 5.6 (Continued) *health_check.py: the health self-check in Python.*

```
    ("smoker", "Do you smoke?", ["yes", "no"]),
    ("diarrhea", "Diarrhea (three or more loose stools a day)?", ["yes", "no"]),
    ("diarrhea days", "For how many days have you had diarrhea?", []),
    ("others ill", "Are others who ate the same food sick too?", ["yes", "no"]),
    ("headache", "Do you have a headache?", ["yes", "no"]),
    ("blood pressure", "Top number of your blood pressure, if measured today?",
     []),
]


def wrap(text, width):
    """Break text into lines of at most width characters, at spaces."""
    lines, line = [], ""
    for word in text.split(" "):
        if line and len(line) + 1 + len(word) > width:
            lines.append(line)
            line = word
        else:
            line = word if not line else line + " " + word
    return lines + [line]


def show(first, rest, text):
    """Print text wrapped, the first line after 'first', the others after
    'rest' (for example '* ' and '  ')."""
    lead = " " * (len(text) - len(text.lstrip(" ")))  # keep the indent
    for i, line in enumerate(wrap(text.lstrip(" "), 72)):
        print((first + lead if i == 0 else rest) + line)


class Console:
    """Reads the user's answers; with echo on, prints them too."""

    def __init__(self, kb, echo):
        self.kb = kb
        self.echo = echo
```

Listing 5.6 (Continued) *health_check.py: the health self-check in Python.*

```
    def read(self, prompt):
        lines = wrap(prompt.rstrip(" >"), 74)       # long questions wrap
        for line in lines[:-1]:
            print(line)
        print(lines[-1] + " > ", end="")
        text = sys.stdin.readline().strip()
        if self.echo:
            print(text)
        return text

    def yes(self, question):
        return self.read(question + " (yes/no) > ") == "yes"

    def ask(self, question):
        var = self.kb.variables.get(question.variable)
        text, choices = question.variable, question.choices
        if var is not None and var.question:
            text, choices = var.question, var.choices
        if choices:
            text += " (" + "/".join(choices) + ")"
        return self.read(text + " > ") or None    # Enter = "I do not know"


def read_texts(path):
    """advice-texts.txt: one 'label = text' per line."""
    texts = {}
    with open(path, encoding="utf-8") as f:
        for line in f:
            if "=" in line and not line.startswith("#"):
                label, text = line.split("=", 1)
                texts[label.strip()] = text.strip()
    return texts


def open_kb(path):
    """Load the knowledge base, or build it from the seed cases."""
    if os.path.exists(path):
        return KnowledgeBase.load(path)
    seed = os.path.join(os.path.dirname(path), "health-seed-cases.json")
    name, description, steps = casefile.read_cases(seed)
    kb = KnowledgeBase(name, description)
    for var, question, choices in QUESTIONS:
        kb.declare(var, question, choices)
    casefile.build(kb, steps)
    kb.save(path)
    return kb
```

Listing 5.6 (Continued) *health_check.py: the health self-check in Python.*

```
def safe(con):
    """The fixed safety questions. True when self-care advice may follow."""
    if not con.yes(ADULT):
        show("", "", NOT_ADULT)
        return False
    for question, advice in RED_FLAGS:
        if con.yes(question):
            show("", "", advice)
            return False
    return True


def built_by(kb, option, node, case):
    """The conditions an option builds from the list of difference."""
    if option in (1, 2, 3):
        d = kb.differences(node.id, case)
        neg = [c.negated() for c in d.a] if option in (1, 3) else []
        return neg + (d.b if option in (2, 3) else [])
    return list(case.facts)                      # options 4 and 5


def teach(kb, con, result, path):
    """The nurse corrects or completes the advice, choosing everything."""
    advice = con.read("Which advice is right or missing? (label) > ")
    if not advice:
        return
    fired = [p.last() for p in result.paths]
    for i, node in enumerate(fired):
        print("  " + str(i + 1) + ") " + node.id)
    pick = con.read("Which node gave wrong or incomplete advice? "
                    "(number, Enter for none) > ")
    node = fired[int(pick) - 1] if pick else None
    facts = [Condition(f.variable, "=", f.value)
             for f in result.memory.facts.values() if f.variable != "advice"]
    while True:
        extra = con.read("Another fact that matters? "
                         "(variable = value, Enter when done) > ")
        if not extra:
            break
        try:
            facts.append(Condition.parse(extra))
        except VcirsError:
            print("Please write it as: variable = value")
    case = Case(facts, [Conclusion("advice", advice)])
```

Listing 5.6 (Continued) *health_check.py: the health self-check in Python.*

```
    if node is not None:
        for line in kb.differences(node.id, case).lines():
            show("", "    ", line)
    option = int(con.read("Option (0 accept, 1-3 exception, 4 beside, "
                          "5 top level) > "))
    if option != 5 and node is None:
        print("Options 0 to 4 need a node; nothing was changed.")
        return
    if option == ACCEPT:
        kb.refine(case, ACCEPT, node.id, author="nurse")
        kb.save(path)
        print("Accepted; the check is recorded.")
        return
    conds = built_by(kb, option, node, case)
    show("", "  ", "Option " + str(option) + " builds: "
         + ", ".join(str(c) for c in conds))
    keep = con.read("Conditions to keep, separated by ';' "
                    "(Enter = as built) > ")
    if keep:
        conds = [Condition.parse(c) for c in keep.split(";")]
    parent = {4: node.parent if node else None, 5: None}.get(
        option, node.id if node else None)
    while True:
        hit = kb.affected_cornerstones(parent, conds, case.conclusions)
        print("Earlier cases whose advice would change: "
              + (", ".join(hit) if hit else "none"))
        more = con.read("Add a condition to tell them apart? "
                        "(Enter to go ahead) > ") if hit else ""
        if not more:
            break
        conds = conds + [Condition.parse(more)]
    try:
        new = kb.refine(case, option, node.id if node else None,
                        chosen=conds, author="nurse")
    except VcirsError as e:
        print("Cannot learn: " + str(e))
        return
    print("Learned: " + str(new))
    kb.save(path)
    print("Saved " + path + ": " + kb.summary())
```

Listing 5.6 (Continued) *health_check.py: the health self-check in Python.*

```
def main(args):
    echo = "--echo" in args
    files = [a for a in args if not a.startswith("--")]
    path = files[0] if files else "../data/health-kb.json"
    kb = open_kb(path)
    if "--tree" in args:
        for line in report.tree(kb):
            print(line)
        return
    texts = read_texts(os.path.join(os.path.dirname(path),
                                    "advice-texts.txt"))
    con = Console(kb, echo)
    print("Health self-check for adults. It is not a diagnosis: it tells you")
    print("what you can do yourself, and when to see a health worker.")
    print("First, the safety questions.")
    if not safe(con):
        return
    print("Now about how you feel. Press Enter if you do not know.")
    result = kb.infer({}, con.ask, multiple=True)
    if not result.paths:
        print("The self-check has no advice for this. If you are worried, "
              "see the puskesmas.")
    for p in result.paths:
        for c in p.conclusions():
            show("* ", "  ", texts.get(c.value, c.value))
        for c in p.intermediate():
            show("  and: ", "  ", texts.get(c.value, c.value))
        show("  (", "  ", p.rule_id() + ": " + ", ".join(
            str(c) for n in p.nodes for c in n.conditions) + ")")
    show("", "", "If anything gets worse or a warning sign appears, see the "
         "puskesmas at once or call 119.")
    if "--nurse" not in args:
        return
    if con.yes("Nurse: is the advice right and complete?"):
        case = Case([Condition(f.variable, "=", f.value)
                     for f in result.memory.facts.values()
                     if f.variable != "advice"], result.conclusions())
        for p in result.paths:
            kb.refine(case, ACCEPT, p.last().id, author="nurse")
        kb.save(path)
        print("Good. The check is recorded.")
    else:
        teach(kb, con, result, path)
```

Listing 5.6 (Continued) *health_check.py: the health self-check in Python.*

```
if __name__ == "__main__":
    main(sys.argv[1:])
```

5.6 Four People at the Puskesmas

Compile and run from `Chapter05/java`:

```
javac -d out -sourcepath ../../vcirs-core/java/src *.java
java -cp out HealthCheck --echo < ../data/session-1.txt
```

or, from `Chapter05/python`, `python health_check.py --echo < ../data/session-1.txt`. Sessions 3 and 4 are run with `--nurse` as well. Run the four sessions in order on a fresh knowledge base, and the transcripts and the saved `health-kb.json` are the same, byte for byte, in both languages.

Session 1: Pak Darmo, on the third day of his fever. His daughter runs the self-check on her phone for him.

```
Health self-check for adults. It is not a diagnosis: it tells you
what you can do yourself, and when to see a health worker.
First, the safety questions.
Is this self-check for an adult (18 or older) who is not pregnant?
(yes/no) > yes
Right now: sudden weakness or numbness on one side, a drooping face,
slurred speech, sudden trouble seeing, or the worst headache ever?
(yes/no) > no
Chest pain or pressure, or trouble breathing? (yes/no) > no
Fever with any of: severe belly pain, vomiting 3 or more times a day,
bleeding gums or nose, blood in vomit or stool, great weakness? (yes/no) > no
Unable to keep water down, or passing very little urine? (yes/no) > no
Now about how you feel. Press Enter if you do not know.
Do you have a fever? (yes/no) > yes
For how many days have you had the fever? > 3
Do you have a cough? (yes/no) > no
Diarrhea (three or more loose stools a day)? (yes/no) > no
Do you have a headache? (yes/no) > yes
Top number of your blood pressure, if measured today? >
* A fever of two days or more needs a check: go to the puskesmas today and
  ask whether a blood test is needed. Until then take paracetamol only,
  not ibuprofen or aspirin.
  and: Rest, drink plenty of water, and take paracetamol as written on the
  label. Measure your temperature morning and evening.
  (dengue check#1: fever = yes, fever days >= 2)
* Rest, drink water, and take paracetamol as written on the label.
  (headache home care#1: headache = yes)
If anything gets worse or a warning sign appears, see the puskesmas at
once or call 119.
```

The five safety questions come first, and all the answers are *no*. Then the knowledge base asks only what its nodes need: the fever and how long it has lasted, the cough, the diarrhea, the headache and — because the headache node has an exception about blood pressure — the top number of his blood pressure, which nobody measured today, so he presses Enter. Two pieces of advice come back, one per branch. The first is the dengue check, because the fever has lasted three days, with the fever home care of the node above it; the second is the headache home care. The line in parentheses under each shows its rule and conditions: `dengue check#1: fever = yes, fever days >= 2`. The last line is the same for everyone: when anything gets worse, go.

The advice says *today*, and *paracetamol only, not ibuprofen or aspirin*, because WHO warns that these two medicines increase the risk of bleeding in dengue [1]. Pak Darmo's daughter takes him to the puskesmas that afternoon.

Session 2: Bu Sum, a jamu seller, with a warning sign. She has had a fever for four days, and this morning her gums bled when she brushed her teeth.

```
Health self-check for adults. It is not a diagnosis: it tells you
what you can do yourself, and when to see a health worker.
First, the safety questions.
Is this self-check for an adult (18 or older) who is not pregnant?
(yes/no) > yes
Right now: sudden weakness or numbness on one side, a drooping face,
slurred speech, sudden trouble seeing, or the worst headache ever?
(yes/no) > no
Chest pain or pressure, or trouble breathing? (yes/no) > no
Fever with any of: severe belly pain, vomiting 3 or more times a day,
bleeding gums or nose, blood in vomit or stool, great weakness? (yes/no) > yes
URGENT: these are warning signs of severe dengue. Go to the IGD now.
```

The third safety question catches it, and the self-check says only one thing: these are warning signs of severe dengue; go to the IGD now [1], [7]. It does not ask about her cough or her headache, and it does not consult the knowledge base at all. This is the arrangement of **Figure 5.1** working as designed.

Session 3: Mas Yanto, and the nurse teaches. Mas Yanto, a parking attendant at the market, has had diarrhea since yesterday. He comes to the posbindu, and Ners Ratna runs the self-check with him in nurse mode.

```
Health self-check for adults. It is not a diagnosis: it tells you
what you can do yourself, and when to see a health worker.
First, the safety questions.
Is this self-check for an adult (18 or older) who is not pregnant?
(yes/no) > yes
Right now: sudden weakness or numbness on one side, a drooping face,
slurred speech, sudden trouble seeing, or the worst headache ever?
(yes/no) > no
Chest pain or pressure, or trouble breathing? (yes/no) > no
Fever with any of: severe belly pain, vomiting 3 or more times a day,
bleeding gums or nose, blood in vomit or stool, great weakness? (yes/no) > no
Unable to keep water down, or passing very little urine? (yes/no) > no
Now about how you feel. Press Enter if you do not know.
Do you have a fever? (yes/no) > no
Do you have a cough? (yes/no) > no
Diarrhea (three or more loose stools a day)? (yes/no) > yes
For how many days have you had diarrhea? > 1
Do you have a headache? (yes/no) > no
* Drink oralit (ORS) after every loose stool, keep eating small meals, and
  wash your hands with soap.
  (ORS at home#1: diarrhea = yes)
If anything gets worse or a warning sign appears, see the puskesmas at
once or call 119.
Nurse: is the advice right and complete? (yes/no) > no
Which advice is right or missing? (label) > report food poisoning
  1) ORS at home#1
Which node gave wrong or incomplete advice? (number, Enter for none) > 1
Another fact that matters? (variable = value, Enter when done) > others ill = yes
Another fact that matters? (variable = value, Enter when done) >
Differences with the cornerstone case of ORS at home#1:
  A (only in the old case): (none)
  B (only in the new case): fever = no, cough = no, diarrhea days = 1,
    headache = no, others ill = yes
Option (0 accept, 1-3 exception, 4 beside, 5 top level) > 2
Option 2 builds: fever = no, cough = no, diarrhea days = 1, headache =
  no, others ill = yes
Conditions to keep, separated by ';' (Enter = as built) > others ill = yes
Earlier cases whose advice would change: none
Learned: report food poisoning#1: others ill = yes -> advice = report food poisoning
Saved ../data/health-kb.json: Health self-check: 11 variables, 10 nodes, 10 rules
```

The knowledge base gives the ORS advice, which is right as far as it goes. But while talking to him, Ners Ratna learns that everyone who ate the *nasi kotak* at last night's wedding party is sick too. That is not just diarrhea; it may be food poisoning, and the puskesmas must know today, so that other guests can be warned. The advice is incomplete, so she answers *no*.

The teaching screen does the rest. She types the missing advice, `report food poisoning`; picks the node that gave the incomplete advice, `ORS at home#1`; and adds the fact that matters, `others ill = yes`. The list of difference against the cornerstone of `ORS at home#1` (which was only `diarrhea = yes`) has an empty section A and a long section B: every answer Mas Yanto gave — *no fever, no cough, one day of diarrhea, no headache* — and the fact she added. This is the price of asking about every complaint: most facts of a real case are irrelevant to the new advice. Option 2

would build all five into the node, and the node would then only ever fire for a person with *exactly* this pattern. She types the one condition that is the reason, `others ill = yes`. No earlier case would change its advice, and `report food poisoning#1` is added under `ORS at home#1`.

Session 4: Bu Tini, the next morning. She was at the same wedding.

```
Health self-check for adults. It is not a diagnosis: it tells you
what you can do yourself, and when to see a health worker.
First, the safety questions.
Is this self-check for an adult (18 or older) who is not pregnant?
(yes/no) > yes
Right now: sudden weakness or numbness on one side, a drooping face,
slurred speech, sudden trouble seeing, or the worst headache ever?
(yes/no) > no
Chest pain or pressure, or trouble breathing? (yes/no) > no
Fever with any of: severe belly pain, vomiting 3 or more times a day,
bleeding gums or nose, blood in vomit or stool, great weakness? (yes/no) > no
Unable to keep water down, or passing very little urine? (yes/no) > no
Now about how you feel. Press Enter if you do not know.
Do you have a fever? (yes/no) > no
Do you have a cough? (yes/no) > no
Diarrhea (three or more loose stools a day)? (yes/no) > yes
For how many days have you had diarrhea? > 2
Are others who ate the same food sick too? (yes/no) > yes
Do you have a headache? (yes/no) > no
* Several people sick after the same food may be food poisoning: tell the
  puskesmas today, so that others can be warned.
  and: Drink oralit (ORS) after every loose stool, keep eating small meals, and
  wash your hands with soap.
  (report food poisoning#1: diarrhea = yes, others ill = yes)
If anything gets worse or a warning sign appears, see the puskesmas at
once or call 119.
Nurse: is the advice right and complete? (yes/no) > yes
Good. The check is recorded.
```

Now the self-check asks a question it never asked before — *Are others who ate the same food sick too?* — because the node learned yesterday needs it, and the question was already declared with its wording. The advice says to tell the puskesmas today, with the ORS advice from the node above. Ners Ratna confirms it, and the check is recorded. By noon the puskesmas has heard from six guests, and the district health office is looking at the caterer.

`--tree` shows the knowledge base after the four sessions:

```
(root)
+- fever home care#1: fever = yes -> advice = fever home care
|  +- dengue check#1: fever days >= 2 -> advice = dengue check
+- cough home care#1: cough = yes -> advice = cough home care
|  +- TB check#1: cough weeks >= 2 -> advice = TB check
|  +- quit smoking help#1: smoker = yes, cough weeks < 2 -> advice = quit smoking help
+- ORS at home#1: diarrhea = yes -> advice = ORS at home
|  +- diarrhea check#1: diarrhea days >= 3 -> advice = diarrhea check
|  +- report food poisoning#1: others ill = yes -> advice = report food poisoning
+- headache home care#1: headache = yes -> advice = headache home care
   +- blood pressure check#1: blood pressure >= 140 -> advice = blood pressure check
```

The only change since the teaching session of Section 5.4 is the new exception under `ORS at home#1`, beside `diarrhea check#1`: both are tried for everyone with diarrhea, and in multiple mode both can answer.

5.7 Making It Better

Stopping advice that has become wrong. Suppose the puskesmas doctor asks that the headache advice no longer mention paracetamol, because too many people take it every day. Ners Ratna stops `headache home care#1` and teaches the new advice as a new node. Exercise 5.7 shows that the stopped node keeps working as a context for its exception — the blood-pressure check still fires — while its own advice is no longer given. Nothing is deleted, and the history records the stop.

Children and pregnancy. The self-check sends children and pregnant women straight to the bidan or the puskesmas, because their danger signs are different and must be taught by people who know them. The right way to cover them is not to add a few nodes to this knowledge base but to build a separate one, with its own fixed safety questions, taught by the bidan — the grouping by patient that the original VCIRS health system used [9].

Numbers from devices. Many families now have a digital blood-pressure monitor and a thermometer. A reading can be typed as a number, and a node such as `blood pressure >= 140` compares it as a number. One reading proves little; WHO asks for two readings on different days before calling blood pressure high [2], which is exactly what the advice says.

The same knowledge, better questions. Pak Darmo was asked about cough, diarrhea and headache even though he came about the fever, because multiple mode tries every branch. For a long list of complaints, a first question — *what is bothering you most?* — and a knowledge base grouped by complaint keeps the self-check short. The Variable table of Chapter 4 shows which variables to put first.

> **PITFALL.** Never "fix" a safety question by teaching the knowledge base. If a red flag is asked too often, or in words people do not understand, change the program and check the source again; do not add a node that answers the danger sign with home care.

Tests. `HealthTest.java` and `test_health.py` check the helpers of Step 1 (negation, comparison, an unknown fact), the teaching session (nine nodes, the conditions of `quit smoking help#1`, every cornerstone keeping its advice), and the app (the word wrap, four red flags, nine seed nodes, two pieces of advice for a fever and a headache, and a wording in `advice-texts.txt` for every piece of advice the knowledge base can give). They end with:

```
all tests passed
```

5.8 Using It for Real

At a posbindu or puskesmas. Run the self-check in nurse mode at the posbindu, with the people who come, and in normal mode on a phone or a laptop at the waiting room for people who are waiting. Teach it only from patients you have actually seen, whose situation you understand. After a few months the knowledge base holds the puskesmas's own experience: the advice for the illnesses that are common in *this* neighborhood, in the words its people understand.

Check your safety questions against current sources. The red flags in this chapter follow the WHO fact sheet on dengue [1], the Ministry of Health's SeGeRa Ke RS campaign for stroke [6] and its guidance on when dengue needs the emergency room [7]. Guidelines change; read them again every year, and whenever the Ministry of Health publishes new guidance, and change the program if they have changed.

Words people use. The questions in this chapter are in English so that the code is the same in both editions of this book. At a real puskesmas they should be in the language of the patients — Indonesian, Javanese, Madurese — and tested on a few people before use. A question that is misunderstood produces a wrong fact, and a wrong fact makes the wrong node fire.

Health data is personal data. The cases in the knowledge base hold symptoms, not names, and that is deliberate. Under Indonesia's Personal Data Protection Law, health data is a specific, sensitive kind of personal data that needs particular care [19]. A knowledge base that is meant to be shared between puskesmas should contain knowledge about situations, never about people: no names, no addresses, no phone numbers, and no combination of facts that would identify one person.

Is the self-check helping? The history of the knowledge base answers part of this question without any extra work. Count, each month, how many consultations the nurse accepted (option 0) and how many she had to correct (options 1 to 5). In the first months corrections will be common; as the knowledge base matures, the share of accepted consultations should rise. A sudden rise in corrections is a signal worth reading — a new illness in the neighborhood, a change in guidance, a misunderstood question. The other part of the question cannot be answered by the program: did people who were told *go today* actually go, and did those told *home care is enough* get better? A simple follow-up at the next posbindu, asking a few people what they did and how they are, is the only honest measure, and it is also the best source of new cases to teach.

Who teaches. Only health workers should teach the knowledge base, and every node records who taught it. When two nurses disagree, the history shows the case and the person behind each node, and the disagreement becomes a conversation about a real case. Chapter 11 builds on this for a clinic team with many experts, versions and an audit trail.

5.9 Common Mistakes, and How to Avoid Them

Putting danger signs into the learned knowledge. A red flag that lives in the knowledge base can be overridden by an exception. *Avoid it:* keep the safety questions in the program, before the knowledge base.

Accepting the option as it is. Options 2 and 3 copy every different fact of a real case, and most of them are irrelevant (session 3). *Avoid it:* read the list of difference, and keep only the facts that explain the new advice — or write the rule you actually use.

Negating when you mean a threshold. Option 1 turns `fever days = 1` into `fever days <> 1`, which also catches zero days. *Avoid it:* when section A holds a number, write the threshold you mean (`fever days >= 2`).

Ignoring the affected cornerstones. A new node that looks right may take earlier patients away from advice they need (case 8). *Avoid it:* read each affected case, and add a condition that tells it apart unless its advice really should change.

Teaching from a guess. A case taught before the patient's outcome is known may teach a mistake. *Avoid it:* teach from cases you have seen through, or at least checked with a colleague.

Deleting or editing old nodes. It destroys the guarantee that earlier cases keep their answers, and the history of who taught what. *Avoid it:* stop a node, and teach the better advice as a new one.

Advice that diagnoses. "You have dengue" is a diagnosis; "a fever of two days or more needs a check at the puskesmas today" is advice about what to do. *Avoid it:* write every conclusion as an action and a place.

Forgetting the wording file. A new advice label without a line in `advice-texts.txt` is printed as its bare label. *Avoid it:* add the wording before teaching the label; the tests check that every label has one.

5.10 Chapter Summary

- A health self-check puts **fixed safety questions** first — stroke signs, chest pain or breathlessness, the warning signs of severe dengue, losing too much water — and sends the person to 119 or the IGD on any *yes*; only then does it consult a knowledge base that can learn (**Figure 5.1**).
- **Case-by-case refinement**: for each case whose answer is missing or wrong, the expert reads the **list of difference** against the **cornerstone case** of the node that answered — section A, only in the old case; section B, only in the new — and chooses one of the **six options** (**Table 5.1**, **Figure 5.2**).
- Options build nodes mechanically, and real cases carry many irrelevant facts; the expert can **choose the conditions herself**. Worked by hand, the options were either too specific or too general, and the nurse's `fever days >= 2` was right (**Listing 5.1**, **Listing 5.2**).
- Before adding a node, the engine lists the **affected cornerstones**; in the teaching session one such case led the nurse to add `cough weeks < 2`, and afterwards every cornerstone kept its advice (**Listing 5.3**, **Listing 5.4**).
- Nodes are never deleted; advice that becomes wrong is **stopped**, and the better advice taught as a new node.
- The **health self-check** (**Listing 5.5**, **Listing 5.6**) gave every piece of advice that applied, with the rule behind each, sent a person with dengue warning signs straight to the IGD, and was taught by the nurse — node, option and conditions chosen by her — to report a probable food poisoning, which it then advised for the next guest.

5.11 Exercises

5.1 The cornerstone case of `cough home care#1` is `cough = yes, cough weeks = 1`. A new case is `cough = yes, cough weeks = 1, fever = yes, smoker = no`, and its right advice is a new one. Write sections A and B of the list of difference, and the node that each of options 1, 2 and 3 would build.

5.2 For each situation, say which option (0 to 5) you would use, and why. (a) The advice was right. (b) A person with a rash and itching, and no fever, cough, diarrhea or headache, gets no advice. (c) A person with diarrhea who has just come back from a place with cholera gets the ORS advice, but must go to the puskesmas today. (d) A person with a fever who has been bitten by a dog gets fever advice, but must get a rabies check whatever the fever.

5.3 In Step 1, option 1 built `fever days <> 1`. Explain in words which patients it catches that it should not, and why the problem is hard to notice.

5.4 Suppose that in case 8 of the teaching session Ners Ratna had put the quit-smoking node at the top level, with the single condition `smoker = yes` (option 5 with a chosen condition). Which

cornerstone cases would be affected? Would the smoker of case 7 lose his TB check in multiple mode?

5.5 A colleague suggests moving the safety questions into the knowledge base "so that nurses can improve them". Give two reasons against it and one thing you could do instead to let nurses propose improvements.

5.6 In session 1 the self-check asked about cough, diarrhea and headache although Pak Darmo came about his fever. Explain why, in terms of multiple mode, and what would change if the app called `infer` without it. Which advice would Pak Darmo have missed?

5.7 The puskesmas doctor asks that the headache advice no longer mention paracetamol. Write a program that loads the seed knowledge base, stops `headache home care#1`, teaches a new top-level node with the advice `headache rest and water`, and shows the advice for a person with a headache and for a person with a headache and a blood pressure of 150, before, after the stop, and after the new node.

5.8 The `wrap` function of the app breaks lines at spaces. What does it do with a single word longer than the width, and why is that acceptable here?

5.9 Sketch the fixed safety questions and the first three seed cases of a separate self-check for children under five, to be taught by a bidan. Which parts of `HealthCheck` would you reuse unchanged?

5.12 Solutions to the Exercises

Solution 5.1. Section A (only in the old case) is empty: both facts of the cornerstone are in the new case. Section B (only in the new case) is `fever = yes, smoker = no`. Option 1 builds nothing — the engine refuses, because an empty list of conditions cannot tell the case apart. Option 2 builds `fever = yes, smoker = no`; option 3, with an empty section A, builds the same. Whether either is right depends on the new advice: if it is about a cough *with* fever, `fever = yes` is the reason and `smoker = no` is noise, and the nurse should keep only the first.

Solution 5.2. (a) Option 0: accept; the check is recorded. (b) Option 5: nothing in the knowledge base is related to a rash, so the case becomes a new top-level node — with the conditions chosen by hand (`rash = yes`), not all the *no* answers. (c) An exception of `ORS at home#1`, option 2 with the chosen condition `cholera area = yes`, placed where the advice came from. (d) The rabies check has nothing to do with the fever; as an exception of the fever node it would only be given to bitten people who also have a fever. It belongs at the top level (option 5), with the condition `animal bite = yes` — and, better still, among the safety questions, because a bite from a possibly rabid animal needs care the same day.

Solution 5.3. `fever days <> 1` is true for every number except one: for 0 (a fever that began today), for 2, 3, 5 and 30. The patients with a fever since today are sent for a dengue check they do not need yet. The problem is hard to notice because the node is right for the case that made it and for most of the cases that come after it; the wrong answers are rare, and they are wrong in a *cautious* direction, so nobody complains. Only a look at the node's condition shows the mistake.

Solution 5.4. The engine checks the new node's conditions against every cornerstone: `smoker = yes` holds for the cornerstone of `TB check#1`, the smoker with a three-week cough, and at the top level no parent condition limits it. So `TB check#1` would be listed as affected. In multiple mode the smoker of case 7 would *not* lose his TB check: the engine follows every branch, so he would get

both the TB check (from the cough branch) and the quit-smoking help (from the new top-level node). That is actually reasonable advice, and Ners Ratna might accept the affected case. In single mode, however, the order of the top-level nodes would decide which one he got — one more reason to look at every affected case rather than trusting the count.

Solution 5.5. First, knowledge that can learn can learn a mistake: an exception taught in a hurry could route a danger sign to home care, and nobody would notice until a patient was harmed. Second, the safety questions come from published guidance [1], [6], [7], and must change when the guidance changes, with the source checked — a different process from learning from cases. Instead, let nurses *propose* changes: the app can keep a simple log of red-flag questions that confused people, or cases where a danger sign was missed, which the team reviews with the source before changing the program.

Solution 5.6. With `multiple` on, the engine tries every top-level node, and each one needs its own first condition answered: `cough`, `diarrhea`, `headache`. Without it, the engine stops at the first branch that answers — the fever branch, which ends in `dengue check#1` — and asks nothing else. Pak Darmo would then have missed the headache advice, and, more importantly, the blood-pressure question that the headache node leads to.

Solution 5.7. The solution programs are `StopDemo.java` and `stop_demo.py`. The Python version is:

```
"""Exercise 5.7: retire a piece of advice without deleting it: stop the
node, and teach the better advice as a new node."""

import pathlib
import sys

sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import TOP_LEVEL, Case, Condition, KnowledgeBase, casefile


PEOPLE = [{"headache": "yes"}, {"headache": "yes", "blood pressure": "150"}]


def show(kb, when):
    for i, facts in enumerate(PEOPLE):
        got = [c.value for c in kb.infer(facts, multiple=True).conclusions()]
        print(when + ", person " + str(i + 1) + ": "
              + (", ".join(got) or "(no advice)"))


name, description, steps = casefile.read_cases("../data/health-seed-cases.json")
kb = casefile.build(KnowledgeBase(name, description), steps)
show(kb, "before")
kb.stop("headache home care#1", "Ratna")
show(kb, "stopped")
case = Case.of(["headache = yes"], ["advice = headache rest and water"])
node = kb.refine(case, TOP_LEVEL, chosen=[Condition.parse("headache = yes")],
                 author="Ratna")
print("new node: " + str(node))
show(kb, "after")
print("nodes kept: " + str(len(kb.nodes)) + ", history: "
      + ", ".join(h["action"] for h in kb.history[-2:]))
```

It prints:

```
before, person 1: headache home care
before, person 2: blood pressure check
stopped, person 1: (no advice)
stopped, person 2: blood pressure check
new node: headache rest and water#1: headache = yes -> advice = headache rest and water
after, person 1: headache rest and water
after, person 2: blood pressure check, headache rest and water
nodes kept: 10, history: stop, add
```

After the stop, the person with only a headache gets no advice — the stopped node fires, but gives nothing — while the person with a high reading still gets the blood-pressure check, because the exception of a stopped node still works. After the new node, both get the new advice. The knowledge base still has all its nodes (ten, with the new one), and the history ends with the stop and the addition.

Solution 5.8. A word longer than the width becomes a line of its own, longer than the width: `wrap` never breaks inside a word. That is acceptable because the texts are sentences of ordinary words; the only long "words" are a few labels and numbers, which are better left whole than cut in the middle.

Solution 5.9. The safety questions would follow child-specific guidance chosen by the bidan — for example the general danger signs that health workers are trained to look for in young children, such as a child who cannot drink or breastfeed or who has had a fit, each of which means *go now* — and the first check would be the child's age. Seed cases might be *fever* → home care for a child; *diarrhea* → oralit and zinc, as WHO advises for children [4]; and *cough with fast breathing* → the puskesmas today. Everything in `HealthCheck` except the constants (`ADULT`, `RED_FLAGS`, `QUESTIONS`), the seed file and the wording file can be reused unchanged: the safety loop, the wrapping, the consultation in multiple mode and the whole teaching screen.

References

[1] World Health Organization, "Dengue and severe dengue," fact sheet, Aug. 21, 2025. [Online]. Available: https://www.who.int/news-room/fact-sheets/detail/dengue-and-severe-dengue

[2] World Health Organization, "Hypertension," fact sheet, Sep. 25, 2025. [Online]. Available: https://www.who.int/news-room/fact-sheets/detail/hypertension

[3] World Health Organization, "Tuberculosis," fact sheet, 2025. [Online]. Available: https://www.who.int/news-room/fact-sheets/detail/tuberculosis

[4] World Health Organization, "Diarrhoeal disease," fact sheet. [Online]. Available: https://www.who.int/news-room/fact-sheets/detail/diarrhoeal-disease

[5] Kementerian Kesehatan Republik Indonesia, "Public Safety Center 119," layanan gawat darurat medis. [Online]. Available: https://psc.kemkes.go.id/web/

[6] Kementerian Kesehatan Republik Indonesia, "Kenali gejala stroke dengan SeGeRa Ke RS," *Ayo Sehat*, Oct. 28, 2023. [Online]. Available: https://ayosehat.kemkes.go.id/kenali-gejala-stroke-dengan-segera-ke-rs

[7] M. D. B. Kholil, "Demam berdarah dengue: kapan harus ke IGD," Direktorat Jenderal Pelayanan Kesehatan, Kementerian Kesehatan RI, Jan. 10, 2025. [Online]. Available: https://keslan.kemkes.go.id/view_artikel/3932/demam-berdarah-dengue-kapan-harus-ke-igd

[8] Kementerian Komunikasi dan Digital Republik Indonesia, "Tentang call center 112," Layanan Panggilan Darurat 112. [Online]. Available: https://layanan112.komdigi.go.id/tentang

[9] I. Subakti and R. Hidayatullah, "Aplikasi sistem pakar untuk diagnosis awal gangguan kesehatan secara mandiri menggunakan Variable-Centered Intelligent Rule System," vol. 6, no. 1, pp. 11–16, Jan. 2007.

[10] Kementerian Kesehatan Republik Indonesia, "Ayo kenali gejala tuberkulosis yang perlu diwaspadai, mulai dari batuk hingga berat badan menurun," *Ayo Sehat*, Mar. 15, 2024. [Online]. Available: https://ayosehat.kemkes.go.id/ayo-kenali-gejala-tuberkulosis-yang-perlu-diwaspadai-mulai-dari-batuk-hingga-berat-badan-menurun

[11] P. Compton and B. H. Kang, *Ripple-Down Rules: The Alternative to Machine Learning*. Boca Raton, FL, USA: CRC Press, 2021.

[12] P. Compton, G. Edwards, B. Kang, L. Lazarus, R. Malor, T. Menzies, P. Preston, A. Srinivasan, and C. Sammut, "Ripple down rules: possibilities and limitations," in *Proc. 6th AAAI Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada, 1991, pp. 6.1–6.20.

[13] M. M. I. Subakti, "A Variable-Centered Intelligent Rule System," M.Sc.Eng. thesis, Dept. of Computer Science and Information Engineering, Taipei, Taiwan, Jun. 2005.

[14] B. H. Kang, P. Compton, and P. Preston, "Multiple classification ripple down rules: evaluation and possibilities," in *Proc. 9th AAAI-Sponsored Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada, 1995, pp. 17.1–17.20.

[15] I. Subakti, "Some revisions in VCIRS and cases reconstructing perspectives," in *Proc. Information and Communication Technology Seminar (ICTS 2006)*, Surabaya, Indonesia, 2006.

[16] I. Subakti, *Variable-Centered Intelligent Rule System*, Indonesian ed. (resume). Surabaya, Indonesia: Dept. of Informatics, Institut Teknologi Sepuluh Nopember, 2006.

[17] D. Richards, "Two decades of ripple down rules research," *The Knowledge Engineering Review*, vol. 24, no. 2, pp. 159–184, 2009.

[18] Kementerian Kesehatan Republik Indonesia, "Ingin berhenti merokok? Hubungi layanan konseling bebas biaya 0-800-177-6565." [Online]. Available: https://kemkes.go.id/id/ingin-berhenti-merokok-hubungi-layanan-konseling-bebas-biaya

[19] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. Jakarta, Indonesia, 2022.

Chapter 6

Crop and Fruit-Tree Problem Advisor for Farmers: Rule Generation

By the end of this chapter you will be able to explain what it means for a piece of knowledge to be *used*, not only stored; compute, by hand and with the engine, the three **usage rates** of VCIRS — the Variable Usage Rate (VUR), the Node Usage Rate (NUR) and the Rule Usage Rate (RUR) — from the Credit, the sharing count and the closeness degree of each condition; say how the rates put the nodes of a knowledge base into a **relative order**; use that order to **generate** candidate rules and nodes; and review the candidates as an expert does, keeping only what is true. You will run the **crop and fruit-tree doctor**, an app in Java and Python that helps farmers with problems of durian, mango, chili and rice, learns from the village extension worker, shows its usage rates, and lets her turn a generated candidate into a new, more general piece of advice that answers farms nobody taught it about.

6.1 Pak Slamet's Orchard

Pak Slamet's family has a fruit and durian orchard on the slope above the village: forty durian trees his father planted, a row of mango trees along the path, and, below them, a few beds of chili and a rice field that the family shares with his brother. It is a typical Indonesian farm in one way that matters. The 2023 agricultural census counted 28.4 million farming households in Indonesia, and 16.89 million of them — six in ten — work less than half a hectare [1]. Farms like these cannot afford to lose a harvest, and they cannot afford an expert on call.

Every season brings its troubles. After the rains, a dark, wet patch appears on the bark of one of the old durian trees and slowly spreads; the disease, a water mold called *Phytophthora palmivora*, causes patch canker, root rot and fruit rot, and in parts of Southeast Asia it has infected up to 30 percent of durian trees and cut yields by a fifth or more [2]. The mangoes get black, sunken spots as they ripen — anthracnose, a fungus that rots the fruit on the tree and after harvest [3] — and some are stung by fruit flies, whose maggots rot the flesh from inside [4]. The chili leaves curl and turn yellow where whiteflies have carried the yellow leaf-curl virus, which can take up to 80 percent of a harvest [5]. And in the rice, brown planthoppers gather at the base of the plants, sometimes so many that whole patches of the field dry out and collapse, an outbreak that is most often caused, ironically, by spraying too much insecticide, which kills the spiders and beetles that eat them [6].

Mbak Nur is the village's agricultural extension worker, the *penyuluh*. She knows all of this, and she covers five villages on a motorcycle. On market days she sits at the farmers' group post and answers the same questions over and over: *what are these spots, what do I do about this patch on the bark, why are my chili leaves curling?* She would like the answers to be there when she is not. And she has noticed something about her own knowledge: much of it is not about one crop at all. The black sunken spots of anthracnose look the same on mango, chili, papaya and dragon fruit, because the same kind of fungus causes them. She taught the mango case and the chili case separately; a farmer with a spotted papaya still gets no answer.

This chapter builds the doctor Mbak Nur wants, and it teaches the part of VCIRS that notices exactly what she noticed. VCIRS keeps count of how its knowledge is used. From those counts it can see which conditions and nodes carry the most weight, and it can **generate** new candidate rules and nodes by combining them — candidates that the expert then confirms or rejects. That is how the doctor learns, with Mbak Nur's approval, that dark sunken spots on *any* fruit call for anthracnose care.

6.2 The Idea in Plain Words

6.2.1 Knowledge that is used

A knowledge base built case by case grows in the places where the cases came from. Some conditions appear in many nodes; some nodes are the start of many rules; some conditions have been confirmed again and again by later cases. Others were written once and never touched again. The thesis that proposed VCIRS calls the analysis of this **value analysis**, or *usage assignment*, and gives it three purposes [7], [8], [9]:

1. to guide the user during knowledge building about which rule, node or variable to visit first;
2. to guide the user and the system during inference in the same way, so that the most used knowledge is tried first;
3. to guide the system, together with the variable analysis of Chapter 4, in **rule generation**.

The measure is three numbers: a **Variable Usage Rate** (VUR) for each condition in each node, a **Node Usage Rate** (NUR) for each node, and a **Rule Usage Rate** (RUR) for each rule. The larger a rate, the more that piece of knowledge is used [7]. They are built from three simple ingredients, and we will meet them one at a time on the smallest corner of Mbak Nur's knowledge base: the rice. It has four nodes: `nutrient check#1` (rice with yellow leaves), its exception `planthopper control#1` (brown hoppers at the base), that node's exception `planthopper emergency#1` (patches drying out), and `blast control#1` (rice with diamond-shaped spots). **Figure 6.1** shows the whole computation for the first of them.

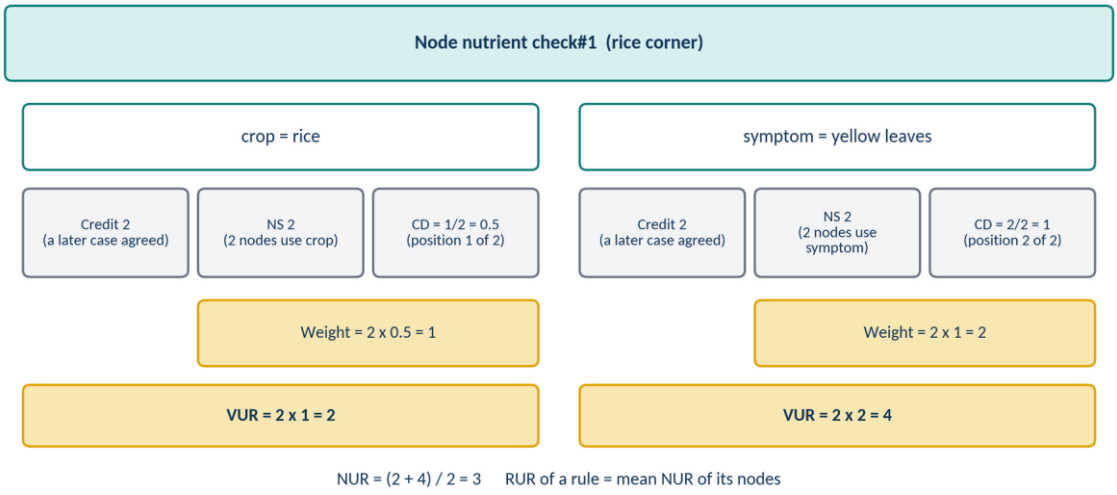

Figure 6.1 How the usage rates of one node are computed. Each condition's Variable Usage Rate is its Credit times its Weight, and the Weight is the number of nodes sharing the variable times the condition's closeness to the conclusion. The node's rate is the mean of its conditions' rates.

6.2.2 Credit: how often a condition was agreed with

When Mbak Nur taught `planthopper control#1`, the case she used — rice, yellow leaves, brown hoppers — *agreed* with both conditions of the parent node, `crop = rice` and `symptom = yellow leaves`. VCIRS records that agreement: each condition of the parent that the new case agrees with earns one more **Credit** [7]. Every condition starts with Credit 1, when its node is created; after the planthopper case, the two conditions of `nutrient check#1` have Credit 2. In the same way, when she taught the emergency node under `planthopper control#1`, the condition `insects = brown hoppers` earned its second Credit.

$$\text{Credit}_i = \text{the number of times condition } i \text{ was used in the Node Structure} \tag{6.1}$$

In plain words, Equation (6.1) says that a condition that many later cases passed through, on their way to a more specific answer, is a condition the knowledge relies on.

6.2.3 NS, CD and Weight: how shared, how close

Two more things make a condition weigh more. The first is how many nodes use its variable: the **NS** (number of nodes sharing) of the variable [7]. In the rice corner, `crop` is used by two nodes (`nutrient check#1` and `blast control#1`), and so is `symptom`; `insects` and `patches` are used by one node each. This is the variable analysis of Chapter 4, reduced to one number per variable.

The second is where the condition stands in its node. A node is a sequence of conditions, and the thesis argues that "the closer a variable is to the conclusion's node, the better it is" — the last condition of a node is the one that finally decides it [7]. The **Closeness Degree** (CD) measures this as the condition's position, its variable order VO, divided by the total number of conditions TV in the node:

$$\text{CD}_i = \frac{\text{VO}_i}{\text{TV}} \tag{6.2}$$

In `nutrient check#1`, `crop = rice` is condition 1 of 2, so its CD is $\frac{1}{2}$; `symptom = yellow leaves` is condition 2 of 2, so its CD is 1. A node with a single condition gives it CD 1. The **Weight** of a condition combines the two:

$$\text{Weight}_i = \text{NS}_i \times \text{CD}_i \tag{6.3}$$

So `crop = rice` in `nutrient check#1` weighs $2 \times 0.5 = 1$, and `symptom = yellow leaves` weighs $2 \times 1 = 2$.

6.2.4 VUR, NUR and RUR

Now the three rates follow in three lines. The Variable Usage Rate of a condition is its Credit times its Weight:

$$\text{VUR}_i = \text{Credit}_i \times \text{Weight}_i \tag{6.4}$$

The Node Usage Rate of a node is the mean of the VURs of its N conditions, and the Rule Usage Rate of a rule is the mean of the NURs of its N nodes [7], [8]:

$$\mathrm{NUR}_j = \frac{1}{N}\sum_{i=1}^{N} \mathrm{VUR}_{ij} \tag{6.5}$$

$$\mathrm{RUR}_k = \frac{1}{N}\sum_{j=1}^{N} \mathrm{NUR}_{jk} \tag{6.6}$$

For `nutrient check#1`: the VUR of `crop = rice` is $2 \times 1 = 2$ and that of `symptom = yellow leaves` is $2 \times 2 = 4$, so its NUR is $(2 + 4)/2 = 3$. The rule `planthopper emergency#1` is the sequence `nutrient check#1 -> planthopper control#1 -> planthopper emergency#1`; if the other two nodes have NUR 2 and 1, its RUR is $(3 + 2 + 1)/3 = 2$. Section 6.3 works every number of the rice corner this way and checks it against the engine.

It is worth saying plainly what the rates are *not*. They do not count how many farmers a node has answered, and they are not probabilities. They measure how much the *structure* of the knowledge leans on each piece: how shared it is, how decisive its position, and how often later teaching passed through it. That is exactly what is needed to decide which pieces are worth combining into something new.

> **KEY INSIGHT.** The usage rates come for free. They are computed from what the Variable-Centered Rule Structure already stores — the order of conditions, which nodes share which variables, and the Credits earned while the expert taught — and they are recomputed whenever the knowledge changes.

6.2.5 From usage to generation

A knowledge base built from cases is **precise but narrow**. Every node was made for real cases and does not disturb the cases before it, so its answers are usually right; but it answers only what it has been taught, and a truly new situation gets no answer at all [10]. The spotted papaya is such a situation: Mbak Nur taught anthracnose for mango and for chili, each node guarded by its crop, and papaya is neither.

Rule generation is VCIRS's answer to narrowness [7], [8]. It looks at the most important knowledge — the nodes shared by the most rules and the variables shared by the most nodes, from the variable analysis of Chapter 4 — and proposes new combinations of it, in an order that does not break any order that already exists in the knowledge base. Two things are generated:

- a **generated rule** is a new sequence of existing nodes that ends in one of the most important nodes;
- a **generated node** is a new sequence of existing conditions that ends in one of the most important variables, with the conclusion of the node it came from.

Both are marked with a `G` in their name (`anthracnose care#1G`) and neither is saved until the expert has confirmed it — and she may keep only part of it. **Figure 6.2** shows the path from the usage rates to a new node.

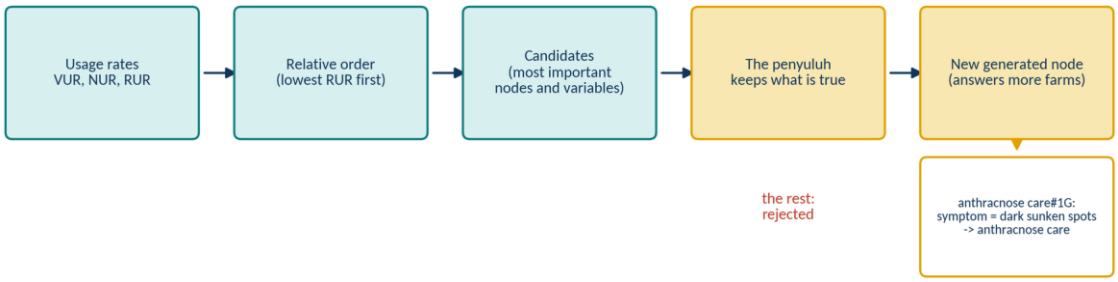

Figure 6.2 Rule generation in VCIRS. The usage rates give a relative order of the knowledge; the order and the most important nodes and variables give candidates; the expert keeps what is true, and the new node answers cases that no taught node could.

6.2.6 The relative order

Where does the order of a candidate come from? The engine walks through the rules starting from the one with the **lowest RUR** — the least used — and lists its nodes; whenever a node is shared with other rules, it brings in those rules' earlier nodes too, placing them by their position and, when positions are equal, by the lower NUR [7]. The result is a **relative node order**: one list of all the nodes of the knowledge base, arranged so that no rule's own order is broken. A generated rule for an important node is the part of this list that comes before it, ending with it. The same walk on conditions, starting from the node with the lowest NUR and using the VURs, gives the **relative variable order**, and a generated node is the part of that list that comes before one of the most important variables [7].

This sounds complicated, and the full algorithm has a dozen steps, but its purpose is simple: to line up everything the knowledge base knows, from the least used to the most used, in an order that respects every rule. A candidate is then just "everything before this important item, in that order".

Take the rice corner of Section 6.2.1 as a small example. Its rule with the lowest RUR is `blast control#1` (1.50, as Section 6.3 computes), so the walk starts there, and `blast control#1` is first. The next lowest is the emergency rule (2.00), and its nodes follow in their own order: `nutrient check#1`, `planthopper control#1`, `planthopper emergency#1`. The first of them is shared with two other rules, but those rules' earlier nodes are already in the list, so nothing moves. The relative node order of the rice corner is therefore `blast control#1`, `nutrient check#1`, `planthopper control#1`, `planthopper emergency#1`. The most important node is `nutrient check#1`, shared by three rules, and its candidate rule is the part of the order up to it: `blast control#1 -> nutrient check#1`, rice with diamond spots followed by rice with yellow leaves. The candidate node for the same node, built the same way from the relative variable order, is `patches = drying, crop = rice, symptom = diamond spots on leaves, symptom = yellow leaves -> nutrient check`. An expert reading it would keep `crop = rice` and `symptom = yellow leaves` — which is the node that already exists — or `symptom = yellow leaves` alone, a general nitrogen rule for every crop, which Section 6.4 shows is a step too far. Even on four nodes, the candidates contain one good idea, some repetition and some nonsense; sorting them is the expert's work.

6.2.7 The expert has the last word

A candidate is a proposal, not knowledge. Most of what it contains is usually wrong for its conclusion: on the Zookeeper demonstration, the thesis found that between 36 and 50 percent of the nodes in each generated rule made sense, and the user crossed out the rest [7]. That is not a

flaw. The machine cannot know what is agriculturally or medically true; it can only notice what is used together. The expert reads the candidate, keeps the conditions or nodes that belong, and rejects the rest. When she keeps only a few conditions, as Mbak Nur does below, the generated node is *more general* than the nodes it came from, and it answers cases none of them could.

In farming the stakes of this review are concrete. A generalization that is too broad sends a farmer to the wrong remedy: a tree with drowning roots given fertilizer, a field of hoppers sprayed with the wrong insecticide, a harvest lost while the real problem spreads. A generalization that is right, on the other hand, carries the penyuluh's experience to crops and farms she never had time to teach. The machine's contribution is to *notice* the pattern across dozens of nodes, which no person can do by eye once a knowledge base grows; the expert's contribution is to *judge* it. Neither is enough alone, and VCIRS is built so that the second always follows the first.

> **DEFINITION.** *Rule generation* in VCIRS proposes new rules (sequences of existing nodes ending in a most important node) and new nodes (sequences of existing conditions ending in a most important variable), ordered by the relative orders that the usage rates produce. Every candidate is marked `G` and saved only with the parts the expert confirms.

6.3 Step 1: The Usage Rates by Hand

Listing 6.1 computes every usage rate of the rice corner by hand, printing each ingredient, and then builds the same four nodes on the engine and checks that the engine gets the same numbers.

Listing 6.1 `UsageByHand.java`: Credit, NS, CD, Weight, VUR, NUR and RUR by hand, checked against vcirs-core.

```
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Locale;
import java.util.Map;

import vcirs.Case;
import vcirs.KnowledgeBase;
import vcirs.Usage;

/**
 * Credit, NS, CD, Weight, VUR, NUR and RUR worked by hand on the rice corner
 * of the farm doctor, then checked against the vcirs-core engine.
 */
public class UsageByHand {
    /** A condition of a node: its variable and its Credit. */
    record Cond(String variable, int credit) { }

    /** A node: its id, its parent (or null) and its conditions. */
    record HandNode(String id, String parent, List<Cond> conds) { }

    // Credit 2 means that a later case agreed with the condition once.
    static final List<HandNode> NODES = List.of(
            new HandNode("nutrient check#1", null,
                    List.of(new Cond("crop", 2), new Cond("symptom", 2))),
            new HandNode("planthopper control#1", "nutrient check#1",
                    List.of(new Cond("insects", 2))),
            new HandNode("planthopper emergency#1", "planthopper control#1",
                    List.of(new Cond("patches", 1))),
            new HandNode("blast control#1", null,
                    List.of(new Cond("crop", 1), new Cond("symptom", 1))));

    static final Map<String, Double> vur = new LinkedHashMap<>();
    static final Map<String, Double> nur = new LinkedHashMap<>();
    static final Map<String, Double> rur = new LinkedHashMap<>();

    static String f2(double x) {
        return String.format(Locale.ROOT, "%.2f", x);  // always a point
    }
```

Listing 6.1 (Continued) *UsageByHand.java: Credit, NS, CD, Weight, VUR, NUR and RUR by hand, checked against vcirs-core.*

```
    static HandNode find(String id) {
        for (HandNode n : NODES) {
            if (n.id().equals(id)) {
                return n;
            }
        }
        throw new IllegalArgumentException(id);
    }

    static void byHand() {
        Map<String, Integer> ns = new LinkedHashMap<>();  // NS: nodes sharing
        for (HandNode n : NODES) {
            for (Cond c : n.conds()) {
                ns.merge(c.variable(), 1, Integer::sum);
            }
        }
        System.out.println("Node                    Variable  Credit  NS  CD"
                + "    Weight  VUR");
        for (HandNode n : NODES) {
            double total = 0;
            for (int i = 0; i < n.conds().size(); i++) {
                Cond c = n.conds().get(i);
                double cd = (i + 1.0) / n.conds().size();    // CD = VO / TV
                double weight = ns.get(c.variable()) * cd;   // Weight = NS x CD
                double v = c.credit() * weight;              // VUR = Credit x W
                vur.put(n.id() + "/" + i, v);
                total += v;
                System.out.println(String.format(Locale.ROOT,
                        "%-25s%-10s%-8d%-4d%-6s%-8s%s", i == 0 ? n.id() : "",
                        c.variable(), c.credit(),
                        ns.get(c.variable()), f2(cd), f2(weight), f2(v)));
            }
            nur.put(n.id(), total / n.conds().size());   // NUR = mean of VURs
        }
        System.out.println();
        for (HandNode n : NODES) {
            List<String> parts = new ArrayList<>();
            for (int i = 0; i < n.conds().size(); i++) {
                parts.add(f2(vur.get(n.id() + "/" + i)));
            }
            System.out.println("NUR " + n.id() + " = (" + String.join(" + ", parts)
                    + ") / " + n.conds().size() + " = " + f2(nur.get(n.id())));
        }
        for (HandNode n : NODES) {
            List<String> path = new ArrayList<>(List.of(n.id()));
            String p = n.parent();
```

Listing 6.1 (Continued) *UsageByHand.java: Credit, NS, CD, Weight, VUR, NUR and RUR by hand, checked against vcirs-core.*

```
            while (p != null) {                    // the rule: the path down
                path.add(0, p);
                p = find(p).parent();
            }
            double sum = 0;
            List<String> parts = new ArrayList<>();
            for (String id : path) {
                sum += nur.get(id);
                parts.add(f2(nur.get(id)));
            }
            rur.put(n.id(), sum / path.size());
            System.out.println("RUR " + n.id() + " = (" + String.join(" + ", parts)
                    + ") / " + path.size() + " = " + f2(rur.get(n.id())));
        }
    }

    static Usage byEngine() {
        KnowledgeBase kb = new KnowledgeBase("rice corner");
        String[][] cases = {
            {"nutrient check", "crop = rice", "symptom = yellow leaves"},
            {"planthopper control", "crop = rice", "symptom = yellow leaves",
                "insects = brown hoppers"},
            {"planthopper emergency", "crop = rice", "symptom = yellow leaves",
                "insects = brown hoppers", "patches = drying"},
            {"blast control", "crop = rice", "symptom = diamond spots on leaves"},
        };
        for (String[] c : cases) {
            List<String> facts = List.of(c).subList(1, c.length);
            kb.learn(Case.of(facts, List.of("advice = " + c[0])));
        }
        return kb.usage();
    }

    public static void main(String[] args) {
        byHand();
        Usage u = byEngine();
        int same = 0;
        int total = 0;
        for (Map.Entry<String, Double> e : vur.entrySet()) {
            String[] key = e.getKey().split("/");
            same += Math.abs(u.vur(key[0], Integer.parseInt(key[1]))
                    - e.getValue()) < 1e-9 ? 1 : 0;
            total++;
        }
        for (Map.Entry<String, Double> e : nur.entrySet()) {
            same += Math.abs(u.nur(e.getKey()) - e.getValue()) < 1e-9 ? 1 : 0;
```

Listing 6.1 (Continued) *UsageByHand.java: Credit, NS, CD, Weight, VUR, NUR and RUR by hand, checked against vcirs-core.*

```
            total++;
        }
        for (Map.Entry<String, Double> e : rur.entrySet()) {
            same += Math.abs(u.rur(e.getKey()) - e.getValue()) < 1e-9 ? 1 : 0;
            total++;
        }
        System.out.println();
        System.out.println("The engine agrees on " + same + " of " + total
                + " numbers.");
    }
}
```

`NODES` describes the rice corner as the thesis's Node Structure would: each node with its parent and its conditions, and each condition with its Credit. The Credits are written in: 2 for the conditions a later case agreed with, 1 for the others, exactly as Section 6.2.2 explained.

`byHand` follows the equations one by one. It counts the NS of each variable by going through all the conditions of all the nodes. For each condition it computes CD as the position divided by the number of conditions (Equation (6.2)), the Weight as NS times CD (Equation (6.3)) and the VUR as Credit times Weight (Equation (6.4)), and prints a row of the table. The NUR of each node is the mean of its VURs (Equation (6.5)). For the RUR, it rebuilds each node's rule by walking up through the parents, as in Chapter 4, and takes the mean of the NURs along it (Equation (6.6)), printing each sum so that you can follow the arithmetic. `f2` prints a number with two decimals; `Locale.ROOT` makes Java write a decimal point on every computer, even one set up for Indonesian, where the default would be a comma.

`byEngine` builds the rice corner the way Mbak Nur would teach it, with four cases and `kb.learn`, and `main` compares all fourteen numbers with the engine's `usage().vur`, `nur` and `rur`. **Listing 6.2** is the Python version.

Listing 6.2 `usage_by_hand.py`: the same computation in Python.

```
"""Credit, NS, CD, Weight, VUR, NUR and RUR worked by hand on the rice
corner of the farm doctor, then checked against the vcirs-core engine."""

import pathlib
import sys

sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import Case, KnowledgeBase

# Each node: its id, its parent, and its conditions as (variable, Credit).
# Credit 2 means that a later case agreed with the condition once.
NODES = [
    ("nutrient check#1", None, [("crop", 2), ("symptom", 2)]),
    ("planthopper control#1", "nutrient check#1", [("insects", 2)]),
    ("planthopper emergency#1", "planthopper control#1", [("patches", 1)]),
    ("blast control#1", None, [("crop", 1), ("symptom", 1)]),
]


def by_hand():
    ns = {}                                     # NS: nodes sharing a variable
    for _, _, conds in NODES:
        for var, _ in conds:
            ns[var] = ns.get(var, 0) + 1
    vur, nur = {}, {}
    print("Node                    Variable  Credit  NS  CD    Weight  VUR")
    for node, _, conds in NODES:
        total = 0.0
        for i, (var, credit) in enumerate(conds):
            cd = (i + 1) / len(conds)           # CD = VO / TV
            weight = ns[var] * cd               # Weight = NS x CD
            v = credit * weight                 # VUR = Credit x Weight
            vur[(node, i)] = v
            total += v
            print(f"{node if i == 0 else '':<25}{var:<10}{credit:<8}"
                  f"{ns[var]:<4}{cd:<6.2f}{weight:<8.2f}{v:.2f}")
        nur[node] = total / len(conds)          # NUR = mean of the VURs
    print()
    for node, _, conds in NODES:
        parts = " + ".join(f"{vur[(node, i)]:.2f}" for i in range(len(conds)))
        print(f"NUR {node} = ({parts}) / {len(conds)} = {nur[node]:.2f}")
    rur = {}
    for node, parent, _ in NODES:
        path, p = [node], parent
        while p is not None:                    # the rule: the path down
```

Listing 6.2 (Continued) *usage_by_hand.py: the same computation in Python.*

```
            path.insert(0, p)
            p = [n for n in NODES if n[0] == p][0][1]
        rur[node] = sum(nur[n] for n in path) / len(path)
        parts = " + ".join(f"{nur[n]:.2f}" for n in path)
        print(f"RUR {node} = ({parts}) / {len(path)} = {rur[node]:.2f}")
    return vur, nur, rur


def by_engine():
    kb = KnowledgeBase("rice corner")
    for facts, advice in [
            (["crop = rice", "symptom = yellow leaves"], "nutrient check"),
            (["crop = rice", "symptom = yellow leaves",
              "insects = brown hoppers"], "planthopper control"),
            (["crop = rice", "symptom = yellow leaves",
              "insects = brown hoppers", "patches = drying"],
             "planthopper emergency"),
            (["crop = rice", "symptom = diamond spots on leaves"],
             "blast control")]:
        kb.learn(Case.of(facts, ["advice = " + advice]))
    return kb.usage()


def main():
    vur, nur, rur = by_hand()
    u = by_engine()
    same = [abs(u.vur(n, i) - v) < 1e-9 for (n, i), v in vur.items()]
    same += [abs(u.nur(n) - v) < 1e-9 for n, v in nur.items()]
    same += [abs(u.rur(n) - v) < 1e-9 for n, v in rur.items()]
    print()
    print("The engine agrees on " + str(sum(same)) + " of " + str(len(same))
          + " numbers.")


if __name__ == "__main__":
    main()
```

Both print:

```
Node                    Variable  Credit  NS  CD    Weight  VUR
nutrient check#1        crop      2       2   0.50  1.00    2.00
                        symptom   2       2   1.00  2.00    4.00
planthopper control#1   insects   2       1   1.00  1.00    2.00
planthopper emergency#1 patches   1       1   1.00  1.00    1.00
blast control#1         crop      1       2   0.50  1.00    1.00
                        symptom   1       2   1.00  2.00    2.00

NUR nutrient check#1 = (2.00 + 4.00) / 2 = 3.00
NUR planthopper control#1 = (2.00) / 1 = 2.00
NUR planthopper emergency#1 = (1.00) / 1 = 1.00
NUR blast control#1 = (1.00 + 2.00) / 2 = 1.50
RUR nutrient check#1 = (3.00) / 1 = 3.00
RUR planthopper control#1 = (3.00 + 2.00) / 2 = 2.50
RUR planthopper emergency#1 = (3.00 + 2.00 + 1.00) / 3 = 2.00
RUR blast control#1 = (1.50) / 1 = 1.50

The engine agrees on 14 of 14 numbers.
```

Read the table from the right. The highest VUR in the rice corner is 4.00, for `symptom = yellow leaves` in `nutrient check#1`: it is shared by two nodes, it stands at the end of its node, and a later case agreed with it. The lowest is 1.00, for `patches = drying` and for `crop = rice` in `blast control#1`: each is used in one way only, and nothing has passed through it yet. The NURs show

the same picture one level up: `nutrient check#1`, the node everything else in the rice corner hangs from, has the highest NUR, 3.00.

The RURs hold a small surprise. The emergency rule has the *lowest* NUR at its end (1.00), but its RUR, 2.00, is higher than that of `blast control#1` (1.50), because it travels through the heavily used `nutrient check#1` on its way. A rule is judged by all the knowledge it passes through, not only by its last node. And the last line is the promise that nothing here is magic: the engine agrees on all fourteen numbers.

6.4 Step 2: Generation on the Farm Doctor

Now the whole of Mbak Nur's first knowledge base: thirteen cases for durian, mango, chili and rice, stored in `farm-seed-cases.json` (Section 6.5 shows it). **Listing 6.3** builds it, prints its rule usage rates and its most important nodes and variables, lists the relative node order, asks the engine for candidate rules and nodes, and plays Mbak Nur's review: one rule candidate rejected, one node candidate accepted with a single condition, and three farms tried before and after.

Listing 6.3 `GenerateDemo.java`: rule and node generation on the farm knowledge base, with the penyuluh's review.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Map;

import vcirs.CaseFile;
import vcirs.Condition;
import vcirs.Facts;
import vcirs.GeneratedNode;
import vcirs.GeneratedRule;
import vcirs.InferenceResult;
import vcirs.KnowledgeBase;
import vcirs.Node;
import vcirs.Num;
import vcirs.Report;
import vcirs.Usage;

/**
 * Rule generation on the farm doctor: the usage rates, the relative node
 * order they produce, the candidate rules and nodes the engine proposes, and
 * Mbak Nur's review, which keeps only what is agriculturally true.
 */
public class GenerateDemo {
    static final List<Map<String, String>> FARMS = List.of(
            Map.of("crop", "mango", "symptom", "dark sunken spots"),
            Map.of("crop", "papaya", "symptom", "dark sunken spots"),
            Map.of("crop", "dragon fruit", "symptom", "dark sunken spots"));

    static void coverage(KnowledgeBase kb, String when) {
        for (Map<String, String> farm : FARMS) {
            InferenceResult result = kb.infer(Facts.map(farm));
            String answer = result.paths.isEmpty() ? "no advice"
                    : result.conclusions().get(0).value + " ("
                    + result.paths.get(0).ruleId() + ")";
            System.out.println("  " + when + ", " + farm.get("crop") + ": "
                    + answer);
        }
    }
```

Listing 6.3 (Continued) *GenerateDemo.java: rule and node generation on the farm knowledge base, with the penyuluh's review.*

```
    public static void main(String[] args) {
        CaseFile seed = CaseFile.read("../data/farm-seed-cases.json");
        KnowledgeBase kb = seed.build(new KnowledgeBase(seed.name,
                seed.description));
        Usage usage = kb.usage();
        System.out.println("Rule usage rates, lowest first:");
        for (String line : Report.rurTable(kb, 4)) {
            System.out.println("  " + line);
        }
        System.out.println("Most important nodes: "
                + String.join(", ", usage.importantNodes()));
        System.out.println("Most important variables: "
                + String.join(", ", usage.importantVariables()));
        System.out.println("Relative node order:");
        List<String> order = kb.relativeNodeOrder().order;
        for (int i = 0; i < order.size(); i++) {
            System.out.println("  " + String.format("%2d", i + 1) + ". "
                    + order.get(i) + "  (NUR "
                    + Num.fixed(usage.nur(order.get(i)), 4) + ")");
        }

        System.out.println("Candidate rules:");
        for (GeneratedRule g : kb.generateRules()) {
            System.out.println("  " + g.id + ": " + g.nodes.size()
                    + " nodes, from " + g.nodes.get(0) + " to "
                    + g.nodes.get(g.nodes.size() - 1));
            System.out.println("  Mbak Nur keeps none of the other nodes: "
                    + "rejected.");
        }
        System.out.println("Candidate nodes:");
        List<GeneratedNode> candidates = kb.generateNodes();
        for (GeneratedNode g : candidates) {
            System.out.println("  " + g.id + ": " + g.conditions.size()
                    + " conditions, the last "
                    + g.conditions.get(g.conditions.size() - 1));
        }
        GeneratedNode chosen = candidates.get(2);
        System.out.println("Reviewing " + chosen.id + ":");
        List<String> texts = new ArrayList<>();
        for (int i = 0; i < chosen.conditions.size(); i++) {
            Condition c = chosen.conditions.get(i);
            texts.add(c.toString());
            System.out.println("  " + String.format("%2d", i + 1) + ". " + c);
        }
        coverage(kb, "before");
        String keep = "symptom = dark sunken spots";
```

Listing 6.3 (Continued) *GenerateDemo.java: rule and node generation on the farm knowledge base, with the penyuluh's review.*

```
        int number = texts.indexOf(keep) + 1;
        Node node = kb.acceptNode(chosen, List.of(keep), "Nur");
        System.out.println("Mbak Nur keeps condition " + number + " only. "
                + "New node: " + node);
        coverage(kb, "after");
        System.out.println(kb.summary());
    }
}
```

The engine calls are few. `kb.usage()` gives the rates; `Report.rurTable(kb, 4)` prints the RURs with four decimals, lowest first; `usage.importantNodes()` and `usage.importantVariables()` give the result of the variable analysis; `kb.relativeNodeOrder().order` is the relative node order.

`kb.generateRules()` and `kb.generateNodes()` return the candidates, and `kb.acceptNode(candidate, keep, "Nur")` saves a generated node with only the conditions in `keep`. `Num.fixed` prints a number with a fixed number of decimals, the same way in Java and in Python. `coverage` asks the knowledge base about three farms with dark sunken spots: a mango, a papaya and a dragon fruit. **Listing 6.4** is the Python version.

Listing 6.4 `generate_demo.py`: the same generation and review in Python.

```
"""Rule generation on the farm doctor: the usage rates, the relative node
order they produce, the candidate rules and nodes the engine proposes, and
Mbak Nur's review, which keeps only what is agriculturally true."""

import pathlib
import sys

sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import KnowledgeBase, casefile, num, report

FARMS = [{"crop": "mango", "symptom": "dark sunken spots"},
         {"crop": "papaya", "symptom": "dark sunken spots"},
         {"crop": "dragon fruit", "symptom": "dark sunken spots"}]


def coverage(kb, when):
    for farm in FARMS:
        result = kb.infer(farm)
        answer = (result.conclusions()[0].value + " (" + result.paths[0]
                  .rule_id() + ")") if result.paths else "no advice"
        print("  " + when + ", " + farm["crop"] + ": " + answer)


def main():
    name, description, steps = casefile.read_cases(
        "../data/farm-seed-cases.json")
    kb = casefile.build(KnowledgeBase(name, description), steps)
    usage = kb.usage()
    print("Rule usage rates, lowest first:")
    for line in report.rur_table(kb, 4):
        print("  " + line)
    print("Most important nodes: " + ", ".join(usage.important_nodes()))
    print("Most important variables: "
          + ", ".join(usage.important_variables()))
    print("Relative node order:")
    for i, node_id in enumerate(kb.relative_node_order().order):
        print("  " + str(i + 1).rjust(2) + ". " + node_id + "  (NUR "
              + num.fixed(usage.nur(node_id), 4) + ")")
```

Listing 6.4 (Continued) *generate_demo.py: the same generation and review in Python.*

```
    print("Candidate rules:")
    for g in kb.generate_rules():
        print("  " + g.id + ": " + str(len(g.nodes)) + " nodes, from "
              + g.nodes[0] + " to " + g.nodes[-1])
        print("  Mbak Nur keeps none of the other nodes: rejected.")
    print("Candidate nodes:")
    candidates = kb.generate_nodes()
    for g in candidates:
        print("  " + g.id + ": " + str(len(g.conditions))
              + " conditions, the last " + str(g.conditions[-1]))
    chosen = candidates[2]
    print("Reviewing " + chosen.id + ":")
    for i, c in enumerate(chosen.conditions):
        print("  " + str(i + 1).rjust(2) + ". " + str(c))
    coverage(kb, "before")
    keep = "symptom = dark sunken spots"
    number = [str(c) for c in chosen.conditions].index(keep) + 1
    node = kb.accept_node(chosen, [keep], "Nur")
    print("Mbak Nur keeps condition " + str(number) + " only. New node: "
          + str(node))
    coverage(kb, "after")
    print(kb.summary())


if __name__ == "__main__":
    main()
```

Both print:

```
Rule usage rates, lowest first:
  RuleID                    RUR
  ------------------------  -------
  planthopper emergency#1   5.6667
  durian fruit rot care#1   6.0000
  anthracnose care#1        6.0000
  fruit fly control#1       6.0000
  anthracnose care#2        6.0000
  blast control#1           6.0000
  yellow virus control#1    6.6250
  planthopper control#1     8.0000
  drain and treat canker#1  9.5000
  canker emergency#1        9.5000
  nutrient check#1          12.0000
  nutrient check#2          12.0000
  patch canker care#1       18.0000
Most important nodes: patch canker care#1, nutrient check#2
Most important variables: crop, symptom
Relative node order:
   1. nutrient check#2  (NUR 12.0000)
   2. planthopper control#1  (NUR 4.0000)
   3. planthopper emergency#1  (NUR 1.0000)
   4. blast control#1  (NUR 6.0000)
   5. anthracnose care#2  (NUR 6.0000)
   6. fruit fly control#1  (NUR 6.0000)
   7. anthracnose care#1  (NUR 6.0000)
   8. durian fruit rot care#1  (NUR 6.0000)
   9. nutrient check#1  (NUR 12.0000)
  10. yellow virus control#1  (NUR 1.2500)
  11. patch canker care#1  (NUR 18.0000)
  12. canker emergency#1  (NUR 1.0000)
  13. drain and treat canker#1  (NUR 1.0000)
Candidate rules:
  patch canker care#1G: 11 nodes, from nutrient check#2 to patch canker care#1
  Mbak Nur keeps none of the other nodes: rejected.
Candidate nodes:
  patch canker care#1G: 16 conditions, the last symptom = wet patch on bark
  durian fruit rot care#1G: 15 conditions, the last symptom = rotting fruit on tree
  anthracnose care#1G: 12 conditions, the last symptom = dark sunken spots
  fruit fly control#1G: 13 conditions, the last symptom = stung fruit
  anthracnose care#2G: 12 conditions, the last symptom = dark sunken spots
  nutrient check#1G: 10 conditions, the last symptom = yellow leaves
  nutrient check#2G: 10 conditions, the last symptom = yellow leaves
  blast control#1G: 8 conditions, the last symptom = diamond spots on leaves
Reviewing anthracnose care#1G:
   1. patches = drying
   2. patch around trunk = yes
   3. soil = waterlogged
   4. leaves curl = yes
   5. insects = whitefly
   6. insects = brown hoppers
   7. crop = rice
   8. symptom = diamond spots on leaves
   9. crop = chili
  10. symptom = yellow leaves
  11. crop = mango
  12. symptom = dark sunken spots
  before, mango: anthracnose care (anthracnose care#1)
  before, papaya: no advice
  before, dragon fruit: no advice
Mbak Nur keeps condition 12 only. New node: anthracnose care#1G: symptom = dark sunken
spots -> advice = anthracnose care
  after, mango: anthracnose care (anthracnose care#1)
  after, papaya: anthracnose care (anthracnose care#1G)
  after, dragon fruit: anthracnose care (anthracnose care#1G)
Crop and fruit-tree doctor: 8 variables, 14 nodes, 14 rules
```

Read the output from the top, as Mbak Nur would.

The rule usage rates. **Figure 6.3** draws the same table as a chart. The three most used rules are `patch canker care#1` (18.00), and the two `nutrient check` rules (12.00 each). They are the nodes most other knowledge hangs from: `patch canker care#1` has two exceptions (water standing at the roots, and a patch around the trunk), and the rice `nutrient check#2` leads down to the planthopper nodes. Their conditions, `crop` and `symptom`, are shared by all eight top-level nodes and have earned extra Credits from their exceptions.

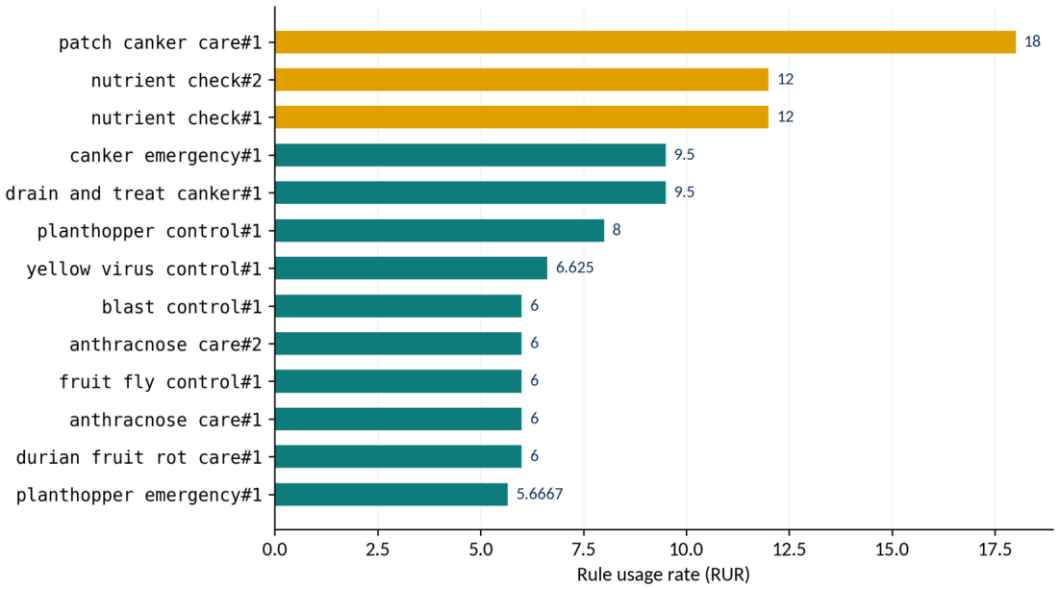

Figure 6.3 The rule usage rates of Mbak Nur's first knowledge base. The most used rules, in amber, are those whose first node has exceptions: much of the knowledge passes through them.

The most important nodes and variables. `patch canker care#1` and `nutrient check#2` are each shared by three rules — their own and those of their exceptions — more than any other node, so they are the most important nodes [7]. (`nutrient check#1` has one exception, so it is shared by only two.) The most important variables are `crop` and `symptom`, each used by eight nodes.

The relative node order. The walk starts from the least used rule, `planthopper emergency#1` (RUR 5.67), and lists its nodes in its own order: `nutrient check#2`, `planthopper control#1`, `planthopper emergency#1`. Next come the five one-node rules with RUR 6.00, the newest first (the tie rule of the engine), then the chili rule with its exception, and finally the durian canker rule with its two exceptions. Every rule's own order is kept: `nutrient check#2` is before `planthopper control#1`, and `patch canker care#1` before both of its exceptions.

The candidate rule. There is only one. For `nutrient check#2`, the part of the order before it is empty, so the candidate would be the one-node rule that already exists, and the engine does not propose it. For `patch canker care#1`, the candidate is the eleven nodes from `nutrient check#2` to `patch canker care#1`: rice, chili and mango advice followed by a durian canker. Mbak Nur keeps none of the other ten nodes — they have nothing to do with a canker on durian bark — and she rejects it. This is the normal fate of most rule candidates in a knowledge base like this one, where every node ends in a piece of advice. Rule generation shines when some nodes *conclude facts that other nodes use*, as in the Zookeeper of the thesis, where one node concludes `mammal = yes` and another begins with it; Exercise 6.8 shows it working there.

The candidate nodes. There are eight, one for each node that uses `symptom`. Why `symptom` and not `crop`? Both are most important, but in every node that uses them `crop` stands first and `symptom` after it, closer to the conclusion; the thesis drops a most important variable that is always *covered* by another in this way, and so does the engine [7]. Each candidate is the relative variable order up to that node's own symptom, with that node's conclusion. The third, `anthracnose care#1G`, comes from the mango anthracnose node and ends with `symptom = dark sunken spots`.

The review. Mbak Nur reads the twelve conditions of `anthracnose care#1G`. Most are nonsense for this conclusion: drying patches in a rice field, whiteflies, waterlogged soil, crops that are not mango. But the twelfth is exactly what she noticed at the market: dark sunken spots, on any fruit, are anthracnose. She keeps condition 12 and nothing else. Before, the mango farm got anthracnose care from `anthracnose care#1`, and the papaya and dragon-fruit farms got nothing. After, the mango farm still gets the same answer from the same node — nothing it knew was disturbed — and the papaya and dragon-fruit farms get anthracnose care from the new node, `anthracnose care#1G`. The knowledge base has fourteen nodes and answers more farms than it did.

She does not accept the other candidates. `nutrient check#1G` offers `symptom = yellow leaves` alone, which would tell *every* farmer with yellow leaves to check nitrogen; but yellow leaves on a durian tree standing in water mean rotting roots, not hunger. Knowing where a generalization stops is the expert's half of rule generation.

6.5 The Crop and Fruit-Tree Doctor

6.5.1 What the doctor starts from

The doctor starts from the thirteen seed cases of `farm-seed-cases.json`:

```
{
  "name": "Crop and fruit-tree doctor",
  "description": "Problems of durian, mango, chili and rice, taught by Mbak Nur, the
village extension worker (penyuluh).",
  "cases": [
    {"facts": ["crop = durian", "symptom = wet patch on bark"], "conclusions": ["advice =
patch canker care"], "option": -1, "author": "Nur"},
    {"facts": ["crop = durian", "symptom = wet patch on bark", "soil = waterlogged"],
"conclusions": ["advice = drain and treat canker"], "option": -1, "author": "Nur"},
    {"facts": ["crop = durian", "symptom = wet patch on bark", "patch around trunk =
yes"], "conclusions": ["advice = canker emergency"], "option": -1, "author": "Nur"},
    {"facts": ["crop = durian", "symptom = rotting fruit on tree"], "conclusions":
["advice = durian fruit rot care"], "option": -1, "author": "Nur"},
    {"facts": ["crop = mango", "symptom = dark sunken spots"], "conclusions": ["advice =
anthracnose care"], "option": -1, "author": "Nur"},
    {"facts": ["crop = mango", "symptom = stung fruit"], "conclusions": ["advice = fruit
fly control"], "option": -1, "author": "Nur"},
    {"facts": ["crop = chili", "symptom = dark sunken spots"], "conclusions": ["advice =
anthracnose care"], "option": -1, "author": "Nur"},
    {"facts": ["crop = chili", "symptom = yellow leaves"], "conclusions": ["advice =
nutrient check"], "option": -1, "author": "Nur"},
    {"facts": ["crop = chili", "symptom = yellow leaves", "leaves curl = yes", "insects =
whitefly"], "conclusions": ["advice = yellow virus control"], "option": -1, "author":
"Nur"},
    {"facts": ["crop = rice", "symptom = yellow leaves"], "conclusions": ["advice =
nutrient check"], "option": -1, "author": "Nur"},
    {"facts": ["crop = rice", "symptom = yellow leaves", "insects = brown hoppers"],
"conclusions": ["advice = planthopper control"], "option": -1, "author": "Nur"},
    {"facts": ["crop = rice", "symptom = yellow leaves", "insects = brown hoppers",
"patches = drying"], "conclusions": ["advice = planthopper emergency"], "option": -1,
"author": "Nur"},
    {"facts": ["crop = rice", "symptom = diamond spots on leaves"], "conclusions":
["advice = blast control"], "option": -1, "author": "Nur"}
  ]
}
```

Each case is a farm Mbak Nur visited, with the crop, the symptom she saw, and anything else that decided her advice. `"option": -1` lets `learn` place each case, so the exceptions — waterlogged soil under a canker, a patch around the trunk, whiteflies on curling chili leaves, brown hoppers and then drying patches in the rice — hang under the node they refine, as in Chapters 3 and 4.

As in Chapter 5, the conclusions are short labels, and the words a farmer reads are kept in `advice-texts.txt`:

```
# The wording of each piece of advice, kept apart from the knowledge.
# One line per advice: label = text. Lines starting with # are notes.
patch canker care = Scrape the soft, rotten bark down to healthy wood, burn the
scrapings, and paint the wound with the fungicide your penyuluh recommends. Keep water
from standing around the trunk.
drain and treat canker = Water standing around the roots feeds this rot. Dig drains so
the soil dries after rain, then treat the patch as for canker.
canker emergency = A patch that goes most of the way around the trunk can kill the tree.
Call the penyuluh this week, and do not wait for the harvest.
durian fruit rot care = Pick up and bury rotten fruit, prune for air and light, and keep
fruit off wet soil. Ask the penyuluh about protecting the tree before the rainy season.
root rot care = Yellow, wilting leaves with water standing at the roots mean the roots
are rotting. Dig drains, keep mulch away from the trunk, and ask the penyuluh about
treatment.
anthracnose care = Dark sunken spots are anthracnose, a fungus that attacks many fruits.
Remove and burn spotted fruit and leaves, prune for air, and never keep seed from spotted
fruit. Ask the penyuluh about hot-water dipping after harvest.
fruit fly control = Collect every fallen and stung fruit and bury it at least 30 cm deep.
Hang methyl eugenol traps, and wrap young fruit in paper bags.
nutrient check = Yellow leaves often mean the plant is short of nitrogen. Ask the
penyuluh about a soil test before adding fertilizer, and give it in smaller doses.
yellow virus control = Curled yellow leaves with whiteflies are the yellow virus. Pull
out and destroy sick plants, hang yellow sticky traps, and plant rows of maize around the
field.
planthopper control = Brown planthoppers at the base of the plants: drain the field for a
few days, do not add more nitrogen, and do not spray insecticides that also kill the
spiders and beetles that eat them.
planthopper emergency = Patches drying out mean a heavy planthopper attack. Call the
penyuluh today, warn your neighbours, and plant at the same time as them next season.
blast control = Diamond-shaped spots with gray centers are blast. Split the nitrogen
fertilizer, keep the field flooded if you can, and choose a resistant variety next
season.
```

The advice follows current agronomy. For durian canker it is to scrape out the rotten bark and treat the wound, and above all to keep water from standing at the roots, which the disease needs [2]. For fruit flies it is the orchard sanitation and male-annihilation traps of the area-wide program that ACIAR published in 2024: every fallen fruit collected and buried at least 30 centimeters deep, and methyl eugenol traps in the orchard [4]. For the chili yellow virus it is the Ministry of Agriculture's advice: pull out and destroy sick plants, yellow sticky traps, and rows of maize around the field as a barrier [11]. For planthoppers it is to drain the field, avoid extra nitrogen and avoid the broad insecticides that cause outbreaks by killing the hoppers' natural enemies [6]; for blast, split nitrogen, flooding where possible and resistant varieties next season [12]; for anthracnose, removing infected fruit and, for mango, asking about hot-water treatment after harvest, which reduced the disease in recent trials [3]. None of the advice names a pesticide brand or a dose; where a chemical is needed, the doctor sends the farmer to the penyuluh, who knows what is registered and effective in the area.

6.5.2 The program

Listing 6.5 is the doctor in Java.

Listing 6.5 `FarmDoctor.java`: the crop and fruit-tree doctor on vcirs-core, with usage rates and generation.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

import vcirs.Case;
import vcirs.CaseFile;
import vcirs.Conclusion;
import vcirs.Condition;
import vcirs.Fact;
import vcirs.Facts;
import vcirs.GeneratedNode;
import vcirs.InferenceResult;
import vcirs.KnowledgeBase;
import vcirs.LearnResult;
import vcirs.Node;
import vcirs.Question;
import vcirs.RdrPath;
import vcirs.Report;
import vcirs.Usage;
import vcirs.Variable;
import vcirs.VcirsException;

/**
 * Crop and fruit-tree doctor: advice for durian, mango, chili and rice, with
 * the usage rates and the rule generation of VCIRS.
 * java FarmDoctor [--echo] [--tree] [--usage] [--generate] [kb.json]
 * A farmer answers a few questions and gets advice with the reason behind
 * it; when the advice is wrong or missing, the penyuluh teaches the right
 * one. --usage prints the usage rates; --generate lets the penyuluh review
 * the nodes the engine proposes and keep only what is true in the field.
 */
public class FarmDoctor {
    static final String[][] QUESTIONS = {
        {"crop", "Which plant has the problem?", "durian", "mango", "chili",
            "rice"},
        {"symptom", "What do you see?", "wet patch on bark",
            "rotting fruit on tree", "dark sunken spots", "stung fruit",
            "yellow leaves", "diamond spots on leaves"},
        {"soil", "How is the soil after rain?", "waterlogged", "drains well"},
```

Listing 6.5 (Continued) *FarmDoctor.java: the crop and fruit-tree doctor on vcirs-core, with usage rates and generation.*

```
        {"patch around trunk", "Does the patch go most of the way around the "
            + "trunk?", "yes", "no"},
        {"leaves curl", "Are the leaves curling?", "yes", "no"},
        {"insects", "Which small insects do you see?", "whitefly",
            "brown hoppers", "none"},
        {"patches", "Are whole patches of the field drying out?", "drying",
            "no"},
    };

    static BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in, StandardCharsets.UTF_8));
    static boolean echo = false;
    static KnowledgeBase kb;

    /** Break text into lines of at most width characters, at spaces. */
    static List<String> wrap(String text, int width) {
        List<String> lines = new ArrayList<>();
        String line = "";
        for (String word : text.split(" ", -1)) {
            if (!line.isEmpty() && line.length() + 1 + word.length() > width) {
                lines.add(line);
                line = word;
            } else {
                line = line.isEmpty() ? word : line + " " + word;
            }
        }
        lines.add(line);
        return lines;
    }

    /** Reads the user's answer; with echo on, prints it too. */
    static String read(String prompt) {
        List<String> lines = wrap(prompt.replaceAll("[ >]+$", ""), 74);
        for (int i = 0; i < lines.size() - 1; i++) {
            System.out.println(lines.get(i));
        }
        System.out.print(lines.get(lines.size() - 1) + " > ");
        String line;
        try {
            line = in.readLine();
        } catch (IOException e) {
            line = null;
        }
        String text = line == null ? "" : line.strip();
        if (echo) {
            System.out.println(text);
```

Listing 6.5 (Continued) *FarmDoctor.java: the crop and fruit-tree doctor on vcirs-core, with usage rates and generation.*

```
        }
        return text;
    }

    static Fact ask(Question q) {
        Variable var = kb.variables.get(q.variable);
        String text = q.variable;
        List<String> choices = q.choices;
        if (var != null && !var.question.isEmpty()) {
            text = var.question;
            choices = var.choices;
        }
        if (!choices.isEmpty()) {
            text += " (" + String.join("/", choices) + ")";
        }
        String answer = read(text + " > ");
        return answer.isEmpty() ? null : new Fact(q.variable, answer); // Enter
    }

    /** advice-texts.txt: one "label = text" per line. */
    static Map<String, String> readTexts(Path path) {
        Map<String, String> texts = new LinkedHashMap<>();
        try {
            for (String line : Files.readAllLines(path, StandardCharsets.UTF_8)) {
                int at = line.indexOf('=');
                if (at >= 0 && !line.startsWith("#")) {
                    texts.put(line.substring(0, at).strip(),
                            line.substring(at + 1).strip());
                }
            }
        } catch (IOException e) {
            throw new VcirsException("cannot read " + path);
        }
        return texts;
    }
```

Listing 6.5 (Continued) *FarmDoctor.java: the crop and fruit-tree doctor on vcirs-core, with usage rates and generation.*

```
    /** Load the knowledge base, or build it from the seed cases. */
    static KnowledgeBase openKb(String path) {
        if (Files.exists(Path.of(path))) {
            return KnowledgeBase.load(path);
        }
        Path parent = Path.of(path).toAbsolutePath().getParent();
        CaseFile seed = CaseFile.read(parent.resolve("farm-seed-cases.json")
                .toString());
        KnowledgeBase k = new KnowledgeBase(seed.name, seed.description);
        for (String[] q : QUESTIONS) {
            List<String> choices = new ArrayList<>();
            for (int i = 2; i < q.length; i++) {
                choices.add(q[i]);
            }
            k.declare(q[0], q[1], choices);
        }
        seed.build(k);
        k.save(path);
        return k;
    }

    static void printUsage() {
        System.out.println("Node usage rates (NUR), lowest first:");
        for (String line : Report.nurTable(kb, 4)) {
            System.out.println("  " + line);
        }
        System.out.println("Rule usage rates (RUR), lowest first:");
        for (String line : Report.rurTable(kb, 4)) {
            System.out.println("  " + line);
        }
        Usage usage = kb.usage();
        System.out.println("Most important nodes: "
                + String.join(", ", usage.importantNodes()));
        System.out.println("Most important variables: "
                + String.join(", ", usage.importantVariables()));
    }
```

Listing 6.5 (Continued) *FarmDoctor.java: the crop and fruit-tree doctor on vcirs-core, with usage rates and generation.*

```
    /** The penyuluh reviews the nodes the engine proposes. */
    static void review(String path) {
        List<GeneratedNode> candidates = kb.generateNodes();
        for (int i = 0; i < candidates.size(); i++) {
            GeneratedNode g = candidates.get(i);
            System.out.println(String.format("%2d", i + 1) + ". " + g.id + ": "
                    + g.conditions.size() + " conditions, from " + g.source);
        }
        while (true) {
            String pick = read("Review which candidate? (number, Enter to "
                    + "finish) > ");
            if (pick.isEmpty()) {
                break;
            }
            GeneratedNode g = candidates.get(Integer.parseInt(pick) - 1);
            for (int i = 0; i < g.conditions.size(); i++) {
                System.out.println("   " + String.format("%2d", i + 1) + ". "
                        + g.conditions.get(i));
            }
            List<String> concl = new ArrayList<>();
            for (Conclusion c : g.conclusions) {
                concl.add(c.toString());
            }
            System.out.println("  -> " + String.join(", ", concl));
            String keep = read("Keep which conditions? (numbers, Enter to "
                    + "reject) > ");
            if (keep.isEmpty()) {
                System.out.println("Rejected.");
                continue;
            }
            List<String> chosen = new ArrayList<>();
            for (String k : keep.split(" ")) {
                chosen.add(g.conditions.get(Integer.parseInt(k) - 1).key());
            }
            Node node = kb.acceptNode(g, chosen, "penyuluh");
            System.out.println("Accepted: " + node);
        }
        kb.save(path);
        System.out.println("Saved " + path + ": " + kb.summary());
    }
```

Listing 6.5 (Continued) *FarmDoctor.java: the crop and fruit-tree doctor on vcirs-core, with usage rates and generation.*

```
    static List<Condition> factsOf(InferenceResult result) {
        List<Condition> facts = new ArrayList<>();
        for (Fact f : result.memory.facts.values()) {
            if (!f.variable.equals("advice")) {
                facts.add(new Condition(f.variable, "=", f.value));
            }
        }
        return facts;
    }

    /** The penyuluh gives the right advice; learn() puts it in its place. */
    static void teach(InferenceResult result, String path) {
        String advice = read("Which advice is right? (label, Enter if none) > ");
        if (advice.isEmpty()) {
            System.out.println("Then there is nothing to learn this time.");
            return;
        }
        List<Condition> facts = factsOf(result);
        while (true) {
            String extra = read("Another fact that matters? "
                    + "(variable = value, Enter when done) > ");
            if (extra.isEmpty()) {
                break;
            }
            try {
                facts.add(Condition.parse(extra));
            } catch (VcirsException e) {
                System.out.println("Please write it as: variable = value");
            }
        }
        LearnResult learned;
        try {
            learned = kb.learn(new Case(facts,
                    List.of(new Conclusion("advice", advice)), 100), "penyuluh");
        } catch (VcirsException e) {
            System.out.println("Cannot learn: " + e.getMessage());
            return;
        }
        System.out.println("Learned: " + learned);
        kb.save(path);
        System.out.println("Saved " + path + ": " + kb.summary());
    }
```

Listing 6.5 (Continued) *FarmDoctor.java: the crop and fruit-tree doctor on vcirs-core, with usage rates and generation.*

```
    public static void main(String[] args) {
        String path = "../data/farm-kb.json";
        List<String> flags = new ArrayList<>();
        for (String a : args) {
            if (a.startsWith("--")) {
                flags.add(a);
            } else {
                path = a;
            }
        }
        echo = flags.contains("--echo");
        kb = openKb(path);
        if (flags.contains("--tree")) {
            for (String line : Report.tree(kb)) {
                System.out.println(line);
            }
            return;
        }
        if (flags.contains("--usage")) {
            printUsage();
            return;
        }
        if (flags.contains("--generate")) {
            review(path);
            return;
        }
        Map<String, String> texts = readTexts(Path.of(path).toAbsolutePath()
                .getParent().resolve("advice-texts.txt"));
        System.out.println("Crop and fruit-tree doctor: " + kb.nodes.size()
                + " pieces of advice so far.");
        System.out.println("Answer each question; press Enter if you do not "
                + "know.");
        InferenceResult result = kb.infer(Facts.none(), FarmDoctor::ask);
        if (!result.paths.isEmpty()) {
            RdrPath p = result.paths.get(0);
            String advice = p.conclusions().get(0).value;
            System.out.println("Advice (" + advice + "):");
            for (String line : wrap(texts.getOrDefault(advice, advice), 72)) {
                System.out.println("  " + line);
            }
            List<String> used = new ArrayList<>();
            for (Node n : p.nodes) {
                for (Condition c : n.conditions) {
                    used.add(c.toString());
                }
            }
```

Listing 6.5 (Continued) *FarmDoctor.java: the crop and fruit-tree doctor on vcirs-core, with usage rates and generation.*

```
            List<String> because = wrap("Because: " + String.join(", ", used)
                    + " (rule " + p.ruleId() + ")", 74);
            for (int i = 0; i < because.size(); i++) {
                System.out.println((i == 0 ? "" : "  ") + because.get(i));
            }
        } else {
            System.out.println("I have no advice for this yet. Ask the "
                    + "penyuluh.");
        }
        if (!result.paths.isEmpty()
                && read("Is the advice right? (yes/no) > ").equals("yes")) {
            kb.learn(new Case(factsOf(result), result.conclusions(), 100),
                    "farmer");
            kb.save(path);
            System.out.println("Good. The case is recorded as checked.");
            return;
        }
        teach(result, path);
    }
}
```

Most of it will look familiar from Chapters 4 and 5: `QUESTIONS` declares the seven variables with their questions and usual answers; `wrap`, `read` and `ask` ask them politely; `readTexts` and `openKb` read the wording and the knowledge base. What is new is in three places.

`--usage` calls `printUsage`, which prints the NUR and RUR tables and the most important nodes and variables — the numbers of Section 6.4 for the knowledge base as it is *now*, after everything the doctor has learned.

`--generate` calls `review`, the penyuluh's screen. It lists every node candidate the engine proposes, with its number of conditions and the node it came from. The penyuluh picks one to review; the screen prints its conditions, numbered, and its conclusion; she types the numbers of the conditions she keeps, or presses Enter to reject it. `kb.acceptNode` saves an accepted candidate as a new top-level node, marked as generated, with its own one-node rule, and records it in the history.

Consulting and teaching are those of Chapter 4: the farmer answers the questions the nodes need, and gets the advice, wrapped for a small screen, and the conditions behind it with the rule's name. If the farmer (or the penyuluh beside him) says the advice is right, the case is recorded as checked; if it is wrong or missing, `teach` asks for the right advice and any other fact that matters and lets `learn` put the new knowledge in its place.

Listing 6.6 is the same doctor in Python.

Listing 6.6 `farm_doctor.py`: the crop and fruit-tree doctor in Python.

```
"""Crop and fruit-tree doctor: advice for durian, mango, chili and rice,
with the usage rates and the rule generation of VCIRS.

    python farm_doctor.py [--echo] [--tree] [--usage] [--generate] [kb.json]

A farmer answers a few questions and gets advice with the reason behind
it; when the advice is wrong or missing, the penyuluh teaches the right one.
--usage prints the usage rates; --generate lets the penyuluh review the
nodes the engine proposes and keep only what is true in the field.
"""

import os
import pathlib
import sys

# Tell Python where the vcirs-core engine is (two folders up, then across).
sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import (Case, Condition, Conclusion, KnowledgeBase, VcirsError,
                   casefile, report)

QUESTIONS = [
    ("crop", "Which plant has the problem?", ["durian", "mango", "chili",
                                              "rice"]),
    ("symptom", "What do you see?", ["wet patch on bark",
                                     "rotting fruit on tree",
                                     "dark sunken spots", "stung fruit",
                                     "yellow leaves",
                                     "diamond spots on leaves"]),
    ("soil", "How is the soil after rain?", ["waterlogged", "drains well"]),
    ("patch around trunk", "Does the patch go most of the way around the "
     "trunk?", ["yes", "no"]),
    ("leaves curl", "Are the leaves curling?", ["yes", "no"]),
    ("insects", "Which small insects do you see?", ["whitefly",
                                                    "brown hoppers", "none"]),
    ("patches", "Are whole patches of the field drying out?",
     ["drying", "no"]),
]
```

Listing 6.6 (Continued) *farm_doctor.py: the crop and fruit-tree doctor in Python.*

```
def wrap(text, width):
    """Break text into lines of at most width characters, at spaces."""
    lines, line = [], ""
    for word in text.split(" "):
        if line and len(line) + 1 + len(word) > width:
            lines.append(line)
            line = word
        else:
            line = word if not line else line + " " + word
    return lines + [line]


class Console:
    """Reads the user's answers; with echo on, prints them too."""

    def __init__(self, kb, echo):
        self.kb = kb
        self.echo = echo

    def read(self, prompt):
        lines = wrap(prompt.rstrip(" >"), 74)
        for line in lines[:-1]:
            print(line)
        print(lines[-1] + " > ", end="")
        text = sys.stdin.readline().strip()
        if self.echo:
            print(text)
        return text

    def ask(self, question):
        var = self.kb.variables.get(question.variable)
        text, choices = question.variable, question.choices
        if var is not None and var.question:
            text, choices = var.question, var.choices
        if choices:
            text += " (" + "/".join(choices) + ")"
        return self.read(text + " > ") or None    # Enter = "I do not know"
```

Listing 6.6 (Continued) *farm_doctor.py: the crop and fruit-tree doctor in Python.*

```
def read_texts(path):
    """advice-texts.txt: one 'label = text' per line."""
    texts = {}
    with open(path, encoding="utf-8") as f:
        for line in f:
            if "=" in line and not line.startswith("#"):
                label, text = line.split("=", 1)
                texts[label.strip()] = text.strip()
    return texts


def open_kb(path):
    """Load the knowledge base, or build it from the seed cases."""
    if os.path.exists(path):
        return KnowledgeBase.load(path)
    seed = os.path.join(os.path.dirname(path), "farm-seed-cases.json")
    name, description, steps = casefile.read_cases(seed)
    kb = KnowledgeBase(name, description)
    for var, question, choices in QUESTIONS:
        kb.declare(var, question, choices)
    casefile.build(kb, steps)
    kb.save(path)
    return kb


def print_usage(kb):
    print("Node usage rates (NUR), lowest first:")
    for line in report.nur_table(kb, 4):
        print("  " + line)
    print("Rule usage rates (RUR), lowest first:")
    for line in report.rur_table(kb, 4):
        print("  " + line)
    usage = kb.usage()
    print("Most important nodes: " + ", ".join(usage.important_nodes()))
    print("Most important variables: "
          + ", ".join(usage.important_variables()))
```

Listing 6.6 (Continued) *farm_doctor.py: the crop and fruit-tree doctor in Python.*

```
def review(kb, con, path):
    """The penyuluh reviews the nodes the engine proposes."""
    candidates = kb.generate_nodes()
    for i, g in enumerate(candidates):
        print(str(i + 1).rjust(2) + ". " + g.id + ": "
              + str(len(g.conditions)) + " conditions, from " + g.source)
    while True:
        pick = con.read("Review which candidate? (number, Enter to finish) > ")
        if not pick:
            break
        g = candidates[int(pick) - 1]
        for i, c in enumerate(g.conditions):
            print("   " + str(i + 1).rjust(2) + ". " + str(c))
        print("   -> " + ", ".join(str(c) for c in g.conclusions))
        keep = con.read("Keep which conditions? (numbers, Enter to reject) > ")
        if not keep:
            print("Rejected.")
            continue
        chosen = [g.conditions[int(k) - 1] for k in keep.split()]
        node = kb.accept_node(g, chosen, "penyuluh")
        print("Accepted: " + str(node))
    kb.save(path)
    print("Saved " + path + ": " + kb.summary())


def teach(kb, con, result, path):
    """The penyuluh gives the right advice; learn() puts it in its place."""
    advice = con.read("Which advice is right? (label, Enter if none) > ")
    if not advice:
        print("Then there is nothing to learn this time.")
        return
    facts = [Condition(f.variable, "=", f.value)
             for f in result.memory.facts.values() if f.variable != "advice"]
    while True:
        extra = con.read("Another fact that matters? "
                         "(variable = value, Enter when done) > ")
        if not extra:
            break
        try:
            facts.append(Condition.parse(extra))
        except VcirsError:
            print("Please write it as: variable = value")
    try:
        learned = kb.learn(Case(facts, [Conclusion("advice", advice)]),
                           "penyuluh")
    except VcirsError as e:
```

Listing 6.6 (Continued) *farm_doctor.py: the crop and fruit-tree doctor in Python.*

```
        print("Cannot learn: " + str(e))
        return
    print("Learned: " + str(learned))
    kb.save(path)
    print("Saved " + path + ": " + kb.summary())


def main(args):
    echo = "--echo" in args
    files = [a for a in args if not a.startswith("--")]
    path = files[0] if files else "../data/farm-kb.json"
    kb = open_kb(path)
    con = Console(kb, echo)
    if "--tree" in args:
        for line in report.tree(kb):
            print(line)
        return
    if "--usage" in args:
        print_usage(kb)
        return
    if "--generate" in args:
        review(kb, con, path)
        return
    texts = read_texts(os.path.join(os.path.dirname(path),
                                    "advice-texts.txt"))
    print("Crop and fruit-tree doctor: " + str(len(kb.nodes))
          + " pieces of advice so far.")
    print("Answer each question; press Enter if you do not know.")
    result = kb.infer({}, con.ask)
    if result.paths:
        p = result.paths[0]
        advice = p.conclusions()[0].value
        print("Advice (" + advice + "):")
        for line in wrap(texts.get(advice, advice), 72):
            print("  " + line)
        because = wrap("Because: " + ", ".join(
            str(c) for n in p.nodes for c in n.conditions)
            + " (rule " + p.rule_id() + ")", 74)
        for i, line in enumerate(because):
            print(("" if i == 0 else "  ") + line)
    else:
        print("I have no advice for this yet. Ask the penyuluh.")
    if result.paths and con.read("Is the advice right? (yes/no) > ") == "yes":
        facts = [Condition(f.variable, "=", f.value)
                 for f in result.memory.facts.values()
                 if f.variable != "advice"]
```

Listing 6.6 (Continued) *farm_doctor.py: the crop and fruit-tree doctor in Python.*

```
        kb.learn(Case(facts, list(result.conclusions())), "farmer")
        kb.save(path)
        print("Good. The case is recorded as checked.")
        return
    teach(kb, con, result, path)


if __name__ == "__main__":
    main(sys.argv[1:])
```

6.6 Five Visits

Compile and run from `Chapter06/java`:

```
javac -d out -sourcepath ../../vcirs-core/java/src *.java
java -cp out FarmDoctor --echo < ../data/session-1.txt
```

or, from `Chapter06/python`, `python farm_doctor.py --echo < ../data/session-1.txt`; session 4 is run with `--generate`. Run the five sessions in order on a fresh knowledge base, and the transcripts and `farm-kb.json` are the same in both languages, byte for byte.

Session 1: Pak Slamet's mangoes.

```
Crop and fruit-tree doctor: 13 pieces of advice so far.
Answer each question; press Enter if you do not know.
Which plant has the problem? (durian/mango/chili/rice) > mango
What do you see? (wet patch on bark/rotting fruit on tree/dark sunken
spots/stung fruit/yellow leaves/diamond spots on leaves) > dark sunken spots
Advice (anthracnose care):
  Dark sunken spots are anthracnose, a fungus that attacks many fruits.
  Remove and burn spotted fruit and leaves, prune for air, and never keep
  seed from spotted fruit. Ask the penyuluh about hot-water dipping after
  harvest.
Because: crop = mango, symptom = dark sunken spots (rule anthracnose
  care#1)
Is the advice right? (yes/no) > yes
Good. The case is recorded as checked.
```

Two questions. Every top-level node begins with `crop`, so the first question is always the crop; the durian nodes fail on it at once, and the first mango node needs the symptom. The advice names the problem, says what to do, and points to the penyuluh for the one step that needs her. The *Because* line is the explanation: the two conditions and the rule `anthracnose care#1`. Pak Slamet says the advice is right, and the check is recorded.

Session 2: Bu Lastri's rice.

```
Crop and fruit-tree doctor: 13 pieces of advice so far.
Answer each question; press Enter if you do not know.
Which plant has the problem? (durian/mango/chili/rice) > rice
What do you see? (wet patch on bark/rotting fruit on tree/dark sunken
spots/stung fruit/yellow leaves/diamond spots on leaves) > yellow leaves
Which small insects do you see? (whitefly/brown hoppers/none) > brown hoppers
Are whole patches of the field drying out? (drying/no) > drying
Advice (planthopper emergency):
  Patches drying out mean a heavy planthopper attack. Call the penyuluh
  today, warn your neighbours, and plant at the same time as them next
  season.
Because: crop = rice, symptom = yellow leaves, insects = brown hoppers,
  patches = drying (rule planthopper emergency#1)
Is the advice right? (yes/no) > yes
Good. The case is recorded as checked.
```

Four questions, down a rule of three nodes: yellow leaves in rice, then brown hoppers at the base, then patches drying out. The answer is the planthopper emergency: call the penyuluh today, warn the neighbors. The advice not to spray broad insecticides, which belongs to the node above, is the one Bu Lastri most needs to hear, because the usual reaction to hoppers is to spray, and spraying is what feeds the outbreak [6]; a later version of the doctor could show the advice of the nodes on the way, as the self-check of Chapter 5 does.

Session 3: a durian with yellow leaves, and the doctor learns. Pak Slamet's youngest durian tree, planted in the low corner of the orchard, has yellow, wilting leaves.

```
Crop and fruit-tree doctor: 13 pieces of advice so far.
Answer each question; press Enter if you do not know.
Which plant has the problem? (durian/mango/chili/rice) > durian
What do you see? (wet patch on bark/rotting fruit on tree/dark sunken
spots/stung fruit/yellow leaves/diamond spots on leaves) > yellow leaves
I have no advice for this yet. Ask the penyuluh.
Which advice is right? (label, Enter if none) > root rot care
Another fact that matters? (variable = value, Enter when done) > soil = waterlogged
Another fact that matters? (variable = value, Enter when done) >
Learned: added: root rot care#1 added at the top level
Saved ../data/farm-kb.json: Crop and fruit-tree doctor: 8 variables, 14 nodes, 14 rules
```

No node matches: the durian nodes are about bark and fruit, and the yellow-leaf nodes are about chili and rice. The doctor says so honestly. Mbak Nur, who comes by on her rounds, sees the water standing at the roots after the night's rain and recognizes root rot, the same *Phytophthora* that causes the canker [2]. She teaches `root rot care` and the fact that matters, `soil = waterlogged`. Since nothing answered, `learn` adds a new top-level node with all three facts.

Session 4: Mbak Nur reviews the generated nodes. At the end of the month, at the farmers' group post, she runs the review.

```
 1. patch canker care#1G: 15 conditions, from patch canker care#1
 2. durian fruit rot care#1G: 10 conditions, from durian fruit rot care#1
 3. anthracnose care#1G: 13 conditions, from anthracnose care#1
 4. fruit fly control#1G: 14 conditions, from fruit fly control#1
 5. anthracnose care#2G: 13 conditions, from anthracnose care#2
 6. nutrient check#1G: 7 conditions, from nutrient check#1
 7. nutrient check#2G: 7 conditions, from nutrient check#2
 8. blast control#1G: 11 conditions, from blast control#1
 9. root rot care#1G: 7 conditions, from root rot care#1
Review which candidate? (number, Enter to finish) > 3
   1. patches = drying
   2. patch around trunk = yes
   3. leaves curl = yes
   4. insects = whitefly
   5. soil = waterlogged
   6. crop = durian
   7. symptom = yellow leaves
   8. crop = chili
   9. crop = rice
  10. symptom = rotting fruit on tree
  11. symptom = diamond spots on leaves
  12. crop = mango
  13. symptom = dark sunken spots
  -> advice = anthracnose care
Keep which conditions? (numbers, Enter to reject) > 13
Accepted: anthracnose care#1G: symptom = dark sunken spots -> advice = anthracnose care
Review which candidate? (number, Enter to finish) >
Saved ../data/farm-kb.json: Crop and fruit-tree doctor: 8 variables, 15 nodes, 15 rules
```

There are nine candidates now, one more than in Section 6.4, because the new root-rot node also uses `symptom`, and the relative variable order has changed with it: the anthracnose candidate has thirteen conditions this time, and the one she wants is number 13. She keeps it, and the doctor saves `anthracnose care#1G: symptom = dark sunken spots -> advice = anthracnose care`. She rejects the rest by simply not reviewing them. The whole review takes three answers.

Session 5: Pak Harto's papaya. Pak Harto, Pak Slamet's neighbor, grows papaya, a crop nobody ever taught the doctor about.

```
Crop and fruit-tree doctor: 15 pieces of advice so far.
Answer each question; press Enter if you do not know.
Which plant has the problem? (durian/mango/chili/rice) > papaya
What do you see? (wet patch on bark/rotting fruit on tree/dark sunken
spots/stung fruit/yellow leaves/diamond spots on leaves) > dark sunken spots
Advice (anthracnose care):
  Dark sunken spots are anthracnose, a fungus that attacks many fruits.
  Remove and burn spotted fruit and leaves, prune for air, and never keep
  seed from spotted fruit. Ask the penyuluh about hot-water dipping after
  harvest.
Because: symptom = dark sunken spots (rule anthracnose care#1G)
Is the advice right? (yes/no) > yes
Good. The case is recorded as checked.
```

The first question is the crop, *papaya*, and every crop-specific node fails on it. The generated node needs only the symptom, which is asked, and it fires: anthracnose care, because of `symptom = dark sunken spots`, from the rule `anthracnose care#1G`. The doctor answered a farm that no taught node could, with knowledge the expert confirmed.

`--tree` shows the knowledge base after the five visits, the generated node marked at the bottom:

```
(root)
+- patch canker care#1: crop = durian, symptom = wet patch on bark -> advice = patch canker
care
|  +- drain and treat canker#1: soil = waterlogged -> advice = drain and treat canker
|  +- canker emergency#1: patch around trunk = yes -> advice = canker emergency
+- durian fruit rot care#1: crop = durian, symptom = rotting fruit on tree -> advice =
durian fruit rot care
+- anthracnose care#1: crop = mango, symptom = dark sunken spots -> advice = anthracnose
care
+- fruit fly control#1: crop = mango, symptom = stung fruit -> advice = fruit fly control
+- anthracnose care#2: crop = chili, symptom = dark sunken spots -> advice = anthracnose
care
+- nutrient check#1: crop = chili, symptom = yellow leaves -> advice = nutrient check
|  +- yellow virus control#1: leaves curl = yes, insects = whitefly -> advice = yellow
virus control
+- nutrient check#2: crop = rice, symptom = yellow leaves -> advice = nutrient check
|  +- planthopper control#1: insects = brown hoppers -> advice = planthopper control
|     +- planthopper emergency#1: patches = drying -> advice = planthopper emergency
+- blast control#1: crop = rice, symptom = diamond spots on leaves -> advice = blast
control
+- root rot care#1: crop = durian, symptom = yellow leaves, soil = waterlogged -> advice =
root rot care
+- anthracnose care#1G: symptom = dark sunken spots -> advice = anthracnose care
[generated]
```

And `--usage` shows how the usage rates have moved:

```
Node usage rates (NUR), lowest first:
  NodeID                    NumOfVariableIDs  NUR
  ------------------------  ----------------  -------
  canker emergency#1        1                 1.0000
  planthopper emergency#1   1                 1.0000
  yellow virus control#1    2                 1.2500
  drain and treat canker#1  1                 2.0000
  root rot care#1           3                 3.8889
  planthopper control#1     1                 4.0000
  durian fruit rot care#1   2                 7.2500
  anthracnose care#1        2                 7.2500
  fruit fly control#1       2                 7.2500
  anthracnose care#2        2                 7.2500
  blast control#1           2                 7.2500
  anthracnose care#1G       1                 10.0000
  nutrient check#1          2                 14.5000
  nutrient check#2          2                 14.5000
  patch canker care#1       2                 21.7500
Rule usage rates (RUR), lowest first:
  RuleID                    RUR
  ------------------------  -------
  root rot care#1           3.8889
  planthopper emergency#1   6.5000
  durian fruit rot care#1   7.2500
  anthracnose care#1        7.2500
  fruit fly control#1       7.2500
  anthracnose care#2        7.2500
  blast control#1           7.2500
  yellow virus control#1    7.8750
  planthopper control#1     9.2500
  anthracnose care#1G       10.0000
  canker emergency#1        11.3750
  drain and treat canker#1  11.8750
  nutrient check#1          14.5000
  nutrient check#2          14.5000
  patch canker care#1       21.7500
Most important nodes: patch canker care#1, nutrient check#2
Most important variables: symptom
```

Two things have changed since Section 6.4. The new root-rot node has three conditions, and its RUR (3.89) is now the lowest of all: it is the newest and least shared knowledge in the base, and the next generation will start from it. And the most important variable is now `symptom` alone: the generated node uses `symptom` without `crop`, so `symptom` is shared by one more node than `crop`. The knowledge base itself has registered what Mbak Nur taught it: symptoms matter more than crops.

6.7 Making It Better

Review on a schedule. The thesis suggests that refinement can run when a knowledge base is opened, on a schedule, or on demand [7]. For a farm doctor, a monthly review at the farmers' group post suits the rhythm of the seasons: enough new cases to change the rates, and not so many candidates that the review becomes a chore.

Show why a candidate was proposed. The review screen lists conditions without their usage rates. Printing each condition's VUR beside it (with `usage().vur(node, i)` for the node it came from) helps the penyuluh see which parts of a candidate the knowledge base leans on most; Exercise 6.6 asks for it.

Generate rules where nodes feed each other. In this doctor every node ends in advice, so generated *rules* are rarely useful. If the knowledge is split into two steps — first the problem

(`problem = anthracnose`), then what to do about it given the crop and the season — nodes begin to conclude facts that other nodes use, and generated rules become chains worth keeping, as in the Zookeeper (Exercise 6.8). Chapter 8 uses such chains for forward and backward reasoning.

Mark generated advice. The tree shows `[generated]` for a node that came from generation, and the history records who accepted it. A doctor could say so to the farmer too — *this advice covers many crops; if it does not help, ask the penyuluh* — so that general knowledge is used with a little more care than knowledge taught for a crop.

Photos. A farmer describes a symptom in words, and words vary. A photo sent to the penyuluh with the consultation lets her check the symptom before she confirms the advice or teaches a new case. Chapter 13 shows, for the Wi-Fi doctor, how to connect such a doctor to a language model that can help turn a farmer's own words into the doctor's variables, with the knowledge itself still checked by people.

Tests. `FarmTest.java` and `test_farm.py` check the rice-corner computation (NUR 3 for `nutrient check#1`, RUR 2 for the emergency rule), the generation on the seed knowledge base (thirteen nodes, the two most important nodes, `nutrient check#2` first in the relative order, one rule candidate and eight node candidates, the papaya answered only after `anthracnose care#1G` is accepted), and the doctor's wrapping and wording file. They end with:

```
all tests passed
```

6.8 Using It for Real

With a farmers' group. The doctor fits the *kelompok tani*, the farmers' group that meets regularly with its penyuluh. Farmers consult it on a phone or on the group's laptop; the penyuluh teaches it on her rounds and reviews the generated candidates at the group meeting, where other farmers can say whether a generalization matches what they see. Over a few seasons, the knowledge base holds the problems of *this* district, in *its* words.

Use the local names. Farmers do not say *Phytophthora*; they say *kanker batang*, *busuk akar*, *patek*, *lalat buah*, *wereng*, *virus kuning*. The advice labels and the choices of the questions should be the names the farmers use, and the advice texts can give the scientific name once, for the penyuluh. The engine does not mind what the values are called, as long as the same thing is always called the same way.

Advice that is safe to follow. A farm doctor must never tell a farmer to spray a particular chemical at a particular dose; that depends on what is registered, on the crop, on the stage and on the weather, and it is the penyuluh's job. The doctor in this chapter gives the steps that are always safe — sanitation, drainage, traps, barrier crops, resistant varieties, not overspraying — and sends the farmer to the penyuluh for the rest. When pesticides are used, the label is the law: protective clothing, the waiting time before harvest, and never near water sources.

Seasons matter. Anthracnose and *Phytophthora* are diseases of the wet season; planthopper outbreaks follow staggered planting and heavy nitrogen [6], [12], [2]. A variable such as `season` (wet, dry), asked only when a node needs it, lets the penyuluh teach seasonal exceptions — *in the dry season, these spots are probably not anthracnose* — in exactly the way Chapter 5 taught exceptions.

Sharing knowledge between villages. Two penyuluh in neighboring districts will teach their doctors differently, and both may be right for their own soils and varieties. Before one doctor's

knowledge base is copied to another village, it should be read with the local penyuluh — the tree and the Rule Structure make this possible in an afternoon — and every node she doubts should be stopped there rather than deleted, so that the history keeps the reason. Merging two knowledge bases node by node, with the cases behind them, is a team task, and Chapter 11 builds the tools for it.

Other crops, other experts. Nothing in `FarmDoctor` is specific to these four crops except the questions and the seed file. A coconut cooperative, a shallot growers' group, a village fish-pond group — each can write its own variables, a dozen seed cases, and let its own expert teach and review. The usage rates and the generation work the same way on any knowledge base.

6.9 Common Mistakes, and How to Avoid Them

Accepting a candidate as it is. A generated candidate is a list of everything that comes before an important item in the relative order; most of it is wrong for the conclusion. *Avoid it:* keep only the conditions (or nodes) that are true for the conclusion, and reject the candidate if none are.

Generalizing too far. `symptom = yellow leaves` alone would send every yellow-leaved plant to a nitrogen check, including a durian drowning in water. *Avoid it:* before keeping a single condition, think of a crop or a situation where it would be wrong; if you can, keep another condition too, or reject the candidate.

Reading the rates as popularity. A high NUR does not mean many farmers got that advice; it means much of the knowledge passes through that node. *Avoid it:* use the rates for what they measure — the structure of the knowledge — and keep a separate count of consultations if you need one.

Comparing rates across knowledge bases. The rates depend on how many nodes a knowledge base has and how they share variables; an NUR of 12 in one base and 12 in another say nothing about each other. *Avoid it:* compare rates only within one knowledge base, and only at one moment.

Forgetting that the rates change. Every new node changes NS, and every new exception earns Credits, so the candidates change too: the anthracnose candidate had twelve conditions in Section 6.4 and thirteen in session 4. *Avoid it:* always review the candidates of the knowledge base as it is now, and read the numbers on the screen rather than remembering them.

Hand-computing with the wrong Credits. Credit is earned only by the conditions of the *parent* of a new node, and only when the new case agrees with them. *Avoid it:* when you check the rates by hand, write down every node's creation and which parent it was hung under.

Advising chemicals. A doctor that names a pesticide and dose can harm a farmer, a harvest and a stream. *Avoid it:* give the safe steps, and send the farmer to the penyuluh for anything chemical.

6.10 Chapter Summary

- **Value analysis** measures how much the structure of a knowledge base uses each piece of it. Each condition has a **Credit** (how often later cases agreed with it), an **NS** (how many nodes share its variable) and a **CD** (its position over the number of conditions in its node); its **Weight** is NS × CD and its **VUR** is Credit × Weight; a node's **NUR** is the mean VUR of its conditions and a rule's **RUR** the mean NUR of its nodes (Equation (6.1)–Equation (6.6), **Figure 6.1**).

- Worked by hand on the rice corner, the rates matched the engine on all fourteen numbers (**Listing 6.1**, **Listing 6.2**).
- The rates give a **relative order** of nodes and conditions, from the least used, that keeps every existing rule's order. **Rule generation** proposes rules and nodes that end in the most important nodes and variables, and the expert keeps only what is true (**Figure 6.2**).
- On the farm knowledge base the single candidate rule was rejected — rule generation needs nodes that feed each other, as in the Zookeeper — and one condition of a candidate node, `symptom = dark sunken spots`, became a generated node that answered papaya and dragon-fruit farms nobody had taught (**Listing 6.3**, **Listing 6.4**, **Figure 6.3**).
- The **crop and fruit-tree doctor** (**Listing 6.5**, **Listing 6.6**) advised on durian, mango, chili and rice with current, safe agronomy, learned root rot from the penyuluh, let her review the generated candidates, and answered a papaya farmer with the node she accepted; its usage rates showed that symptoms had become more important than crops.

6.11 Exercises

6.1 In the rice corner, suppose Mbak Nur teaches one more exception under `nutrient check#1`: rice with yellow leaves and *white* insects at the base, for which the advice is different. Which Credits change? Recompute the VURs and the NUR of `nutrient check#1`.

6.2 A node has three conditions, and their variables are shared by 1, 3 and 2 nodes. All Credits are 1. Compute the CD, Weight and VUR of each condition and the NUR of the node. Which condition weighs most, and why?

6.3 Explain in your own words why the thesis gives a condition near the end of a node more weight than one at the start. Use `crop = rice, symptom = yellow leaves` as the example.

6.4 In Section 6.4 the RUR of `planthopper emergency#1` (5.67) is lower than that of the one-node rules with RUR 6.00, although its first node, `nutrient check#2`, has NUR 12.00. Compute it from the NURs of its three nodes, and explain the result.

6.5 Why did the engine propose no candidate rule for `nutrient check#2`, which is one of the two most important nodes?

6.6 Change the review screen of the doctor so that it prints, beside each condition of a candidate, the VUR of that condition in the node it came from (or a dash if it comes from another node). Which engine calls do you need?

6.7 Mbak Nur accepted `symptom = dark sunken spots` alone. Suggest a situation in which this generated node could give wrong advice, and say how she would correct it without deleting the node.

6.8 Run rule generation on the Zookeeper cases of the thesis (`vcirs-core/data/zookeeper-cases.json`), print all the candidate rules, accept `carnivore#1G` keeping the nodes `mammal#2`, `mammal#1` and `carnivore#1`, and print the IF-THEN rule it becomes after knowledge base transformation. Why is this candidate worth keeping, when the farm's was not?

6.9 The doctor asks the crop first because every seed case began with it. After the generated node, would it be better to ask the symptom first? Discuss, using the usage rates printed after session 5 and the idea of `questionOrder()` from Chapter 4.

6.12 Solutions to the Exercises

Solution 6.1. The new case agrees with both conditions of the parent, so `crop = rice` and `symptom = yellow leaves` in `nutrient check#1` each earn one more Credit: 3. NS does not change (the new exception uses `insects`, which is now shared by two nodes, but `crop` and `symptom` are still used by two). The VURs become $3 \times 1 = 3$ and $3 \times 2 = 6$, and the NUR $(3 + 6)/2 = 4.5$. The new exception's own condition, `insects = white`, starts with Credit 1; and because `insects` is now shared by two nodes, the VUR of `insects = brown hoppers` in `planthopper control#1` doubles too, to $2 \times 2 = 4$.

Solution 6.2. The CDs are $\frac{1}{3}$, $\frac{2}{3}$ and 1. The Weights are $1 \times \frac{1}{3} \approx 0.33$, $3 \times \frac{2}{3} = 2$ and $2 \times 1 = 2$. With Credits of 1, the VURs equal the Weights, and the NUR is $(0.33 + 2 + 2)/3 \approx 1.44$. The second and third conditions weigh the same: the second is shared more widely, the third stands closer to the conclusion.

Solution 6.3. Conditions early in a node narrow the field in general ways; the last ones decide the case. `crop = rice` only says that the node is about rice; many rice nodes share it. `symptom = yellow leaves` is what makes this node *this* advice rather than blast control. The closeness degree rewards the condition that finally decides, which is also the condition an expert would name if asked "why this advice?".

Solution 6.4. The rule is `nutrient check#2 -> planthopper control#1 -> planthopper emergency#1`, with NURs 12.00, 4.00 and 1.00. Its RUR is $(12 + 4 + 1)/3 = 17/3 \approx 5.67$. The two exception nodes each have one condition used nowhere else, with low VURs, and they pull the mean down: a rule is judged by all its nodes, and this one ends in very specific, little-shared knowledge.

Solution 6.5. A generated rule for an important node is the part of the relative node order that comes before it, ending with it. `nutrient check#2` is the very first node of the order, so its candidate would be the one-node sequence `nutrient check#2` — which is already a rule of the knowledge base. The engine does not propose a rule that already exists [7].

Solution 6.6. For each condition of the candidate, find a node of the knowledge base whose conditions include it at some position `i` — the candidate's source node, `kb.node(candidate.source)` (Python `kb.node(g.source)`), is the natural first place to look — and print `kb.usage().vur(node.id, i)` beside it, formatted with `Num.fixed(x, 4)`, or a dash if the condition comes only from other nodes. The calls are `kb.node`, `node.conditions`, `Condition.same` to compare, and `usage().vur`. Computing `kb.usage()` once, before the loop, keeps it fast.

Solution 6.7. Some fruit diseases and injuries also leave dark sunken marks — for example sunburn followed by rot, or bruises from a fall — and a farmer who describes them as "dark sunken spots" will be told it is anthracnose. Mbak Nur corrects this the RDR way: the next time such a case comes, she teaches an exception under `anthracnose care#1G` with the fact that tells it apart (for example `marks only on the sunny side = yes`), and the generated node keeps working for everything else. If she decides the generalization was a mistake, she stops the node, as in Chapter 5; it is never deleted.

Solution 6.8. The solution programs are `ZooGeneration.java` and `zoo_generation.py`. The Python version is:

```
"""Exercise 6.8: rule generation where it shines, on the Zookeeper of the
VCIRS thesis, whose nodes conclude facts that other nodes use."""

import pathlib
import sys

CORE = pathlib.Path(__file__).resolve().parents[2] / "vcirs-core"
sys.path.insert(0, str(CORE / "python"))
from vcirs import KnowledgeBase, casefile

name, description, steps = casefile.read_cases(
    str(CORE / "data" / "zookeeper-cases.json"))
kb = casefile.build(KnowledgeBase(name, description), steps)
candidates = kb.generate_rules()
for g in candidates:
    print(g.id + ": " + " -> ".join(g.nodes))
carnivore = [g for g in candidates if g.id == "carnivore#1G"][0]
rule = kb.accept_rule(carnivore, ["mammal#2", "mammal#1", "carnivore#1"],
                      "zookeeper")
print("Kept: " + str(rule))
for r in kb.transform():
    if r.id == rule.id:
        print("\n".join(r.lines()))
```

It prints:

```
carnivore#1G: bird#2 -> albatross#1 -> bird#1 -> mammal#2 -> mammal#1 -> carnivore#1
cheetah#1G: bird#2 -> albatross#1 -> bird#1 -> mammal#2 -> mammal#1 -> carnivore#1 ->
carnivore#2 -> ungulate#1 -> ungulate#2 -> ostrich#1 -> penguin#1 -> giraffe#1 -> zebra#1 -
> cheetah#1
giraffe#1G: bird#2 -> albatross#1 -> bird#1 -> mammal#2 -> mammal#1 -> carnivore#1 ->
carnivore#2 -> ungulate#1 -> ungulate#2 -> ostrich#1 -> penguin#1 -> giraffe#1
ostrich#1G: bird#2 -> albatross#1 -> bird#1 -> mammal#2 -> mammal#1 -> carnivore#1 ->
carnivore#2 -> ungulate#1 -> ungulate#2 -> ostrich#1
ungulate#1G: bird#2 -> albatross#1 -> bird#1 -> mammal#2 -> mammal#1 -> carnivore#1 ->
carnivore#2 -> ungulate#1
Kept: carnivore#1G: mammal#2 -> mammal#1 -> carnivore#1
carnivore#1G [RULE CF=50]
  IF   give milk = yes [CF=50]
  AND  has hair = yes [CF=50]
  AND  mammal = yes [CF=50]
  AND  eat meat = yes [CF=50]
  THEN carnivore = yes
```

The five candidate rules are those of the thesis, including `giraffe#1G`, which the thesis's table omits although it ties with the others (the engine's design notes explain the difference). Keeping `mammal#2`, `mammal#1` and `carnivore#1` gives a rule that, after transformation, reads *if it gives milk, has hair, is a mammal and eats meat, then it is a carnivore* — true, if more specific than necessary, and exactly the kept rule of the thesis [7]. It is worth keeping because the nodes feed each other: `mammal#1` and `mammal#2` conclude `mammal = yes`, which `carnivore#1` uses. In the farm knowledge base no node concludes anything another node uses, so a sequence of farm nodes is not a chain but a list of unrelated advice.

Solution 6.9. After session 5, `symptom` is the most important variable, shared by ten nodes against nine for `crop`, and the generated node needs only the symptom. Asking the symptom first would let the doctor answer the papaya farmer with one question, and would rule out most nodes at once for any farmer. But nine of the ten nodes still need the crop too, so most consultations would ask both anyway. `questionOrder()` ranks variables by their highest VUR, and after the generated node `symptom` ranks first; an app that follows it asks the symptom first, which is also how Mbak Nur herself starts at the market: *what do you see?* Chapter 7 puts the question order to work.

References

[1] CNBC Indonesia, "Lahan pertanian makin sempit, petani gurem naik pesat" (Sensus Pertanian 2023, Badan Pusat Statistik), Dec. 4, 2023. [Online]. Available: https://www.cnbcindonesia.com/news/20231204161252-4-494410/lahan-pertanian-makin-sempit-petani-gurem-naik-pesat

[2] A. Singh, C. Chow, K. Nathaniel, Y. L. Vun, S. Javad, and K. Jabeen, "Management of *Phytophthora* and *Phytopythium* oomycete diseases in durian (*Durio zibethinus*)," *Crop Protection*, vol. 190, Art. no. 107086, 2025.

[3] A. Worku, A. Alemu, and B. Tsedaley, "Combining plant extracts and hot water treatments for the management of postharvest mango anthracnose (*Colletotrichum gloeosporioides*)," *Scientific Reports*, vol. 15, Art. no. 5074, 2025.

[4] S. De Faveri, S. Vijaysegaran, and J. Cheesman, *Area-Wide Management of Methyl Eugenol-Attracted Fruit Flies in Mango: A Systems Approach*, ACIAR Monograph No. 222. Canberra, Australia: Australian Centre for International Agricultural Research, 2024.

[5] Y. M. A. Sandra, "Identification of recovery traits in chili (*Capsicum* spp.): implications for breeding begomovirus-resistant varieties," *Plumula: Berkala Ilmiah Agroteknologi*, vol. 13, no. 2, pp. 63–70, Jul. 2025.

[6] Pacific Pests, Pathogens and Weeds, "Rice brown planthopper (064)," fact sheet, v10. [Online]. Available: https://apps.lucidcentral.org/pppw_v10/text/web_full/entities/rice_brown_planthopper_064.htm

[7] M. M. I. Subakti, "A Variable-Centered Intelligent Rule System," M.Sc.Eng. thesis, Dept. of Computer Science and Information Engineering, Taipei, Taiwan, Jun. 2005.

[8] I. Subakti, "A variable-centered intelligent rule system," in *Proc. 1st Annual Int. Conf. Information and Communication Technology Seminar (ICTS 2005)*, vol. 1, no. 1, Surabaya, Indonesia, Aug. 2005, pp. 167–174.

[9] I. Subakti, *Variable-Centered Intelligent Rule System*, Indonesian ed. (resume). Surabaya, Indonesia: Dept. of Informatics, Institut Teknologi Sepuluh Nopember, 2006.

[10] P. Compton and B. H. Kang, *Ripple-Down Rules: The Alternative to Machine Learning*. Boca Raton, FL, USA: CRC Press, 2021.

[11] L. Sebayang, *Teknik Pengendalian Penyakit Kuning pada Tanaman Cabai*. Medan, Indonesia: Balai Pengkajian Teknologi Pertanian Sumatera Utara, 2013; published by Direktorat Jenderal Hortikultura, Kementerian Pertanian. [Online]. Available: https://hortikultura.pertanian.go.id/wp-content/uploads/2024/11/Teknik-Pengendalian-Penyakit-Kuning-Pada-Tanaman-Cabai_watermark.pdf

[12] International Rice Research Institute, "Blast (leaf and collar)," *Rice Knowledge Bank*. [Online]. Available: http://www.knowledgebank.irri.org/training/fact-sheets/pest-management/diseases/item/blast-leaf-collar

Chapter 7

Small-Shop Customer-Message Router: Usage and Credit Statistics

By the end of this chapter you will be able to turn the WhatsApp messages of a small shop, written in Indonesian, in English or in a mixture of both, into a handful of yes-or-no **signals**; teach a VCIRS knowledge base to send every message to the right person in the shop, and correct it with one sentence when it is wrong; explain what the **Credit** of a condition counts and how VCIRS turns Credits into the three usage rates of the thesis — the **Variable Usage Rate** (VUR), the **Node Usage Rate** (NUR) and the **Rule Usage Rate** (RUR); compute those rates by hand and check them against the engine; and read the numbers the way a shopkeeper would: which signals matter most, which rules are corrected most often, which question to listen for first, and where two rules compete for the same message. The app of the chapter, the **message router** of Toko Rina, is written in Java and in Python, and both print exactly the same thing.

7.1 Bu Rina's Phone

Toko Rina is a *toko kelontong*, a neighborhood grocery, on the corner of a housing estate. It sells rice, cooking oil, eggs, sugar, instant noodles, LPG cylinders and drinking water in nineteen-liter gallons, and like most small shops in Indonesia today it takes a good share of its orders through WhatsApp. That is no surprise: in the Digital 2026 report for Indonesia, WhatsApp is the most used social platform in the country, used every month by about nine in ten internet users [1]. For a shop, it means that the counter now has a second door, and this one never closes.

Three people answer that door. **Bu Rina**, the owner, knows the prices and the stock and takes the orders. Her daughter **Dinda** keeps the money: she checks transfers, QRIS payments and the small debts of neighbors who pay at the end of the month. Her husband **Pak Darto** does the deliveries on his motorcycle. They share one phone, the shop's WhatsApp Business number, and on a busy morning it receives a message every few minutes. Some are simple: *Bu, gula pasir 1 kg harganya berapa?* Some are urgent: *Minyaknya bocor di plastik, tolong diganti*. Some are in English, from the students and young families who rent in the estate: *Do you deliver to Jalan Melati?* And some are rubbish: *Selamat! Anda menang undian, klik www…*

The problem is not answering; everyone in the family can answer. The problem is **who** answers. A payment message read by Pak Darto waits until Dinda comes home. A complaint read late becomes an angry customer. A question about the delivery fee answered by Bu Rina has to be passed on to Pak Darto anyway. WhatsApp Business offers labels and quick replies to keep chats organized [2], but someone still has to read each message and choose the label. Bu Rina would like the phone to do that first reading for her: look at each message, decide who should handle it, and put it in that person's pile — and, when it gets one wrong, to learn from her correction, the way the Wi-Fi doctor of Chapter 3 learned from Mas Anto.

This chapter builds that router. But it also answers a second question that every shop owner asks sooner or later about any system that learns: *what has it learned, and what matters most in it?* After a few weeks of corrections, Bu Rina wants to know which kinds of messages cause the most

trouble, which words the router relies on most, and what a new helper should listen for first when a customer sends a voice note instead of text. VCIRS answers those questions with numbers that it keeps anyway, as a side effect of learning: the **Credit** of every condition and the **usage rates** computed from it [3], [4]. Those numbers are the real subject of this chapter.

7.2 The Idea in Plain Words

7.2.1 Signals, not understanding

A computer program that truly understood Indonesian and English messages would be wonderful, and Chapter 13 connects a local language model to an expert system. But a shop does not need understanding to route most messages. It needs to notice a few kinds of words. A message that contains *harga*, *berapa* or *how much* is about a price. One with *transfer*, *tf*, *bayar* or *QRIS* is about a payment. *Antar*, *kirim*, *ongkir* and *deliver* are about delivery; *rusak*, *bocor*, *kecewa*, *broken* and *missing* are complaints.

We call each kind of word a **signal**. The router knows ten signals: `price`, `stock`, `order`, `payment`, `delivery`, `complaint`, `hours` (opening hours), `late` (an order that has not arrived), `proof` (a proof of payment) and `link` (a web address, usually spam). Each signal is either present in a message or not, so each becomes a VCIRS variable with the value `yes` or `no`. The list of words for each signal lives in a small text file that the family can edit without programming. **Figure 7.1** shows the whole path of one message.

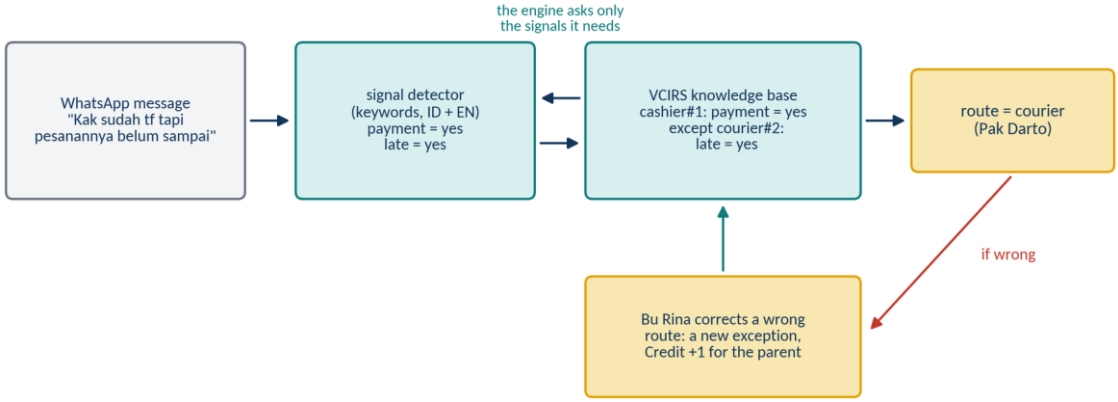

Figure 7.1 The path of one message through the router. The detector turns words into yes/no signals, but only when the knowledge base asks for them; the knowledge base picks the route; when Bu Rina corrects a wrong route, a new exception is added and its parent's conditions earn Credit.

Two design decisions are worth saying out loud. First, the detector does not look for all ten signals in advance. The VCIRS engine asks for a signal only when a node it is checking needs it, exactly as the Wi-Fi doctor asked a customer only the questions it needed. Here the "person" answering is the detector, and its answers cost nothing, but the same knowledge base can be used by a human helper who listens to a voice note and answers the questions — and then every question saved is time saved. Second, the detector is deliberately simple and honest: it matches whole words and short phrases, and nothing else. It will make mistakes that a person would not make, and we will see them. The point of VCIRS is that such mistakes can be corrected in the knowledge base, one message at a time, by the person who knows the right answer.

7.2.2 Routing is classification

Deciding who handles a message is a classification problem, the same kind of problem the Wi-Fi doctor solved: the facts are the signals, and the conclusion is `route = owner`, `route = cashier`, `route = courier`, `route = owner now` (a complaint, which Bu Rina wants to see at once), `route = auto reply` (the quick reply with the opening hours) or `route = ignore`. When no node gives a route, the message goes to Bu Rina, who reads it herself; the program calls this *owner reads it*.

Bu Rina starts with seven rules, one per signal: a price question goes to the owner, a stock question goes to the owner, an order goes to the owner, a payment goes to the cashier, a delivery question to the courier, a complaint to the owner at once, and a question about opening hours gets the auto reply. These are exactly the "rules that work most of the time" of Chapter 3, and exactly as there, they are over-general. *Berapa ongkir ke Perum Griya Asri?* contains *berapa* (price) and *ongkir* (delivery); the first rule sends it to the owner, but the delivery fee is Pak Darto's business. So Bu Rina adds an exception in the context where the mistake happened: under the price rule, *if delivery is also mentioned, route to the courier*. This is Ripple Down Rules [5] as the VCIRS engine does it, with one small refinement that Section 7.5 explains: Bu Rina may say which signal made the difference, and the engine then builds the exception from that signal alone.

7.2.3 Credit: counting agreement

Every condition stored in a VCIRS node carries a whole number called its **Credit** [3], [6]. When a node is created, each of its conditions starts with Credit 1. After that, the Credit of a condition grows by one every time an expert creates a new node *under* its node — an exception — for a case that **agrees** with the condition (or that restates its variable). The thesis puts it in one sentence: Credit "increases when the user creates a node which agrees with the values of the old case" [3].

Think about what that means in the shop. The seed rule `price = yes → route = owner` starts with Credit 1. When Bu Rina corrects *Berapa ongkir…?* by adding an exception under it, the message did mention a price, so it agreed with the condition `price = yes`, and the Credit becomes 2. When she corrects *Harga telur kok naik terus, kecewa saya* (price and a complaint), it becomes 3; and so on. A high Credit therefore says two things at once: this condition is **true for many of the cases that came to this node**, and this node is a **place where the expert keeps refining** knowledge. The condition is well used; the node is busy.

Notice what Credit does *not* count. It does not count how many messages were routed by a node. A rule that routes a hundred messages a day perfectly never gets a correction, and its Credit stays 1. Credit is about the **structure** of the knowledge, grown by teaching, not about the traffic. Section 7.2.6 comes back to this difference, because a shop needs both kinds of counting.

Two more properties of Credit are worth knowing before we use it. It **never goes down**: VCIRS never deletes a node, so a correction, once made, stays in the count, even if the node is later stopped. And it is **local**: the Credit of `price = yes` in the price rule says nothing about a `price = yes` condition in another node, which has its own Credit. Think of each condition as having its own small notebook, in which the expert makes a mark every time she refines the knowledge right below it for a case that agreed with it. After a month, the notebooks with many marks show where the expert's attention went.

7.2.4 Weight, closeness and the three usage rates

Credit is the raw material. VCIRS combines it with two other facts about a variable to measure how much the variable is used; the combination is called **value analysis**, or **usage assignment**, in the thesis [3], [4]. The measure for one variable inside one node is the **Variable Usage Rate**:

$$VUR_i = Credit_i \times Weight_i \tag{7.1}$$

The weight of a variable in a node has two parts. The first is **NS**, the number of nodes that share the variable: a variable used in four nodes is more important to the knowledge base than one used in a single node. The second is the **Closeness Degree** CD, which says how close the variable stands to the node's conclusion:

$$Weight_i = NS_i \times CD_i \tag{7.2}$$

$$CD_i = \frac{VO_i}{TV} \tag{7.3}$$

Here VO_i is the order of variable i in the node (1 for the first condition, 2 for the second, and so on) and TV is the total number of variables in the node. In a node with two conditions, the first has closeness $1/2$ and the second $2/2 = 1$. The idea is that a node is a *sequence* of conditions that leads to its conclusion, and the condition checked last is the one that finally decides; it stands closest to the conclusion [3].

A node's usage is the average of the usage of its variables, and a rule's usage is the average of the usage of its nodes. With N variables in node j and M nodes in rule k:

$$NUR_j = \frac{1}{N}\sum_{i=1}^{N} VUR_{ij} \tag{7.4}$$

$$RUR_k = \frac{1}{M}\sum_{j=1}^{M} NUR_{jk} \tag{7.5}$$

Remember from Chapter 3 that a **rule**, in the VCIRS structure, is the path of nodes from the top level down to a node: the rule of an exception contains its parent and itself. So RUR measures the usage of a whole line of reasoning, and NUR of one step of it.

A small worked example makes the formulas concrete. Suppose the price node `owner#1` has one condition, `price = yes`, with Credit 3, and `price` appears in no other node. Then $NS = 1$, $CD = 1/1 = 1$, the weight is $1 \times 1 = 1$, and by Equation (7.1) its VUR is $3 \times 1 = 3$. Suppose `delivery = yes` appears in three nodes and one of them is `owner#2: order = yes, delivery = yes`. In that node, `delivery` is the second of two conditions, so $CD = 2/2 = 1$, the weight by Equation (7.2) is $3 \times 1 = 3$, and with Credit 1 its VUR is 3. `order` is the first of the two, so its closeness by Equation (7.3) is $1/2$; if it is used in no other node its weight is $1 \times 1/2 = 1/2$, and its VUR is $1/2$. By Equation (7.4) the NUR of `owner#2` is $(1/2 + 3)/2 = 1.75$. Section 7.3 computes exactly these numbers with a program, and then lets the engine compute them too. Finally, Equation (7.5) averages NURs along a rule: an exception with NUR 1 under a node with NUR 3 makes a rule with RUR $(3 + 1)/2 = 2$.

> **DEFINITION.** The *Credit* of a condition counts how many times an expert created a node under its node for a case that agreed with it (plus one for its own creation). The *Variable Usage Rate* of a condition is its Credit times the number of nodes sharing its variable times its closeness to the conclusion; the *Node Usage Rate* is the average VUR of a node's conditions; the *Rule Usage Rate* is the average NUR of a rule's nodes. Larger means more used.

7.2.5 What the numbers are for

The thesis gives value analysis three purposes [3]. The first is a guideline **while building** knowledge: which rules, nodes and variables to visit first, because they are the most used. The second is a guideline **while consulting** the knowledge base: which variables to ask about first, and which parts of the knowledge are the most interesting to the user. The third is a guideline for **rule generation**, the subject of Chapter 6, where the usage rates and the sharing counts together decide which new rules are worth proposing.

Beside the usage rates, VCIRS keeps the plainer measure of **variable analysis**: how many nodes share each variable (NS) and how many rules share each node. The variables shared by the most nodes — at least two — are the **most important variables**, and the nodes shared by the most rules are the most important nodes [3], [4]. The engine gives both, together with a **question order**: the variables sorted by their largest VUR, most used first.

For Toko Rina these become very practical questions. Which signal is most important, so that its word list deserves the most care? Which rule is corrected most often, so that Bu Rina should look at it first when she reviews the knowledge? And when a helper listens to a voice note, what should she listen for first?

7.2.6 Two kinds of counting

A shop needs a second kind of number that VCIRS does not keep in its structure: **traffic**. How many messages did each person receive this morning? How many did each node route? Those counts come from running the router, not from teaching it, and they answer different questions: whether Dinda is overloaded, whether the auto reply is worth keeping, whether a rule is never used at all.

The router of this chapter prints both. The Credits and usage rates tell Bu Rina about her **knowledge**: where it was refined, which signals it leans on. The traffic counts tell her about her **customers**: what they write about. A rule with high usage and low traffic is a subtle rule, much refined for rare messages; a rule with low usage and high traffic is a simple rule that works. Both are fine; they call for different kinds of attention.

7.3 Usage Rates by Hand

Before trusting the engine's numbers we compute them ourselves, as Chapter 3 built Ripple Down Rules from scratch. **Listing 7.1** describes five nodes as plain data — their identifiers, their parents and their conditions with Credits — computes NS, CD, VUR, NUR and RUR with Equation (7.1) to Equation (7.5), and then builds the same five nodes with vcirs-core and prints the engine's tables.

Listing 7.1 `UsageByHand.java`: the usage rates of five nodes computed by hand, then by vcirs-core.

```
import java.util.ArrayList;
import java.util.List;

import vcirs.Case;
import vcirs.KnowledgeBase;
import vcirs.Num;
import vcirs.Report;

/**
 * Credit, VUR, NUR and RUR computed by hand for five nodes, then by
 * vcirs-core for the same five nodes, to show that they agree.
 */
public class UsageByHand {
    record Cond(String variable, int credit) {
    }

    record Node(String id, String parent, List<Cond> conds) {
    }

    // node id, parent, the conditions in their order with their Credit
    static final List<Node> NODES = List.of(
            new Node("owner#1", null, List.of(new Cond("price", 3))),
            new Node("owner#2", null, List.of(new Cond("order", 1),
                    new Cond("delivery", 1))),
            new Node("courier#1", null, List.of(new Cond("delivery", 1))),
            new Node("courier#2", "owner#1", List.of(new Cond("delivery", 1))),
            new Node("owner now#1", "owner#1",
                    List.of(new Cond("complaint", 1))));

    /** How many nodes use the variable (NS). */
    static int ns(String variable) {
        int n = 0;
        for (Node node : NODES) {
            if (node.conds.stream().anyMatch(c -> c.variable.equals(variable))) {
                n++;
            }
        }
        return n;
    }

    static double vur(List<Cond> conds, int i) {
        Cond c = conds.get(i);
        double cd = (double) (i + 1) / conds.size();  // Closeness Degree VO/TV
        return c.credit * ns(c.variable) * cd;         // Credit x NS x CD
    }
```

Listing 7.1 (Continued) *UsageByHand.java: the usage rates of five nodes computed by hand, then by vcirs-core.*

```
    static double nur(List<Cond> conds) {
        double sum = 0;
        for (int i = 0; i < conds.size(); i++) {
            sum += vur(conds, i);
        }
        return sum / conds.size();
    }

    /** The nodes from the top level down to nodeId. */
    static List<Node> ruleOf(String nodeId) {
        List<Node> path = new ArrayList<>();
        while (nodeId != null) {
            String id = nodeId;
            Node node = NODES.stream().filter(n -> n.id.equals(id))
                    .findFirst().get();
            path.add(0, node);
            nodeId = node.parent;
        }
        return path;
    }

    static String pad(String s, int width) {
        return String.format("%-" + width + "s", s);
    }

    public static void main(String[] args) {
        System.out.println("By hand:");
        for (Node node : NODES) {
            List<String> rates = new ArrayList<>();
            for (int i = 0; i < node.conds.size(); i++) {
                rates.add(Num.fmt(vur(node.conds, i), 2));
            }
            List<Node> rule = ruleOf(node.id);
            double rur = 0;
            for (Node n : rule) {
                rur += nur(n.conds);
            }
            rur /= rule.size();
            System.out.println("  " + pad(node.id, 12) + " VUR "
                    + pad(String.join(", ", rates), 10) + " NUR "
                    + pad(Num.fmt(nur(node.conds), 2), 5) + " RUR "
                    + Num.fmt(rur, 2));
        }
```

Listing 7.1 (Continued) *UsageByHand.java: the usage rates of five nodes computed by hand, then by vcirs-core.*

```
        KnowledgeBase kb = new KnowledgeBase("five nodes");
        kb.refine(Case.of(List.of("price = yes"), List.of("route = owner")),
                KnowledgeBase.TOP_LEVEL, null);
        kb.refine(Case.of(List.of("order = yes", "delivery = yes"),
                List.of("route = owner")), KnowledgeBase.TOP_LEVEL, null);
        kb.refine(Case.of(List.of("delivery = yes"), List.of("route = courier")),
                KnowledgeBase.TOP_LEVEL, null);
        kb.refine(Case.of(List.of("price = yes", "delivery = yes"),
                List.of("route = courier")), KnowledgeBase.ADD_NEW, "owner#1");
        kb.refine(Case.of(List.of("price = yes", "complaint = yes"),
                List.of("route = owner now")), KnowledgeBase.ADD_NEW, "owner#1");
        System.out.println("By vcirs-core:");
        List<String> lines = new ArrayList<>(Report.vurTable(kb, 2));
        lines.addAll(Report.rurTable(kb, 2));
        for (String line : lines) {
            System.out.println("  " + line);
        }
    }
}
```

The five nodes are the worked example of Section 7.2.4 made complete. `owner#1` routes price questions and has two exceptions, `courier#2` (a price question that also mentions delivery) and `owner now#1` (a price question with a complaint), so its condition `price = yes` has Credit 3. `owner#2` is a node with two conditions, `order = yes, delivery = yes`, which shows the Closeness Degree at work. `courier#1` routes delivery questions. `ns` counts the nodes whose conditions use a variable; `vur` applies Equation (7.3), Equation (7.2) and Equation (7.1) in three lines; `nur` averages (Equation (7.4)); `ruleOf` climbs from a node to the top level through the parents, and `main` averages the NURs along that path (Equation (7.5)).

The second half builds the same knowledge base with the engine's `refine`: three top-level nodes (option 5) and two exceptions under `owner#1` (option 2, section B of the list of difference). We never set a Credit ourselves; the engine raises the Credit of `price = yes` each time an exception is created under `owner#1` for a case that agrees with it. **Listing 7.2** is the same program in Python.

Listing 7.2 `usage_by_hand.py`: the same computation in Python.

```
"""Credit, VUR, NUR and RUR computed by hand for five nodes, then by
vcirs-core for the same five nodes, to show that they agree."""

import pathlib
import sys

sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import ADD_NEW, TOP_LEVEL, Case, KnowledgeBase, num, report

# node id, parent, [(variable, credit)] -- the conditions in their order
NODES = [
    ("owner#1", None, [("price", 3)]),
    ("owner#2", None, [("order", 1), ("delivery", 1)]),
    ("courier#1", None, [("delivery", 1)]),
    ("courier#2", "owner#1", [("delivery", 1)]),
    ("owner now#1", "owner#1", [("complaint", 1)]),
]


def ns(variable):
    """How many nodes use the variable (NS)."""
    return sum(1 for _, _, conds in NODES
               if any(v == variable for v, _ in conds))


def vur(conds, i):
    variable, credit = conds[i]
    cd = (i + 1) / len(conds)             # Closeness Degree = VO / TV
    return credit * ns(variable) * cd     # Credit x Weight, Weight = NS x CD


def nur(conds):
    return sum(vur(conds, i) for i in range(len(conds))) / len(conds)


def rule_of(node_id):
    """The nodes from the top level down to node_id."""
    path = []
    while node_id is not None:
        node = next(n for n in NODES if n[0] == node_id)
        path.insert(0, node)
        node_id = node[1]
    return path
```

Listing 7.2 (Continued) *usage_by_hand.py: the same computation in Python.*

```
print("By hand:")
for node_id, _, conds in NODES:
    rates = [num.fmt(vur(conds, i), 2) for i in range(len(conds))]
    rule = rule_of(node_id)
    rur = sum(nur(n[2]) for n in rule) / len(rule)
    print("  " + node_id.ljust(12) + " VUR " + ", ".join(rates).ljust(10)
          + " NUR " + num.fmt(nur(conds), 2).ljust(5)
          + " RUR " + num.fmt(rur, 2))

kb = KnowledgeBase("five nodes")
kb.refine(Case.of(["price = yes"], ["route = owner"]), TOP_LEVEL)
kb.refine(Case.of(["order = yes", "delivery = yes"], ["route = owner"]),
          TOP_LEVEL)
kb.refine(Case.of(["delivery = yes"], ["route = courier"]), TOP_LEVEL)
kb.refine(Case.of(["price = yes", "delivery = yes"], ["route = courier"]),
          ADD_NEW, "owner#1")
kb.refine(Case.of(["price = yes", "complaint = yes"],
                  ["route = owner now"]), ADD_NEW, "owner#1")
print("By vcirs-core:")
for line in report.vur_table(kb, 2) + report.rur_table(kb, 2):
    print("  " + line)
```

Both print:

```
By hand:
  owner#1      VUR 3          NUR 3     RUR 3
  owner#2      VUR 0.5, 3     NUR 1.75  RUR 1.75
  courier#1    VUR 3          NUR 3     RUR 3
  courier#2    VUR 3          NUR 3     RUR 3
  owner now#1  VUR 1          NUR 1     RUR 2
By vcirs-core:
  VariableID  NodeID       VariableOrder  VUR
  ----------  -----------  -------------  ----
  order       owner#2      1              0.50
  complaint   owner now#1  1              1.00
  price       owner#1      1              3.00
  delivery    owner#2      2              3.00
  delivery    courier#1    1              3.00
  delivery    courier#2    1              3.00
  RuleID       RUR
  -----------  ----
  owner#2      1.75
  owner now#1  2.00
  owner#1      3.00
  courier#1    3.00
  courier#2    3.00
```

Check a few lines against Section 7.2.4. `owner#2` has two VURs, 0.5 for `order` (Credit 1, NS 1, closeness 1/2) and 3 for `delivery` (Credit 1, NS 3, closeness 1); its NUR is their average, 1.75, and because it is a top-level node its rule contains only itself, so its RUR is also 1.75. `owner now#1` has NUR 1 (Credit 1, `complaint` used once), but its rule is the path `owner#1 → owner now#1`, so its RUR is $(3 + 1)/2 = 2$. The engine's tables list the same numbers, sorted from the smallest to the largest as the thesis sorts them, and its Credits came from the teaching alone: the two `refine` calls under `owner#1` raised `price = yes` from 1 to 3.

> **KEY INSIGHT.** The usage rates are not extra data that someone must maintain. They are computed from the structure the expert builds anyway — which conditions are in which nodes, in what order, and how often later cases agreed with them. Teaching the knowledge base *is* collecting the statistics.

7.4 The Signal Detector

The detector reads the word lists from `data/signals.csv`, one keyword or phrase per line, with the signal's name in front:

```
# signal,keyword or phrase (lower case; Indonesian and English)
price,harga
price,berapa
price,brp
price,how much
...
payment,transfer
payment,tf
payment,transferan
...
late,belum sampai
late,belum datang
late,not arrived
...
link,http
```

The file holds 81 keywords and phrases for ten signals. It mixes Indonesian, English and the abbreviations of chat (*brp* for *berapa*, *tf* for *transfer*), because that is how customers write. **Listing 7.3** is the detector.

Listing 7.3 `Signals.java`: the signal detector reads the word lists and spots whole words and phrases in a message.

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Locale;
import java.util.Map;

/**
 * Signals: the kinds of words a customer's message contains. A signal is a
 * name such as "price" with a list of keywords and phrases, in Indonesian
 * and in English, read from data/signals.csv. The detector only spots
 * words; it does not understand the message.
 */
public class Signals {
    private final Map<String, List<List<String>>> keywords =
            new LinkedHashMap<>();

    /** "Harganya BERAPA, kak?" -> [harganya, berapa, kak] */
    public static List<String> words(String text) {
        StringBuilder clean = new StringBuilder();
        for (char ch : text.toLowerCase(Locale.ROOT).toCharArray()) {
            boolean keep = (ch >= 'a' && ch <= 'z') || (ch >= '0' && ch <= '9');
            clean.append(keep ? ch : ' ');
        }
        String s = clean.toString().strip();
        return s.isEmpty() ? List.of() : Arrays.asList(s.split(" +"));
    }

    public Signals(String path) {
        List<String> lines;
        try {
            lines = Files.readAllLines(Path.of(path), StandardCharsets.UTF_8);
        } catch (IOException e) {
            throw new IllegalStateException("cannot read " + path);
        }
        for (String line : lines) {
            line = line.strip();
            if (line.isEmpty() || line.startsWith("#")) {
                continue;
            }
            int comma = line.indexOf(',');
            keywords.computeIfAbsent(line.substring(0, comma),
```

Listing 7.3 (Continued) *Signals.java: the signal detector reads the word lists and spots whole words and phrases in a message.*

```
                    k -> new ArrayList<>()).add(words(line.substring(comma + 1)));
        }
    }

    public List<String> names() {
        return new ArrayList<>(keywords.keySet());
    }

    /** True when one of the signal's phrases appears in the message. */
    public boolean found(String text, String signal) {
        List<String> tokens = words(text);
        for (List<String> phrase : keywords.getOrDefault(signal, List.of())) {
            int n = phrase.size();
            for (int i = 0; i + n <= tokens.size(); i++) {
                if (tokens.subList(i, i + n).equals(phrase)) {
                    return true;
                }
            }
        }
        return false;
    }

    public List<String> allFound(String text) {
        List<String> out = new ArrayList<>();
        for (String s : keywords.keySet()) {
            if (found(text, s)) {
                out.add(s);
            }
        }
        return out;
    }
}
```

`words` turns a message into lower-case words: every character that is not a letter from a to z or a digit becomes a space, and the result is split on spaces. So *Harganya BERAPA, kak?* becomes `harganya`, `berapa`, `kak`. The constructor reads the file and stores, for each signal, its phrases as lists of words. `found` looks for any of a signal's phrases as consecutive words of the message: `how much` matches *How much is a 12 kg LPG refill?*, but `stock` does not match *stocking*, because only whole words count. `allFound` lists every signal present, in the order of the file; the router uses it when it has to remember a message as a case. **Listing 7.4** is the Python version.

Listing 7.4 `signals.py`: the same detector in Python.

```
"""Signals: the kinds of words a customer's message contains.

A signal is a name such as "price" with a list of keywords and phrases, in
Indonesian and in English, read from data/signals.csv. The detector only
spots words; it does not understand the message.
"""


def words(text):
    """'Harganya BERAPA, kak?' -> ['harganya', 'berapa', 'kak']"""
    clean = "".join(ch if "a" <= ch <= "z" or "0" <= ch <= "9" else " "
                    for ch in text.lower())
    return clean.split()


class Signals:
    def __init__(self, path):
        self.keywords = {}                      # signal -> list of phrases
        with open(path, encoding="utf-8") as f:
            for line in f:
                line = line.strip()
                if not line or line.startswith("#"):
                    continue
                signal, phrase = line.split(",", 1)
                self.keywords.setdefault(signal, []).append(words(phrase))

    def names(self):
        return list(self.keywords)

    def found(self, text, signal):
        """True when one of the signal's phrases appears in the message."""
        tokens = words(text)
        for phrase in self.keywords.get(signal, []):
            n = len(phrase)
            for i in range(len(tokens) - n + 1):
                if tokens[i:i + n] == phrase:
                    return True
        return False

    def all_found(self, text):
        return [s for s in self.keywords if self.found(text, s)]
```

Matching whole words has a price that the family will meet quickly: *kurirnya* (the courier) is a different word from *kurir*, and *pesanan* (an order) from *pesan*. That is why the file lists some words in several forms. It also has a benefit: the detector never finds *tf* inside *gift* or *ada* inside *adalah*, and every decision it makes can be explained by pointing at a word.

7.5 The Router on vcirs-core

Listing 7.5 holds the three things the app does with the knowledge base: build it from Bu Rina's seed rules, route a message, and learn from a correction.

Listing 7.5 `Router.java`: building the knowledge base, routing a message and teaching a correction.

```
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;

import vcirs.Case;
import vcirs.CaseFile;
import vcirs.Condition;
import vcirs.Conclusion;
import vcirs.Fact;
import vcirs.Facts;
import vcirs.InferenceResult;
import vcirs.KnowledgeBase;
import vcirs.Node;
import vcirs.Question;

/** The shop's message router on vcirs-core: route, teach, count. */
public class Router {
    public static final String NOBODY = "owner reads it";  // no node routes it
    static final String[][] QUESTIONS = {
        {"price", "Does the message ask about a price?"},
        {"stock", "Does it ask whether something is in stock?"},
        {"order", "Does the customer want to order or buy?"},
        {"payment", "Is it about a payment or a transfer?"},
        {"delivery", "Is it about delivery or a delivery fee?"},
        {"complaint", "Is the customer complaining?"},
        {"hours", "Does it ask when the shop is open?"},
        {"late", "Does the customer say an order has not arrived?"},
        {"proof", "Does it send or mention a proof of payment?"},
        {"link", "Does it contain a web link?"},
    };

    /** A knowledge base with Bu Rina's seven first rules. */
    public static KnowledgeBase newKb(String data) {
        CaseFile seed = CaseFile.read(Path.of(data, "seed-cases.json")
                .toString());
        KnowledgeBase kb = new KnowledgeBase(seed.name, seed.description);
        for (String[] q : QUESTIONS) {
            kb.declare(q[0], q[1], List.of("yes", "no"));
        }
        return seed.build(kb);
    }
```

Listing 7.5 (Continued) *Router.java: building the knowledge base, routing a message and teaching a correction.*

```
    /** The engine's asker: "does the message contain this signal?" */
    static class Detector implements vcirs.Asker {
        final Signals signals;
        final String text;
        final List<String> checked = new ArrayList<>();

        Detector(Signals signals, String text) {
            this.signals = signals;
            this.text = text;
        }

        public Fact ask(Question q) {
            checked.add(q.variable);
            return new Fact(q.variable,
                    signals.found(text, q.variable) ? "yes" : "no");
        }
    }

    /** The route, the engine's result and the signals checked. */
    public static class Routed {
        public String route;
        public InferenceResult result;
        public List<String> checked;
    }

    /**
     * Route a message. With an order, the signals are checked in that order
     * until one is found; the engine does the rest.
     */
    public static Routed route(KnowledgeBase kb, Signals signals, String text,
                               List<String> order) {
        Detector det = new Detector(signals, text);
        Map<String, Fact> facts = Facts.none();
        for (String signal : order) {
            det.checked.add(signal);
            boolean yes = signals.found(text, signal);
            facts.put(signal, new Fact(signal, yes ? "yes" : "no"));
            if (yes) {
                break;
            }
        }
        Routed r = new Routed();
        r.result = kb.infer(facts, det);
        List<Conclusion> found = r.result.conclusions();
        r.route = found.isEmpty() ? NOBODY : found.get(0).value;
        r.checked = det.checked;
```

Listing 7.5 (Continued) *Router.java: building the knowledge base, routing a message and teaching a correction.*

```
        return r;
    }

    public static Routed route(KnowledgeBase kb, Signals signals, String text) {
        return route(kb, signals, text, List.of());
    }

    /** What teach() did with one message. */
    public static class Taught {
        public String routed;
        public String fired;                    // node that routed it, or null
        public Node node;                       // the new node, or null
        public List<String> affected = List.of();
    }

    /** Route one message; if the route is wrong, learn the right one. */
    public static Taught teach(KnowledgeBase kb, Signals signals, String text,
                               String right, String because, String author) {
        Routed r = route(kb, signals, text);
        Taught t = new Taught();
        t.routed = r.route;
        t.fired = r.result.paths.isEmpty() ? null
                : r.result.paths.get(0).last().id;
        if (r.route.equals(right)) {
            kb.refine(new Case(), KnowledgeBase.ACCEPT, t.fired, null, author);
            return t;
        }
        List<Condition> facts = new ArrayList<>();
        for (String s : signals.allFound(text)) {
            facts.add(new Condition(s, "=", "yes"));
        }
        Case c = new Case(facts, List.of(new Conclusion("route", right)), 100);
        List<Condition> chosen = because == null ? null
                : List.of(new Condition(because, "=", "yes"));
        int option = t.fired != null ? KnowledgeBase.ADD_NEW
                : KnowledgeBase.TOP_LEVEL;
        List<Condition> conds = chosen != null ? chosen
                : t.fired != null ? kb.differences(t.fired, c).b : c.facts;
        t.affected = kb.affectedCornerstones(t.fired, conds, c.conclusions);
        t.node = kb.refine(c, option, t.fired, chosen, author);
        return t;
    }
}
```

Building. `newKb` reads `seed-cases.json`, which holds Bu Rina's seven first rules as seven cases with option 5 (a new top-level node each), and declares the ten signals with the question a human helper would be asked (`QUESTIONS`). The order of the seed cases matters, as the order of the top-level nodes always does in RDR: the price rule comes first, then stock, order, payment, delivery, complaint and hours. Because three of them conclude `route = owner`, the engine names them `owner#1`, `owner#2` and `owner#3` — one identifier per node, numbered in the order they were made.

Routing. The `Detector` is the engine's **asker**: when the engine needs a signal it does not know, it calls `ask`, and the detector answers `yes` or `no` from the words of the message, noting the signal in `checked`. `route` starts the engine with no facts at all, so the engine walks the top-level nodes in order and asks exactly what it needs. The optional `order` checks some signals *before* the engine starts, in a given order, until one is found; Section 7.9 uses it to test the question order of value analysis. If no node fires, the route is `owner reads it`.

Teaching. `teach` routes the message first. If the route is right, it records the check with option 0, *accept*, so the history of the knowledge base shows that the node was confirmed by a real message. If it is wrong, it builds a case from the signals found in the message (only the present ones, all `= yes`) with the right route as its conclusion. If Bu Rina named the signal that made the difference, that signal alone becomes the condition (the engine's `chosen` conditions); otherwise the new node takes section B of the list of difference, as in Chapter 3 [7]. The new node is hung under the node that gave the wrong route (option 2), or at the top level if no node routed the message (option 5). Before changing anything, `teach` asks the engine for the **affected cornerstones** of the new node, the earlier cases whose route it would change.

Why only the signals that are present? Because a message is described well by what it contains, and poorly by the long list of things it does not contain. A node built from `payment = yes, complaint = yes` reads like Bu Rina's reasoning; a node built from `payment = yes, complaint = yes, price = no, stock = no, order = no, …` would fire only for messages that avoid seven signals, which is not what she meant. The engine's `chosen` conditions let the app say exactly what the expert meant. **Listing 7.6** is the Python version.

Listing 7.6 `router.py`: the same three operations in Python.

```
"""The shop's message router on vcirs-core: route, teach, count."""

import os
import pathlib
import sys

# Tell Python where the vcirs-core engine is (two folders up, then across).
sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import (ACCEPT, ADD_NEW, TOP_LEVEL, Case, Condition, Conclusion,
                   KnowledgeBase, casefile)

NOBODY = "owner reads it"          # when no node routes the message
QUESTIONS = [
    ("price", "Does the message ask about a price?"),
    ("stock", "Does it ask whether something is in stock?"),
    ("order", "Does the customer want to order or buy?"),
    ("payment", "Is it about a payment or a transfer?"),
    ("delivery", "Is it about delivery or a delivery fee?"),
    ("complaint", "Is the customer complaining?"),
    ("hours", "Does it ask when the shop is open?"),
    ("late", "Does the customer say an order has not arrived?"),
    ("proof", "Does it send or mention a proof of payment?"),
    ("link", "Does it contain a web link?"),
]


def new_kb(data):
    """A knowledge base with Bu Rina's seven first rules."""
    name, description, steps = casefile.read_cases(
        os.path.join(data, "seed-cases.json"))
    kb = KnowledgeBase(name, description)
    for var, question in QUESTIONS:
        kb.declare(var, question, ["yes", "no"])
    return casefile.build(kb, steps)


class Detector:
    """The engine's asker: 'does the message contain this signal?'"""

    def __init__(self, signals, text):
        self.signals = signals
        self.text = text
        self.checked = []
```

Listing 7.6 (Continued) *router.py: the same three operations in Python.*

```
    def __call__(self, question):
        self.checked.append(question.variable)
        return "yes" if self.signals.found(self.text, question.variable) \
            else "no"


def route(kb, signals, text, order=()):
    """(route, result, checked signals). With an `order`, the signals are
    checked in that order until one is found; the engine does the rest."""
    det = Detector(signals, text)
    facts = {}
    for signal in order:
        det.checked.append(signal)
        facts[signal] = "yes" if signals.found(text, signal) else "no"
        if facts[signal] == "yes":
            break
    result = kb.infer(facts, det)
    found = result.conclusions()
    return (found[0].value if found else NOBODY), result, det.checked


def teach(kb, signals, text, right, because, author):
    """Route one message; if the route is wrong, learn the right one.
    Returns (routed, fired node id or None, new node or None, affected)."""
    routed, result, _ = route(kb, signals, text)
    fired = result.paths[0].last().id if result.paths else None
    if routed == right:
        kb.refine(Case(), ACCEPT, fired, author=author)
        return routed, fired, None, []
    case = Case([Condition(s, "=", "yes") for s in signals.all_found(text)],
                [Conclusion("route", right)])
    chosen = [Condition(because, "=", "yes")] if because else None
    option = ADD_NEW if fired else TOP_LEVEL
    conds = chosen if chosen else (kb.differences(fired, case).b if fired
                                   else case.facts)
    affected = kb.affected_cornerstones(fired, conds, case.conclusions)
    node = kb.refine(case, option, fired, chosen, author)
    return routed, fired, node, affected
```

7.6 The App

Listing 7.7 is the whole app. It has five commands, one per job in the shop.

Listing 7.7 `ShopRouter.java`: the message router of Toko Rina, with its five commands.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

import vcirs.Condition;
import vcirs.Fact;
import vcirs.Facts;
import vcirs.InferenceResult;
import vcirs.KnowledgeBase;
import vcirs.Node;
import vcirs.Num;
import vcirs.Report;
import vcirs.Usage;

/**
 * Toko Rina's message router: sends each WhatsApp message to the right
 * person and learns from Bu Rina's corrections.
 *   java ShopRouter teach [file ...]  build the knowledge base again from
 *                                     the seed rules and teach it
 *   java ShopRouter stats             Credit, VUR, NUR, RUR, order
 *   java ShopRouter inbox             route Monday's messages
 *   java ShopRouter route "text"      route one message, explain it
 *   java ShopRouter voice [--echo]    a person answers the questions
 */
public class ShopRouter {
    static final String DATA = "../data";
    static final String KB_FILE = DATA + "/shop-kb.json";
```

Listing 7.7 (Continued) *ShopRouter.java: the message router of Toko Rina, with its five commands.*

```
    /** [message, right route, signal or null] from a data file. */
    static List<String[]> labelled(String name) {
        List<String[]> out = new ArrayList<>();
        try {
            for (String line : Files.readAllLines(Path.of(DATA, name),
                    StandardCharsets.UTF_8)) {
                if (line.isBlank() || line.startsWith("#")) {
                    continue;
                }
                String[] p = line.split("\\|");
                out.add(new String[] {p[0].strip(), p[1].strip(),
                    p.length > 2 ? p[2].strip() : null});
            }
        } catch (IOException e) {
            throw new IllegalStateException("cannot read " + name);
        }
        return out;
    }

    static String rjust(int n, int width) {
        return String.format("%" + width + "d", n);
    }

    static String ljust(String s, int width) {
        return String.format("%-" + width + "s", s);
    }

    static void teach(Signals signals, List<String> files) {
        KnowledgeBase kb = Router.newKb(DATA);
        System.out.println("Seed rules: " + kb.summary());
        List<String[]> seen = new ArrayList<>();
        int right = 0;
        for (String name : files) {
            System.out.println("Teaching " + name + ":");
            for (String[] m : labelled(name)) {
                seen.add(m);
                Router.Taught t = Router.teach(kb, signals, m[0], m[1], m[2],
                        "Rina");
                System.out.println(rjust(seen.size(), 2) + ". " + m[0]);
                List<String> found = signals.allFound(m[0]);
                System.out.println("    signals: " + (found.isEmpty()
                        ? "(none)" : String.join(", ", found)));
                String by = t.fired != null ? " (" + t.fired + ")" : "";
                if (t.node == null) {
                    right++;
                    System.out.println("    routed to " + t.routed + by
```

Listing 7.7 (Continued) *ShopRouter.java: the message router of Toko Rina, with its five commands.*

```
                        + ": right");
                continue;
            }
            System.out.println("    routed to " + t.routed + by
                    + ": WRONG, Rina says " + m[1]);
            String where = t.fired != null ? "exception of " + t.fired
                    : "top level";
            System.out.println("    learned " + t.node + " (" + where
                    + ")");
            System.out.println("    earlier cases whose route would "
                    + "change: " + (t.affected.isEmpty() ? "none"
                    : String.join(", ", t.affected)));
        }
    }
    System.out.println("Right the first time: " + right + " of "
            + seen.size());
    int still = 0;
    for (String[] m : seen) {
        if (Router.route(kb, signals, m[0]).route.equals(m[1])) {
            still++;
        }
    }
    System.out.println("All taught messages again: " + still + " of "
            + seen.size() + " right");
    kb.save(KB_FILE);
    System.out.println("Saved shop-kb.json: " + kb.summary());
}

static void stats(KnowledgeBase kb) {
    System.out.println("Credits of every condition:");
    List<List<Object>> rows = new ArrayList<>();
    for (Node n : kb.nodes.values()) {
        for (Condition c : n.conditions) {
            rows.add(List.of(n.id, c.toString(), c.credit));
        }
    }
    for (String line : Report.table(List.of("NodeID", "Condition",
            "Credit"), rows)) {
        System.out.println("  " + line);
    }
    String[] titles = {"Variable Usage Rates:", "Node Usage Rates:",
        "Rule Usage Rates:"};
    List<List<String>> tables = List.of(Report.vurTable(kb, 2),
            Report.nurTable(kb, 2), Report.rurTable(kb, 2));
    for (int i = 0; i < 3; i++) {
        System.out.println(titles[i]);
```

Listing 7.7 (Continued) *ShopRouter.java: the message router of Toko Rina, with its five commands.*

```
            for (String line : tables.get(i)) {
                System.out.println("  " + line);
            }
        }
        Usage usage = kb.usage();
        List<String> important = new ArrayList<>();
        for (String v : usage.importantVariables()) {
            important.add(v + " (NS " + usage.ns(v) + ")");
        }
        System.out.println("Most important variables: "
                + String.join(", ", important));
        System.out.println("Question order: "
                + String.join(", ", usage.questionOrder()));
    }

    static void count(Map<String, Integer> table, String key) {
        table.merge(key, 1, Integer::sum);
    }

    static void showCounts(String title, Map<String, Integer> table) {
        System.out.println(title);
        List<String> keys = new ArrayList<>(table.keySet());
        keys.sort((a, b) -> table.get(a).equals(table.get(b))
                ? a.compareTo(b) : table.get(b) - table.get(a));
        for (String k : keys) {
            System.out.println("  " + ljust(k, 15) + table.get(k));
        }
    }

    static void inbox(KnowledgeBase kb, Signals signals) {
        Map<String, Integer> perRoute = new LinkedHashMap<>();
        Map<String, Integer> perNode = new LinkedHashMap<>();
        List<String> wrong = new ArrayList<>();
        int[] checked = {0, 0};
        List<String> order = kb.usage().questionOrder();
        List<String[]> messages = labelled("inbox.txt");
        for (int i = 0; i < messages.size(); i++) {
            String text = messages.get(i)[0];
            String good = messages.get(i)[1];
            Router.Routed r = Router.route(kb, signals, text);
            checked[0] += r.checked.size();
            Router.Routed other = Router.route(kb, signals, text, order);
            checked[1] += other.checked.size();
            boolean ok = r.route.equals(good);
            if (!ok) {
                wrong.add(String.valueOf(i + 1));
```

Listing 7.7 (Continued) *ShopRouter.java: the message router of Toko Rina, with its five commands.*

```
            }
            System.out.println(rjust(i + 1, 2) + ". " + (ok ? "  " : "X ")
                    + ljust(r.route, 15)
                    + text.substring(0, Math.min(48, text.length())));
            count(perRoute, r.route);
            count(perNode, r.result.paths.isEmpty() ? "(no node)"
                    : r.result.paths.get(0).last().id);
            if (!other.route.equals(r.route)) {
                System.out.println("       (important-first order would say "
                        + other.route + ")");
            }
        }
        int n = messages.size();
        System.out.println("Right: " + (n - wrong.size()) + " of " + n
                + (wrong.isEmpty() ? "" : "; wrong: " + String.join(", ",
                wrong)));
        showCounts("Messages per person:", perRoute);
        showCounts("Messages per node:", perNode);
        System.out.println("Signals checked per message: "
                + Num.fmt((double) checked[0] / n, 2) + " in the engine's "
                + "order, " + Num.fmt((double) checked[1] / n, 2)
                + " important-first");
    }

    static void route(KnowledgeBase kb, Signals signals, String text) {
        Router.Routed r = Router.route(kb, signals, text);
        System.out.println("Message: " + text);
        List<String> seen = new ArrayList<>();
        for (String v : r.checked) {
            seen.add(v + " " + r.result.memory.facts.get(v).value);
        }
        System.out.println("Signals checked: " + String.join(", ", seen));
        System.out.println("Route: " + r.route);
        if (r.route.equals(Router.NOBODY)) {
            System.out.println("Why: no node matched, so Bu Rina reads it "
                    + "herself.");
            return;
        }
        System.out.println("Why:");
        for (String line : r.result.how()) {
            System.out.println("  " + line);
        }
    }
```

Listing 7.7 (Continued) *ShopRouter.java: the message router of Toko Rina, with its five commands.*

```
    static void voice(KnowledgeBase kb, boolean echo) {
        BufferedReader in = new BufferedReader(
                new InputStreamReader(System.in, StandardCharsets.UTF_8));
        List<String> listen = kb.usage().questionOrder().subList(0, 3);
        System.out.println("Voice note or photo? Listen above all for: "
                + String.join(", ", listen));
        InferenceResult result = kb.infer(Facts.none(), q -> {
            System.out.print(kb.variables.get(q.variable).question
                    + " (yes/no) > ");
            String line;
            try {
                line = in.readLine();
            } catch (IOException e) {
                line = null;
            }
            String answer = line == null ? "" : line.strip();
            if (echo) {
                System.out.println(answer);
            }
            return answer.isEmpty() ? null : new Fact(q.variable, answer);
        });
        System.out.println("Route: " + (result.conclusions().isEmpty()
                ? Router.NOBODY : result.conclusions().get(0).value));
    }

    public static void main(String[] args) {
        Signals signals = new Signals(DATA + "/signals.csv");
        String command = args.length > 0 ? args[0] : "inbox";
        if (command.equals("teach")) {
            List<String> files = args.length > 1
                    ? Arrays.asList(args).subList(1, args.length)
                    : List.of("teach-week.txt");
            teach(signals, files);
            return;
        }
        KnowledgeBase kb = Files.exists(Path.of(KB_FILE))
                ? KnowledgeBase.load(KB_FILE) : Router.newKb(DATA);
        if (command.equals("stats")) {
            stats(kb);
        } else if (command.equals("route")) {
            route(kb, signals, String.join(" ",
                    Arrays.asList(args).subList(1, args.length)));
        } else if (command.equals("voice")) {
            voice(kb, Arrays.asList(args).contains("--echo"));
        } else {
            inbox(kb, signals);
```

Listing 7.7 (Continued) *ShopRouter.java: the message router of Toko Rina, with its five commands.*

```
        }
    }
}
```

- `teach [file …]` always starts again from the seed rules, then reads one or more files of labelled messages — the message, the route Bu Rina chose, and optionally the signal that made the difference, separated by `|` — and teaches them one by one, printing what happens. At the end it routes every taught message again, to prove that nothing learned later broke anything learned earlier, and saves `shop-kb.json`. Starting from the seeds each time makes the result exactly reproducible; in daily use you would keep the saved file and teach only new messages.
- `stats` prints the Credit of every condition, the three usage-rate tables of the thesis (VUR, NUR and RUR, smallest first), the most important variables with their NS, and the question order.

- `inbox` routes a file of labelled messages that were *not* taught, marks the wrong ones with `X`, counts the messages per person and per node (the traffic of Section 7.2.6), and compares the number of signals checked in the engine's own order with the number checked when the most important signal is looked for first.
- `route "text"` routes one message and explains the decision with the engine's How explanation.
- `voice` is for messages the detector cannot read, such as voice notes and photos: the helper answers the engine's questions herself, after the app reminds her which signals matter most.

Listing 7.8 is the Python version. Its helper functions `count` and `show_counts` are the same as the Java ones, including the sort order (most messages first, then alphabetically), so that the two versions print the same lines.

Listing 7.8 `shop_router.py`: the same app in Python.

```
"""Toko Rina's message router: sends each WhatsApp message to the right
person and learns from Bu Rina's corrections.

    python shop_router.py teach [file ...]  build the knowledge base again
                                            from the seed rules and teach it
    python shop_router.py stats             Credit, VUR, NUR, RUR, order
    python shop_router.py inbox             route Monday's messages
    python shop_router.py route "text"      route one message, explain it
    python shop_router.py voice [--echo]    a person answers the questions
"""

import os
import sys

from router import NOBODY, new_kb, route, teach
from signals import Signals
from vcirs import KnowledgeBase, num, report

DATA = os.path.join(os.path.dirname(os.path.abspath(__file__)), "..", "data")
KB_FILE = os.path.join(DATA, "shop-kb.json")


def labelled(name):
    """[(message, right route, signal or None)] from a data file."""
    out = []
    with open(os.path.join(DATA, name), encoding="utf-8") as f:
        for line in f:
            if line.strip() and not line.startswith("#"):
                parts = [p.strip() for p in line.split("|")]
                out.append((parts[0], parts[1],
                            parts[2] if len(parts) > 2 else None))
    return out


def cmd_teach(signals, files):
    kb = new_kb(DATA)
    print("Seed rules: " + kb.summary())
    seen, right = [], 0
    for name in files:
        print("Teaching " + name + ":")
        for text, good, because in labelled(name):
            seen.append((text, good))
            routed, fired, node, hit = teach(kb, signals, text, good,
                                             because, "Rina")
            print(str(len(seen)).rjust(2) + ". " + text)
            print("    signals: " + (", ".join(signals.all_found(text))
```

Listing 7.8 (Continued) *shop_router.py: the same app in Python.*

```
                                    or "(none)"))
            by = " (" + fired + ")" if fired else ""
            if node is None:
                right += 1
                print("    routed to " + routed + by + ": right")
                continue
            print("    routed to " + routed + by + ": WRONG, Rina says "
                  + good)
            where = "exception of " + fired if fired else "top level"
            print("    learned " + str(node) + " (" + where + ")")
            print("    earlier cases whose route would change: "
                  + (", ".join(hit) if hit else "none"))
    print("Right the first time: " + str(right) + " of " + str(len(seen)))
    still = sum(1 for text, good in seen
                if route(kb, signals, text)[0] == good)
    print("All taught messages again: " + str(still) + " of "
          + str(len(seen)) + " right")
    kb.save(KB_FILE)
    print("Saved shop-kb.json: " + kb.summary())


def cmd_stats(kb):
    print("Credits of every condition:")
    rows = [[n.id, str(c), c.credit] for n in kb.nodes.values()
            for c in n.conditions]
    for line in report.table(["NodeID", "Condition", "Credit"], rows):
        print("  " + line)
    for title, lines in (("Variable Usage Rates:", report.vur_table(kb, 2)),
                         ("Node Usage Rates:", report.nur_table(kb, 2)),
                         ("Rule Usage Rates:", report.rur_table(kb, 2))):
        print(title)
        for line in lines:
            print("  " + line)
    usage = kb.usage()
    print("Most important variables: "
          + ", ".join(v + " (NS " + str(usage.ns(v)) + ")"
                      for v in usage.important_variables()))
    print("Question order: " + ", ".join(usage.question_order()))


def count(table, key):
    table[key] = table.get(key, 0) + 1
```

Listing 7.8 (Continued) *shop_router.py: the same app in Python.*

```
def show_counts(title, table):
    print(title)
    for k in sorted(table, key=lambda k: (-table[k], k)):
        print("  " + k.ljust(15) + str(table[k]))


def cmd_inbox(kb, signals):
    per_route, per_node, wrong = {}, {}, []
    checked = [0, 0]
    order = kb.usage().question_order()
    messages = labelled("inbox.txt")
    for i, (text, good, _) in enumerate(messages):
        routed, result, asked = route(kb, signals, text)
        checked[0] += len(asked)
        other = route(kb, signals, text, order)
        checked[1] += len(other[2])
        mark = "  " if routed == good else "X "
        if routed != good:
            wrong.append(str(i + 1))
        print(str(i + 1).rjust(2) + ". " + mark + routed.ljust(15)
              + text[:48])
        count(per_route, routed)
        count(per_node, result.paths[0].last().id if result.paths
              else "(no node)")
        if other[0] != routed:
            print("       (important-first order would say " + other[0]
                  + ")")
    n = len(messages)
    print("Right: " + str(n - len(wrong)) + " of " + str(n)
          + ("; wrong: " + ", ".join(wrong) if wrong else ""))
    show_counts("Messages per person:", per_route)
    show_counts("Messages per node:", per_node)
    print("Signals checked per message: " + num.fmt(checked[0] / n, 2)
          + " in the engine's order, " + num.fmt(checked[1] / n, 2)
          + " important-first")
```

Listing 7.8 (Continued) *shop_router.py: the same app in Python.*

```
def cmd_route(kb, signals, text):
    routed, result, checked = route(kb, signals, text)
    print("Message: " + text)
    print("Signals checked: " + ", ".join(
        v + " " + result.memory.facts[v].value for v in checked))
    print("Route: " + routed)
    if routed == NOBODY:
        print("Why: no node matched, so Bu Rina reads it herself.")
        return
    print("Why:")
    for line in result.how():
        print("  " + line)


def cmd_voice(kb, echo):
    def ask(question):
        print(kb.variables[question.variable].question + " (yes/no) > ",
              end="")
        answer = sys.stdin.readline().strip()
        if echo:
            print(answer)
        return answer or None

    listen = kb.usage().question_order()[:3]
    print("Voice note or photo? Listen above all for: " + ", ".join(listen))
    result = kb.infer({}, ask)
    found = result.conclusions()
    print("Route: " + (found[0].value if found else NOBODY))


def main(args):
    signals = Signals(os.path.join(DATA, "signals.csv"))
    command = args[0] if args else "inbox"
    if command == "teach":
        cmd_teach(signals, args[1:] or ["teach-week.txt"])
        return
    kb = KnowledgeBase.load(KB_FILE) if os.path.exists(KB_FILE) \
        else new_kb(DATA)
    if command == "stats":
        cmd_stats(kb)
    elif command == "route":
        cmd_route(kb, signals, " ".join(args[1:]))
    elif command == "voice":
        cmd_voice(kb, "--echo" in args)
    else:
        cmd_inbox(kb, signals)
```

Listing 7.8 (Continued) *shop_router.py: the same app in Python.*

```


if __name__ == "__main__":
    main(sys.argv[1:])
```

To run it, compile once from the `Chapter07/java` folder with `javac -d out -sourcepath ../../vcirs-core/java/src *.java` and then type `java -cp out ShopRouter teach`, or, from `Chapter07/python`, `python shop_router.py teach`. The chapter's README lists every command for Windows, macOS and Linux.

7.7 Bu Rina's First Week

During her first week with the router, Bu Rina writes down fifteen messages as they arrive, each with the route she would choose, in `teach-week.txt`. When the router got one wrong, she also notes the signal that made the difference:

```
# One message per line: message | right route | the signal that made the difference
Bu, beras 5 kg masih ada? Harganya berapa sekarang? | owner
Sudah saya transfer ya bu 85rb, tolong dicek | cashier
Kak sudah tf tapi pesanannya belum sampai dari tadi siang | courier | late
Berapa ongkir ke Perum Griya Asri? | courier | delivery
...
```

`teach` replays her week:

```
Seed rules: Toko Rina message router: 11 variables, 7 nodes, 7 rules
Teaching teach-week.txt:
 1. Bu, beras 5 kg masih ada? Harganya berapa sekarang?
    signals: price, stock
    routed to owner (owner#1): right
 2. Sudah saya transfer ya bu 85rb, tolong dicek
    signals: payment
    routed to cashier (cashier#1): right
 3. Kak sudah tf tapi pesanannya belum sampai dari tadi siang
    signals: payment, late
    routed to cashier (cashier#1): WRONG, Rina says courier
    learned courier#2: late = yes -> route = courier (exception of cashier#1)
    earlier cases whose route would change: none
 4. Berapa ongkir ke Perum Griya Asri?
    signals: price, delivery
    routed to owner (owner#1): WRONG, Rina says courier
    learned courier#3: delivery = yes -> route = courier (exception of owner#1)
    earlier cases whose route would change: none
 5. Minyak goreng yang kemarin bocor bu, gimana ini
    signals: complaint
    routed to owner now (owner now#1): right
 6. Harga telur kok naik terus, kecewa saya
    signals: price, complaint
    routed to owner (owner#1): WRONG, Rina says owner now
    learned owner now#2: complaint = yes -> route = owner now (exception of owner#1)
    earlier cases whose route would change: none
 7. Hi, is the shop open on Sunday?
    signals: hours
    routed to auto reply (auto reply#1): right
 8. Do you deliver to Jalan Melati? How much is the delivery fee?
    signals: price, delivery
    routed to courier (courier#3): right
 9. PROMO GRATIS saldo 500rb!!! klik https://bit.ly/saldo-gratis
    signals: link
    routed to owner reads it: WRONG, Rina says ignore
    learned ignore#1: link = yes -> route = ignore (top level)
    earlier cases whose route would change: none
10. Cek harga minyak murah di sini www.promo-murah.xyz
    signals: price, link
    routed to owner (owner#1): WRONG, Rina says ignore
    learned ignore#2: link = yes -> route = ignore (exception of owner#1)
    earlier cases whose route would change: none
11. Pesan gas 3 kg 2 tabung ya bu, diantar ke rumah
    signals: order, delivery
    routed to owner (owner#3): right
12. Bu salah kirim, saya pesan gula pasir yang datang gula merah
    signals: order, delivery, complaint
    routed to owner (owner#3): WRONG, Rina says owner now
    learned owner now#3: complaint = yes -> route = owner now (exception of owner#3)
    earlier cases whose route would change: none
13. Kak tf nya kelebihan 10rb, dikembalikan atau jadi saldo aja?
    signals: payment
    routed to cashier (cashier#1): right
14. Is the 19 liter gallon available? I want 2
    signals: stock, order
    routed to owner (owner#2): right
15. The eggs I paid for yesterday were broken, 3 of them
    signals: payment, complaint
    routed to cashier (cashier#1): WRONG, Rina says owner now
    learned owner now#4: complaint = yes -> route = owner now (exception of cashier#1)
    earlier cases whose route would change: none
Right the first time: 8 of 15
All taught messages again: 15 of 15 right
Saved shop-kb.json: Toko Rina message router: 11 variables, 14 nodes, 14 rules
```

Read the week as Bu Rina lived it.

Messages 1 and 2 are routed right by the seed rules. The first contains two signals, `price` and `stock`; the price node comes first and answers, and since both go to the owner, it does not matter which.

Message 3 is the first lesson. *Kak sudah tf tapi pesanannya belum sampai* — I transferred, but my order has not arrived — contains `payment` and `late`. The payment rule sends it to Dinda, but the customer is not asking about money; she is asking where her rice is. Bu Rina says the route is the courier, and that the signal that made the difference is `late`. The engine hangs `courier#2: late = yes` under `cashier#1`. The affected-cornerstones check says that no earlier case would change route.

Messages 4, 6 and 10 refine the price rule three times: a delivery fee (`delivery`), a complaint about prices (`complaint`) and a price hidden in a spam link (`link`). Each becomes an exception of `owner#1`. **Message 8** — *Do you deliver to Jalan Melati? How much is the delivery fee?* — then arrives in English and is routed right at once by the exception learned from message 4, in Indonesian. The knowledge is about signals, not about a language.

Message 9 is spam, and no node routes it: `link` has never been seen. The engine adds `ignore#1: link = yes` at the top level. One message later, spam that also mentions a price is caught by the price rule, and becomes the exception `ignore#2` of `owner#1`.

Messages 12 and 15 are complaints hidden behind other signals: a wrong delivery (*salah kirim*, which also contains *pesan* and *kirim*) and broken eggs that were already paid for. Both become exceptions that send the message to the owner at once.

At the end, eight of the fifteen messages were routed right the first time, seven were taught, and — the line that matters most — all fifteen are routed right when tried again. The knowledge base has grown from 7 to 14 nodes.

The saved file also keeps the **history** of the week, one entry per step: seven `add` entries for the seed rules, then eight `accept` entries (option 0) for the messages routed right and seven `add` entries for the corrections — six with option 2, as exceptions, and one with option 5, the spam rule at the top level. Nothing in that history had to be written by hand. It is the diary of the week, and together with the Credits it answers the question every new system raises in its first weeks: is it getting better? Here the answer is visible in the order of the lines. The first corrections come thick and fast; later messages, like the English delivery question, are more and more often answered by what was learned from earlier ones. A shop that keeps teaching for a month will see the share of accepted messages grow week by week, and a sudden jump of corrections will tell it that something new is happening among its customers.

7.8 Reading the Numbers

Now the question of this chapter: what has the router learned, and what matters most in it? `stats` begins with the Credits:

```
Credits of every condition:
  NodeID        Condition        Credit
  ------------  ---------------  ------
  owner#1       price = yes      4
  owner#2       stock = yes      1
  owner#3       order = yes      2
  cashier#1     payment = yes    3
  courier#1     delivery = yes   1
  owner now#1   complaint = yes  1
  auto reply#1  hours = yes      1
  courier#2     late = yes       1
  courier#3     delivery = yes   1
  owner now#2   complaint = yes  1
  ignore#1      link = yes       1
  ignore#2      link = yes       1
  owner now#3   complaint = yes  1
  owner now#4   complaint = yes  1
```

Every new node has Credit 1 on its condition. The seed rules that were corrected have more: `price = yes` has Credit 4 (three exceptions were made under `owner#1`), `payment = yes` has 3 (two exceptions under `cashier#1`) and `order = yes` has 2 (one). The Credits have written, without anyone asking, the history of the week: the price rule was the most corrected rule of Toko Rina, followed by the payment rule. If Bu Rina reviews her knowledge for ten minutes on Sunday, those are the two nodes to read first.

The Variable Usage Rates follow:

```
Variable Usage Rates:
  VariableID  NodeID        VariableOrder  VUR
  ----------  ------------  -------------  ----
  stock       owner#2       1              1.00
  hours       auto reply#1  1              1.00
  late        courier#2     1              1.00
  order       owner#3       1              2.00
  delivery    courier#1     1              2.00
  delivery    courier#3     1              2.00
  link        ignore#1      1              2.00
  link        ignore#2      1              2.00
  payment     cashier#1     1              3.00
  price       owner#1       1              4.00
  complaint   owner now#1   1              4.00
  complaint   owner now#2   1              4.00
  complaint   owner now#3   1              4.00
  complaint   owner now#4   1              4.00
```

Every node here has a single condition, so every Closeness Degree is 1 and each VUR is simply Credit times NS. `complaint` has VUR 4 in each of its four nodes although each Credit is 1, because it is **shared** by four nodes (`owner now#1` at the top level and three exceptions); `price` has VUR 4 in one node because its Credit is 4. The two routes to importance are different — one variable is used in many places, the other is refined in one place — and the VUR puts them on the same scale.

The Node Usage Rates repeat the VURs, since a node with one condition has an NUR equal to its VUR. The Rule Usage Rates are more interesting, because they average along the path of an exception:

```
Rule Usage Rates:
  RuleID        RUR
  ------------  ----
  owner#2       1.00
  auto reply#1  1.00
  owner#3       2.00
  courier#1     2.00
  courier#2     2.00
  ignore#1      2.00
  cashier#1     3.00
  courier#3     3.00
  ignore#2      3.00
  owner now#3   3.00
  owner now#4   3.50
  owner#1       4.00
  owner now#1   4.00
  owner now#2   4.00
Most important variables: complaint (NS 4)
Question order: price, complaint, payment, order, delivery, link, stock, hours, late
```

`owner now#4`, the exception *complaint* under the payment rule, has RUR 3.5: the average of `cashier#1` (NUR 3) and itself (NUR 4). `courier#2`, the exception *late* under the same payment rule, has only 2: `late` is used in one node, with Credit 1. The last two lines are the verdict of variable analysis and value analysis. `complaint` is the most important variable, shared by four nodes. The question order starts with `price` and `complaint` (both with VUR 4), then `payment`, `order` and `delivery`, and ends with the signals that are used least.

EXAMPLE. Reading `stats` in five steps. (1) Find the conditions with the largest Credit: those nodes were corrected most, so read them first. (2) Find the variables with the largest NS in the last lines: they are shared by many nodes, so their word lists matter most. (3) In the VUR table, look for a variable whose VUR jumped since last time: something new is happening around it. (4) In the RUR table, the largest rates belong to the busiest lines of reasoning, usually an exception under a much-refined rule. (5) Compare with the traffic counts of `inbox` before deciding anything: a much-used rule that routes few messages is subtle, not broken.

PITFALL. “Most important” in VCIRS means “most used by the knowledge”, not “most urgent for the business”. Complaints happen to be both in Toko Rina, but a signal can be rare in the knowledge and still matter a great deal — a customer who writes that the LPG cylinder smells of gas needs attention at once, even if no rule mentions *gas* yet. Use the numbers to decide where to look, and your own judgment to decide what matters.

7.9 Monday Morning

On Monday the router works alone for the first time, while Bu Rina serves customers at the counter. Later she checks its routes against the ones she would have chosen; the file `inbox.txt` has the 25 messages of the morning with her routes. `inbox` routes them all:

```
 1.   owner          Bu, gula pasir 1 kg harganya berapa?
 2.   cashier        Sudah tf 120rb ya kak, buktinya saya kirim
 3.   owner          Pesan beras 10 kg sama minyak 2 liter
 4. X owner          Pesanan saya belum sampai bu, sudah 2 jam
 5.   auto reply     Is the shop open today? It's a public holiday
 6.   owner          How much is a 12 kg LPG refill?
 7.   cashier        Kak, bisa bayar pakai QRIS?
 8.   owner now      Telurnya ada yang pecah 2, gimana bu?
 9.   courier        Ongkir ke Blok C berapa ya?
10.   owner          Bu, air galon masih ada? Mau ambil sore
11.   owner reads it Assalamualaikum bu, terima kasih ya kemarin
12.   ignore         Selamat! Anda menang undian, klik www.hadiah-und
13.   owner          Kak, pesen rokok 1 slop, anter ke pos ronda ya
14.   owner now      Minyaknya bocor di plastik, tolong diganti
15.   owner          Do you have Indomie goreng in stock? I need 1 bo
16.   cashier        Bu, sudah saya bayar lunas yang kemarin
17.   owner now      Harga beras naik lagi? Kecewa bu
18. X owner          Jam berapa tutup malam ini?
19.   courier        Kurirnya sudah jalan belum?
20.   owner          Bu tolong dicatat, besok saya ambil pesanan
21.   owner now      The sugar I paid for is missing from the bag
22.   owner          Ready sampo sachet renceng?
23.   cashier        Mbak, transferan saya sudah masuk?
24.   courier        Bisa kirim sekarang? Rumah saya dekat masjid
25. X owner          Bu Rina, anak saya titip uang 50rb buat bayar ut
       (important-first order would say cashier)
Right: 22 of 25; wrong: 4, 18, 25
Messages per person:
  owner          11
  cashier        4
  owner now      4
  courier        3
  auto reply     1
  ignore         1
  owner reads it 1
Messages per node:
  owner#3        5
  cashier#1      4
  owner#1        3
  owner#2        3
  courier#1      2
  owner now#1    2
  (no node)      1
  auto reply#1   1
  courier#3      1
  ignore#1       1
  owner now#2    1
  owner now#4    1
Signals checked per message: 4.72 in the engine's order, 4.28 important-first
```

Twenty-two of twenty-five are right, and the three wrong ones are instructive.

Message 4, *Pesanan saya belum sampai bu, sudah 2 jam*, contains `order` (*pesanan*) and `late`. The order rule `owner#3` answers, and it has no exception for `late` — the exception learned from message 3 of last week lives under the *payment* rule, in the context where that mistake happened. RDR is doing what it promises: knowledge learned in one context does not leak into another. Bu Rina has to teach this context too.

Message 18, *Jam berapa tutup malam ini?* — what time do you close tonight? — contains *berapa*, a price word, and *tutup*, an hours word. The price rule comes first and wins. Here the detector is not wrong: *berapa* really does appear. The knowledge needs an exception.

Message 25, *Bu Rina, anak saya titip uang 50rb buat bayar utang* — my child is bringing 50 thousand to pay my debt — contains `order` (*titip*, which usually means "please set aside for me") and `payment` (*bayar*). The order rule wins; the cashier should have it.

The line under message 25 is the value-analysis experiment. `inbox` routes every message twice: once in the engine's order, and once after checking the signals in the **question order** until one is found — here `price`, then `complaint`, then `payment`, which finds *bayar* and starts the engine from the payment rule. Only for message 25 do the two disagree, and in this case the important-first order happens to be right. That tells Bu Rina something useful: the message matches **two top-level rules** with different routes, and only the order of the nodes decides between them. Whenever the two orders disagree, the knowledge base has two rules competing for one message, and an exception should settle it. Across the morning the important-first order checks 4.28 signals per message instead of 4.72. Why does the engine need almost five checks for a message that has only one or two signals? Because it walks the top-level nodes in the order Bu Rina taught them, and each node that fails costs one check: a message about opening hours is asked about price, stock, order, payment, delivery and complaint before the hours rule is reached. For the detector that costs nothing. For a person answering questions it would be tedious, which is why the voice mode of Section 7.10 reminds her of the important signals first.

Should the router therefore always check the important signals first? Not blindly. The order in which an RDR knowledge base is searched is part of the knowledge the expert taught — Chapter 3 showed what happens when the order of rules is changed behind the expert's back. The router keeps the engine's order for its decisions, and uses the important-first order as a **check** that points at competing rules, and as a guideline for people, in the voice mode of Section 7.10.

The traffic counts answer different questions. Bu Rina herself received 11 of the 25 messages, Dinda 4, Pak Darto 3; `owner#3`, the order rule, routed 5 messages, more than any other node, although its Credit is only 2 — a simple rule that works. Only one message reached no node at all: *Assalamualaikum bu, terima kasih ya kemarin*, a greeting and a thank-you, which is exactly the kind of message Bu Rina wants to read herself.

The `route` command explains one decision. For a message that combines a payment and a complaint:

```
Message: Bu, sudah tf ya, tapi telurnya pecah 2
Signals checked: price no, stock no, order no, payment yes, late no, complaint yes
Route: owner now
Why:
  owner now#4 fired because:
    payment = yes  (answered by the user, CF 100)
    complaint = yes  (answered by the user, CF 100)
    so route = owner now  [rule CF 100, conclusion CF 100]
```

The detector checked six signals, one per node the engine tried: price, stock and order failed at once; payment fired the payment rule; its first exception, `late`, did not fire; its second, `complaint`, did. The explanation calls the detector's answers "answered by the user", because to the engine the detector *is* the user who answers its questions. And for the thank-you message:

```
Message: Assalamualaikum bu, terima kasih ya kemarin
Signals checked: price no, stock no, order no, payment no, delivery no, complaint no, hours
no, link no
Route: owner reads it
Why: no node matched, so Bu Rina reads it herself.
```

7.10 Making It Better

7.10.1 Teaching Monday's three mistakes

Bu Rina writes Monday's three mistakes in `teach-week2.txt`, with the signal that made the difference for each, and teaches the two weeks together with `teach teach-week.txt teach-week2.txt`. The last lines of the output show the new knowledge:

```
Teaching teach-week2.txt:
16. Pesanan saya belum sampai bu, sudah 2 jam
    signals: order, late
    routed to owner (owner#3): WRONG, Rina says courier
    learned courier#4: late = yes -> route = courier (exception of owner#3)
    earlier cases whose route would change: none
17. Jam berapa tutup malam ini?
    signals: price, hours
    routed to owner (owner#1): WRONG, Rina says auto reply
    learned auto reply#2: hours = yes -> route = auto reply (exception of owner#1)
    earlier cases whose route would change: none
18. Bu Rina, anak saya titip uang 50rb buat bayar utang
    signals: order, payment
    routed to owner (owner#3): WRONG, Rina says cashier
    learned cashier#2: payment = yes -> route = cashier (exception of owner#3)
    earlier cases whose route would change: none
Right the first time: 8 of 18
All taught messages again: 18 of 18 right
Saved shop-kb.json: Toko Rina message router: 11 variables, 17 nodes, 17 rules
```

Each correction became an exception in the right context: *late* under the order rule, *hours* under the price rule, *payment* under the order rule. The line *All taught messages again: 18 of 18 right* is the regression check: the three new nodes did not change the route of any earlier message, including *Pesan gas 3 kg 2 tabung ya bu, diantar ke rumah* (order and delivery), which still goes to the owner, because the new exception of the order rule needs `late`, not `delivery`. Monday's inbox now reads *Right: 25 of 25* — as it should, since three of its messages were just taught; the real test is next Monday's messages.

The numbers have moved. The Credits and the end of the usage tables now say:

```
Credits of every condition:
  NodeID        Condition        Credit
  ------------  ---------------  ------
  owner#1       price = yes      5
  owner#2       stock = yes      1
  owner#3       order = yes      4
  cashier#1     payment = yes    3
  courier#1     delivery = yes   1
  owner now#1   complaint = yes  1
  auto reply#1  hours = yes      1
  courier#2     late = yes       1
  courier#3     delivery = yes   1
  owner now#2   complaint = yes  1
  ignore#1      link = yes       1
  ignore#2      link = yes       1
  owner now#3   complaint = yes  1
  owner now#4   complaint = yes  1
  courier#4     late = yes       1
  auto reply#2  hours = yes      1
  cashier#2     payment = yes    1
```

and at the very end:

```
Most important variables: complaint (NS 4)
Question order: payment, price, order, complaint, delivery, hours, late, link, stock
```

`payment` has become the variable with the largest VUR, 6 in `cashier#1`: its Credit is still 3, but `payment` is now shared by two nodes, the payment rule and the new exception `cashier#2`, so its weight doubled. `order = yes` went from Credit 2 to 4. **Figure 7.2** compares the largest VUR of every signal after the first week and after Monday's corrections.

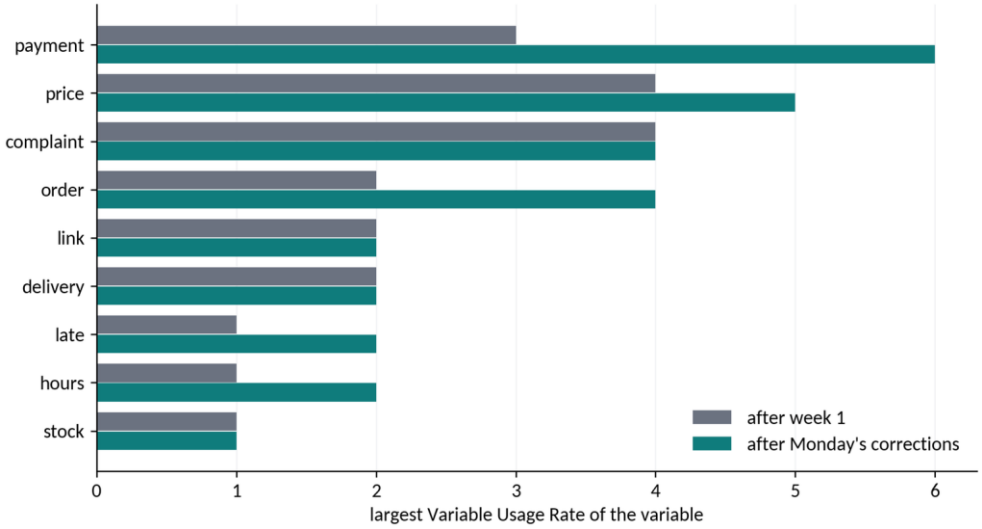

Figure 7.2 The largest Variable Usage Rate of each signal after the first week (grey) and after Monday's three corrections (teal). Payment, order, late and hours gained; the others did not change. The signal `proof` has no bar because no node uses it yet.

7.10.2 The voice mode

When a customer sends a voice note, the detector has nothing to read. The helper — say Dinda's cousin, who minds the phone on Saturday — listens, and the router asks her the questions it needs. Before the first question it reminds her of the three signals that matter most, taken from the question order:

```
Voice note or photo? Listen above all for: payment, price, order
Does the message ask about a price? (yes/no) > no
Does it ask whether something is in stock? (yes/no) > no
Does the customer want to order or buy? (yes/no) > no
Is it about a payment or a transfer? (yes/no) > yes
Does the customer say an order has not arrived? (yes/no) > no
Is the customer complaining? (yes/no) > yes
Route: owner now
```

The questions come in the engine's order, the same order that routes typed messages, so a voice note gets exactly the route a typed message with the same signals would get. The reminder is value analysis used as the thesis intended, as a **guideline for the user**: listen above all for payment, price and order, because the knowledge relies on them most.

7.10.3 Maintaining the word lists

Most wrong routes in a real shop come from one of two sources, and it pays to tell them apart. Either the **detector** missed or misread a word — *kurirnya* was not in the list, *berapa* appeared in a question about the time — or the **knowledge** gave the wrong route for signals that were read correctly. The first is fixed in `signals.csv`, the second by teaching. The `route` command shows

which it is: if the signals checked are wrong, fix the words; if they are right and the route is wrong, teach.

The usage rates help here too. A signal with a high VUR is one the knowledge leans on in many places, so a missing word in its list misroutes many kinds of messages. In Toko Rina that means the lists for `payment`, `price` and `complaint` deserve a careful look every few weeks, with the family's own chat habits in mind (*tf*, *trf*, *transfer-an*, *sdh bayar*).

7.10.4 Tests

`RouterTest.java` and `test_router.py` check the detector (whole words, phrases, the order of the signals found), the first week (14 nodes, Credit 4 for `price = yes`, `complaint` as the most important variable, two routes) and the second (17 nodes, an old answer kept, a late order routed to the courier, `payment` first in the question order). They end with:

```
all tests passed
```

7.11 Using It for Real

Your own shop. Nothing in `ShopRouter` is about groceries except three data files. Write your signals and their words in `signals.csv`, your people or piles as routes, and three to ten first rules in `seed-cases.json`. A phone-credit counter might route *pulsa*, *paket data* and *token listrik* messages differently; a laundry might have `pickup`, `ready`, `price` and `complaint`; a small bakery `order`, `custom cake`, `pickup time` and `delivery`. Then run the router beside you for a week, and write down each wrong route with the right one and the signal that made the difference.

The same numbers, another shop. Imagine Mbak Sari's laundry after a month. Her `stats` shows that `ready` (*sudah jadi?*, *is it done?*) has Credit 5 in its rule: she corrected it five times, each time for a message that also mentioned something else — an express order, a missing shirt, a pickup by a neighbor. The rule is the busiest place in her knowledge, and reading its five exceptions together tells her something about her business: customers who ask whether their laundry is ready often have a second, more important question. She decides to answer *ready* messages herself for a while. Meanwhile `price` has Credit 1 and routes a quarter of all messages: a simple rule that works, which she can leave alone.

From the phone to the program. The router reads text. The simplest way to feed it is to copy messages from WhatsApp into a text file at the end of the day, or to export a chat (WhatsApp can export a chat as a text file) and let a small script take out the messages. Use the routes to set the labels of WhatsApp Business by hand [2]. Connecting a program directly to the WhatsApp Business Platform is possible, but it needs a registered business account and a server, and it is not needed to get the benefit of this chapter: the knowledge and its statistics are the same.

Weekly ten minutes. Once a week, run `stats` and `inbox` on a few days of messages. Read the nodes with the highest Credit first: they are where your customers keep surprising the rules. Look at any message where the important-first order disagrees with the engine: two rules compete there. Look at the traffic: a node that never routes anything for a month may describe a situation that no longer happens.

Keep the messages, not the people. The cases in `shop-kb.json` hold only signals — `payment = yes`, `late = yes` — never a customer's name, number or message. Keep it that way. The messages themselves are personal data of your customers and fall under Indonesia's personal data protection law [8]; keep the teaching files on the shop's own computer, delete old ones you no

longer need, and never put phone numbers into a knowledge base that you might share with another shop.

Growing further. When the knowledge base has a few dozen nodes, Chapter 6's rule generation can propose general rules from the most important variables, and Chapter 10 shows how to put a knowledge base like the router's on a small web page that the whole family can use from their phones.

7.12 Common Mistakes, and How to Avoid Them

Reading Credit as traffic. A rule with Credit 1 may route half of all messages. *Avoid it:* read Credit and usage rates as "how much the knowledge was refined and shared", and use the traffic counts for "how many messages".

Reading "most important" as "most urgent". The usage rates know nothing about the business. *Avoid it:* decide urgency yourself; use the numbers to decide where to look first.

Fixing a detector problem with a rule. If *kurirnya* is not recognized, teaching a node will not help, because the signal never becomes `yes`. *Avoid it:* look at the signals checked in `route` first; fix the word list when a word was missed.

Teaching without naming the difference. Without a named signal, the new node takes every signal in section B, and may become too narrow to fire again. *Avoid it:* name the one signal that explains the different route whenever you know it.

Reordering the seed rules to "fix" a mistake. It changes the routes of messages you never looked at, as the flat list of Chapter 3 did. *Avoid it:* teach an exception in the context of the mistake.

Expecting knowledge to travel between contexts. The *late* exception under the payment rule does not help a late message routed by the order rule. *Avoid it:* when the same mistake appears under another rule, teach it there too; after a while, Chapter 6's rule generation can suggest a general rule.

Words that belong to two signals. *Berapa* is a price word and part of *jam berapa*. *Avoid it:* either add the phrase to the more specific signal and teach the exception, as Bu Rina did, or remove the ambiguous word and list longer phrases instead.

Comparing usage rates between shops. A VUR of 6 in one knowledge base means nothing in another; the numbers depend on how many nodes each has. *Avoid it:* compare nodes and variables within one knowledge base, and watch how they change over time.

Storing personal data in cases. *Avoid it:* keep only signals in the knowledge base.

7.13 Chapter Summary

- A small shop can route most WhatsApp messages by **signals** — kinds of words in Indonesian and English — without understanding the language; the detector spots whole words and phrases and answers the engine's questions only when asked (**Figure 7.1**, **Listing 7.3**, **Listing 7.4**).
- Routing is classification; seven over-general seed rules plus exceptions taught message by message gave a router that sent all 25 messages of a morning to the right person after two short teaching sessions (**Listing 7.5**, **Listing 7.6**, **Listing 7.7**, **Listing 7.8**).

- The **Credit** of a condition grows when an expert creates a node under its node for a case that agrees with it; it records where knowledge was refined, not how many messages passed.
- **VUR** is Credit times Weight, Weight is NS times the Closeness Degree VO/TV; **NUR** averages VURs over a node and **RUR** averages NURs along a rule (Equation (7.1), Equation (7.2), Equation (7.3), Equation (7.4), Equation (7.5)); the rates computed by hand equal the engine's (**Listing 7.1**, **Listing 7.2**).
- Variable analysis gives the **most important variables** (the most shared) and value analysis the **question order**; in Toko Rina they pointed to the most-refined rules, to the word lists that deserve most care, and — where the important-first order disagreed with the engine's — to rules competing for the same message (**Figure 7.2**).
- Traffic counts (messages per person, per node) are a second, different kind of statistic, and a shop needs both.

7.14 Exercises

7.1 A node has three conditions: `payment = yes` (Credit 2), `late = yes` (Credit 1) and `complaint = yes` (Credit 1). In the knowledge base, `payment` is used by 2 nodes, `late` by 3 and `complaint` by 4. Compute the three VURs and the node's NUR by hand.

7.2 Using the Credits printed in Section 7.8, explain in one sentence each why `owner#1` has Credit 4 and why `owner#2` (the stock rule) still has Credit 1, although the stock rule routed three Monday messages.

7.3 In the output of Section 7.8, `courier#3` has RUR 3 and `courier#2` has RUR 2, although both are exceptions with one condition and Credit 1. Explain the difference.

7.4 After Monday's corrections, `route` gives this for a new message:

```
Message: Kak, harga telur berapa? Sekalian dianter ya
Signals checked: price yes, delivery yes
Route: courier
Why:
  courier#3 fired because:
    price = yes  (answered by the user, CF 100)
    delivery = yes  (answered by the user, CF 100)
    so route = courier  [rule CF 100, conclusion CF 100]
```

Is the route right for Toko Rina? If Bu Rina says the owner should get it, can she teach it with the signals the detector knows? What would you add?

7.5 A student in the estate writes *The rice has not arrived and it is already 8 pm*. The router says:

```
Message: The rice has not arrived and it is already 8 pm
Signals checked: price no, stock no, order no, payment no, delivery no, complaint no, hours
no, link no
Route: owner reads it
Why: no node matched, so Bu Rina reads it herself.
```

Why does `late` never get checked, although *not arrived* is in its word list? Write the line you would add to a teaching file to fix it, and say where the new node would go.

7.6 Why does `teach` record a message that was routed right with option 0 instead of doing nothing? Name one question about the knowledge base that those records let you answer later.

7.7 Bu Rina's cousin opens a phone-credit counter. Design five signals with at least three Indonesian and two English keywords each, three routes, and four seed rules. Which of your signals do you expect to become the most important variable after a month, and why?

7.8 Explain, with the example of message 25 of Section 7.9, why a disagreement between the engine's order and the important-first order is useful information, and why the router still decides in the engine's order.

7.15 Solutions to the Exercises

Solution 7.1. The node has $TV = 3$, so the closeness degrees are $1/3$, $2/3$ and $3/3$. `payment`: $2 \times 2 \times 1/3 = 4/3 \approx 1.33$. `late`: $1 \times 3 \times 2/3 = 2$. `complaint`: $1 \times 4 \times 1 = 4$. The NUR is $(4/3 + 2 + 4)/3 = 22/9 \approx 2.44$. Notice how the last condition, closest to the conclusion, weighs most.

Solution 7.2. `owner#1` started with Credit 1 and three exceptions were created under it (for messages 4, 6 and 10 of the first week), each for a message that mentioned a price, so it gained three Credits. `owner#2` routed messages correctly but was never *corrected*: no node was ever created under it, so its Credit did not change. Credit counts refinement, not traffic.

Solution 7.3. RUR averages the NURs of all nodes on the rule's path. `courier#3` is an exception of `owner#1`, whose NUR is 4, and its own NUR is 2 (Credit 1, `delivery` shared by two nodes), so its RUR is $(4 + 2)/2 = 3$. `courier#2` is an exception of `cashier#1`, whose NUR is 3, and its own NUR is 1 (`late` used once), so its RUR is $(3 + 1)/2 = 2$.

Solution 7.4. The customer asks a price *and* asks for delivery: she is ordering eggs to be delivered. In Toko Rina an order goes to the owner, who prepares it and sends Pak Darto, so the route is wrong. But the message has exactly the signals of the cornerstone case of `courier#3` (*Berapa ongkir…?*: `price` and `delivery`), so no difference exists between them, and the engine refuses to learn — the list of difference is empty. A new signal is needed that tells an order apart from a question about the fee, for example `order` keywords such as *sekalian* ("while you are at it") or *mau* ("I want"), or a new signal `product` that recognizes the names of goods (*telur*, *beras*, *gas*). With `order` detected, Bu Rina can teach the exception `order = yes → owner` under `courier#3`.

Solution 7.5. `late` appears only in exceptions: `courier#2` under the payment rule and `courier#4` under the order rule. The message contains neither a payment nor an order word, so neither parent fires and their exceptions are never tried; no top-level node uses `late`. Add to a teaching file the line `The rice has not arrived and it is already 8 pm | courier | late`. No node routes the message, so `teach` adds `courier#5: late = yes → route = courier` at the top level (option 5), where every message can reach it.

Solution 7.6. An accepted message is a verification: a real message confirmed that the node's route was right. The records let you answer questions such as "how often was each node confirmed this month?", "has this node ever been confirmed since it was taught?" and "is the share of accepted messages growing?", which is the everyday measure of whether the router is getting better, as Chapter 3 described for the Wi-Fi doctor.

Solution 7.7. One possible design: signals `pulsa` (*pulsa*, *isi*, *top up*, *credit*, *reload*), `data` (*paket*, *kuota*, *data*, *internet*, *gb*), `token` (*token*, *listrik*, *pln*, *meter*, *electricity*), `payment` (*tf*, *transfer*, *bayar*, *paid*, *qris*) and `failed` (*belum masuk*, *gagal*, *pending*, *not received*, *failed*); routes `sell`, `cashier` and `check the provider`; seed rules `pulsa → sell`, `data → sell`, `token → sell`, `payment → cashier`. The signal `failed` will probably become the most important variable: it will appear as an

exception under every selling rule (a top-up that did not arrive needs checking, whatever was bought), so it will be shared by several nodes.

Solution 7.8. Message 25 contains `order` and `payment`, so it matches the order rule and the payment rule, two top-level rules with different routes. In the engine's order the order rule comes first and wins; in the important-first order, `payment` is found first and the payment rule wins. The disagreement proves that two rules compete and that only the order of the nodes decides, which is exactly the situation where the expert should teach an exception. The router still decides in the engine's order because that order is part of what Bu Rina taught: her earlier corrections were made under the nodes that the engine's order reached, and deciding in another order would bypass them for messages nobody has checked.

References

[1] Campaign Brief Asia, "Digital 2026: Indonesia reveals social media user identities increased 26% to 180 million," Nov. 6, 2025. [Online]. Available: https://campaignbriefasia.com/2025/11/06/digital-2026-indonesia-reveals-social-media-user-identities-increased-26-to-180-million/

[2] WhatsApp, "Staying organized as a small business using lists and quick replies on WhatsApp," WhatsApp Business resource library. [Online]. Available: https://business.whatsapp.com/resources/resource-library/staying-organized-smb-using-lists-quick-replies (accessed Sep. 2026).

[3] M. M. I. Subakti, "A Variable-Centered Intelligent Rule System," M.Sc.Eng. thesis, Dept. of Computer Science and Information Engineering, Taipei, Taiwan, Jun. 2005.

[4] I. Subakti, "A variable-centered intelligent rule system," in *Proc. 1st Annual Int. Conf. Information and Communication Technology Seminar (ICTS 2005)*, vol. 1, no. 1, Surabaya, Indonesia, Aug. 2005, pp. 167–174.

[5] P. Compton and B. H. Kang, *Ripple-Down Rules: The Alternative to Machine Learning*. Boca Raton, FL, USA: CRC Press, 2021.

[6] I. Subakti, *Variable-Centered Intelligent Rule System*, Indonesian ed. (resume). Surabaya, Indonesia: Dept. of Informatics, Institut Teknologi Sepuluh Nopember, 2006.

[7] I. Subakti, "Some revisions in VCIRS and cases reconstructing perspectives," in *Proc. Information and Communication Technology Seminar (ICTS 2006)*, Surabaya, Indonesia, 2006.

[8] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. Jakarta, Indonesia, Oct. 17, 2022. [Online]. Available: https://peraturan.bpk.go.id/Details/229798/uu-no-27-tahun-2022

Chapter 8

Motorcycle Fault Finder for Ojek Drivers: Forward and Backward Inference with Explanations

By the end of this chapter you will be able to explain the two directions in which a rule-based system can reason — **forward**, from what you see to what follows, and **backward**, from a suspicion to the questions that confirm or reject it; build both from scratch in a few dozen lines of Java and Python; turn a VCIRS knowledge base built case by case into ordinary IF-THEN rules; and run the **fault finder**, an app for ojek drivers that first rules out danger, then looks for the cause of a motorcycle problem with as few questions as possible, and answers the three questions every user of an expert system is entitled to ask: *what* did you conclude, *how* did you reach it, and *why* are you asking me this. The app knows the popular scooters on Indonesian roads today and gives safe advice: what the driver can do himself, what needs a bengkel today, and when to stop riding at once.

8.1 Mas Yudi's Morning

Mas Yudi drives an online ojek in Surabaya. His working tool is a Honda BeAT, like a great many of his colleagues' — it is cheap to run, light in traffic and easy to repair. Indonesians bought 6,412,769 new motorcycles in 2025, and 91.7 percent of them were automatic scooters [1]; the BeAT family has been the best-selling model for years, followed by scooters such as the Vario, the Scoopy and the PCX [2]. When ojek drivers compare motorcycles for their work in 2026, the same names come back: the BeAT for its fuel economy, the Vario 125 for its storage space, the Yamaha Gear 125 for its hooks and flat deck, the Scoopy for its comfort [3].

For a driver like Mas Yudi, a motorcycle that will not start is a morning without income. At six o'clock, in front of his boarding house, it does not start. Is it the battery? The fuel? Something expensive? He could push it to a bengkel and wait, or he could call Cak Man, the mechanic who has a small workshop next to the drivers' base camp and who has been fixing their scooters for twenty years. Cak Man always asks the same few questions, in an order that seems obvious once you have heard it: *Is the horn loud or weak? What does the starter do when you press it? Does the fuel gauge show anything?* After three or four answers he knows what is wrong, and he knows whether Mas Yudi can ride to him, should kick-start it and come today, or must not ride it at all.

Cak Man cannot answer every driver's phone call. He is one man with two hands, and at seven in the morning, when half the base camp's motorcycles have been parked in the rain all night, his phone rings without pause. Many of those calls end the same way: *it is the battery, kick-start it and come this afternoon*, or *you are out of petrol, Mas*. Those are the calls a program could answer, leaving Cak Man free for the scooters that really need his hands. This chapter puts his questions and his knowledge into a program that any driver can run on a phone or a laptop at the base camp. It is a small expert system in the classic sense: it reasons with IF-THEN rules, it can reason in both directions, and it explains itself. Unlike the classic systems of the 1980s, its rules were not

written by a programmer. They were built by VCIRS from Cak Man's cases, and they come with the safety rules that he insists on.

8.2 The Idea in Plain Words

8.2.1 Two directions of reasoning

There are two natural ways to use a set of rules, and both are as old as expert systems [4], [5].

Forward chaining starts from the facts. Mas Yudi tells the program everything he sees — the engine does not start, the horn is weak, the fuel gauge shows fuel — and the program fires every rule whose conditions are all true, adds the conclusions to what it knows, and repeats until nothing new follows. It is *data-driven*: it answers "what follows from all this?" It is the natural way to reason when the facts come first, for example when a form has already been filled in.

Backward chaining starts from a goal. Cak Man suspects the battery. To prove *cause = weak battery*, the program looks for the rules that conclude it; to use such a rule, it must prove the rule's conditions, each of which becomes a smaller goal (a *subgoal*); a condition that no rule concludes is asked of the user. It is *goal-driven*: it answers "is it this?" and it asks only the questions that the goal needs. It is the natural way to reason when asking is expensive — and for a driver standing next to his motorcycle in the rain, every question is expensive.

Figure 8.1 shows both on the same two rules of the fault finder. On the left, facts climb upward through the rules to a conclusion. On the right, a goal goes down through a rule to a subgoal and to two questions.

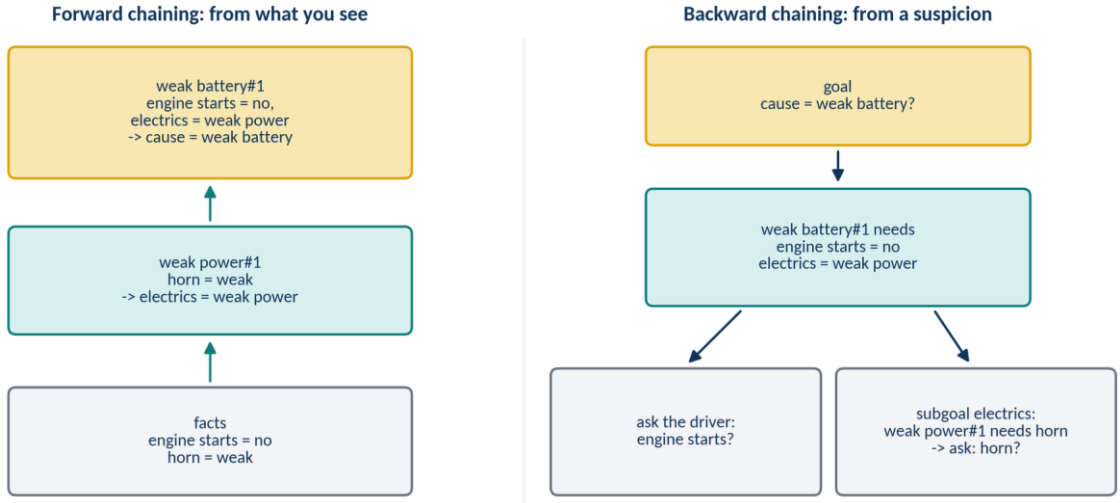

Figure 8.1 The same two rules used in both directions. Forward chaining (left) climbs from the facts through the rules to a conclusion; backward chaining (right) starts from a goal, turns the rule's conditions into subgoals and asks the driver only what no rule can work out.

8.2.2 Rules that feed rules

Look again at the two rules of the figure. The first does not name a cause at all. It says that a weak horn means the **electrics** have weak power. The second uses that conclusion as one of its conditions: if the engine does not start and the electrics have weak power, the cause is a weak battery. A conclusion that is used as a condition by another rule is an **intermediate conclusion**, and it is what makes chaining more than a lookup table.

Intermediate conclusions are how mechanics think. Cak Man first decides *which system* is at fault — the electrics, the fuel, the transmission, the cooling — and only then *which part*. A weak horn and a slow starter both point to weak electrical power; he does not need a separate rule for every combination of symptoms and parts. The fault finder uses four intermediate variables: `electrics` (no power or weak power), `transmission` (a CVT problem, the continuously variable transmission of a scooter, with its V-belt and clutch), and `cooling` (air-cooled or liquid-cooled, which follows from the model of the motorcycle). The fourth is `safety`, which is used as a goal on its own.

8.2.3 What, How and Why

An expert who gives advice without reasons is hard to trust, and a program even more so. From the first expert systems onward, a rule-based system has been expected to answer three questions during and after inference [4], [6]:

- **What** is the result? The conclusions reached, final and intermediate.
- **How** was it reached? The rules that fired, and the facts that made each of them fire, including facts that were themselves concluded by other rules.
- **Why** is this question being asked? The rule the program is trying to use, the condition it needs, and the chain of rules from the goal down to this question.

VCIRS keeps an event log during inference from which all three answers are read [4]; the vcirs-core engine offers them as `what()`, `how()` and `Question.why()`. The health self-diagnosis system built on VCIRS used the same backward chaining with explanations for patients describing their symptoms at home [7]; the fault finder does the same for motorcycles. For a driver, *why* is the most useful of the three: when the program asks “How many km since the V-belt was changed?”, he may not know, and knowing *why* it asks tells him whether it is worth finding out. Imagine the conversation with a person instead. The mechanic asks about the belt; the driver asks “why do you want to know?”; the mechanic answers “because if it is over twenty-four thousand, I think your belt is worn, and a worn belt can snap”. That answer has three parts — the rule being tried, the condition it needs, and what follows if the condition holds — and those are exactly the three lines of `Question.why()`. When the question is deep inside a chain, the program adds a fourth: the chain of rules from the goal down to the question, which is the mechanic saying “I am checking the cooling because I think it is overheating”.

8.2.4 How sure is the answer?

Not every rule is equally certain. That a weak horn means weak electrical power is almost always true; that a shudder when pulling away means a dirty CVT clutch is usually true but not always. Each node of a VCIRS knowledge base has a **certainty factor** (CF), a whole number from 0 to 100 given by the expert, and each condition has one too. When the knowledge base is turned into rules, the CF of a rule is [4]:

$$CF_{\text{rule}} = \frac{\min_i CF_{\text{clause } i} \times CF_{\text{node}}}{100} \tag{8.1}$$

When the rule fires, the certainty of its conclusion combines the certainty of the facts it used with the certainty of the rule, in the classical way of rule-based systems [8]:

$$CF_{\text{conclusion}} = \frac{\min_i CF_{\text{fact } i} \times CF_{\text{rule}}}{100} \tag{8.2}$$

The weakest link decides. A driver's answer counts as CF 100. The rule *horn = weak → electrics = weak power* has CF 90, so by Equation (8.2) the intermediate conclusion has CF $100 \times 90/100 = 90$. The rule *engine starts = no, electrics = weak power → cause = weak battery* has CF 100 by Equation (8.1) (all its clauses and its node have CF 100), but one of its facts, the intermediate conclusion, has CF 90, so the weak battery is concluded with CF $\min(100,90) \times 100/100 = 90$. A chain of reasoning is never more certain than its least certain step.

8.2.5 From cases to rules

In Chapter 3 the knowledge of the Wi-Fi doctor was a tree of nodes and exceptions, used by walking the tree. Chaining needs something else: a flat set of IF-THEN rules in which any conclusion can serve as a condition of another rule. VCIRS bridges the two with **knowledge base transformation** [4], [6]: every node becomes one rule, with the node's conditions as its IF part and its conclusions as its THEN part; and every path of two or more nodes (a node with its exceptions) becomes one more rule that combines the conditions along the path. The expert keeps teaching cases, the RDR way; the engine keeps the rules, the RBS way. This is exactly the combination VCIRS was designed for: the easy knowledge building of Ripple Down Rules and the strong inference of rule-based systems [4], [9].

Cak Man teaches the fault finder whole cases — "when the horn is weak, the electrics have weak power", "when the engine does not start and the electrics are weak, it is the battery" — and each becomes a top-level node, so each node becomes exactly one rule. `vcirs-core` does the transformation with `transform()`, and the chaining engines use its rules.

8.2.6 Advice, not repair

A fault finder for motorcycles must be safe before it is clever. It should never suggest a repair that can hurt someone, and it should say clearly when the motorcycle must not be ridden. Cak Man's rules follow four levels of urgency, shown in **Figure 8.2**. Soft brakes and a smell of petrol mean *stop riding now*: a brake that does not bite and a fuel leak near a hot engine are the two faults that cause accidents and fires. A weak battery, overheating, worn brake pads, an injection fault or a blown fuse mean *go to a bengkel today*. A worn V-belt, a failing starter relay or a dirty CVT clutch mean *this week*. Only an empty tank, and starting a weak battery with the kick starter, are things the driver does himself. The app is an adviser: it names the likely cause and the sensible next step, and everything that involves opening the engine, the brakes or the fuel system is for a mechanic.

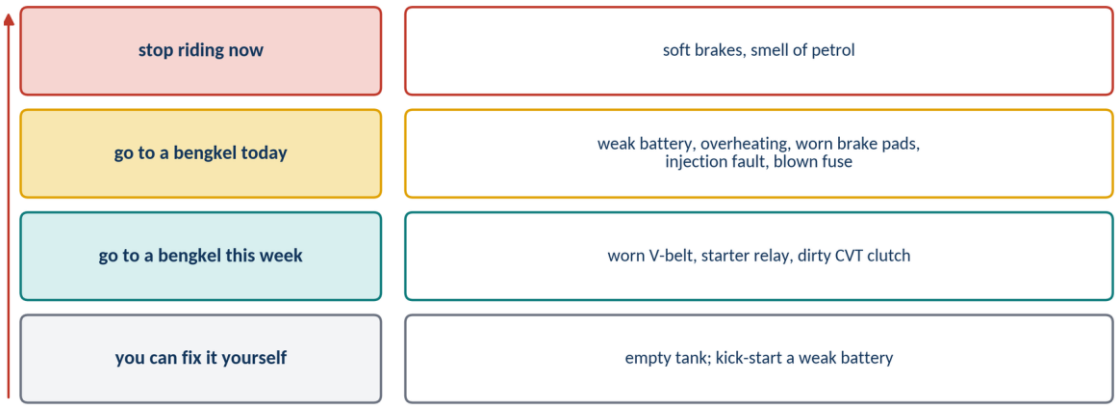

Figure 8.2 The four levels of urgency of the fault finder's advice, from the most urgent at the top. Brake and fuel faults stop the ride; most other faults go to a bengkel, today or this week; only a few things are for the driver himself.

The app enforces the top level in its structure, not only in its words. Before looking for any cause, it runs backward chaining on one goal, *safety = stop riding now*, which asks at most two questions. Only when that goal is not proved does it go on.

8.2.7 Which direction, when?

The two directions are not rivals; they suit different situations, and a good app uses both. Forward chaining is the better choice when the facts arrive by themselves and cost nothing: readings from a sensor, a form already filled in, the log of a machine. It finds *everything* that follows, including things nobody asked about, which is useful when you want a complete picture — a mechanic at the end of a service who wants every warning sign listed. Backward chaining is the better choice when facts must be asked of a person, when there are many possible questions and only a few matter for any one problem, and when there is a hypothesis to test. It stops as soon as the goal is proved, and it can always say why it asks.

In practice, expert systems mix them. The fault finder runs backward chaining twice — once for danger and once for a cause — and offers forward chaining for drivers who prefer to describe everything first. The same knowledge serves all three uses without change, because it is kept as rules that do not know in which direction they will be read. That is a property worth protecting: a rule written as "if the horn is weak, the electrics are weak" can be used forward and backward; a rule written as a procedure, "first ask about the horn, then …", can only be used one way.

8.3 Chaining from Scratch

Before using the engine, we build both directions ourselves on four rules. **Listing 8.1** is the whole program in Java.

Listing 8.1 `Chain.java`: forward and backward chaining from scratch, on four rules of the fault finder.

```
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

/**
 * Forward and backward chaining from scratch, on four rules of the fault
 * finder. A rule has a name, conditions and one conclusion.
 */
public class Chain {
    record Rule(String name, Map<String, String> when, String var,
                String value) {
    }

    static Map<String, String> map(String... kv) {
        Map<String, String> m = new LinkedHashMap<>();
        for (int i = 0; i < kv.length; i += 2) {
            m.put(kv[i], kv[i + 1]);
        }
        return m;
    }

    static final List<Rule> RULES = List.of(
            new Rule("R1", map("horn", "weak"), "electrics", "weak power"),
            new Rule("R2", map("starter sound", "slow"), "electrics",
                    "weak power"),
            new Rule("R3", map("engine starts", "no", "electrics", "weak power"),
                    "cause", "weak battery"),
            new Rule("R4", map("engine starts", "no", "fuel gauge", "empty"),
                    "cause", "empty tank"));

    /**
     * Fire every rule whose conditions are all known facts, again and
     * again, until nothing new can be concluded.
     */
    static Map<String, String> forward(Map<String, String> given) {
        Map<String, String> facts = new LinkedHashMap<>(given);
        boolean changed = true;
        while (changed) {
            changed = false;
            for (Rule r : RULES) {
                boolean all = true;
                for (Map.Entry<String, String> c : r.when.entrySet()) {
                    all = all && c.getValue().equals(facts.get(c.getKey()));
                }
                if (!facts.containsKey(r.var) && all) {
                    facts.put(r.var, r.value);
```

Listing 8.1 (Continued) *Chain.java: forward and backward chaining from scratch, on four rules of the fault finder.*

```
                    System.out.println("  " + r.name + " fires: " + r.var
                            + " = " + r.value);
                    changed = true;
                }
            }
        }
        return facts;
    }

    /**
     * Is var = value? Try the rules that conclude it; prove their
     * conditions the same way; ask about what no rule concludes.
     */
    static boolean backward(String var, String value, Map<String, String> facts,
                            Map<String, String> answers, int depth) {
        String pad = "  ".repeat(depth);
        if (facts.containsKey(var)) {
            return facts.get(var).equals(value);
        }
        List<Rule> rules = RULES.stream()
                .filter(r -> r.var.equals(var) && r.value.equals(value))
                .toList();
        if (rules.isEmpty()) {
            facts.put(var, answers.getOrDefault(var, "unknown"));
            System.out.println(pad + "ask " + var + ": " + facts.get(var));
            return facts.get(var).equals(value);
        }
        for (Rule r : rules) {
            System.out.println(pad + "try " + r.name + " for " + var + " = "
                    + value);
            boolean all = true;
            for (Map.Entry<String, String> c : r.when.entrySet()) {
                if (!backward(c.getKey(), c.getValue(), facts, answers,
                        depth + 1)) {
                    all = false;
                    break;
                }
            }
            if (all) {
                facts.put(var, value);
                System.out.println(pad + r.name + " proves " + var + " = "
                        + value);
                return true;
            }
        }
        return false;
```

Listing 8.1 (Continued) *Chain.java: forward and backward chaining from scratch, on four rules of the fault finder.*

```
    }

    public static void main(String[] args) {
        System.out.println("Forward, from: engine starts = no, horn = weak");
        forward(map("engine starts", "no", "horn", "weak"));
        System.out.println("Backward, goal: cause = weak battery");
        Map<String, String> driver = map("engine starts", "no", "horn",
                "normal", "starter sound", "slow");
        System.out.println(backward("cause", "weak battery",
                new LinkedHashMap<>(), driver, 1) ? "  proved" : "  not proved");
    }
}
```

A `Rule` has a name, its conditions (a map from variable to value, all of which must hold) and one conclusion, a variable and a value. R1 and R2 conclude the intermediate `electrics = weak power` from a weak horn or a slow starter; R3 concludes a weak battery; R4 an empty tank.

Forward. `forward` copies the facts and loops: for every rule whose conclusion is not yet known and whose conditions all hold, it adds the conclusion and prints which rule fired. When a full pass adds nothing, it stops. Because a new conclusion may make another rule true, the loop runs until there is no change: that is the *chain*.

Backward. `backward` answers "is `var` equal to `value`?" If the variable is already known, it compares. If no rule concludes the goal, it is a question for the driver, answered here from a map of prepared answers. Otherwise it tries each rule that concludes the goal, proving its conditions one after the other by calling itself — each condition becomes a subgoal, printed one level deeper — and stops at the first rule whose conditions all hold. **Listing 8.2** is the Python version.

Listing 8.2 `chain.py`: the same two directions in Python.

```
"""Forward and backward chaining from scratch, on four rules of the fault
finder. A rule is (name, conditions, conclusion)."""

RULES = [
    ("R1", {"horn": "weak"}, ("electrics", "weak power")),
    ("R2", {"starter sound": "slow"}, ("electrics", "weak power")),
    ("R3", {"engine starts": "no", "electrics": "weak power"},
     ("cause", "weak battery")),
    ("R4", {"engine starts": "no", "fuel gauge": "empty"},
     ("cause", "empty tank")),
]


def forward(facts):
    """Fire every rule whose conditions are all known facts, again and
    again, until nothing new can be concluded."""
    facts = dict(facts)
    changed = True
    while changed:
        changed = False
        for name, when, (var, value) in RULES:
            if var not in facts and all(facts.get(v) == x
                                        for v, x in when.items()):
                facts[var] = value
                print("  " + name + " fires: " + var + " = " + value)
                changed = True
    return facts


def backward(var, value, facts, answers, depth=1):
    """Is var = value? Try the rules that conclude it; prove their
    conditions the same way; ask about what no rule concludes."""
    pad = "  " * depth
    if var in facts:
        return facts[var] == value
    rules = [r for r in RULES if r[2] == (var, value)]
    if not rules:
        facts[var] = answers.get(var, "unknown")
        print(pad + "ask " + var + ": " + facts[var])
        return facts[var] == value
    for name, when, _ in rules:
        print(pad + "try " + name + " for " + var + " = " + value)
        if all(backward(v, x, facts, answers, depth + 1)
               for v, x in when.items()):
            facts[var] = value
            print(pad + name + " proves " + var + " = " + value)
```

Listing 8.2 (Continued) *chain.py: the same two directions in Python.*

```
            return True
    return False


print("Forward, from: engine starts = no, horn = weak")
forward({"engine starts": "no", "horn": "weak"})
print("Backward, goal: cause = weak battery")
driver = {"engine starts": "no", "horn": "normal", "starter sound": "slow"}
print("  proved" if backward("cause", "weak battery", {}, driver)
      else "  not proved")
```

Both print:

```
Forward, from: engine starts = no, horn = weak
  R1 fires: electrics = weak power
  R3 fires: cause = weak battery
Backward, goal: cause = weak battery
  try R3 for cause = weak battery
    ask engine starts: no
    try R1 for electrics = weak power
      ask horn: normal
    try R2 for electrics = weak power
      ask starter sound: slow
    R2 proves electrics = weak power
  R3 proves cause = weak battery
  proved
```

Forward chaining from a weak horn fired R1 and then R3: two steps up the chain. Backward chaining for the weak battery tried R3, which needed `engine starts` (asked: no) and `electrics = weak power`, a subgoal. The first rule for the subgoal, R1, needed a weak horn and failed (the horn was normal); the second, R2, needed a slow starter and succeeded. Then R3 was proved. Notice what backward chaining did *not* ask: the fuel gauge, which only R4 needs. That is its whole advantage.

It is worth tracing one more case by hand, because the two directions can disagree about *how much* they learn. Give forward chaining the facts `engine starts = no` and `fuel gauge = empty` only. On the first pass R1 and R2 fail (no horn or starter facts), R3 fails (the electrics are unknown), and R4 fires: *empty tank*. On the second pass nothing new happens. Now give backward chaining the goal *weak battery* for the same driver, with the horn normal and the starter normal. It tries R3, asks about the engine, tries R1 and R2 for the electrics, fails both, and reports *not proved* — without ever learning that the tank is empty, because no rule for its goal mentions fuel. Backward chaining is only as wide as its goal. That is why the fault finder's default goal is the variable `cause` with no value: "any cause", not one particular cause.

Our small engines lack three things that a real app needs: certainty factors, explanations that a user can read, and an order of rules that tries the most certain first. vcirs-core provides all three, with the same two directions [4].

8.4 Cak Man's Knowledge

Cak Man's knowledge is 21 cases in `data/fault-cases.json`, each with its facts, its conclusions, its CF and option 5 (a new top-level node):

```
{"facts": ["brake feel = soft"], "conclusions": ["safety = stop riding now",
  "cause = brake fluid leak or air in the brake line", "advice = do not ride;
  push it or call a bengkel that comes to you"], "cf": 100, ...}
{"facts": ["horn = weak"], "conclusions": ["electrics = weak power"], "cf": 90, ...}
{"facts": ["engine starts = no", "electrics = weak power"], "conclusions":
  ["cause = weak battery", "advice = kick-start it if it has a kick starter;
  charge or replace the battery at a bengkel today"], "cf": 100, ...}
{"facts": ["transmission = CVT problem", "km since belt change >= 24000"],
  "conclusions": ["cause = worn V-belt", "advice = replace the V-belt at a
  bengkel this week; a worn belt can snap on the road"], "cf": 90, ...}
{"facts": ["model = Yamaha NMAX"], "conclusions": ["cooling = liquid-cooled"], ...}
```

(The lines are wrapped here; in the file each case is one line.) **Table 8.1** summarizes what the cases know.

How were these cases collected? Not by asking Cak Man to "write down his rules" — experts are notoriously bad at that, and the attempt usually produces rules that are too general or that contradict each other. Instead, over two weeks, the drivers at the base camp noted every breakdown Cak Man diagnosed: what the driver saw, what Cak Man concluded, and how sure he was. Each case is one such conversation, reduced to its facts. Where Cak Man reasoned in two steps ("the horn is weak, so the battery side is weak; the engine will not start, so it is the battery"), the case was split into two, and the first became an intermediate conclusion. This is knowledge acquisition case by case, the VCIRS way [4], [6]: the expert talks about concrete motorcycles, never about rules in the abstract, and the structure of the knowledge follows from the cases.

All 21 cases use option 5, a new top-level node, rather than letting `learn` decide. For a knowledge base that will be read as rules, this is the clearest choice: each case becomes one node and each node becomes one rule, with no exceptions whose meaning depends on their parent. When a case contradicts an earlier one — when Cak Man says "not the battery this time, because the lights were bright" — the RDR options of Chapter 3 are still there, and transformation will turn the exception into its own rule with the full path of conditions.

Table 8.1 What Cak Man taught the fault finder: symptoms, causes and urgency. Rules without a CF in brackets have CF 100.

Symptoms (and intermediate conclusions)	Cause	Urgency
brakes feel soft	brake fluid leak or air in the line	stop riding now
strong smell of petrol	fuel leak	stop riding now
brakes grind	worn brake pads (CF 90)	bengkel today
no start; horn none → no power	blown main fuse or loose battery cable	bengkel today if it blows again
no start; horn weak or starter slow → weak power	weak battery	kick-start, bengkel today
no start; starter clicks; horn normal	starter relay or starter motor (CF 80)	kick-start, this week
no start; fuel gauge empty	empty tank	fill up
no start; fuel ok; engine check light blinking	injection or sensor fault (CF 80)	bengkel today
starts; top speed low → CVT problem; 24,000 km or more since the belt	worn V-belt (CF 90)	this week
CVT problem; shudders when pulling away	dirty CVT clutch (CF 80)	at a bengkel
temperature light on; model → liquid-cooled	overheating	stop and cool, bengkel today

The V-belt rule uses a number. Honda's dealers advise replacing the V-belt of an automatic scooter at about 25,000 km [9]; Cak Man starts to suspect it a little earlier, at 24,000, because ojek

drivers ride far more than most owners and a snapped belt leaves them stranded. The condition `km since belt change >= 24000` is compared as a number, so an answer of 26000 satisfies it and 18000 does not. The six model cases tell the engine which popular scooters are air-cooled (BeAT, Scoopy, Gear 125) and which have a radiator (Vario 125, Vario 160, NMAX); only a liquid-cooled engine has a coolant temperature warning light and a coolant reserve tank to check. Why ask for the model and not simply "is your engine liquid-cooled?" Because every driver knows his model and few know how their engine is cooled. A good question asks for something the user can see or knows for certain, and leaves the technical inference to the rules. The same idea is behind asking how loud the horn is instead of asking about the battery's voltage.

The command `rules` prints the rules that transformation makes from the cases. The first twenty lines:

```
stop riding now#1 [RULE CF=100]
  IF   brake feel = soft [CF=100]
  THEN safety = stop riding now
  AND  cause = brake fluid leak or air in the brake line
  AND  advice = do not ride; push it or call a bengkel that comes to you
stop riding now#2 [RULE CF=100]
  IF   fuel smell = yes [CF=100]
  THEN safety = stop riding now
  AND  cause = fuel leak
  AND  advice = switch off, keep fire and cigarettes away, do not ride; call a bengkel
worn brake pads#1 [RULE CF=90]
  IF   brake feel = grinding [CF=100]
  THEN cause = worn brake pads
  AND  advice = ride slowly to the nearest bengkel today and replace the pads
no power#1 [RULE CF=90]
  IF   horn = none [CF=100]
  THEN electrics = no power
weak power#1 [RULE CF=90]
  IF   horn = weak [CF=100]
  THEN electrics = weak power
```

Each node became one rule with its CF: the `[RULE CF=90]` of `weak power#1` comes from the case's CF 90 and Equation (8.1). Each rule's name is its node's identifier, made from the first conclusion's value, so the three electrics rules are called `no power#1`, `weak power#1` and `weak power#2`.

8.5 The Fault Finder

Listing 8.3 is the app.

Listing 8.3 `FaultFinder.java`: the fault finder with its safety check and three ways of reasoning.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.Set;

import vcirs.CaseFile;
import vcirs.ChainResult;
import vcirs.Conclusion;
import vcirs.Fact;
import vcirs.KnowledgeBase;
import vcirs.Question;
import vcirs.RbsRule;
import vcirs.Report;

/**
 * Motorcycle fault finder for ojek drivers: forward and backward chaining
 * with explanations, on vcirs-core.
 *   java FaultFinder [--echo]                safety check, then find the
 *                                            cause (backward)
 *   java FaultFinder diagnose [--echo]       describe it all first
 *                                            (forward chaining)
 *   java FaultFinder check "cause = X" [--echo]   is it X?
 *   java FaultFinder rules                   the rules it reasons with
 * Add a file name such as fault-cases-plus.json to use other cases, and
 * --log to see every step of the search for a cause.
 * Type "why" at any question to hear why it is asked; press Enter if you do
 * not know. The finder gives advice, not repairs: brakes, fuel leaks and
 * anything it cannot name go to a bengkel.
 */
public class FaultFinder {
    static final String DATA = "../data";
    static final String DANGER = "safety = stop riding now";
    static final String[][] QUESTIONS = {
        {"brake feel", "How do the brakes feel when you squeeze the levers?",
            "normal", "soft", "grinding"},
        {"fuel smell", "Is there a strong smell of petrol around the motorcycle?",
            "yes", "no"},
        {"engine starts", "Does the engine start?", "yes", "no"},
        {"starter sound", "What happens when you press the starter button?",
            "none", "click", "slow", "normal"},
```

Listing 8.3 (Continued) *FaultFinder.java: the fault finder with its safety check and three ways of reasoning.*

```
        {"horn", "How loud is the horn?", "normal", "weak", "none"},
        {"fuel gauge", "What does the fuel gauge show?", "empty", "ok"},
        {"engine check light", "Is the engine check light blinking?",
            "yes", "no"},
        {"top speed", "Can it still reach its usual speed?", "normal", "low"},
        {"km since belt change",
            "How many km since the V-belt was changed? (a number)"},
        {"shudder", "Does it shudder when you pull away?", "yes", "no"},
        {"temperature light", "Is the temperature warning light on?",
            "yes", "no"},
        {"model", "Which model is it?", "Honda BeAT", "Honda Scoopy",
            "Yamaha Gear 125", "Honda Vario 125", "Honda Vario 160",
            "Yamaha NMAX"},
    };
    static final List<String> CHECKLIST = List.of("model", "engine starts",
            "starter sound", "horn", "fuel gauge", "engine check light",
            "top speed", "temperature light");

    static KnowledgeBase openKb(String file) {
        CaseFile cases = CaseFile.read(DATA + "/" + file);
        KnowledgeBase kb = new KnowledgeBase(cases.name, cases.description);
        for (String[] q : QUESTIONS) {
            List<String> choices = new ArrayList<>();
            for (int i = 2; i < q.length; i++) {
                choices.add(q[i]);
            }
            kb.declare(q[0], q[1], choices);
        }
        return cases.build(kb);
    }

    /**
     * Asks the driver; answers "why" with the engine's reason. It never
     * asks for what a rule can work out (such as "electrics").
     */
    static class Driver {
        final KnowledgeBase kb;
        final boolean echo;
        int count = 0;
        final Set<String> derived = new HashSet<>();
        final Set<String> skipped = new HashSet<>();  // left empty in checklist
        boolean log = false;                          // every step (--log)
        final BufferedReader in = new BufferedReader(
                new InputStreamReader(System.in, StandardCharsets.UTF_8));
```

Listing 8.3 (Continued) *FaultFinder.java: the fault finder with its safety check and three ways of reasoning.*

```
        Driver(KnowledgeBase kb, boolean echo) {
            this.kb = kb;
            this.echo = echo;
            for (RbsRule r : kb.transform()) {
                for (Conclusion c : r.conclusions) {
                    derived.add(c.variable);
                }
            }
        }

        String read(String prompt) {
            System.out.print(prompt);
            String line;
            try {
                line = in.readLine();
            } catch (IOException e) {
                line = null;
            }
            String text = line == null ? "" : line.strip();
            if (echo) {
                System.out.println(text);
            }
            return text;
        }

        Fact ask(Question q) {
            String var = q.variable;
            if (derived.contains(var) || skipped.contains(var)) {
                return null;                // worked out by rules, or skipped
            }
            List<String> choices = kb.variables.get(var).choices;
            String hint = choices.isEmpty() ? ""
                    : " (" + String.join("/", choices) + ")";
            String answer;
            while (true) {
                answer = read(kb.variables.get(var).question + hint + " > ");
                if (!answer.equals("why")) {
                    break;
                }
                for (String line : q.why()) {
                    System.out.println("  " + line);
                }
            }
            count++;
            return answer.isEmpty() ? null : new Fact(var, answer);
        }
```

Listing 8.3 (Continued) *FaultFinder.java: the fault finder with its safety check and three ways of reasoning.*

```
    }

    static void show(ChainResult result, String what) {
        for (String line : result.what()) {
            System.out.println(line);
        }
        System.out.println(what);
        for (String line : result.how()) {
            System.out.println("  " + line);
        }
    }

    /** Backward chaining on the one goal that must be ruled out first. */
    static Map<String, Fact> safetyFirst(KnowledgeBase kb, Driver driver) {
        System.out.println("Safety first.");
        ChainResult result = kb.backward(DANGER, new LinkedHashMap<>(),
                driver::ask);
        if (result.proved) {
            System.out.println("STOP: do not ride this motorcycle now.");
            show(result, "How I know:");
            return null;
        }
        return new LinkedHashMap<>(result.memory.facts);
    }

    /** Backward chaining on "cause": only the questions a cause needs. */
    static void find(KnowledgeBase kb, Driver driver) {
        Map<String, Fact> facts = safetyFirst(kb, driver);
        if (facts == null) {
            return;
        }
        System.out.println("Looking for the cause.");
        ChainResult result = kb.backward("cause", facts, driver::ask);
        if (!result.proved) {
            System.out.println("I cannot name the fault. Take it to a bengkel "
                    + "and tell the mechanic what you told me.");
        }
        show(result, "How I know:");
        System.out.println("Questions asked: " + driver.count);
        if (driver.log) {
            System.out.println("Every step:");
            for (String line : result.log.lines()) {
                System.out.println("  " + line);
            }
        }
    }
```

Listing 8.3 (Continued) *FaultFinder.java: the fault finder with its safety check and three ways of reasoning.*

```
    /** Forward chaining: the driver describes everything first. */
    static void diagnose(KnowledgeBase kb, Driver driver) {
        Map<String, Fact> facts = safetyFirst(kb, driver);
        if (facts == null) {
            return;
        }
        System.out.println("Describe the motorcycle (Enter to skip a line).");
        for (String var : CHECKLIST) {
            driver.count++;
            String answer = driver.read(kb.variables.get(var).question + " > ");
            if (!answer.isEmpty()) {
                facts.put(var, new Fact(var, answer));
            } else {
                driver.skipped.add(var);
            }
        }
        ChainResult result = kb.forward(facts, driver::ask);
        show(result, "How I know:");
        System.out.println("Questions asked: " + driver.count);
    }

    /** Backward chaining on one cause the mechanic suspects. */
    static void check(KnowledgeBase kb, Driver driver, String goal) {
        Map<String, Fact> facts = safetyFirst(kb, driver);
        if (facts == null) {
            return;
        }
        System.out.println("Checking: " + goal);
        ChainResult result = kb.backward(goal, facts, driver::ask);
        System.out.println(result.proved ? "Proved."
                : "Not proved: look for another cause.");
        show(result, "How:");
        System.out.println("Questions asked: " + driver.count);
    }
```

Listing 8.3 (Continued) *FaultFinder.java: the fault finder with its safety check and three ways of reasoning.*

```
    public static void main(String[] args) {
        boolean echo = false;
        boolean log = false;
        String file = "fault-cases.json";
        List<String> words = new ArrayList<>();
        for (String a : args) {
            if (a.equals("--echo")) {
                echo = true;
            } else if (a.equals("--log")) {
                log = true;
            } else if (a.endsWith(".json")) {
                file = a;
            } else {
                words.add(a);
            }
        }
        KnowledgeBase kb = openKb(file);
        Driver driver = new Driver(kb, echo);
        driver.log = log;
        String command = words.isEmpty() ? "find" : words.get(0);
        if (command.equals("rules")) {
            for (String line : Report.transformed(kb)) {
                System.out.println(line);
            }
        } else if (command.equals("check")) {
            check(kb, driver, words.get(1));
        } else if (command.equals("diagnose")) {
            diagnose(kb, driver);
        } else {
            find(kb, driver);
        }
    }
}
```

Read it from the bottom. `main` builds the knowledge base from the case file (another file name ending in `.json` may be given) and runs one of four commands.

The driver. `Driver` is the asker the engines call when they need a fact. It prints the declared question with the usual answers, reads the reply, and counts the questions. If the reply is `why`, it prints the engine's reason, `Question.why()`, and asks again. Two kinds of question are never put to the driver: a variable that some rule concludes (`electrics`, `transmission`, `cooling`, `cause`, …), because the rules can work it out and a driver cannot be expected to know what "weak power" is; and a line he left empty in the checklist, because he already said he does not know. The set `derived` is computed once from the rules of `kb.transform()`.

Safety first. `safetyFirst` runs `kb.backward(DANGER, …)` with the goal *safety = stop riding now*. Two rules conclude it; backward chaining tries them and asks about the brakes and the smell of petrol. If the goal is proved, the app prints STOP with the explanation and ends. Otherwise it returns the facts it learned, so that nothing is asked twice.

Find. `find`, the default command, runs backward chaining on the goal `cause` with no value: "find any cause". The engine tries every rule that concludes a cause, most certain first, and stops at the first one it can prove — asking only what each attempt needs.

Diagnose. `diagnose` is forward chaining. The driver first answers a checklist of eight easy observations (the model, whether the engine starts, the starter, the horn, the fuel gauge, the engine check light, the top speed and the temperature light); then `kb.forward` fires everything that follows, and may still ask for a fact that a rule needs, such as the kilometers since the belt was changed.

Check. `check "cause = worn V-belt"` is backward chaining on a suspicion, the way Cak Man works when a driver says "I think it is the belt".

Every command ends with the engine's What (`result.what()`) and How (`result.how()`); with `--log`, `find` also prints every step of its search (Section 8.7.2). **Listing 8.4** is the Python version; the knowledge base, the questions and the output are the same.

Listing 8.4 `fault_finder.py`: the same app in Python.

```
"""Motorcycle fault finder for ojek drivers: forward and backward chaining
with explanations, on vcirs-core.

    python fault_finder.py [--echo]              safety check, then find
                                                 the cause (backward)
    python fault_finder.py diagnose [--echo]     describe it all first
                                                 (forward chaining)
    python fault_finder.py check "cause = X" [--echo]   is it X?
    python fault_finder.py rules                 the rules it reasons with
    Add a file name such as fault-cases-plus.json to use other cases, and
    --log to see every step of the search for a cause.

Type "why" at any question to hear why it is asked; press Enter if you do
not know. The finder gives advice, not repairs: brakes, fuel leaks and
anything it cannot name go to a bengkel.
"""

import os
import pathlib
import sys

# Tell Python where the vcirs-core engine is (two folders up, then across).
sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import KnowledgeBase, casefile, report

DATA = os.path.join(os.path.dirname(os.path.abspath(__file__)), "..", "data")
DANGER = "safety = stop riding now"
QUESTIONS = [
    ("brake feel", "How do the brakes feel when you squeeze the levers?",
     ["normal", "soft", "grinding"]),
    ("fuel smell", "Is there a strong smell of petrol around the motorcycle?",
     ["yes", "no"]),
    ("engine starts", "Does the engine start?", ["yes", "no"]),
    ("starter sound", "What happens when you press the starter button?",
     ["none", "click", "slow", "normal"]),
    ("horn", "How loud is the horn?", ["normal", "weak", "none"]),
    ("fuel gauge", "What does the fuel gauge show?", ["empty", "ok"]),
    ("engine check light", "Is the engine check light blinking?",
     ["yes", "no"]),
    ("top speed", "Can it still reach its usual speed?", ["normal", "low"]),
    ("km since belt change",
     "How many km since the V-belt was changed? (a number)", []),
    ("shudder", "Does it shudder when you pull away?", ["yes", "no"]),
    ("temperature light", "Is the temperature warning light on?",
     ["yes", "no"]),
```

Listing 8.4 (Continued) *fault_finder.py: the same app in Python.*

```
    ("model", "Which model is it?",
     ["Honda BeAT", "Honda Scoopy", "Yamaha Gear 125", "Honda Vario 125",
      "Honda Vario 160", "Yamaha NMAX"]),
]
CHECKLIST = ["model", "engine starts", "starter sound", "horn", "fuel gauge",
             "engine check light", "top speed", "temperature light"]


def open_kb(cases):
    name, description, steps = casefile.read_cases(os.path.join(DATA, cases))
    kb = KnowledgeBase(name, description)
    for var, question, choices in QUESTIONS:
        kb.declare(var, question, choices)
    return casefile.build(kb, steps)


class Driver:
    """Asks the driver; answers 'why' with the engine's reason. It never
    asks for what a rule can work out (such as 'electrics')."""

    def __init__(self, kb, echo):
        self.kb = kb
        self.echo = echo
        self.count = 0
        self.derived = {c.variable for r in kb.transform()
                        for c in r.conclusions}
        self.skipped = set()              # left empty in the checklist
        self.log = False                  # print every step (--log)

    def read(self, prompt):
        print(prompt, end="")
        text = sys.stdin.readline().strip()
        if self.echo:
            print(text)
        return text
```

Listing 8.4 (Continued) *fault_finder.py: the same app in Python.*

```
    def ask(self, question):
        var = question.variable
        if var in self.derived or var in self.skipped:
            return None                    # worked out by rules, or skipped
        choices = self.kb.variables[var].choices
        hint = " (" + "/".join(choices) + ")" if choices else ""
        while True:
            answer = self.read(self.kb.variables[var].question + hint
                               + " > ")
            if answer != "why":
                break
            for line in question.why():
                print("  " + line)
        self.count += 1
        return answer or None


def show(result, what):
    for line in result.what():
        print(line)
    print(what)
    for line in result.how():
        print("  " + line)


def safety_first(kb, driver):
    """Backward chaining on the one goal that must be ruled out first."""
    print("Safety first.")
    result = kb.backward(DANGER, {}, driver.ask)
    if result.proved:
        print("STOP: do not ride this motorcycle now.")
        show(result, "How I know:")
        return None
    return result.memory.facts
```

Listing 8.4 (Continued) *fault_finder.py: the same app in Python.*

```
def find(kb, driver):
    """Backward chaining on 'cause': only the questions a cause needs."""
    facts = safety_first(kb, driver)
    if facts is None:
        return
    print("Looking for the cause.")
    result = kb.backward("cause", facts, driver.ask)
    if not result.proved:
        print("I cannot name the fault. Take it to a bengkel and tell the "
              "mechanic what you told me.")
    show(result, "How I know:")
    print("Questions asked: " + str(driver.count))
    if driver.log:
        print("Every step:")
        for line in result.log.lines():
            print("  " + line)


def diagnose(kb, driver):
    """Forward chaining: the driver describes everything first."""
    facts = safety_first(kb, driver)
    if facts is None:
        return
    facts = dict(facts)
    print("Describe the motorcycle (Enter to skip a line).")
    for var in CHECKLIST:
        driver.count += 1
        answer = driver.read(kb.variables[var].question + " > ")
        if answer:
            facts[var] = answer
        else:
            driver.skipped.add(var)
    result = kb.forward(facts, driver.ask)
    show(result, "How I know:")
    print("Questions asked: " + str(driver.count))
```

Listing 8.4 (Continued) *fault_finder.py: the same app in Python.*

```
def check(kb, driver, goal):
    """Backward chaining on one cause the mechanic suspects."""
    facts = safety_first(kb, driver)
    if facts is None:
        return
    print("Checking: " + goal)
    result = kb.backward(goal, facts, driver.ask)
    print("Proved." if result.proved else "Not proved: look for another "
          "cause.")
    show(result, "How:")
    print("Questions asked: " + str(driver.count))


def main(args):
    echo = "--echo" in args
    files = [a for a in args if a.endswith(".json")]
    words = [a for a in args if a != "--echo" and a not in files]
    kb = open_kb(files[0] if files else "fault-cases.json")
    driver = Driver(kb, echo)
    driver.log = "--log" in args
    words = [w for w in words if w != "--log"]
    if words and words[0] == "rules":
        for line in report.transformed(kb):
            print(line)
    elif words and words[0] == "check":
        check(kb, driver, words[1])
    elif words and words[0] == "diagnose":
        diagnose(kb, driver)
    else:
        find(kb, driver)


if __name__ == "__main__":
    main(sys.argv[1:])
```

The data folder holds the answers of the sessions in this chapter, one per line, so that each can be replayed with `--echo`, exactly as in Chapter 3: from `Chapter08/java`, compile with `javac -d out -sourcepath ../../vcirs-core/java/src *.java` and run, for example, `java -cp out FaultFinder --echo < ../data/find-beat.txt`; from `Chapter08/python`, `python fault_finder.py --echo < ../data/find-beat.txt`.

8.6 Mornings at the Base Camp

8.6.1 Mas Yudi's BeAT will not start

Six o'clock, in front of the boarding house. Mas Yudi runs the finder:

```
Safety first.
How do the brakes feel when you squeeze the levers? (normal/soft/grinding) > normal
Is there a strong smell of petrol around the motorcycle? (yes/no) > no
Looking for the cause.
Does the engine start? (yes/no) > no
How loud is the horn? (normal/weak/none) > weak
Result: electrics = weak power  (rule weak power#1, CF 90)
Result: cause = weak battery  (rule weak battery#1, CF 90)
Result: advice = kick-start it if it has a kick starter; charge or replace the battery at a
bengkel today  (rule weak battery#1, CF 90)
How I know:
  weak power#1 fired because:
    horn = weak  (answered by the user, CF 100)
    so electrics = weak power  [rule CF 90, conclusion CF 90]
  weak battery#1 fired because:
    engine starts = no  (answered by the user, CF 100)
    electrics = weak power  (concluded by weak power#1, CF 90)
    so cause = weak battery, advice = kick-start it if it has a kick starter; charge or
replace the battery at a bengkel today  [rule CF 100, conclusion CF 90]
Questions asked: 4
```

Four questions. The first two are the safety check: the brakes are normal and there is no smell of petrol, so the goal *stop riding now* is not proved. Then backward chaining looks for a cause. The two safety rules also conclude causes, but their facts are already known and false, so they fail without a question. The next rule by certainty is *blown main fuse*, which needs `engine starts = no` (asked: no) and `electrics = no power`, a subgoal. To prove anything about `electrics`, the engine tries the rules that conclude it: `no power#1` needs no horn at all (asked: the horn is weak — it fails), and `weak power#1` needs a weak horn, which holds, so `electrics = weak power` is concluded with CF 90. The fuse rule still fails, since the electrics have weak power, not none. The next rule, *weak battery*, needs exactly what is now known, and fires with CF 90, as Equation (8.2) predicted in Section 8.2.4.

The How explanation shows the chain: the horn made the electrics weak, and the weak electrics with a dead engine made the battery the cause. The advice tells Mas Yudi what to do *now* (kick-start it, since the BeAT has a kick starter) and *today* (charge or replace the battery). He is on the road by a quarter past six.

8.6.2 The same morning, forward

What if Mas Yudi had described everything first? `diagnose` does that:

```
Safety first.
How do the brakes feel when you squeeze the levers? (normal/soft/grinding) > normal
Is there a strong smell of petrol around the motorcycle? (yes/no) > no
Describe the motorcycle (Enter to skip a line).
Which model is it? > Honda BeAT
Does the engine start? > no
What happens when you press the starter button? > slow
How loud is the horn? > weak
What does the fuel gauge show? > ok
Is the engine check light blinking? > no
Can it still reach its usual speed? >
Is the temperature warning light on? >
Result: cooling = air-cooled  (rule air-cooled#1, CF 100)
Result: electrics = weak power  (rule weak power#2, CF 80)
Result: cause = weak battery  (rule weak battery#1, CF 80)
Result: advice = kick-start it if it has a kick starter; charge or replace the battery at a
bengkel today  (rule weak battery#1, CF 80)
How I know:
  air-cooled#1 fired because:
    model = Honda BeAT  (given, CF 100)
    so cooling = air-cooled  [rule CF 100, conclusion CF 100]
  weak power#2 fired because:
    starter sound = slow  (given, CF 100)
    so electrics = weak power  [rule CF 80, conclusion CF 80]
  weak power#1 fired because:
    horn = weak  (given, CF 100)
    so electrics = weak power  [rule CF 90, conclusion CF 90]
  weak battery#1 fired because:
    engine starts = no  (given, CF 100)
    electrics = weak power  (concluded by weak power#2, CF 80)
    so cause = weak battery, advice = kick-start it if it has a kick starter; charge or
replace the battery at a bengkel today  [rule CF 100, conclusion CF 80]
Questions asked: 10
```

Ten questions instead of four, and the same cause. Forward chaining cannot know in advance which observations matter, so it collects them all, and then derives everything they imply — including `cooling = air-cooled` from the model, which nobody asked about and which does not matter today. That is the character of data-driven reasoning: thorough, and not economical with the user's time. It has its place, though. At the end of a regular service, Cak Man could fill in the checklist for every motorcycle as a matter of routine — model, starter, horn, top speed, kilometers since the belt — and let forward chaining list *every* warning the knowledge base can derive, including a belt that is due before any symptom appears. There the facts cost nothing, because the mechanic is looking at the motorcycle anyway, and completeness is exactly what he wants.

Two details are worth reading closely. First, the order of the How explanation follows the engine's queue of variables: the model came first in the checklist, so its rule fired first; the starter's rule `weak power#2` (CF 80) fired before the horn's `weak power#1` (CF 90), because *starter sound* came before *horn* in the checklist. Second, `electrics` was concluded once, by the first rule that fired, with CF 80; when `weak power#1` fired a moment later, the engine noted that the value was already known and did not change it. So the weak battery was concluded with CF 80 here, and with CF 90 in the backward session. The engine takes the first conclusion it reaches; it does not add up evidence from two rules. Section 8.7 comes back to this.

8.6.3 Pak Heru's Vario 160 has no power

Pak Heru rides a Vario 160. For a week it has felt weak on the hills, and the top speed is down. Cak Man suspects the V-belt and checks it with `check "cause = worn V-belt"`. At the question about kilometers, Pak Heru types `why`:

```
Safety first.
How do the brakes feel when you squeeze the levers? (normal/soft/grinding) > normal
Is there a strong smell of petrol around the motorcycle? (yes/no) > no
Checking: cause = worn V-belt
Does the engine start? (yes/no) > yes
Can it still reach its usual speed? (normal/low) > low
How many km since the V-belt was changed? (a number) > why
  I need to know 'km since belt change'.
  Rule worn V-belt#1 needs km since belt change >= 24000;
  if it holds, the rule concludes cause = worn V-belt, advice = replace the V-belt at a
bengkel this week; a worn belt can snap on the road.
How many km since the V-belt was changed? (a number) > 26000
Proved.
Result: transmission = CVT problem  (rule CVT problem#1, CF 90)
Result: cause = worn V-belt  (rule worn V-belt#1, CF 81)
Result: advice = replace the V-belt at a bengkel this week; a worn belt can snap on the
road  (rule worn V-belt#1, CF 81)
How:
  CVT problem#1 fired because:
    engine starts = yes  (answered by the user, CF 100)
    top speed = low  (answered by the user, CF 100)
    so transmission = CVT problem  [rule CF 90, conclusion CF 90]
  worn V-belt#1 fired because:
    transmission = CVT problem  (concluded by CVT problem#1, CF 90)
    km since belt change >= 24000  (km since belt change = 26000, answered by the user, CF
100)
    so cause = worn V-belt, advice = replace the V-belt at a bengkel this week; a worn belt
can snap on the road  [rule CF 90, conclusion CF 81]
Questions asked: 5
```

The *why* answer names the rule, the exact condition and what the rule would conclude. Pak Heru finds the service booklet under the seat: the belt was changed 26,000 km ago. The goal is proved through the intermediate `transmission = CVT problem`, with CF 81: the CVT rule has CF 90, the V-belt rule has CF 90, and $90 \times 90/100 = 81$ by Equation (8.2). The advice is *this week*: a worn belt can snap, and on a Vario 160 with a passenger that is a dangerous way to stop.

Suppose instead that the booklet had said 18,000 km. The same check ends differently:

```
Safety first.
How do the brakes feel when you squeeze the levers? (normal/soft/grinding) > normal
Is there a strong smell of petrol around the motorcycle? (yes/no) > no
Checking: cause = worn V-belt
Does the engine start? (yes/no) > yes
Can it still reach its usual speed? (normal/low) > low
How many km since the V-belt was changed? (a number) > 18000
Not proved: look for another cause.
Result: transmission = CVT problem  (rule CVT problem#1, CF 90)
How:
  CVT problem#1 fired because:
    engine starts = yes  (answered by the user, CF 100)
    top speed = low  (answered by the user, CF 100)
    so transmission = CVT problem  [rule CF 90, conclusion CF 90]
Questions asked: 5
```

The intermediate conclusion still holds — something is wrong in the CVT — but the belt is too young for the V-belt rule, so the goal is not proved, and the finder says so: *look for another cause*. It does not quietly switch to another goal; a check answers the question it was asked. Pak Heru can now run the finder without a goal, and backward chaining on `cause` will try the dirty clutch next and ask whether the scooter shudders. A “no” from a check is as useful as a “yes”: it rules out the expensive part and points to the cheap one.

8.6.4 A warning light on an NMAX

Mbak Wulan rides a Yamaha NMAX for a food-delivery app. In the midday traffic a warning light came on, and she runs the finder. At the question about the model she asks why:

```
Safety first.
How do the brakes feel when you squeeze the levers? (normal/soft/grinding) > normal
Is there a strong smell of petrol around the motorcycle? (yes/no) > no
Looking for the cause.
Does the engine start? (yes/no) > yes
Is the temperature warning light on? (yes/no) > yes
Which model is it? (Honda BeAT/Honda Scoopy/Yamaha Gear 125/Honda Vario 125/Honda Vario
160/Yamaha NMAX) > why
  I need to know 'model'.
  Rule air-cooled#1 needs model = Honda BeAT;
  if it holds, the rule concludes cooling = air-cooled.
  Chain of rules from the goal: overheating#1 > air-cooled#1.
Which model is it? (Honda BeAT/Honda Scoopy/Yamaha Gear 125/Honda Vario 125/Honda Vario
160/Yamaha NMAX) > Yamaha NMAX
Result: cooling = liquid-cooled  (rule liquid-cooled#3, CF 100)
Result: cause = overheating  (rule overheating#1, CF 100)
Result: advice = stop in a safe place and let the engine cool; check the coolant reserve
tank only when cold; go to a bengkel today  (rule overheating#1, CF 100)
How I know:
  liquid-cooled#3 fired because:
    model = Yamaha NMAX  (answered by the user, CF 100)
    so cooling = liquid-cooled  [rule CF 100, conclusion CF 100]
  overheating#1 fired because:
    temperature light = yes  (answered by the user, CF 100)
    cooling = liquid-cooled  (concluded by liquid-cooled#3, CF 100)
    so cause = overheating, advice = stop in a safe place and let the engine cool; check
the coolant reserve tank only when cold; go to a bengkel today  [rule CF 100, conclusion CF
100]
Questions asked: 5
```

The *why* answer shows a **chain of rules from the goal**: the engine is trying to prove `overheating#1`, which needs to know whether the engine is liquid-cooled; to find out, it is trying `air-cooled#1`, the first rule about cooling, which needs `model = Honda BeAT`. The answer, *Yamaha NMAX*, fails the three air-cooled rules and proves `liquid-cooled#3`. Then the overheating rule fires. Five questions, and one of them — the model — is asked only because a rule needed it.

The advice is careful on purpose. A hot engine's coolant is under pressure, and opening the radiator cap of a hot engine can scald. Mbak Wulan is told to stop in a safe place, let the engine cool, look at the coolant reserve tank only when it is cold, and go to a bengkel today.

8.6.5 Soft brakes

One evening a young driver says his front brake lever goes almost to the handlebar. The finder stops at the first question:

```
Safety first.
How do the brakes feel when you squeeze the levers? (normal/soft/grinding) > soft
STOP: do not ride this motorcycle now.
Result: safety = stop riding now  (rule stop riding now#1, CF 100)
Result: cause = brake fluid leak or air in the brake line  (rule stop riding now#1, CF 100)
Result: advice = do not ride; push it or call a bengkel that comes to you  (rule stop
riding now#1, CF 100)
How I know:
  stop riding now#1 fired because:
    brake feel = soft  (answered by the user, CF 100)
    so safety = stop riding now, cause = brake fluid leak or air in the brake line, advice
= do not ride; push it or call a bengkel that comes to you  [rule CF 100, conclusion CF
100]
```

No diagnosis, no list of causes, no fuel question: the goal *stop riding now* was proved by the first rule, and the app ended. This is the safety design of Section 8.2.6 at work. The explanation still says why, so the driver knows it is not the program being fussy. He pushes the motorcycle to the side of the road, calls a mobile mechanic, and that night nobody is hurt.

> **KEY INSIGHT.** The order of the goals is a safety decision. By proving *stop riding now* before looking for any cause, the app guarantees that no driver with soft brakes is ever told about his battery first. Nothing in the rules themselves could guarantee this; it is the structure of the program around them.

8.6.6 Counting the questions

Table 8.2 puts the sessions side by side. The numbers are the "Questions asked" lines of the transcripts, which include the two questions of the safety check; the soft-brake session asked one.

Table 8.2 Questions asked in the sessions of this chapter, including the safety check.

Session	Mode	Questions	Result
Mas Yudi's BeAT (8.6.1)	backward, find a cause	4	weak battery, CF 90
the same BeAT (8.6.2)	forward, checklist first	10	weak battery, CF 80
Pak Heru's Vario 160 (8.6.3)	backward, check the V-belt	5	worn V-belt, CF 81
Mbak Wulan's NMAX (8.6.4)	backward, find a cause	5	overheating, CF 100
soft brakes (8.6.5)	backward, safety goal	1	stop riding now
Pak Slamet's Scoopy (8.7.1)	backward, find a cause	8	no cause, then worn spark plug

Backward chaining asked four or five questions when the fault was one it knew, and eight when it had to rule out every cause it knew before giving up. Forward chaining asked ten for a fault that backward chaining found with four. Neither number is fixed: it depends on the order of the rules and on the fault. But the pattern is general, and it is the reason goal-driven reasoning became the standard for consultation systems in which a person answers the questions [5].

8.7 Making It Better

8.7.1 A fault it cannot name

Pak Slamet's Scoopy will not start. The starter spins normally, the horn is loud, there is fuel, and no warning light is on:

```
Safety first.
How do the brakes feel when you squeeze the levers? (normal/soft/grinding) > normal
Is there a strong smell of petrol around the motorcycle? (yes/no) > no
Looking for the cause.
Does the engine start? (yes/no) > no
How loud is the horn? (normal/weak/none) > normal
What happens when you press the starter button? (none/click/slow/normal) > normal
What does the fuel gauge show? (empty/ok) > ok
Is the temperature warning light on? (yes/no) > no
Is the engine check light blinking? (yes/no) > no
I cannot name the fault. Take it to a bengkel and tell the mechanic what you told me.
No rule fired.
How I know:
  No rule fired.
Questions asked: 8
```

Eight questions, and no rule could be proved. The finder says so honestly and tells him to take it to a bengkel with the answers he gave — which is exactly what a mechanic wants to hear. Cak Man finds a worn, fouled spark plug and replaces it in ten minutes. He then teaches the fault finder the case: *engine does not start, starter normal, fuel ok, no engine check light → worn spark plug*, with CF 80, because other faults can look the same. The case is one more line in a copy of the case file, `fault-cases-plus.json`. The next driver with the same symptoms gets an answer:

```
Safety first.
How do the brakes feel when you squeeze the levers? (normal/soft/grinding) > normal
Is there a strong smell of petrol around the motorcycle? (yes/no) > no
Looking for the cause.
Does the engine start? (yes/no) > no
How loud is the horn? (normal/weak/none) > normal
What happens when you press the starter button? (none/click/slow/normal) > normal
What does the fuel gauge show? (empty/ok) > ok
Is the temperature warning light on? (yes/no) > no
Is the engine check light blinking? (yes/no) > no
Result: cause = worn spark plug  (rule worn spark plug#1, CF 80)
Result: advice = have the spark plug checked and replaced at a bengkel; it takes ten
minutes  (rule worn spark plug#1, CF 80)
How I know:
  worn spark plug#1 fired because:
    engine starts = no  (answered by the user, CF 100)
    starter sound = normal  (answered by the user, CF 100)
    fuel gauge = ok  (answered by the user, CF 100)
    engine check light = no  (answered by the user, CF 100)
    so cause = worn spark plug, advice = have the spark plug checked and replaced at a
bengkel; it takes ten minutes  [rule CF 80, conclusion CF 80]
Questions asked: 8
```

The new rule was tried last, because its CF, 80, is the lowest; all the more certain rules were ruled out first by the same eight answers. This is the everyday loop of VCIRS from Chapter 3 applied to chaining: when the system has no answer, the expert teaches the case, and the knowledge base transformation turns it into a rule that forward and backward chaining can use at once.

How did Cak Man choose CF 80? Not by measuring anything. A certainty factor is the expert's judgment, and it helps to agree on a simple scale at the base camp and keep to it: 100 for "always, I have never seen otherwise" (an empty tank stops the engine); 90 for "almost always" (a weak horn means weak power); 80 for "usually, but other faults look the same" (the spark plug, the starter relay, the dirty clutch); 70 or less for "a reasonable first guess". With a shared scale, two mechanics teaching the same knowledge base give comparable numbers, and a driver reading *CF 72* knows it means "fairly likely, worth checking", not "certain".

8.7.2 Seeing every step

The How explanation shows the rules that fired. Sometimes a mechanic wants more: every rule the engine *tried*, including the ones that failed, and exactly where each failed. That is the engine's event log, the EventLog of the thesis [4], and the option `--log` prints it after a search for a cause. For Mas Yudi's morning (`--echo --log < ../data/find-beat.txt`), the end of the output is:

```
Questions asked: 4
Every step:
   1. backward chaining for cause
   2. goal cause: rules stop riding now#1, stop riding now#2, blown main fuse or loose
battery cable#1, weak battery#1, empty tank#1, overheating#1, worn brake pads#1, worn V-
belt#1, starter relay or starter motor#1, injection or sensor fault#1, dirty CVT clutch#1
   3. try stop riding now#1 for cause
   4. stop riding now#1: brake feel = soft fails (brake feel = normal)
   5. stop riding now#1 does not fire
   6. try stop riding now#2 for cause
   7. stop riding now#2: fuel smell = yes fails (fuel smell = no)
   8. stop riding now#2 does not fire
   9. try blown main fuse or loose battery cable#1 for cause
  10. ask engine starts (for blown main fuse or loose battery cable#1: engine starts = no)
  11. engine starts = no [CF 100] (user)
  12. blown main fuse or loose battery cable#1: engine starts = no holds (engine starts =
no)
  13. subgoal electrics (needed by blown main fuse or loose battery cable#1)
  14. goal electrics: rules no power#1, weak power#1, weak power#2
  15. try no power#1 for electrics
  16. ask horn (for no power#1: horn = none)
  17. horn = weak [CF 100] (user)
  18. no power#1: horn = none fails (horn = weak)
  19. no power#1 does not fire
  20. try weak power#1 for electrics
  21. weak power#1: horn = weak holds (horn = weak)
  22. weak power#1 fires: electrics = weak power [CF 90]
  23. blown main fuse or loose battery cable#1: electrics = no power fails (electrics =
weak power)
  24. blown main fuse or loose battery cable#1 does not fire
  25. try weak battery#1 for cause
  26. weak battery#1: engine starts = no holds (engine starts = no)
  27. weak battery#1: electrics = weak power holds (electrics = weak power)
  28. weak battery#1 fires: cause = weak battery, advice = kick-start it if it has a kick
starter; charge or replace the battery at a bengkel today [CF 90]
  29. goal cause proved
```

Line 2 is the agenda of backward chaining: the eleven rules that conclude a cause, in the order they will be tried — certainty first, then the order of the case file. Lines 3 to 8 are the two safety rules failing on facts already known, without a question. Line 13 is the moment a condition becomes a subgoal: `electrics` is needed by the fuse rule, so the engine opens a new, smaller agenda (line 14) of the three rules that conclude it. Line 16 is a question, with its reason in brackets: *ask horn, for no power#1: horn = none*. Line 22 is the intermediate conclusion with its CF, and lines 26 and 27 show the weak-battery rule reusing both facts without asking again. Twenty-nine steps, four questions.

A log like this is how a knowledge base is debugged. When the finder asks a question that seems silly, the log shows which rule asked it and why that rule was being tried at that moment; when it misses a cause, the log shows which condition failed. Cak Man reads it the way he reads the sound of an engine.

8.7.3 Combining evidence

In Section 8.6.2 the electrics were concluded with CF 80 by the starter's rule, and the horn's rule, with CF 90, was ignored because the value was already known. Many rule-based systems **combine** the certainty of two rules that reach the same conclusion: two independent pieces of evidence make the conclusion more certain than either alone. MYCIN's classic combination for two positive factors, written on a scale of 100, is $CF_1 + CF_2(100 - CF_1)/100$; for 80 and 90 it gives $80 + 90 \times 20/100 = 98$ [5]. The VCIRS engine follows the thesis and keeps the first conclusion. For a fault finder this is the cautious choice — it never becomes more certain than one rule makes it — and the How explanation shows both rules that fired, so a mechanic reading it sees the second piece of evidence. Exercise 8.6 asks you to compute what combining would change.

8.7.4 The order of rules

Backward chaining tries the rules for a goal in order of certainty, and for equal certainty in the order they were taught. That order decides which *questions come first*, not which answer is right: a rule is only proved when all its conditions hold. But the order does matter for the driver's time. In Section 8.6.4 Mbak Wulan was first asked whether the engine starts — a question for the fuse and battery rules, which come earlier in the file — and only then about the temperature light and the model, because the overheating rule comes after them. When a group of drivers mostly brings one kind of fault, putting the most common cases earlier in the case file makes the finder ask fewer questions on average. The usage rates of Chapter 7 can guide that choice.

8.7.5 Tests

`FinderTest.java` and `test_finder.py` check the knowledge base (21 nodes and 21 rules), the safety goal (proved for soft brakes, not for normal brakes and no petrol smell), that backward chaining finds the weak battery with four questions, and that forward chaining concludes the worn V-belt from 30,000 km and overheating from an NMAX with the light on. They end with:

```
all tests passed
```

8.8 Using It for Real

At the base camp. Put the finder on the laptop or an old phone at the base camp, with the case file that the local mechanic trusts. Teach it the cases that come up again and again in your area — the dust of dry-season roads, the floods of the rainy season, which leave water in the CVT housing and the air filter — and let the mechanic read the rules with `rules` now and then. A driver who is stuck elsewhere can run the same program on his phone with a Python app, or simply call a colleague at the base camp who runs it for him.

Your own models. The six model cases are only the start. Add the models your colleagues ride — a Honda PCX 160, a Yamaha Aerox, a Honda Genio, an electric scooter — with the right cooling, after checking the owner's manual. Electric motorcycles need their own rules (battery charge, controller warnings) and their own mechanic; do not stretch petrol-engine rules to cover them.

Knowing your numbers. The V-belt threshold of 24,000 km, the idea that a weak horn means a weak battery, the CFs of the rules — all come from one mechanic's experience and from the dealers' schedules [9]. Your mechanic may use other numbers, and he should. The point of the

fault finder is that his numbers, once taught, are applied the same way for every driver, and that every answer can be traced back to them.

Three minutes before the finder. Some checks are faster than any program, and the finder's questions assume the driver has done them: is the side stand up (most scooters will not start with the stand down), is the brake lever squeezed while pressing the starter (a scooter's starter works only with a brake held), is the key fully turned, and is the kill switch on the right handlebar in the *run* position? Put these four on a card at the base camp. A good part of the "it will not start" calls to any mechanic end with one of them, and teaching a program to ask them is less useful than teaching drivers to check them first.

Safety is not a setting. Keep the two stop-riding rules at the top of the knowledge base and keep the safety check in front of every diagnosis. Never teach the finder a repair that means working on the brakes, the fuel system or anything hot. When in doubt, the advice is a bengkel.

In Indonesian. The questions of the fault finder are plain text in `QUESTIONS`; the knowledge uses short English values such as `weak` and `no power`. To give the drivers Indonesian questions — *Bagaimana bunyi klakson?* — change only the question texts, and keep the values, or translate the values in the case file and the choices together. The rules do not care which language the driver reads; they care that the answers match the values in the conditions.

Keeping a logbook. Ask the drivers to save the transcript of every consultation — the questions, the answers and the advice — and to write one more line when the motorcycle has been repaired: what the mechanic actually found. Once a month, compare. When the finder's cause and the mechanic's agree, the knowledge is confirmed; when they disagree, the transcript shows exactly which answers led the finder astray, and Cak Man can teach the case that was missing, as he did for Pak Slamet's spark plug. After a few months the logbook is also a small history of the base camp's motorcycles: which faults are common, in which season, on which model.

Other machines. The same program, with other cases, becomes a fault finder for a water pump, a generator, a rice cooker at a laundromat, or the cold-storage box of a fish seller. The two directions of reasoning and the three explanations are the same.

8.9 Common Mistakes, and How to Avoid Them

Asking the user for an intermediate conclusion. A driver cannot answer "are the electrics at weak power?" *Avoid it:* let rules conclude intermediate variables, and never ask for a variable that some rule concludes, as `Driver.ask` does.

Using forward chaining when questions are expensive. Forward chaining needs the facts first and asks for many that do not matter. *Avoid it:* use backward chaining when a person must answer; use forward chaining when the facts arrive anyway, from a form or a sensor.

Letting safety depend on the order of rules. If the stop-riding rules were ordinary causes among others, a driver could be told about his battery before anyone asked about his brakes. *Avoid it:* check the danger goal first, on its own.

Advising repairs that can hurt. *Avoid it:* the advice names the cause and the next step; opening brakes, fuel lines and hot cooling systems is for a mechanic.

Answering a number with words. `km since belt change >= 24000` needs a number; *about twenty thousand* is not compared as a number and the condition fails. *Avoid it:* the question says "(a number)"; if the driver does not know, pressing Enter is better than a guess.

Different words for the same model. `Honda Beat` and `Honda BeAT` are different values. *Avoid it:* use the choices the question shows.

Reading CF as a probability. A CF of 81 does not mean 81 percent of such scooters have a worn belt. It is a measure of how strongly the rules support the conclusion. *Avoid it:* read it as "strong", "fairly strong" or "weak" support, and let the mechanic decide.

Forgetting the driver's safety beyond the motorcycle. A driver who has stopped on a busy road is in danger whatever the fault. *Avoid it:* the first advice of every consultation on the road is to push the motorcycle to a safe place, off the traffic lane, before answering any question.

Letting the finder replace the service. A finder that answers well can tempt drivers to skip the regular service. *Avoid it:* periodic replacements — oil, belt, brake pads, spark plug — follow the dealer's schedule [9]; the finder is for the days when something has already gone wrong.

Teaching an unverified cause. If the spark plug was replaced *and* the fuel filter too, it is not clear which fixed it. *Avoid it:* teach only what the repair proved.

8.10 Chapter Summary

- **Forward chaining** reasons from facts to everything they imply; **backward chaining** reasons from a goal to the rules and questions that prove it, and asks only what the goal needs (**Figure 8.1**, **Listing 8.1**, **Listing 8.2**).
- **Intermediate conclusions** such as `electrics = weak power` let rules feed rules, the way a mechanic first finds the system and then the part.
- A rule-based system answers **What** (the conclusions), **How** (the rules and facts that produced them) and **Why** (the rule and condition behind a question, and the chain of rules from the goal).
- The certainty of a rule and of a conclusion follow Equation (8.1) and Equation (8.2): the weakest link decides, and a chain is never more certain than its least certain step.
- VCIRS turns a knowledge base built from cases into IF-THEN rules by **knowledge base transformation**, so the easy teaching of RDR and the strong inference of rule-based systems work together (**Table 8.1**).
- The **fault finder** (**Listing 8.3**, **Listing 8.4**) checks the danger goal first, finds causes by backward chaining in four or five questions, can also diagnose by forward chaining, explains every step, and gives advice by urgency — stop now, bengkel today, this week, or do it yourself (**Figure 8.2**, **Table 8.2**).

8.11 Exercises

8.1 Using the four rules of **Listing 8.2**, write by hand the trace of forward chaining from `engine starts = no`, `starter sound = slow`, `fuel gauge = empty`. Which rules fire, in which order, and what is concluded about the cause?

8.2 For the same four rules, what does backward chaining ask, in which order, to prove `cause = empty tank` for a driver whose engine does not start and whose fuel gauge shows empty? Why is the horn never asked?

8.3 Compute with Equation (8.1) and Equation (8.2) the CF with which the fault finder concludes *dirty CVT clutch* for a scooter that starts, has low top speed and shudders. Show both steps.

8.4 A driver runs `check "cause = dirty CVT clutch"`. List the questions the finder asks after the safety check, in order, if the answers are: the engine starts, the top speed is low, it shudders.

8.5 In Section 8.6.1, the *blown main fuse* rule was tried before the *weak battery* rule. Explain from the rules and their CFs why, and why trying it cost no extra question.

8.6 In Section 8.6.2, what CF would the weak battery have if the engine combined the two rules for the electrics with the MYCIN formula of Section 8.7.3 before firing the battery rule?

8.7 Mas Yudi's colleague rides a Honda PCX 160, which is liquid-cooled. Write the case you would add to `fault-cases.json`, and explain what the finder would ask a PCX rider whose temperature light is on, before and after your change.

8.8 Design two new cases for the rainy season: a scooter that *starts but dies after wading through a flood*, and one whose *CVT makes a loud noise after rain*. For each, give facts, conclusions, a CF and the urgency level of **Figure 8.2**, and say whether the finder should ever advise the driver to ride on.

8.12 Solutions to the Exercises

Solution 8.1. First pass: R1 does not hold (no horn fact); R2 holds (`starter sound = slow`) and fires, `electrics = weak power`; R3 now holds (`engine starts = no`, `electrics = weak power`) and fires, `cause = weak battery`; R4 would hold too, but its conclusion variable `cause` is already known, so it does not fire. Second pass: nothing new; stop. The finder concludes *weak battery*, although the tank is empty — forward chaining in this small engine takes the first cause it reaches. The real fault finder tries rules in order of certainty and would still report only one cause; a driver who sees an empty gauge should fill up first, which is why Exercise 8.2 matters.

Solution 8.2. Goal `cause = empty tank`: the only rule is R4. Its first condition, `engine starts = no`, is concluded by no rule, so it is asked (no). Its second, `fuel gauge = empty`, is asked (empty). R4 is proved. The horn is never asked because no rule for the goal, and no rule for any of its conditions, mentions the horn: backward chaining only follows the rules that can lead to the goal.

Solution 8.3. First, the CVT rule: its clauses have CF 100 and its node CF 90, so by Equation (8.1) its CF is $100 \times 90/100 = 90$; the driver's answers have CF 100, so by Equation (8.2) `transmission = CVT problem` has CF $100 \times 90/100 = 90$. Then the clutch rule: its CF is $100 \times 80/100 = 80$; its facts have CF 90 (the CVT problem) and 100 (the shudder), so the conclusion has CF $\min(90,100) \times 80/100 = 72$.

Solution 8.4. The goal's only rule is `dirty CVT clutch#1`, which needs `transmission = CVT problem` and `shudder = yes`. The first is a subgoal: the rule `CVT problem#1` asks *Does the engine start?* (yes) and *Can it still reach its usual speed?* (low), and fires. Then *Does it shudder when you pull away?* (yes). Three questions after the two of the safety check, five in all.

Solution 8.5. Backward chaining for `cause` tries the rules that conclude a cause in order of decreasing CF, and for equal CF in the order of the case file. The two safety rules (CF 100) come first, then *blown main fuse* (CF 100), which precedes *weak battery* (CF 100) in the file. The fuse rule's first condition, `engine starts = no`, is asked, and the weak-battery rule needs the same fact; its second, `electrics = no power`, is a subgoal, whose first rule asks about the horn — and that same answer then proves `electrics = weak power`, which the weak-battery rule also needs. Both questions asked for the fuse rule would have been asked for the battery rule anyway, so trying the fuse first cost nothing.

Solution 8.6. Combined, the electrics would have CF $80 + 90 \times (100 - 80)/100 = 98$. The battery rule has CF 100 and its facts would have CF 100 (the engine) and 98 (the electrics), so the weak battery would be concluded with CF $\min(100,98) \times 100/100 = 98$ instead of 80.

Solution 8.7. Add the line `{"facts": ["model = Honda PCX 160"], "conclusions": ["cooling = liquid-cooled"], "option": 5}` (and add *Honda PCX 160* to the choices of the model question). Before the change, the model rules fail for a PCX, `cooling` cannot be proved, the finder cannot ask for it (it is concluded by rules), so the overheating rule fails and the finder goes on to other causes, ending with "I cannot name the fault" if none holds. After the change, the answer *Honda PCX 160* proves `cooling = liquid-cooled` and the overheating rule fires, with the same five questions as for Mbak Wulan's NMAX.

Solution 8.8. One possible design. Flood: facts `engine starts = yes`, `dies after flood = yes`; conclusions `cause = water in the air filter or the exhaust`, `advice = do not keep starting it; push it out of the water and take it to a bengkel today`; CF 80; urgency *bengkel today*. Repeatedly cranking an engine that has sucked in water can damage it badly, so the advice must not be "try again". CVT noise: facts `engine starts = yes`, `CVT noise after rain = yes`; conclusions `transmission = CVT problem`, then a separate case `transmission = CVT problem`, `CVT noise after rain = yes` → `cause = water in the CVT housing`, `advice = ride gently to a bengkel this week to have the CVT drained and cleaned`; CF 70; urgency *this week*. The finder should never advise riding on through deep water; for the CVT, riding gently to the bengkel is acceptable only if the noise does not come with a loss of drive.

References

[1] F. Sandi, "Ramalan nasib penjualan motor tahun 2026 usai naik tipis di 2025," *CNBC Indonesia*, Jan. 12, 2026. [Online]. Available: https://www.cnbcindonesia.com/news/20260112131535-4-701584/ramalan-nasib-penjualan-motor-tahun-2026-usai-naik-tipis-di-2025

[2] Warta Ekonomi, "Sepeda motor terlaris di Indonesia sepanjang 2025," Dec. 29, 2025. [Online]. Available: https://wartaekonomi.co.id/read595218/sepeda-motor-terlaris-di-indonesia-sepanjang-2025

[3] A. Wibowo, "5 motor paling irit dan nyaman untuk ojek online 2026," *Garduoto*, Sep. 3, 2026. [Online]. Available: https://www.garduoto.com/142984/5-motor-paling-irit-dan-nyaman-untuk-ojek-online-2026

[4] M. M. I. Subakti, "A Variable-Centered Intelligent Rule System," M.Sc.Eng. thesis, Dept. of Computer Science and Information Engineering, Taipei, Taiwan, Jun. 2005.

[5] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Hoboken, NJ, USA: Pearson, 2020.

[6] I. Subakti, "A variable-centered intelligent rule system," in *Proc. 1st Annual Int. Conf. Information and Communication Technology Seminar (ICTS 2005)*, vol. 1, no. 1, Surabaya, Indonesia, Aug. 2005, pp. 167–174.

[7] I. Subakti and R. Hidayatullah, "Aplikasi sistem pakar untuk diagnosis awal gangguan kesehatan secara mandiri menggunakan Variable-Centered Intelligent Rule System," vol. 6, no. 1, pp. 11–16, Jan. 2007.

[8] G. R. Baur and D. V. Pigford, *Expert Systems for Business: Concepts and Applications*. Boston, MA, USA: Boyd & Fraser, 1990.

[9] Wahana Honda, "Komponen motor matic yang wajib diganti berkala," Wahana Honda blog. [Online]. Available: https://www.wahanahonda.com/blog/komponen-motor-matic-yang-wajib-diganti-berkala (accessed Sep. 2026).

Chapter 9

Cooperative-Loan Eligibility Advisor: Fuzzy VCIRS

By the end of this chapter you will be able to explain why words such as *low*, *medium* and *high* have soft edges, and how a **fuzzy set** gives each number a degree of belonging to a word; apply the hedges *very* and *somewhat*; compute the degree to which a member's repayment ratio is *low* by hand and check it against the engine's fuzzy hooks; tell apart the two kinds of imprecision of the fuzzy VCIRS paper, **fuzziness** and **uncertainty**, and carry uncertainty as a small **fuzzy number**; and run the **loan advisor** of a savings-and-loan cooperative, which prepares the committee's monthly meeting with a recommendation for every application, the reason for it in plain words, a warning when a case is borderline, and the loan amount that would change the answer. The advisor is built to be fair and transparent: it refuses to learn anything from protected attributes such as gender, religion or ethnicity, and every answer can be traced to the numbers that produced it.

9.1 The Committee of Koperasi Sumber Makmur

Koperasi Sumber Makmur is a savings-and-loan cooperative in a village south of Malang. Its four hundred members are vegetable sellers, rice farmers, tailors, warung owners, ojek drivers and caterers. Every month each member pays a small compulsory saving, and from the pooled savings the cooperative lends to its members — only to its members, which makes it what regulators call a *closed-loop* cooperative. Cooperatives that also serve the public are *open-loop*, and under the financial-sector law of 2023 (UU P2SK) they are moving under the supervision of the Financial Services Authority, OJK; the Ministry of Cooperatives handed OJK its list of such cooperatives in January 2025 [1]. Savings-and-loan business inside cooperatives is governed by the Minister of Cooperatives' regulation on savings and loans, Permenkop UKM No. 8 of 2023 [2], under the Cooperatives Law of 1992, which is still in force while a new law is being prepared; in July 2026 the minister said it was targeted to pass within the year [3], [4]. Cooperatives are also growing fast: by July 2026, 83,382 village cooperatives had been formed under the Koperasi Desa Merah Putih program [5], and many of them lend to their members. Their members' incomes are as varied as their work: a vegetable seller earns a little every day, a tailor earns by the order, a civil servant's spouse by the month, and a rice farmer mostly twice a year, at harvest. A good loan decision has to see those differences, and a fair one must see nothing else.

On the first Saturday of every month the committee — the *pengurus* — meets in the village hall to decide the month's loan applications. Bu Wati, the treasurer, prepares the files. She has done this for eleven years, and she knows what the committee usually decides: a member whose installments would eat a small part of her income gets the loan; one whose installments would eat half of it is asked to borrow less or to save first; a new member starts with a small loan; a member who is often late is discussed in person. What she wants is not a machine that decides — in a cooperative, the members' meeting and the committee decide, and that is how it should be. She wants a **consistent first reading** of every file, with the reasons written down, so that the

committee spends its time on the hard cases, so that two members with the same situation get the same answer, and so that every member can be told *why*.

There is a difficulty that anyone who has sat in such a meeting knows. The committee's knowledge is written in words: "if the repayment is **low** compared with the income", "if the member has **enough** savings", "if she is **rarely** late". But the files contain numbers: an income of Rp 3,600,000 a month, installments of Rp 1,010,000, one late payment last year. Where does *low* end? If the answer is "at 25 percent", then a member at 24.9 percent and one at 25.1 percent get different answers although nobody could honestly say their situations differ. This chapter's answer is the one the fuzzy VCIRS paper proposed: let the words keep their soft edges, and let the engine reason with *degrees* [6].

> **PITFALL.** The advisor in this chapter prepares a decision; it does not make one, and it is not financial advice to anyone. The committee decides each loan, members decide whether to borrow, and the cooperative's own rules, set by its members' meeting, always come first.

9.2 The Idea in Plain Words

9.2.1 Words with soft edges

A classical set has a hard edge: a number is in it or not. The **fuzzy set** of Lotfi Zadeh [7] lets a number belong to a set *to a degree* between 0 and 1. For the word *low* applied to the repayment ratio, a ratio of 15 percent is low to degree 1, a ratio of 35 percent to degree 0, and a ratio of 25 percent to degree 0.5 — half low, half not. The rule that turns a number into a degree is the **membership function** of the word. We use such words all day without noticing. A siomay seller says the price is *a bit expensive* at Rp 15,000 and *cheap* at Rp 10,000, and nobody asks him at which rupiah one becomes the other. A driver says the market is *close* at two kilometers and *far* at ten. The words are useful precisely because they are soft: they carry a judgment that does not change abruptly with a small change in the number. Fuzzy sets write that judgment down so that a program can use it the same way every time.

The simplest useful membership function is a **trapezoid**, given by four numbers $a \le b \le c \le d$: the degree is 0 up to a, rises in a straight line to 1 at b, stays 1 until c, and falls back to 0 at d:

$$\mu(x) = \begin{cases} 1 & b \le x \le c \\ \frac{x-a}{b-a} & a < x < b \\ \frac{d-x}{d-c} & c < x < d \\ 0 & \text{otherwise} \end{cases} \tag{9.1}$$

A triangle is a trapezoid with $b = c$, and a set that starts at the very left, such as *low*, has $a = b = 0$. The cooperative's committee agreed on three words for the repayment ratio: *low* is (0, 0, 20, 30), *medium* is (20, 30, 30, 40) and *high* is (30, 40, 100, 100). **Figure 9.1** draws them. Between 20 and 30 percent a ratio is partly low and partly medium; between 30 and 40, partly medium and partly high. There is no single point at which the committee's opinion jumps.

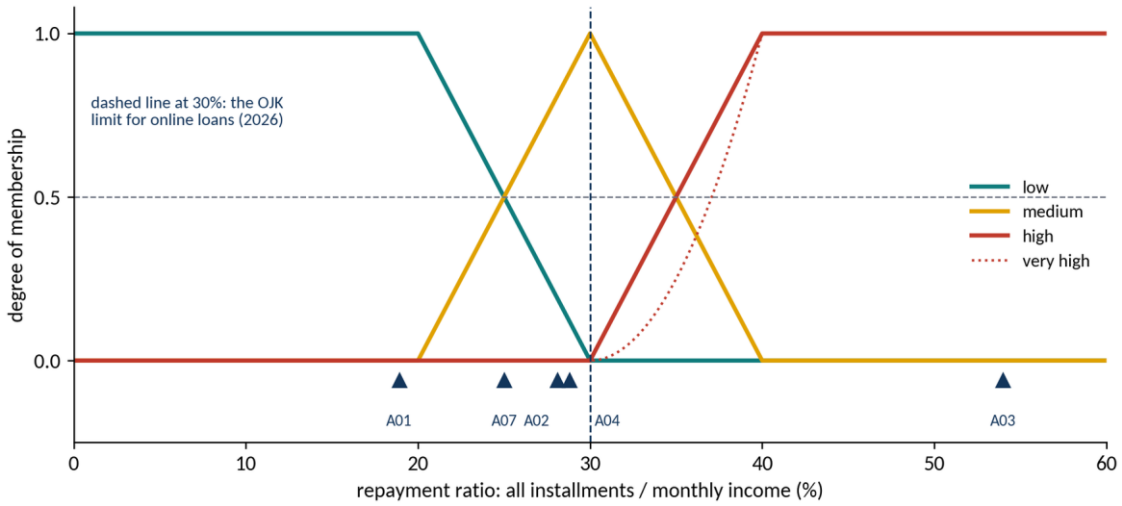

Figure 9.1 The three fuzzy words for the repayment ratio, with the hedge *very high* (dotted). The horizontal dashed line is the threshold 0.5 at which a fuzzy condition counts as true; the triangles mark the ratios of five members of Section 9.6.

The committee described the other numbers in the same way: *membership* is *new* (0, 0, 6, 12) or *established* (6, 12, 1200, 1200) months; *savings cover*, the member's savings as a percentage of the loan, is *small* (0, 0, 10, 20) or *enough* (10, 20, 1000, 1000); *late payments* in the last twelve months are *rare* (0, 0, 1, 3) or *frequent* (1, 3, 12, 12). The last number of a set such as 1200 only has to be larger than any value that can occur. All nine sets live in the file `fuzzy-sets.csv`, one line each, where the committee can read and change them.

9.2.2 Very, somewhat, not

People sharpen and soften their words with **hedges**: *very* high, *somewhat* low, *not* low. Zadeh proposed treating a hedge as an operation on the degree [7]. *Very* concentrates the word by squaring the degree; *somewhat* dilates it by taking the square root; *not* takes the complement:

$$\mu_{\text{very } A}(x) = \mu_A(x)^2, \qquad \mu_{\text{somewhat } A}(x) = \sqrt{\mu_A(x)}, \qquad \mu_{\text{not } A}(x) = 1 - \mu_A(x) \tag{9.2}$$

So a ratio that is high to degree 0.5 is *very high* only to degree 0.25, and a ratio that is low to degree 0.5 is *somewhat low* to degree 0.71. The vcirs-core engine understands *very*, *extremely* (the cube), *somewhat*, *rather* and *fairly* (the square root) and *not*, written in front of a term in a condition, such as `repayment ratio = very high`.

Hedges also make the committee's knowledge easier to read aloud. "If the repayment ratio is *somewhat low* and the member is established, recommend" sounds like a sentence a committee member would say, and it means exactly what she means: a little more generous at the border than plain *low*. Because a hedge is only an operation on a degree, it needs no new fuzzy set and no new numbers in `fuzzy-sets.csv`; the three words the committee agreed on carry all their shades.

9.2.3 Fuzzy conditions in VCIRS

The engine's fuzzy hook is small, and it is enough [6], [8]. A variable may have fuzzy sets. When a condition such as `repayment ratio = medium` meets a fact that is a *number*, such as `repayment ratio = 28.1`, the condition is not simply true or false: it is true **to the degree** $\mu_{\text{medium}}(28.1) = 0.81$. The condition **holds** when the degree reaches the knowledge base's threshold, 0.5 by

default. When a node fires, the degree becomes part of the certainty of its conclusion — the fuzzy form of the classical certainty rule [9] used in Chapter 8:

$$CF_{\text{conclusion}} = \frac{\min_i(CF_{\text{fact } i} \times \mu_i) \times CF_{\text{rule}}}{100} \tag{9.3}$$

Here μ_i is the degree of the i-th condition (1 for a crisp condition). A member whose ratio is medium to degree 0.81, with a fact of CF 100 and a rule of CF 100, gets the rule's advice with CF $100 \times 0.81 \times 100/100 = 81$. The advice is not only given; it is given *with a strength* that says how clearly the member's numbers fit the words of the rule.

Two practical consequences follow. First, the committee's knowledge can be taught in words, exactly as the committee speaks: a case says `repayment ratio = medium`, `savings cover = enough`, and nothing about percentages. Second, a member exactly on the border between two words — a ratio of 25 percent, low to degree 0.5 and medium to degree 0.5 — satisfies *both* conditions, and the engine can be asked to follow both. That is how the advisor finds borderline cases.

How can the same knowledge base be taught in words and consulted with numbers? Because the engine decides how to compare by looking at the fact. When Bu Wati teaches the case `repayment ratio = medium`, the fact is the word *medium*, and a condition `repayment ratio = medium` is simply equal to it, as in every earlier chapter; the case is learned, checked against its cornerstone and placed by the ordinary RDR steps. When a member's file is consulted, the fact is the number 28.1, and the same condition is measured through the fuzzy set. Nothing in the knowledge base has to be converted. The words are the knowledge; the fuzzy sets are the bridge from the numbers to the words, and the committee can move the bridge without touching the knowledge.

9.2.4 Two kinds of imprecision

The fuzzy VCIRS paper points out that a rule-based system meets two different kinds of imprecision, and treats them with two different tools [6]. **Fuzziness** is in the words: *low* has no sharp edge. That is what fuzzy sets and degrees handle. **Uncertainty** is in the facts and the rules: we are not sure the member's income is really Rp 3,600,000, and the committee is not sure its rule is always right. The paper models uncertainty with **fuzzy numbers**, such as *about 0.8* or *close to 1.0*, and combines them with fuzzy arithmetic; the certainty of a conclusion is the fuzzy product of the uncertainty of the rule and that of the fact.

A **triangular fuzzy number** (a, b, c) says "about b, certainly not below a nor above c". For positive triangles the product is well approximated by multiplying the corners:

$$(a_1, b_1, c_1) \otimes (a_2, b_2, c_2) \approx (a_1 a_2,\ b_1 b_2,\ c_1 c_2) \tag{9.4}$$

The advisor uses this in one place where it matters most: how sure is the income? When a member shows a sales book, a pay slip or bank statements, Bu Wati marks the income *verified*, about 0.95, between 0.85 and 1; when the member only states it, *stated*, about 0.7, between 0.5 and 0.9. The advice's CF, read as a crisp number (for example 0.81 as the triangle (0.81, 0.81, 0.81)), is multiplied by the income's triangle, and the result is printed as "how sure: about 0.57 (0.41 to 0.73)". A committee member sees at once whether a recommendation rests on firm ground.

9.2.5 The repayment ratio and the 30 percent line

The most important number in any loan decision is how much of the borrower's income the installments will take. The advisor calls it the **repayment ratio**: all the member's monthly installments — to the cooperative and elsewhere — including the new loan's, divided by the monthly income:

$$r = \frac{E + I}{M} \times 100, \qquad I = \left\lceil \frac{A}{n} \right\rceil + A \times 1\% \tag{9.5}$$

where E is the existing installments, M the monthly income, A the amount of the new loan, n the number of months and I the new installment: the amount divided by the months, rounded up to a whole rupiah, plus the cooperative's flat interest of 1 percent of the amount per month (the rate is the example cooperative's; each cooperative's members' meeting sets its own).

Where should *low* end? Regulators give a useful reference. For online lending platforms (*pindar*), OJK has been tightening the limit on a borrower's total repayments step by step, to 30 percent of income in 2026 [10]. That rule does not bind a closed-loop cooperative, but it is a sensible yardstick, and the committee chose its words around it: at 30 percent a ratio is no longer low at all and fully medium; the rule of thumb "keep it below a quarter of the income" is the point where *low* and *medium* cross.

Notice that the ratio counts *all* installments, not only the cooperative's. A member who already pays a motorcycle installment to a dealer, a phone installment to a shop and a small online loan may look comfortable to the cooperative alone and be stretched thin in reality. Bu Wati asks every applicant to list every installment, and she writes down what the member says even when it cannot be checked — the income proof line will show how firm the file is.

9.2.6 Fair and transparent by construction

A loan advisor decides who gets money, and money changes lives. Indonesia's financial consumer-protection regulation, POJK No. 22 of 2023, lists fair treatment and transparency among the principles that financial service providers must follow [11], and a cooperative that wants its members' trust should hold itself to the same standard. The advisor does three things for fairness and transparency, and they are built into its structure rather than left to good intentions.

First, **no protected attributes**. The advisor looks only at five facts, all about the ability to repay: the repayment ratio, the months of membership, the savings cover, the number of late payments and the income pattern (daily, monthly or at harvest). The application form also has the member's name, village and the like; those are shown to the committee and never given to the engine. And the knowledge base itself refuses any case that mentions a protected attribute — religion, ethnicity, gender, marital status, age, disability, political view or place of origin, in English or Indonesian. A cooperative that wants to be careful can also watch for **proxies**: a harmless-looking variable that stands in for a protected one, such as the name of a hamlet where one group lives. The rule is simple: every variable must be about repayment, and the committee must be able to say why.

Second, **reasons for every answer**. Each recommendation comes with the rule that produced it and the degree to which the member's numbers fit each word, as the engine's How explanation. A member who asks "why was I told to borrow less?" can be shown: *your repayment ratio of 28.1 percent is medium to degree 0.81, and the committee's rule for a medium ratio is to recommend a smaller amount.*

Third, **what would change the answer**. For every member whose ratio is not low, the advisor computes the largest loan, in steps of Rp 100,000, that would bring the ratio down to *low* (25 percent or less). “Rp 4,800,000 would be low for you” is more useful and more respectful than “no”.

Together these make the advisor **auditable**. Its knowledge is a readable file of cases in words; its fuzzy sets are a readable file of numbers; every change is recorded in the history with the name of whoever taught it; and every recommendation can be reproduced from the application and the knowledge base of that month. When a member, an auditor or a supervisor asks how a decision was prepared, the cooperative can show them, line by line.

Figure 9.2 shows the whole path of an application.

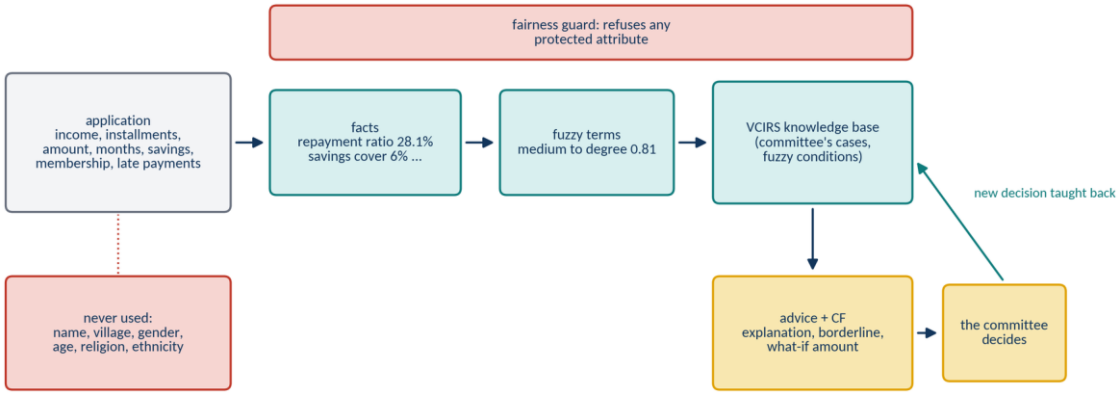

Figure 9.2 The path of an application through the advisor. Only facts about repayment reach the engine; a fairness guard refuses any knowledge that uses a protected attribute; the advice comes with its CF, its explanation, a borderline warning and a what-if amount; the committee decides, and its decisions are taught back.

9.2.7 One member, from start to finish

Before any code, follow one file by hand, the way the advisor will. Mas Joko, an ojek driver, asks for Rp 6,000,000 over 12 months. His income is Rp 3,600,000 a month, stated but not documented; he already pays Rp 450,000 a month for his motorcycle; his savings are Rp 400,000; he has been a member for 30 months and was late once last year.

Step 1, the numbers. By Equation (9.5) his new installment is $I = 500{,}000 + 60{,}000 = 560{,}000$ and his repayment ratio is $r = 1{,}010{,}000/3{,}600{,}000 \times 100 \approx 28.1$ percent. His savings cover 400,000/6,000,000, 6 percent of the loan.

Step 2, the words. By Equation (9.1), a ratio of 28.1 is *low* to degree 0.19 and *medium* to degree 0.81. Thirty months of membership are *established* to degree 1 and *new* to 0. A savings cover of 6 percent is *small* to degree 1 and *enough* to 0. One late payment is *rare* to degree 1 and *frequent* to 0.

Step 3, the rules. The *low* node needs a degree of 0.5; 0.19 is not enough, so neither it nor its exceptions fire. The *medium* node holds with 0.81 and fires: *recommend a smaller amount*. Its exception needs *enough* savings, degree 0: it does not fire. The *high* node needs *high*, degree 0.

Step 4, the strength. By Equation (9.3), $CF = 100 \times 0.81 \times 100/100 = 81$. The income is only stated, so by Equation (9.4) the recommendation is *about 0.57*, between 0.41 and 0.73.

Step 5, the way out. For the ratio to be at most 25 percent, all his installments may be at most $0.25 \times 3{,}600{,}000 = 900{,}000$, so the new one at most 450,000. An installment is about $A/12 +$

$A/100$ of the amount A, so A can be at most about $450{,}000/(1/12 + 1/100) \approx 4{,}821{,}000$; in steps of Rp 100,000 that is Rp 4,800,000.

Five small steps, each of which a committee member can check with a calculator. Section 9.6 shows the advisor printing exactly these numbers.

9.2.8 Why not a credit score?

Banks and online lenders usually decide with a **credit score**: a statistical model, trained on thousands of past borrowers, that turns an application into a number. For a large lender with a large history that works well. A village cooperative has neither the history nor the statisticians, and it has something a score lacks: a committee that knows its members and can say why it decides. The VCIRS approach keeps the knowledge in the committee's own words and grows it one decision at a time, exactly as Ripple Down Rules grew the Wi-Fi doctor's knowledge. Every rule was put there by a person for a reason, every answer can be read aloud, and a wrong answer is corrected where it happened, without retraining anything.

There is also a fairness argument. A statistical model can learn to use a protected attribute through a proxy without anyone noticing, because nobody reads its weights. A knowledge base of a few dozen readable rules, with a guard that refuses protected variables, can be inspected by any member of the cooperative in ten minutes.

9.3 Membership by Hand

As in the earlier chapters, we first compute the degrees ourselves. **Listing 9.1** writes the trapezoid of Equation (9.1) in a dozen lines, applies the hedges of Equation (9.2) by hand, and asks the engine for the same degrees through a condition, to check that they agree.

Listing 9.1 `FuzzyTerms.java`: membership degrees and hedges by hand, checked against vcirs-core.

```
import java.util.List;
import java.util.Map;

import vcirs.Condition;
import vcirs.Fact;
import vcirs.KnowledgeBase;
import vcirs.Num;

/**
 * How true is "the repayment ratio is low"? Membership degrees by hand,
 * with the hedges "very" and "somewhat", checked against vcirs-core.
 */
public class FuzzyTerms {
    static final String[] NAMES = {"low", "medium", "high"};
    static final double[][] POINTS = {{0, 0, 20, 30}, {20, 30, 30, 40},
        {30, 40, 100, 100}};

    /** A trapezoid: 0 up to a, rising to 1 at b, 1 until c, 0 again at d. */
    static double membership(double x, double[] p) {
        double a = p[0], b = p[1], c = p[2], d = p[3];
        if (b <= x && x <= c) {
            return 1.0;
        }
        if (a < x && x < b) {
            return (x - a) / (b - a);
        }
        if (c < x && x < d) {
            return (d - x) / (d - c);
        }
        return 0.0;
    }

    static String pad(String s, int width) {
        return String.format("%-" + width + "s", s);
    }
```

Listing 9.1 (Continued) *FuzzyTerms.java: membership degrees and hedges by hand, checked against vcirs-core.*

```
    public static void main(String[] args) {
        KnowledgeBase kb = new KnowledgeBase("terms");
        for (int i = 0; i < NAMES.length; i++) {
            kb.defineFuzzy("repayment ratio", NAMES[i], POINTS[i]);
        }
        System.out.println("ratio  low   medium  high  very high  somewhat low"
                + "  engine agrees");
        List<String> asked = List.of("low", "medium", "high", "very high",
                "somewhat low");
        for (double x : new double[] {15, 20, 25, 28.1, 30, 33.5, 35, 40, 45}) {
            double low = membership(x, POINTS[0]);
            double med = membership(x, POINTS[1]);
            double high = membership(x, POINTS[2]);
            double[] byHand = {low, med, high, high * high, Math.sqrt(low)};
            Map<String, Fact> fact = Map.of("repayment ratio",
                    new Fact("repayment ratio", Num.fmt(x, 1)));
            boolean same = true;
            for (int i = 0; i < asked.size(); i++) {
                double e = kb.evaluate(new Condition("repayment ratio", "=",
                        asked.get(i)), fact);
                same = same && Num.key(e) == Num.key(byHand[i]);
            }
            String[] cells = new String[5];
            for (int i = 0; i < 5; i++) {
                cells[i] = Num.fixed(byHand[i], 2);
            }
            System.out.println(String.format("%5s", Num.fmt(x, 1)) + "  "
                    + cells[0] + "  " + pad(cells[1], 6) + "  " + cells[2]
                    + "  " + pad(cells[3], 9) + "  " + pad(cells[4], 12) + "  "
                    + (same ? "yes" : "NO"));
        }
    }
}
```

`membership` is Equation (9.1) line for line. `main` defines the three words for the engine with `defineFuzzy`, then, for nine ratios, computes by hand the degrees of *low*, *medium*, *high*, *very high* (the square of *high*) and *somewhat low* (the square root of *low*), and asks the engine to `evaluate` the conditions `repayment ratio = low`, `= medium`, `= high`, `= very high` and `= somewhat low` against the fact. Numbers are compared through `Num.key`, the engine's way of deciding that two doubles are equal. **Listing 9.2** is the Python version.

Listing 9.2 `fuzzy_terms.py`: the same table in Python.

```
"""How true is "the repayment ratio is low"? Membership degrees by hand,
with the hedges "very" and "somewhat", checked against vcirs-core."""

import math
import pathlib
import sys

sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import Condition, Fact, KnowledgeBase, num

TERMS = {"low": (0, 0, 20, 30), "medium": (20, 30, 30, 40),
         "high": (30, 40, 100, 100)}


def membership(x, points):
    """A trapezoid: 0 up to a, rising to 1 at b, 1 until c, 0 again at d."""
    a, b, c, d = points
    if b <= x <= c:
        return 1.0
    if a < x < b:
        return (x - a) / (b - a)
    if c < x < d:
        return (d - x) / (d - c)
    return 0.0


kb = KnowledgeBase("terms")
for term, points in TERMS.items():
    kb.define_fuzzy("repayment ratio", term, list(points))

print("ratio  low   medium  high  very high  somewhat low  engine agrees")
for x in [15, 20, 25, 28.1, 30, 33.5, 35, 40, 45]:
    low, med, high = (membership(x, TERMS[t]) for t in TERMS)
    by_hand = [low, med, high, high * high, math.sqrt(low)]
    fact = {"repayment ratio": Fact("repayment ratio", num.fmt(x, 1))}
    engine = [kb.evaluate(Condition("repayment ratio", "=", t), fact)
              for t in ["low", "medium", "high", "very high", "somewhat low"]]
    same = all(num.key(a) == num.key(b) for a, b in zip(by_hand, engine))
    cells = [num.fixed(v, 2) for v in by_hand]
    print(num.fmt(x, 1).rjust(5) + "  " + cells[0] + "  " + cells[1].ljust(6)
          + "  " + cells[2] + "  " + cells[3].ljust(9) + "  "
          + cells[4].ljust(12) + "  " + ("yes" if same else "NO"))
```

Both print:

```
ratio  low   medium  high  very high  somewhat low  engine agrees
   15  1.00  0.00    0.00  0.00       1.00          yes
   20  1.00  0.00    0.00  0.00       1.00          yes
   25  0.50  0.50    0.00  0.00       0.71          yes
 28.1  0.19  0.81    0.00  0.00       0.44          yes
   30  0.00  1.00    0.00  0.00       0.00          yes
 33.5  0.00  0.65    0.35  0.12       0.00          yes
   35  0.00  0.50    0.50  0.25       0.00          yes
   40  0.00  0.00    1.00  1.00       0.00          yes
   45  0.00  0.00    1.00  1.00       0.00          yes
```

Read the rows as the committee would. At 15 and 20 percent the ratio is simply low. At 25 it is exactly half low and half medium, and *somewhat low* to 0.71: a member here is on the fence, and the advisor will say so. At 28.1, Mas Joko's ratio in Section 9.6, it is medium to 0.81 and low only to 0.19, below the threshold. At 33.5 it is medium to 0.65 and high to 0.35 — and *very high* to only 0.12, because squaring a small degree makes it much smaller: *very* is a demanding word. From 40 up it is simply high. In every row the engine agrees with the hand computation.

Why trapezoids, and not smooth curves such as bells? Because a committee can set a trapezoid with four numbers it understands — "definitely low up to 20, definitely not low from 30" — and check it with a pencil. Smooth curves give almost the same advice and are much harder to explain to a member. The fuzzy VCIRS paper, too, builds its fuzzy terms from simple shapes [6]. Simplicity is part of transparency. A shape that can be drawn on the blackboard of the village hall is a shape the whole committee can argue about and agree on.

9.4 The Cooperative's Arithmetic

Listing 9.3 holds everything the advisor computes from an application, in whole rupiah so that Java and Python agree to the last digit.

Listing 9.3 `Loans.java`: installments, the repayment ratio and the what-if amount, in whole rupiah.

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Locale;
import java.util.Map;

/**
 * Loan arithmetic of the cooperative, in whole rupiah. The cooperative
 * lends at a flat 1% of the amount per month (its members' meeting sets the
 * rate). An installment is the amount divided by the number of months,
 * rounded up, plus the monthly interest.
 */
public class Loans {
    static final long RATE_BP = 100;  // interest per month, in 1/100 of a %

    static long installment(long amount, long months) {
        return (amount + months - 1) / months + amount * RATE_BP / 10000;
    }

    /** 100 * numerator / denominator, rounded to one decimal, as tenths. */
    static long tenths(long numerator, long denominator) {
        return (numerator * 1000 + denominator / 2) / denominator;
    }

    /** 253 -> "25.3" */
    static String decimal(long t) {
        return t / 10 + "." + t % 10;
    }

    /** 4800000 -> "Rp 4,800,000" */
    static String rupiah(long x) {
        return "Rp " + String.format(Locale.ROOT, "%,d", x);
    }

    static class Application {
        static final String[] FIELDS = {"id", "name", "work", "income pattern",
            "monthly income", "existing installments", "amount", "months",
            "savings", "membership months", "late payments", "income proof"};
        final Map<String, String> data = new LinkedHashMap<>();
```

Listing 9.3 (Continued) *Loans.java: installments, the repayment ratio and the what-if amount, in whole rupiah.*

```
        Application(String line) {
            String[] parts = line.strip().split(",");
            for (int i = 0; i < FIELDS.length; i++) {
                data.put(FIELDS[i], parts[i]);
            }
        }

        long number(String field) {
            return Long.parseLong(data.get(field));
        }

        /** The repayment ratio: all installments / income, in tenths of %. */
        long ratioTenths(long amount) {
            long total = number("existing installments")
                    + installment(amount, number("months"));
            return tenths(total, number("monthly income"));
        }

        long ratioTenths() {
            return ratioTenths(number("amount"));
        }

        /**
         * What the advisor may look at. Names, villages, gender, age or
         * religion are never facts.
         */
        Map<String, String> facts() {
            Map<String, String> f = new LinkedHashMap<>();
            f.put("repayment ratio", decimal(ratioTenths()));
            f.put("membership", String.valueOf(number("membership months")));
            f.put("savings cover",
                    String.valueOf(number("savings") * 100 / number("amount")));
            f.put("late payments", String.valueOf(number("late payments")));
            f.put("income pattern", data.get("income pattern"));
            return f;
        }
```

Listing 9.3 (Continued) *Loans.java: installments, the repayment ratio and the what-if amount, in whole rupiah.*

```
        /**
         * The largest amount, in steps of Rp 100,000, that keeps the
         * repayment ratio at or below the limit (null if none does).
         */
        Long largestLow(long limitTenths) {
            long amount = number("amount");
            while (amount > 0 && ratioTenths(amount) > limitTenths) {
                amount -= 100000;
            }
            return amount > 0 ? amount : null;
        }
    }

    static List<Application> readApplications(String path) {
        List<Application> out = new ArrayList<>();
        try {
            for (String line : Files.readAllLines(Path.of(path),
                    StandardCharsets.UTF_8)) {
                if (!line.isBlank() && !line.startsWith("#")) {
                    out.add(new Application(line));
                }
            }
        } catch (IOException e) {
            throw new IllegalStateException("cannot read " + path);
        }
        return out;
    }
}
```

`installment` is Equation (9.5)'s I: the amount divided by the months and rounded up, plus 1 percent of the amount (`RATE_BP` is the rate in hundredths of a percent, so the cooperative can change it in one place). `tenths` computes a percentage rounded to one decimal, as a whole number of tenths, with integers only; `decimal` writes 281 as `28.1`, and `rupiah` writes 4800000 as `Rp 4,800,000`.

An `Application` is one line of `applications.csv`:

```
# id,name,work,income pattern,monthly income,existing installments,amount,...
A01,Bu Siti,vegetable seller,daily,4500000,300000,5000000,10,1200000,48,0,verified
A02,Mas Joko,ojek driver,daily,3600000,450000,6000000,12,400000,30,1,stated
```

The remaining columns are the months of the loan, the member's savings, the months of membership, the late payments of the last twelve months and whether the income is *verified* or *stated*. `ratioTenths` applies Equation (9.5). `facts` is the most important method of the chapter for fairness: it builds the **only** facts the engine ever sees — the ratio, the membership months, the savings cover as a whole percentage, the late payments and the income pattern. `largestLow` steps the amount down by Rp 100,000 until the ratio is 25.0 percent or less. Why whole rupiah and tenths of a percent, instead of ordinary decimal numbers? Two reasons. Money must never be rounded differently in two places — a member's installment printed as Rp 560,000 on one page and Rp 559,999.99 on another destroys trust in the whole paper. And the Java and Python versions must print the same thing, which is only guaranteed when the arithmetic is done in integers; the only decimal numbers in the advisor are the degrees and certainties, which vcirs-core prints the same way in both languages. **Listing 9.4** is the Python version.

Listing 9.4 `loans.py`: the same arithmetic in Python.

```
"""Loan arithmetic of the cooperative, in whole rupiah.

The cooperative lends at a flat 1% of the amount per month (its members'
meeting sets the rate). An installment is the amount divided by the number
of months, rounded up, plus the monthly interest.
"""

RATE_BP = 100                     # interest per month, in 1/100 of a percent


def installment(amount, months):
    return (amount + months - 1) // months + amount * RATE_BP // 10000


def tenths(numerator, denominator):
    """100 * numerator / denominator, rounded to one decimal, as tenths."""
    return (numerator * 1000 + denominator // 2) // denominator


def decimal(t):
    """253 -> '25.3'"""
    return str(t // 10) + "." + str(t % 10)


def rupiah(x):
    """4800000 -> 'Rp 4,800,000'"""
    return "Rp " + "{:,}".format(x)


class Application:
    FIELDS = ["id", "name", "work", "income pattern", "monthly income",
              "existing installments", "amount", "months", "savings",
              "membership months", "late payments", "income proof"]

    def __init__(self, line):
        self.data = dict(zip(Application.FIELDS, line.strip().split(",")))

    def number(self, field):
        return int(self.data[field])
```

Listing 9.4 (Continued) *loans.py: the same arithmetic in Python.*

```
    def ratio_tenths(self, amount=None):
        """The repayment ratio: all installments / income, in tenths of %."""
        amount = self.number("amount") if amount is None else amount
        total = self.number("existing installments") \
            + installment(amount, self.number("months"))
        return tenths(total, self.number("monthly income"))

    def facts(self):
        """What the advisor may look at. Names, villages, gender, age or
        religion are never facts."""
        amount = self.number("amount")
        return {
            "repayment ratio": decimal(self.ratio_tenths()),
            "membership": str(self.number("membership months")),
            "savings cover": str(self.number("savings") * 100 // amount),
            "late payments": str(self.number("late payments")),
            "income pattern": self.data["income pattern"],
        }

    def largest_low(self, limit_tenths=250):
        """The largest amount, in steps of Rp 100,000, that keeps the
        repayment ratio at or below the limit (None if none does)."""
        amount = self.number("amount")
        while amount > 0 and self.ratio_tenths(amount) > limit_tenths:
            amount -= 100000
        return amount if amount > 0 else None


def read_applications(path):
    with open(path, encoding="utf-8") as f:
        return [Application(line) for line in f
                if line.strip() and not line.startswith("#")]
```

For Mas Joko, by Equation (9.5): $I = \lceil 6{,}000{,}000/12 \rceil + 60{,}000 = 560{,}000$; with $E = 450{,}000$ and $M = 3{,}600{,}000$, $r = 1{,}010{,}000/3{,}600{,}000 \times 100 = 28.06$, which rounds to 28.1 percent.

9.5 The Advisor

The committee's knowledge is seven cases in `loan-cases.json`, taught by Bu Wati from past decisions and built with `learn`, the everyday RDR loop of Chapter 3:

```
repayment ratio = low                          -> recommend
repayment ratio = medium                       -> recommend a smaller amount
repayment ratio = high                         -> not now: lower the amount or save first
repayment ratio = low, late payments = frequent      -> refer to the committee
repayment ratio = low, membership = new              -> start with a smaller first loan
repayment ratio = medium, savings cover = enough     -> recommend
repayment ratio = medium, savings cover = enough,
    late payments = frequent                         -> refer to the committee
```

(This is a summary; in the file each case is one line of JSON.) The first three become top-level nodes. The fourth and fifth are wrong answers of the first node, so `learn` hangs them under it as exceptions: *low, but often late* goes to the committee; *low, but a new member* starts small. The sixth is an exception of the medium node — a member with enough savings to cover a good part of the loan can borrow the full amount — and the seventh an exception of the sixth. Everything is in words; the numbers come only at advice time. **Listing 9.5** is the advisor.

Listing 9.5 `LoanAdvisor.java`: the fair, explained, fuzzy loan advisor.

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.Set;

import vcirs.Case;
import vcirs.CaseFile;
import vcirs.Condition;
import vcirs.Conclusion;
import vcirs.Facts;
import vcirs.InferenceResult;
import vcirs.KnowledgeBase;
import vcirs.Node;
import vcirs.Num;
import vcirs.RdrPath;

/**
 * Cooperative-loan eligibility advisor: fuzzy VCIRS with explanations.
 *   java LoanAdvisor advise        advice for every application
 *   java LoanAdvisor teach         learn the committee's decisions
 *   java LoanAdvisor fairness [cases.json]   check the knowledge
 * The advisor prepares the committee's meeting; the committee decides.
 */
public class LoanAdvisor {
    static final String DATA = "../data";
    static final String KB_FILE = DATA + "/loan-kb.json";
    static final List<String> PROTECTED = List.of("religion", "ethnicity",
            "gender", "marital status", "age", "disability", "political view",
            "place of origin", "agama", "suku", "jenis kelamin",
            "status perkawinan", "usia", "umur", "disabilitas",
            "pilihan politik", "asal daerah");
    // How sure the income figure is, as a triangular fuzzy number (a, b, c).
    static final Map<String, double[]> PROOF = Map.of(
            "verified", new double[] {0.85, 0.95, 1.0},
            "stated", new double[] {0.5, 0.7, 0.9});
```

Listing 9.5 (Continued) *LoanAdvisor.java: the fair, explained, fuzzy loan advisor.*

```
    /** Raised when knowledge would use a protected attribute. */
    static class FairnessException extends RuntimeException {
        FairnessException(String message) {
            super(message);
        }
    }

    static void checkFair(String variable, String where) {
        if (PROTECTED.contains(variable.strip().toLowerCase())) {
            throw new FairnessException(where + " uses the protected attribute '"
                    + variable + "'");
        }
    }

    static List<String> lines(String name) {
        List<String> out = new ArrayList<>();
        try {
            for (String line : Files.readAllLines(Path.of(DATA, name),
                    StandardCharsets.UTF_8)) {
                if (!line.isBlank() && !line.startsWith("#")) {
                    out.add(line.strip());
                }
            }
        } catch (IOException e) {
            throw new IllegalStateException("cannot read " + name);
        }
        return out;
    }

    static KnowledgeBase buildKb(String casesFile) {
        CaseFile cases = CaseFile.read(DATA + "/" + casesFile);
        for (int i = 0; i < cases.steps.size(); i++) {
            for (Condition fact : cases.steps.get(i).aCase.facts) {
                checkFair(fact.variable, "case " + (i + 1));
            }
        }
        KnowledgeBase kb = new KnowledgeBase(cases.name, cases.description);
        for (String line : lines("fuzzy-sets.csv")) {
            String[] p = line.split(",");
            kb.defineFuzzy(p[0], p[1], new double[] {Double.parseDouble(p[2]),
                Double.parseDouble(p[3]), Double.parseDouble(p[4]),
                Double.parseDouble(p[5])});
        }
        return cases.build(kb);
    }
```

Listing 9.5 (Continued) *LoanAdvisor.java: the fair, explained, fuzzy loan advisor.*

```
    static KnowledgeBase openKb() {
        return Files.exists(Path.of(KB_FILE)) ? KnowledgeBase.load(KB_FILE)
                : buildKb("loan-cases.json");
    }

    static void adviseOne(KnowledgeBase kb, Loans.Application app) {
        Map<String, String> d = app.data;
        Map<String, String> facts = app.facts();
        System.out.println(d.get("id") + " " + d.get("name") + " ("
                + d.get("work") + "): " + Loans.rupiah(app.number("amount"))
                + " over " + d.get("months") + " months");
        System.out.println("  facts: repayment ratio "
                + facts.get("repayment ratio") + "%, membership "
                + facts.get("membership") + " months, savings cover "
                + facts.get("savings cover") + "%, late payments "
                + facts.get("late payments") + ", income "
                + facts.get("income pattern"));
        InferenceResult result = kb.infer(Facts.map(facts), null, true);
        if (result.paths.isEmpty()) {
            System.out.println("  advice: none; the committee decides");
            return;
        }
        RdrPath first = result.paths.get(0);
        String advice = first.conclusions().get(0).value;
        System.out.println("  advice: " + advice + "  (CF "
                + Num.fmt(first.cf, 2) + ")");
        System.out.println("  because:");
        for (String line : result.how()) {
            System.out.println("    " + line);
        }
        List<String> others = new ArrayList<>();
        for (RdrPath p : result.paths.subList(1, result.paths.size())) {
            if (!p.conclusions().isEmpty()
                    && !p.conclusions().get(0).value.equals(advice)) {
                others.add(p.conclusions().get(0).value);
            }
        }
        if (!others.isEmpty()) {
            System.out.println("  BORDERLINE, also possible: "
                    + String.join("; ", others));
        }
        double[] t = PROOF.get(d.get("income proof"));
        System.out.println("  how sure (income " + d.get("income proof")
                + "): about " + Num.fmt(first.cf / 100.0 * t[1], 2) + " ("
                + Num.fmt(first.cf / 100.0 * t[0], 2) + " to "
                + Num.fmt(first.cf / 100.0 * t[2], 2) + ")");
```

Listing 9.5 (Continued) *LoanAdvisor.java: the fair, explained, fuzzy loan advisor.*

```
        Long best = app.largestLow(250);
        if (best != null && best < app.number("amount")) {
            System.out.println("  " + Loans.rupiah(best) + " would bring the "
                    + "repayment ratio down to low ("
                    + Loans.decimal(app.ratioTenths(best)) + "%)");
        }
    }

    /** The fuzzy term the number belongs to most (the first on a tie). */
    static String bestTerm(KnowledgeBase kb, String variable, String value) {
        if (!kb.fuzzySets.containsKey(variable)) {
            return value;
        }
        String best = null;
        double degree = -1;
        for (Map.Entry<String, Double> e : kb.fuzzify(variable,
                Double.parseDouble(value))) {
            if (Num.key(e.getValue()) > Num.key(degree)) {
                best = e.getKey();
                degree = e.getValue();
            }
        }
        return best;
    }

    static void teach(KnowledgeBase kb) {
        Map<String, Loans.Application> apps = new LinkedHashMap<>();
        for (Loans.Application a : Loans.readApplications(DATA
                + "/applications.csv")) {
            apps.put(a.data.get("id"), a);
        }
        for (String row : lines("decisions.csv")) {
            String[] r = row.split(",");
            Map<String, String> facts = apps.get(r[0]).facts();
            Condition chosen = Condition.parse(r[2]);
            checkFair(chosen.variable, "the decision for " + r[0]);
            List<Condition> terms = new ArrayList<>();
            for (Map.Entry<String, String> f : facts.entrySet()) {
                terms.add(new Condition(f.getKey(), "=",
                        bestTerm(kb, f.getKey(), f.getValue())));
            }
            Case c = new Case(terms, List.of(new Conclusion("advice", r[1])),
                    100);
            System.out.println(r[0] + ": " + c);
            InferenceResult result = kb.infer(Facts.map(facts));
            String fired = result.paths.isEmpty() ? null
```

Listing 9.5 (Continued) *LoanAdvisor.java: the fair, explained, fuzzy loan advisor.*

```
                : result.paths.get(0).last().id;
        if (!result.conclusions().isEmpty()
                && result.conclusions().get(0).value.equals(r[1])) {
            kb.refine(c, KnowledgeBase.ACCEPT, fired, null, "committee");
            System.out.println("  the advisor already agrees");
            continue;
        }
        Node node = kb.refine(c, KnowledgeBase.ADD_NEW, fired,
                List.of(chosen), "committee");
        System.out.println("  learned " + node + " (exception of " + fired
                + ")");
    }
    kb.save(KB_FILE);
    System.out.println("Saved loan-kb.json: " + kb.summary());
}

static void fairness(KnowledgeBase kb, List<Loans.Application> apps) {
    List<String> used = new ArrayList<>();
    for (String v : kb.variables.keySet()) {
        for (Node n : kb.nodes.values()) {
            if (n.variables().contains(v)) {
                used.add(v);
                break;
            }
        }
    }
    System.out.println("Variables the advice may depend on: "
            + String.join(", ", used));
    List<String> bad = new ArrayList<>();
    for (String v : used) {
        if (PROTECTED.contains(v.toLowerCase())) {
            bad.add(v);
        }
    }
    System.out.println("Protected attributes used: "
            + (bad.isEmpty() ? "none" : String.join(", ", bad)));
    int same = 0;
    for (Loans.Application app : apps) {
        Set<String> answers = new HashSet<>();
        for (String value : List.of("female", "male")) {
            Map<String, String> facts = new LinkedHashMap<>(app.facts());
            facts.put("gender", value);
            answers.add(kb.infer(Facts.map(facts)).conclusions().toString());
        }
        same += answers.size() == 1 ? 1 : 0;
    }
```

Listing 9.5 (Continued) *LoanAdvisor.java: the fair, explained, fuzzy loan advisor.*

```
        System.out.println("Same advice for gender = female and gender = male: "
                + same + " of " + apps.size() + " applications");
    }

    public static void main(String[] args) {
        String command = args.length > 0 ? args[0] : "advise";
        List<Loans.Application> apps = Loans.readApplications(DATA
                + "/applications.csv");
        if (command.equals("teach")) {
            teach(buildKb("loan-cases.json"));
        } else if (command.equals("fairness")) {
            KnowledgeBase kb;
            try {
                kb = args.length > 1 ? buildKb(args[1]) : openKb();
            } catch (FairnessException e) {
                System.out.println("Refused: " + e.getMessage() + ".");
                return;
            }
            fairness(kb, apps);
        } else {
            KnowledgeBase kb = openKb();
            System.out.println(kb.summary());
            for (Loans.Application app : apps) {
                adviseOne(kb, app);
            }
        }
    }
}
```

Building. `buildKb` reads the case file and, before anything else, runs every fact of every case through `checkFair`, which throws `FairnessException` for a protected attribute. Then it defines the nine fuzzy sets from `fuzzy-sets.csv` with `defineFuzzy` and builds the knowledge base from the cases. `openKb` uses the saved `loan-kb.json` if the committee has already taught the advisor, and the case file otherwise.

Advising. `adviseOne` prints the member's request and facts and runs `kb.infer(facts, null, true)`: RDR inference with **multiple** conclusions, so that every node that fires on a separate path is reported. The first path gives the advice and its CF, by Equation (9.3); the How explanation lists every node that fired, with the degree of each fuzzy condition. If another path gives a *different* advice, the case is **borderline**. Then comes the uncertainty of the income, by Equation (9.4), and the what-if amount.

Teaching. `teach` reads `decisions.csv`, the committee's decisions that differ from the advisor's, each with the fact that made the difference. It turns the member's numbers into words with `bestTerm` — each number becomes the word it belongs to most, so 54.0 percent becomes *high* — because the knowledge is kept in words. Then, exactly as in Chapter 7, it checks the advice, accepts it (option 0) if the advisor already agreed, or hangs the committee's decision as an exception of the node that gave the old advice, with the named fact as its only condition (option 2 with a chosen condition). The named fact is checked for fairness too.

Checking fairness. `fairness` lists the variables the advice may depend on, confirms that none is protected, and runs a **counterfactual test**: it gives the engine every application twice, once with `gender = female` added to the facts and once with `gender = male`, and counts the applications whose advice stays the same. Since no node mentions gender, the answer must be all of them; the test is there so that anyone can see it, and so that a future change that breaks it is caught.

Listing 9.6 is the Python version.

Listing 9.6 `loan_advisor.py`: the same advisor in Python.

```
"""Cooperative-loan eligibility advisor: fuzzy VCIRS with explanations.

    python loan_advisor.py advise        advice for every application
    python loan_advisor.py teach         learn the committee's decisions
    python loan_advisor.py fairness [cases.json]   check the knowledge

The advisor prepares the committee's meeting; the committee decides.
"""

import os
import pathlib
import sys

# Tell Python where the vcirs-core engine is (two folders up, then across).
sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import (ACCEPT, ADD_NEW, Case, Condition, Conclusion,
                   KnowledgeBase, casefile, num)
from loans import decimal, read_applications, rupiah

DATA = os.path.join(os.path.dirname(os.path.abspath(__file__)), "..", "data")
KB_FILE = os.path.join(DATA, "loan-kb.json")
PROTECTED = ["religion", "ethnicity", "gender", "marital status", "age",
             "disability", "political view", "place of origin", "agama",
             "suku", "jenis kelamin", "status perkawinan", "usia", "umur",
             "disabilitas", "pilihan politik", "asal daerah"]
# How sure the income figure is, as a triangular fuzzy number (a, b, c).
PROOF = {"verified": (0.85, 0.95, 1.0), "stated": (0.5, 0.7, 0.9)}


class FairnessError(Exception):
    """Raised when knowledge would use a protected attribute."""


def check_fair(variable, where):
    if variable.strip().lower() in PROTECTED:
        raise FairnessError(where + " uses the protected attribute '"
                            + variable + "'")
```

Listing 9.6 (Continued) *loan_advisor.py: the same advisor in Python.*

```
def build_kb(cases_file):
    name, description, steps = casefile.read_cases(
        os.path.join(DATA, cases_file))
    for i, step in enumerate(steps):
        for fact in step.case.facts:
            check_fair(fact.variable, "case " + str(i + 1))
    kb = KnowledgeBase(name, description)
    with open(os.path.join(DATA, "fuzzy-sets.csv"), encoding="utf-8") as f:
        for line in f:
            if line.strip() and not line.startswith("#"):
                var, term, a, b, c, d = line.strip().split(",")
                kb.define_fuzzy(var, term, [a, b, c, d])
    return casefile.build(kb, steps)


def open_kb():
    return KnowledgeBase.load(KB_FILE) if os.path.exists(KB_FILE) \
        else build_kb("loan-cases.json")


def times(cf, triangle):
    """A crisp certainty times a triangular fuzzy number."""
    return tuple(cf / 100.0 * x for x in triangle)


def advise_one(kb, app):
    d = app.data
    facts = app.facts()
    print(d["id"] + " " + d["name"] + " (" + d["work"] + "): "
          + rupiah(app.number("amount")) + " over " + d["months"]
          + " months")
    print("  facts: repayment ratio " + facts["repayment ratio"]
          + "%, membership " + facts["membership"] + " months, savings "
          + "cover " + facts["savings cover"] + "%, late payments "
          + facts["late payments"] + ", income " + facts["income pattern"])
    result = kb.infer(facts, None, True)
    if not result.paths:
        print("  advice: none; the committee decides")
        return
    first = result.paths[0]
    print("  advice: " + first.conclusions()[0].value + "  (CF "
          + num.fmt(first.cf, 2) + ")")
    print("  because:")
    for line in result.how():
        print("    " + line)
    others = [p.conclusions()[0].value for p in result.paths[1:]
```

Listing 9.6 (Continued) *loan_advisor.py: the same advisor in Python.*

```
                if p.conclusions() and p.conclusions()[0].value
                != first.conclusions()[0].value]
    if others:
        print("  BORDERLINE, also possible: " + "; ".join(others))
    a, b, c = times(first.cf, PROOF[d["income proof"]])
    print("  how sure (income " + d["income proof"] + "): about "
          + num.fmt(b, 2) + " (" + num.fmt(a, 2) + " to " + num.fmt(c, 2)
          + ")")
    best = app.largest_low()
    if best is not None and best < app.number("amount"):
        print("  " + rupiah(best) + " would bring the repayment ratio down to low ("
              + decimal(app.ratio_tenths(best)) + "%)")


def best_term(kb, variable, value):
    """The fuzzy term the number belongs to most (the first on a tie)."""
    if variable not in kb.fuzzy_sets:
        return value
    terms = kb.fuzzify(variable, float(value))
    return max(terms, key=lambda t: num.key(t[1]))[0]


def teach(kb):
    apps = {a.data["id"]: a for a in
            read_applications(os.path.join(DATA, "applications.csv"))}
    with open(os.path.join(DATA, "decisions.csv"), encoding="utf-8") as f:
        rows = [line.strip().split(",") for line in f
                if line.strip() and not line.startswith("#")]
    for member, advice, because in rows:
        facts = apps[member].facts()
        chosen = Condition.parse(because)
        check_fair(chosen.variable, "the decision for " + member)
        case = Case([Condition(v, "=", best_term(kb, v, x))
                     for v, x in facts.items()],
                    [Conclusion("advice", advice)])
        print(member + ": " + str(case))
        result = kb.infer(facts)
        fired = result.paths[0].last().id if result.paths else None
        if result.conclusions() and result.conclusions()[0].value == advice:
            kb.refine(case, ACCEPT, fired, author="committee")
            print("  the advisor already agrees")
            continue
        node = kb.refine(case, ADD_NEW, fired, [chosen], "committee")
        print("  learned " + str(node) + " (exception of " + fired + ")")
    kb.save(KB_FILE)
    print("Saved loan-kb.json: " + kb.summary())
```

Listing 9.6 (Continued) *loan_advisor.py: the same advisor in Python.*

```
def fairness(kb, apps):
    used = [v for v in kb.variables
            if any(v in n.variables() for n in kb.nodes.values())]
    print("Variables the advice may depend on: " + ", ".join(used))
    bad = [v for v in used if v.lower() in PROTECTED]
    print("Protected attributes used: " + (", ".join(bad) or "none"))
    same = 0
    for app in apps:
        answers = set()
        for value in ["female", "male"]:
            facts = dict(app.facts())
            facts["gender"] = value
            answers.add(str(kb.infer(facts).conclusions()))
        same += 1 if len(answers) == 1 else 0
    print("Same advice for gender = female and gender = male: " + str(same)
          + " of " + str(len(apps)) + " applications")


def main(args):
    command = args[0] if args else "advise"
    apps = read_applications(os.path.join(DATA, "applications.csv"))
    if command == "teach":
        teach(build_kb("loan-cases.json"))
    elif command == "fairness":
        try:
            kb = build_kb(args[1]) if len(args) > 1 else open_kb()
        except FairnessError as e:
            print("Refused: " + str(e) + ".")
            return
        fairness(kb, apps)
    else:
        kb = open_kb()
        print(kb.summary())
        for app in apps:
            advise_one(kb, app)


if __name__ == "__main__":
    main(sys.argv[1:])
```

9.6 The Committee Meeting

Seven applications are on the table this month. From the `Chapter09/java` folder, `java -cp out LoanAdvisor advise` (after compiling with `javac -d out -sourcepath ../../vcirs-core/java/src *.java`), or from `Chapter09/python`, `python loan_advisor.py advise`, prints the committee's papers. The first two:

```
Koperasi Sumber Makmur loan advisor: 5 variables, 7 nodes, 7 rules
A01 Bu Siti (vegetable seller): Rp 5,000,000 over 10 months
  facts: repayment ratio 18.9%, membership 48 months, savings cover 24%, late payments 0,
income daily
  advice: recommend  (CF 100)
  because:
    recommend#1 fired because:
      repayment ratio = low  (repayment ratio = 18.9, given, CF 100)
      so advice = recommend  [rule CF 100, conclusion CF 100]
  how sure (income verified): about 0.95 (0.85 to 1)
A02 Mas Joko (ojek driver): Rp 6,000,000 over 12 months
  facts: repayment ratio 28.1%, membership 30 months, savings cover 6%, late payments 1,
income daily
  advice: recommend a smaller amount  (CF 81)
  because:
    recommend a smaller amount#1 fired because:
      repayment ratio = medium  (repayment ratio = 28.1, given, CF 100, to degree 0.81)
      so advice = recommend a smaller amount  [rule CF 100, conclusion CF 81]
  how sure (income stated): about 0.57 (0.41 to 0.73)
  Rp 4,800,000 would bring the repayment ratio down to low (24.9%)
```

Bu Siti sells vegetables at the morning market. Her installments would take 18.9 percent of her income: low, to degree 1. She has been a member for four years, her savings cover 24 percent of the loan and she has never been late, so no exception fires. The advice is *recommend*, with CF 100, and her income is verified by her sales book: *about 0.95*.

Mas Joko drives an ojek. His new installment of Rp 560,000, on top of Rp 450,000 he already pays for his motorcycle, would take 28.1 percent of his income: medium to degree 0.81. His savings cover only 6 percent — *small* — so the exception for enough savings does not fire. The advice is *recommend a smaller amount*, CF 81; his income is only stated, so *about 0.57, between 0.41 and 0.73*; and Rp 4,800,000 instead of Rp 6,000,000 would bring his ratio down to 24.9 percent, low. That last line is what Bu Wati will say to him: not “no”, but “this much, yes”.

Then Pak Darmo, the rice farmer:

```
A03 Pak Darmo (rice farmer): Rp 8,000,000 over 8 months
  facts: repayment ratio 54.0%, membership 72 months, savings cover 18%, late payments 0,
income harvest
  advice: not now: lower the amount or save first  (CF 100)
  because:
    not now: lower the amount or save first#1 fired because:
      repayment ratio = high  (repayment ratio = 54.0, given, CF 100)
      so advice = not now: lower the amount or save first  [rule CF 100, conclusion CF 100]
  how sure (income verified): about 0.95 (0.85 to 1)
  Rp 3,700,000 would bring the repayment ratio down to low (25.0%)
```

His ratio is 54.0 percent — *high*, to degree 1 — and the advice is *not now*. On paper that is right: Rp 1,080,000 a month from an average income of Rp 2,000,000 is too much. But the committee knows Pak Darmo, and it knows something the numbers do not show: a rice farmer’s income does not come monthly. It comes at harvest. Section 9.7 comes back to him. For now it is enough to note that the advisor did not miscalculate; it has simply never been taught about income that comes at harvest, and so it cannot yet see what the committee sees.

The rest of the table:

```
A04 Mbak Rani (tailor): Rp 4,000,000 over 10 months
  facts: repayment ratio 28.8%, membership 36 months, savings cover 50%, late payments 0,
income monthly
  advice: recommend  (CF 88)
  because:
    recommend#2 fired because:
      repayment ratio = medium  (repayment ratio = 28.8, given, CF 100, to degree 0.88)
      savings cover = enough  (savings cover = 50, given, CF 100)
      so advice = recommend  [rule CF 100, conclusion CF 88]
  how sure (income verified): about 0.84 (0.75 to 0.88)
  Rp 2,200,000 would bring the repayment ratio down to low (24.8%)
A05 Pak Hasan (warung owner): Rp 3,000,000 over 6 months
  facts: repayment ratio 8.8%, membership 4 months, savings cover 10%, late payments 0,
income daily
  advice: start with a smaller first loan  (CF 100)
  because:
    start with a smaller first loan#1 fired because:
      repayment ratio = low  (repayment ratio = 8.8, given, CF 100)
      membership = new  (membership = 4, given, CF 100)
      so advice = start with a smaller first loan  [rule CF 100, conclusion CF 100]
  how sure (income stated): about 0.7 (0.5 to 0.9)
A06 Bu Yati (caterer): Rp 5,000,000 over 10 months
  facts: repayment ratio 18.8%, membership 40 months, savings cover 10%, late payments 4,
income monthly
  advice: refer to the committee  (CF 100)
  because:
    refer to the committee#1 fired because:
      repayment ratio = low  (repayment ratio = 18.8, given, CF 100)
      late payments = frequent  (late payments = 4, given, CF 100)
      so advice = refer to the committee  [rule CF 100, conclusion CF 100]
  how sure (income verified): about 0.95 (0.85 to 1)
```

Mbak Rani, a tailor, has a medium ratio (28.8 percent, degree 0.88), but her savings cover half the loan, so the exception *medium, savings enough* fires and the advice is *recommend*, CF 88: the rule's degree is the smaller of the two conditions' degrees, 0.88 and 1. **Pak Hasan** opened a warung and joined the cooperative four months ago: his ratio is a comfortable 8.8 percent, but *membership = new* fires the exception *start with a smaller first loan*. **Bu Yati**, a caterer, has a low ratio but four late payments in a year — *frequent* — and the exception sends her file to the committee for a conversation, not a refusal. Notice also the what-if lines. For Pak Darmo, Rp 3,700,000 would bring his ratio to 25.0 percent, a figure the committee may find useful even though, as Section 9.7 shows, it will decide differently. For Mbak Rani the advisor still prints Rp 2,200,000, although her advice is already *recommend*: the line answers a different question — "how much would be low without counting the savings?" — and the committee can ignore it when the savings settle the matter.

Last, Mas Andi:

```
A07 Mas Andi (phone-credit seller): Rp 4,000,000 over 10 months
  facts: repayment ratio 25.0%, membership 24 months, savings cover 7%, late payments 0,
income daily
  advice: recommend  (CF 50)
  because:
    recommend#1 fired because:
      repayment ratio = low  (repayment ratio = 25.0, given, CF 100, to degree 0.5)
      so advice = recommend  [rule CF 100, conclusion CF 50]
    recommend a smaller amount#1 fired because:
      repayment ratio = medium  (repayment ratio = 25.0, given, CF 100, to degree 0.5)
      so advice = recommend a smaller amount  [rule CF 100, conclusion CF 50]
  BORDERLINE, also possible: recommend a smaller amount
  how sure (income stated): about 0.35 (0.25 to 0.45)
```

His ratio is exactly 25.0 percent: low to degree 0.5 and medium to degree 0.5. Both top-level nodes hold at the threshold, the multiple inference follows both, and the advisor prints both explanations and the word **BORDERLINE**. Each advice has CF 50, the weakest a fuzzy conclusion can have and still fire, and his income is only stated, so the whole recommendation is *about 0.35*. This is exactly the file the committee should discuss: the numbers do not decide it, and the advisor says so instead of pretending.

What did the committee decide that Saturday? It approved Bu Siti, Mbak Rani and — for Rp 4,800,000 — Mas Joko, as the advisor suggested. It gave Pak Hasan a first loan of Rp 1,500,000, as the rule for new members says, and told him that after a year of regular payments he could come back for more. It invited Bu Yati to talk: her late payments had come during the months when a big catering customer paid her late, and the committee agreed on a loan with installments on the fifth of the month, after her customers pay. It discussed Mas Andi and approved his loan, since he had two years of regular savings, but asked him to bring his sales records next time so that his income could be marked *verified*. And it spent most of its time on Pak Darmo. Six files were settled in half an hour because the advisor had prepared them the same way the committee would have; the meeting's time went to the two files where people were needed.

> **KEY INSIGHT.** Fuzziness turns a knife-edge into a slope. With crisp limits, Mas Andi at 25.0 percent would get one answer and a neighbor at 25.1 percent the opposite one, with full confidence. With fuzzy words, both get a weak answer with an honest CF, and the case that sits on the border is flagged for people to decide.

9.7 Teaching the Committee's Decision

The committee discusses Pak Darmo. He has been a member for six years and has never been late. His loan is for fertilizer and seed; after the harvest in four months he will be able to pay a large part at once. The committee approves the loan with a **harvest-time schedule** (in Java, *pinjaman yarnen*, "pay after harvest"), a common arrangement in farming villages. Bu Wati writes the decision in `decisions.csv`:

```
# id,the committee's advice,the fact that made the difference
A03,consider a harvest-time schedule,income pattern = harvest
```

and runs `teach`:

```
A03: repayment ratio = high, membership = established, savings cover = enough, late
payments = rare, income pattern = harvest -> advice = consider a harvest-time schedule
  learned consider a harvest-time schedule#1: income pattern = harvest -> advice = consider
a harvest-time schedule (exception of not now: lower the amount or save first#1)
Saved loan-kb.json: Koperasi Sumber Makmur loan advisor: 6 variables, 8 nodes, 8 rules
```

The case, in words, is what the committee saw: a high ratio, an established member with enough savings who is rarely late, whose income comes at harvest. The advisor had said *not now*; the difference, named by the committee, is `income pattern = harvest`, and the new node is an exception of the *not now* node with exactly that condition. Nothing else changes. After teaching, the advice for Pak Darmo is:

```
A03 Pak Darmo (rice farmer): Rp 8,000,000 over 8 months
  facts: repayment ratio 54.0%, membership 72 months, savings cover 18%, late payments 0,
income harvest
  advice: consider a harvest-time schedule  (CF 100)
  because:
    consider a harvest-time schedule#1 fired because:
      repayment ratio = high  (repayment ratio = 54.0, given, CF 100)
      income pattern = harvest  (given, CF 100)
      so advice = consider a harvest-time schedule  [rule CF 100, conclusion CF 100]
  how sure (income verified): about 0.95 (0.85 to 1)
  Rp 3,700,000 would bring the repayment ratio down to low (25.0%)
```

The exception fired: the ratio is high (degree 1), the income pattern is harvest (a crisp condition), and the advice is to consider a harvest-time schedule. The same knowledge will now prepare the next farmer's file the same way — and only a farmer's: a warung owner with a high ratio still gets *not now*. Next season, when Pak Sukri, another farmer, asks for a loan for his chili field with a ratio of 48 percent, his page will already say *consider a harvest-time schedule*, and the committee will start the discussion where it ended for Pak Darmo. The other six members get exactly the advice they got before, which `advise` confirms line by line.

Why an exception and not a new top-level rule "income pattern = harvest → harvest-time schedule"? Because the committee did not say that every farmer should get a harvest schedule; it said that when a farmer's ratio is *high*, the harvest schedule is the answer instead of *not now*. A farmer whose ratio is low borrows on ordinary terms, and the knowledge base should not suggest otherwise. The exception keeps the new knowledge in the context where the committee gave it — the lesson of Chapter 3, now with fuzzy words.

Finally, the fairness check, before and after the teaching:

```
Variables the advice may depend on: repayment ratio, membership, savings cover, late
payments, income pattern
Protected attributes used: none
Same advice for gender = female and gender = male: 7 of 7 applications
```

The advice may now depend on five variables, the income pattern included; none is protected; and adding a gender to the facts never changed any member's advice. And when someone tries to teach a case that uses gender — the file `bad-cases.json` holds such a case — the advisor refuses to build the knowledge base at all (`fairness bad-cases.json`):

```
Refused: case 1 uses the protected attribute 'gender'.
```

9.8 Making It Better

9.8.1 Tuning the words with the committee

The fuzzy sets are the committee's words written as numbers, and the committee should own them. A good way to set them is to take a dozen past files, sort them by ratio, and ask the committee members to mark, for each, whether they would call it low, medium or high. Where almost everyone says *low*, the degree should be near 1; where opinions split, near 0.5. The corners of the trapezoids follow from those marks. Change `fuzzy-sets.csv`, run `advise` on the past files, and see whether the advice matches what the committee decided then. Exercise 9.6 does this for the border between *low* and *medium*.

9.8.2 The threshold

The knowledge base's threshold, 0.5, decides when a fuzzy condition counts as true. A cautious committee might raise it to 0.6 for the *recommend* rules — but the threshold is one number for the whole knowledge base, so raising it also makes *not now* and *refer to the committee* harder to reach, which is not caution at all. It is usually better to keep 0.5 and move the corners of the words, where the meaning lives. An example makes the difference plain. With a threshold of 0.6, Mas Andi's ratio of 25.0 percent — low and medium to degree 0.5 each — would satisfy *no* word at all, and no node would fire: the most delicate file of the month would get no advice and no borderline warning. Moving the corners of *low* instead, as in Exercise 9.6, changes the meaning of one word and nothing else.

9.8.3 Hedges in the knowledge

The committee can use hedges when it teaches. A case with `repayment ratio = very high` catches only ratios well above 40 percent with a strong degree; a case with `repayment ratio = somewhat low` is generous at the border. For example, a committee that wants to refer to a financial counselor any member whose repayments would be *very* high could teach that as an exception of the *not now* node. Hedges let the words of the rules follow the words of the meeting, without new fuzzy sets.

9.8.4 Keeping a record of the overrides

Every time the committee's decision differs from the advisor's, `teach` records it in the history of `loan-kb.json`, marked as taught by the *committee*. Once a year, count the overrides by the node they refined. A node that the committee overrides again and again is a rule the committee no longer believes, or a word whose edges have moved — the *low* of a year of high prices may not be the *low* of a quiet year. That count, like the Credits of Chapter 7, tells the committee where to look first when it reviews its own knowledge at the annual members' meeting.

9.8.5 Tests

`LoanTest.java` and `test_loans.py` check the arithmetic (an installment of Rp 560,000, Mas Joko's ratio of 28.1 percent, his what-if amount, the rupiah format), the fuzzy advice (*recommend a smaller amount* with CF 81, two paths for the borderline case) and fairness (the bad case file is refused, and adding a gender never changes any advice). They end with:

```
all tests passed
```

9.9 Using It for Real

Your cooperative's words. Start from your own rules, not from this chapter's. Write down, with the committee, the words it uses when it decides — *income is enough*, *the member is known*, *she has savings* — and turn each into a variable with fuzzy sets or a crisp value. Keep the variables about repayment. Then teach ten or twenty past decisions as cases, and run `advise` on last month's files to see whether the advisor would have prepared them the way the committee decided.

Starting carefully. For the first three months, run the advisor beside the committee without showing its advice during the meeting. Bu Wati prints it afterwards and compares it with what the committee decided. Where they agree, nothing is needed; where they differ, the committee

teaches the case. Only when the advisor agrees with most decisions — and when the committee understands why it disagrees with the rest — should its pages go on the table before the discussion. This is how the knowledge base earns the committee's trust, one meeting at a time.

The members' meeting sets the rules. The rate, the maximum loan, the savings a member must hold before borrowing and the length of loans belong to the cooperative's statutes (AD/ART) and the members' meeting, which the law makes the highest decision-making body of a cooperative [3], and to the regulations that apply to savings and loans [2]. The advisor reads the rate from one constant; update it when the meeting changes it, and add any hard limit of the statutes as a crisp condition of the knowledge base.

Data about people. An application file is personal data. Keep it on the cooperative's computer, show each member her own file, and do not send it anywhere else; Indonesia's personal data protection law applies to cooperatives as to everyone [12]. The knowledge base itself holds cases in words — *repayment ratio = high* — and no names, which is one more reason to keep it that way.

Every decision is the committee's. Print the advisor's page for each file, let the committee write its decision beside it, and teach the differences once a month, as Bu Wati did. Over a year the advisor will learn the committee's habits, and the history in `loan-kb.json` will show which rules the committee overrode most often — the place to talk about whether a rule or a word should change.

A second treasurer. Bu Wati will not be treasurer forever. The advisor, with its cases, its words and its history, is also the best way to hand over eleven years of experience: a new treasurer who prepares the files with it gives the same first reading Bu Wati would give, and can read in the knowledge base *why* the committee decides as it does. Ask the new treasurer to prepare three months of files beside Bu Wati, comparing notes, before she takes over.

Explaining to a member. When a member asks why, read her the explanation: which word her numbers fit, to what degree, and which rule followed. Then read her the what-if amount. A member who understands the reason can plan: save a little more, borrow a little less, or come back in three months as an established member.

In the members' language. The advisor prints its explanations in the engine's words (*fired because*, *to degree 0.81*). For members, write a short sheet in Indonesian that says what each line means — *derajat 0,81 berarti angka Anda sebagian besar termasuk "sedang"* — or translate the lines of `adviseOne` into Indonesian; the knowledge itself does not change.

Other decisions with soft edges. The same advisor fits any decision made from numbers by people who speak in words: whether a student qualifies for a village scholarship (*low* family income, *good* attendance), whether a small shop can take a supplier's credit (*high* turnover, *short* history), how much fertilizer credit a farmer group can carry.

9.10 Common Mistakes, and How to Avoid Them

Hard limits dressed as fuzzy sets. A trapezoid with $a = b$ and $c = d$ is a crisp interval, and brings back the knife-edge. *Avoid it:* give every word a real slope, as wide as the committee's hesitation.

Words that leave gaps. If *low* ends at 25 and *medium* starts at 30, a ratio of 27 belongs to nothing, and no node fires. *Avoid it:* let neighboring words overlap, so that every number belongs to some word to degree 0.5 or more, as in **Figure 9.1**.

Reading the CF as a probability of repayment. CF 81 does not mean an 81 percent chance that Mas Joko repays. It says how well his numbers fit the rule's words. *Avoid it:* read it as the strength of the recommendation, and read the uncertainty line for how firm the data are.

Using protected attributes, openly or through proxies. *Avoid it:* keep the fairness guard; before adding any variable, ask whether it is about repayment and whether it could stand in for a protected group.

Teaching in numbers instead of words. A case with `repayment ratio = 54.0` would fire only for exactly 54.0. *Avoid it:* teach in the committee's words; `teach` does this with `bestTerm`.

Mixing the ratio's parts. Leaving out installments paid elsewhere — to a bank, a pindar, a motorcycle dealer — makes every ratio look better than it is. *Avoid it:* ask for all installments, and mark the income *stated* when it cannot be checked.

Forgetting to update the income. A member's income changes with the seasons, a new job or an illness. *Avoid it:* take the income and installments from the current application, not from last year's file, and mark it *stated* when there is no fresh proof.

Treating a borderline flag as a refusal. *Avoid it:* a borderline file is one the numbers cannot decide; it needs the committee's knowledge of the member, not a "no".

Changing a word without testing it. Moving the corners of *low* by two points changes the advice for every member near the border. *Avoid it:* run `advise` on last month's files before and after the change, and let the committee look at every file whose advice moved.

Letting the advisor decide. *Avoid it:* the advisor prepares; the committee decides, and says so to the member.

9.11 Chapter Summary

- A **fuzzy set** gives a number a degree of belonging to a word; the trapezoid of Equation (9.1) is enough for most committee words, and the hedges of Equation (9.2) sharpen or soften them (**Figure 9.1**, **Listing 9.1**, **Listing 9.2**).
- In VCIRS, a condition on a fuzzy variable holds when its degree reaches the threshold, and the degree enters the conclusion's certainty by Equation (9.3); knowledge is taught in words and applied to numbers.
- **Fuzziness** (soft words) and **uncertainty** (unsure data and rules) are different; the advisor carries the uncertainty of the income as a triangular **fuzzy number** combined by Equation (9.4).
- The **repayment ratio** of Equation (9.5), computed in whole rupiah (**Listing 9.3**, **Listing 9.4**), is the committee's key number; OJK's 30 percent limit for online lenders in 2026 is a sensible yardstick.
- The **loan advisor** (**Listing 9.5**, **Listing 9.6**) gives each application an advice with its CF, its explanation, a borderline warning when two words hold equally, the uncertainty of the income and the loan amount that would be low; the committee's decisions are taught back as exceptions (**Figure 9.2**).

- Fairness and transparency are built in: only repayment facts reach the engine, knowledge using a protected attribute is refused, a counterfactual test shows that gender never changes the advice, and every answer can be explained to the member.

9.12 Exercises

9.1 Using Equation (9.1) and the committee's sets, compute by hand the degrees of *low*, *medium* and *high* for a repayment ratio of 22 percent and of 37 percent. Which nodes of the knowledge base can fire for each?

9.2 With Equation (9.2), compute *very medium*, *somewhat medium* and *not medium* for 33.5 percent. Which of them hold at the threshold 0.5?

9.3 Bu Lastri asks for Rp 3,000,000 over 6 months. Her income is Rp 3,000,000 a month and she already pays Rp 250,000. Compute her installment and repayment ratio by Equation (9.5), the degrees of the three words, and the advice and CF the advisor gives if she is an established member with enough savings who is rarely late.

9.4 Mas Joko's advice has CF 81 and his income is *stated*. Compute his "how sure" triangle with Equation (9.4). What would it be if he brought three months of bank statements and his income became *verified*?

9.5 Explain, with Mas Andi's file, why the advisor calls `infer` with multiple conclusions. What would it print for him if it did not?

9.6 The committee decides that 25 percent should still be "clearly low" and moves *low* to (0, 0, 22, 32) and *medium* to (22, 32, 32, 42). Recompute Mas Andi's and Mas Joko's degrees. Is Mas Andi still borderline? What is Mas Joko's advice and CF now?

9.7 A committee member proposes adding the variable `village` because "members from Dusun Krajan always repay". Answer him, using the idea of a proxy and the advisor's fairness guard.

9.8 Design one more exception for the knowledge base: members whose repayment ratio is *very high* should be referred to a free financial counselor before any loan is discussed. Write the case, say under which node it goes, and explain how it interacts with the harvest-time exception taught in Section 9.7.

9.13 Solutions to the Exercises

Solution 9.1. At 22 percent: *low* has $c = 20 < 22 < d = 30$, so $\mu = (30 - 22)/(30 - 20) = 0.8$; *medium* has $a = 20 < 22 < b = 30$, so $\mu = (22 - 20)/10 = 0.2$; *high* is 0. Only nodes whose first condition is `repayment ratio = low` can fire (0.8 reaches 0.5; 0.2 does not). At 37 percent: *low* 0; *medium* $(40 - 37)/10 = 0.3$; *high* $(37 - 30)/10 = 0.7$. Only the *high* node and its exceptions can fire.

Solution 9.2. *Medium* at 33.5 is 0.65. *Very medium* is $0.65^2 = 0.4225$; *somewhat medium* is $\sqrt{0.65} \approx 0.806$; *not medium* is $1 - 0.65 = 0.35$. Only *somewhat medium* holds at 0.5.

Solution 9.3. $I = [3{,}000{,}000/6] + 30{,}000 = 530{,}000$; $r = (250{,}000 + 530{,}000)/3{,}000{,}000 \times 100 = 26.0$ percent. *Low*: $(30 - 26)/10 = 0.4$; *medium*: $(26 - 20)/10 = 0.6$; *high*: 0. The *medium* node fires with degree 0.6; its exception *savings cover = enough* also fires (degree 1), and the exception below it needs frequent late payments, which she does not have. The advice is *recommend*, with CF $\min(100 \times 0.6, 100 \times 1) \times 100/100 = 60$.

Solution 9.4. Stated: $0.81 \times (0.5,0.7,0.9) = (0.405,0.567,0.729)$, printed *about 0.57 (0.41 to 0.73)*. Verified: $0.81 \times (0.85,0.95,1.0) = (0.6885,0.7695,0.81)$, *about 0.77 (0.69 to 0.81)*. The advice does not change; the committee's confidence in it does.

Solution 9.5. At exactly 25.0 percent, *low* and *medium* both hold with degree 0.5, so two top-level nodes fire and give different advice. Without multiple conclusions, the engine stops at the first path, the *low* node, and prints *recommend (CF 50)* with one explanation and no warning. The committee would see a weak recommendation but not that the opposite recommendation is exactly as strong. Multiple inference makes the tie visible.

Solution 9.6. Mas Andi, 25.0 percent: with the new corners $22 < 25 < 32$, so *low* is $(32 - 25)/10 = 0.7$ and *medium* is $(25 - 22)/10 = 0.3$. Only *low* holds: he is no longer borderline, and the advice is *recommend* with CF 70. Mas Joko, 28.1 percent: *low* $(32 - 28.1)/10 = 0.39$, *medium* $(28.1 - 22)/10 = 0.61$. The advice is still *recommend a smaller amount*, now with CF 61. Moving the corners changed the strength of both answers and removed one borderline case — which is why such a change belongs to the committee, tested on past files.

Solution 9.7. A village is not about the ability to repay; it describes where a person lives, and in many regions where a person lives is linked to ethnicity, religion or origin. Using it would let the advisor treat two members with the same income, savings and history differently because of their hamlet — a **proxy** for a protected attribute. The fairness guard lists the protected attributes by name and cannot recognize every proxy, so the committee's rule must be that every variable is about repayment and can be justified to the member. What makes Dusun Krajan's members reliable — savings, a good history — is already in the knowledge base as their own facts.

Solution 9.8. The case is `repayment ratio = very high` → `advice = refer to a financial counselor first`. It goes under the *not now* node (option 2 with the chosen condition `repayment ratio = very high`), because it refines that node's answer for the most extreme cases. The *not now* node now has two exceptions, *harvest* and *very high*. For a farmer with a very high ratio both fire, and the engine, following the exceptions in the order they were taught, gives the harvest-time advice first; with multiple conclusions the advisor would print both and flag the file as borderline. If the committee wants the counselor first in every case, the very-high exception should itself get a harvest exception, or be taught before the harvest one in a rebuilt knowledge base.

References

[1] Otoritas Jasa Keuangan, "Siaran pers bersama: Kemenkop RI serahkan daftar koperasi yang menjalankan kegiatan di sektor jasa keuangan kepada OJK," Jan. 14, 2025. [Online]. Available: https://ojk.go.id/id/berita-dan-kegiatan/siaran-pers/Pages/Kemenkop-RI-Serahkan-Daftar-Koperasi-yang-Menjalankan-Kegiatan-di-Sektor-Jasa-Keuangan-kepada-OJK.aspx

[2] Kementerian Koperasi dan UKM, *Peraturan Menteri Koperasi dan UKM Nomor 8 Tahun 2023 tentang Usaha Simpan Pinjam oleh Koperasi*. Jakarta, Indonesia, 2023. [Online]. Available: https://peraturan.bpk.go.id/Details/260414/permenkop-ukm-no-8-tahun-2023

[3] Republik Indonesia, *Undang-Undang Nomor 25 Tahun 1992 tentang Perkoperasian*. Jakarta, Indonesia, 1992. [Online]. Available: https://peraturan.bpk.go.id/Details/46650/uu-no-25-tahun-1992

[4] ANTARA News, "Menkop: RUU Perkoperasian baru ditargetkan disahkan tahun ini," Jul. 28, 2026. [Online]. Available: https://m.antaranews.com/amp/berita/5669913/menkop-ruu-perkoperasian-baru-ditargetkan-disahkan-tahun-ini

[5] Liputan6.com, "83 ribu Koperasi Merah Putih terbentuk, nilai transaksi tembus Rp 56,69 miliar," Jul. 13, 2026. [Online]. Available: https://www.liputan6.com/bisnis/read/8245255/83-ribu-koperasi-merah-putih-terbentuk-nilai-transaksi-tembus-rp-5669-miliar

[6] I. Subakti and O. Wijayanto, "Penerapan konsep fuzzy dalam Variable-Centered Intelligent Rule System (studi kasus: pemilihan jurusan di Chinese University of Hongkong)," Dept. of Informatics, Institut Teknologi Sepuluh Nopember, Surabaya, Indonesia.

[7] L. A. Zadeh, "Fuzzy sets," *Information and Control*, vol. 8, no. 3, pp. 338–353, 1965.

[8] M. M. I. Subakti, "A Variable-Centered Intelligent Rule System," M.Sc.Eng. thesis, Dept. of Computer Science and Information Engineering, Taipei, Taiwan, Jun. 2005.

[9] G. R. Baur and D. V. Pigford, *Expert Systems for Business: Concepts and Applications*. Boston, MA, USA: Boyd & Fraser, 1990.

[10] VOI, "Mulai bertahap, OJK terapkan batas rasio utang pinjol 30 persen pada 2026," Jan. 12, 2026. [Online]. Available: https://voi.id/ekonomi/550592/mulai-bertahap-ojk-terapkan-batas-rasio-utang-pinjol-30-persen-pada-2026

[11] Otoritas Jasa Keuangan, *Peraturan OJK Nomor 22 Tahun 2023 tentang Pelindungan Konsumen dan Masyarakat di Sektor Jasa Keuangan*. Jakarta, Indonesia, Dec. 2023. [Online]. Available: https://ojk.go.id/id/regulasi/Pages/Pelindungan-Konsumen-dan-Masyarakat-di-Sektor-Jasa-Keuangan.aspx

[12] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. Jakarta, Indonesia, Oct. 17, 2022. [Online]. Available: https://peraturan.bpk.go.id/Details/229798/uu-no-27-tahun-2022

Chapter 10

Laundry Stain-Removal Advisor as a Web App with a Database

By the end of this chapter you will be able to explain why an expert system that several people use at once belongs in a web app with a database; keep a VCIRS knowledge base in the tables of a SQLite database and read it with SQL; ask an expert system's questions over the web with the *re-run pattern*, in which the browser remembers the answers and the server remembers nothing between calls; and build the **stain advisor** of a small laundry twice — a Java server on the standard library's `HttpServer` with the SQLite JDBC driver, and a Python server on Flask — that serve the same pages, answer every call with the same bytes and write the same database. You will run it in a browser, teach it two new lessons, look at what it learned, and count its consultations with SQL.

10.1 Bu Rini's Laundry

Bu Rini's laundry is a shophouse on the street behind a campus, between a photocopy kiosk and a food stall. On the front counter there is a scale, a stack of plastic bags and an order book; in the back there are four washing machines, two dryers, a long ironing table and, on most days, Joko. Students from the boarding houses around bring their clothes by the kilogram and collect them two days later, washed, ironed and folded. Most of the work is routine. The part that is not routine is the stains.

A white shirt with tea on the collar. A pair of jeans with motorcycle grease on the knee. A school uniform with a ballpoint line where a pen leaked in the pocket. A silk scarf with a drop of cooking oil. Each of these has a right treatment and several wrong ones, and the wrong ones are not harmless. Hot water fixes a blood stain into cotton for good. Chlorine bleach on a rust mark makes it darker. Rubbing alcohol on silk can leave a ring worse than the stain. A stain that goes through the dryer before it is treated usually stays forever, and a customer whose favorite shirt comes back ruined does not come back.

Bu Rini knows the treatments. She learned some from her mother, some from the labels of the cleaning products she buys, and most of them from twelve years of stains. But she is not always in the shop. On Mondays she buys supplies at the market; on Thursdays she minds the second shop across town. Dewi, who takes the orders at the counter, and Joko, who washes, have to decide on their own. For a while they used a notebook that hung on a nail by the washing machines. The notebook got wet, the pages stuck together, and nobody could find the page for ink when they needed it.

What Bu Rini wants is simple to say. She wants her treatments to be available on the phone at the counter and on the old laptop in the back, both at the same time, both showing the same advice. She wants Dewi and Joko to be able to add what they learn, with their names on it, so that she can read on Thursday evening what was learned on Thursday morning. And she wants a few numbers at the end of the week: how many stains were treated, how often the advice worked, which treatments are used most. Chapter 3 built a troubleshooter that learns from each fix, but it ran in

one terminal on one computer and kept its knowledge in a file. This chapter takes the same kind of knowledge base to a **web app** with a **database**.

10.2 The Idea in Plain Words

10.2.1 From one computer to several: a web app

A **web app** is a program in two parts that talk over a network. The **server** is a program that runs on one computer — in the laundry, the laptop in the back — and waits. The **client** is a web browser, on the same laptop or on any phone connected to the same Wi-Fi, which shows pages and sends requests. When Dewi opens the advisor on her phone, the browser asks the server for a page; the page contains a small program in JavaScript, and that program asks the server questions such as "what should I ask next, given these answers?" The server answers, the page shows the answer, and the conversation goes on.

The requests and answers travel as **HTTP**, the language of the web, and their content is written in **JSON**, the same plain-text format our knowledge bases have used since Chapter 3. A request to our server looks like this:

```
POST /api/consult
{"facts": {"stain": "tea"}, "unknown": []}
```

and its answer like this:

```
{"status": "question", "variable": "color",
 "question": "Is the item white or colored?", "choices": ["white", "colored"], ...}
```

The advantage for the laundry is that the knowledge lives in one place. There are not two copies that drift apart, one on the phone and one on the laptop; there is one knowledge base in the server, and every browser sees it. When Joko teaches the advisor something in the back room, Dewi's next consultation at the counter already uses it.

A web app can also mean a server on the internet that the whole world can reach. That is not what the laundry needs, and it brings problems the laundry does not want: passwords, attacks, a monthly bill. Our server runs on the shop's own computer and, unless told otherwise, accepts connections only from that computer. With one option it accepts them from the shop's Wi-Fi, which is enough for the phone at the counter.

10.2.2 Why a database, and why SQLite

Chapter 3 kept its knowledge base in a JSON file, loaded at the start and saved after every lesson. For one person at one keyboard that is enough. For a shop it has three weaknesses.

First, **two people at once**. If Dewi and Joko teach the advisor at the same moment, two programs may write the same file at the same time, and one lesson can be lost or the file damaged. A database is built for exactly this: every change is a **transaction** that happens completely or not at all, and two changes are never mixed [1].

Second, **questions about the records**. Bu Rini's weekly numbers — how many consultations, how many worked, which treatment was given most — are questions about many records at once. A database answers such questions with a few words of **SQL** (Structured Query Language): `SELECT Treatment, COUNT(*) FROM Consultation GROUP BY Treatment`. With a JSON file we would have to write a loop for every question.

Third, **records that are not knowledge**. The laundry also wants to keep the steps of each treatment, and a log of consultations. These belong next to the knowledge base but are not part of it, and a database holds them as tables of their own.

SQLite is the database we use. Unlike the databases that run as separate servers, SQLite is a small library that reads and writes one ordinary file — here `laundry.db` — so there is nothing to install and nothing to administer; you can copy the file to a flash disk as a backup. It is also the most widely deployed database engine in the world: it is inside every Android and iOS phone and every major web browser, and its developers estimate that over a trillion SQLite databases are in use [2]. The current release line, as of September 2026, is 3.53 (3.53.4 was released in July 2026) [1]. Python has SQLite built in, as the `sqlite3` module; Java reaches it through **JDBC**, Java's standard way of talking to databases, with the SQLite JDBC driver, a single `.jar` file (version 3.53.2.0, released in June 2026, which carries SQLite 3.53.2) [3].

10.2.3 The knowledge base as tables

When the thesis that defined VCIRS implemented the method, it kept the knowledge base in a database, Microsoft Access at the time, with one main table and several supporting ones [4]. The main table is **NodeStructure**: one row for every condition of every node, with the variable, its position in the node, the operator, the value, the certainty factor and the Credit. The names of variables, nodes and rules are kept in the **Variable**, **Node** and **Rule** tables; the conclusions in **Conclusion**; and the order of nodes in each rule in **RuleStructure** [4], [5]. vcirs-core's `SqliteStore` writes exactly these tables, adds a **History** table for the audit trail and a small **Meta** table for the name and format, and fills in the usage rates (VUR, NUR, RUR) so that the tables can be browsed. **Figure 10.1** shows them, together with the two tables of the laundry's own.

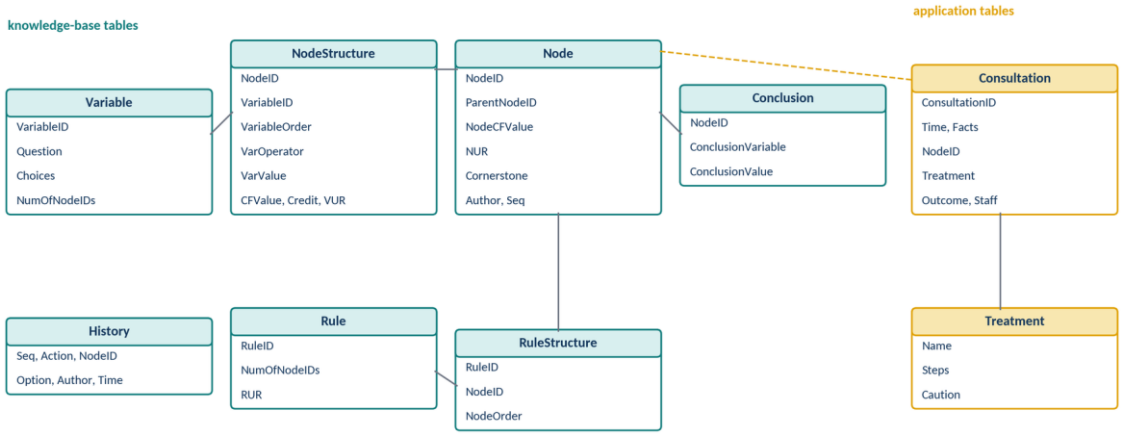

Figure 10.1 The tables of the stain advisor's database. The teal tables hold the VCIRS knowledge base as in Appendix E of the thesis; the amber tables belong to the laundry: the steps of each treatment and the log of consultations.

Why tables and not a JSON text in one column? Because tables answer questions. "Which nodes use the variable `fabric`?" is `SELECT NodeID FROM NodeStructure WHERE VariableID = 'fabric'`. "Which rules have more than one node?" is a join of RuleStructure with NodeStructure. The variable-centered design of VCIRS — a node is a sequence of *variable* conditions, and every use of a variable is recorded with it — fits a relational database naturally, which is why the thesis chose one [4].

10.2.4 Asking questions over the web: the re-run pattern

In Chapter 3 the engine asked its questions through an *asker*, a function that printed the question and waited for the answer typed in the terminal. The engine stopped in the middle of its work, waited, and continued. A web server cannot work like that. When the browser sends a request, the server must answer at once, and it cannot keep a half-finished consultation waiting in memory for an answer that may never come (Dewi may put her phone down and serve a customer). The usual rule of web design is that the server keeps no memory of a conversation between one request and the next: every request carries everything the server needs [6].

So we turn the conversation around. The **browser** keeps the answers given so far. Each time, it sends *all* of them to the server. The server starts the engine from the beginning with those answers and an asker of a special kind: an asker that, instead of waiting for an answer, stops the engine at once and reports the question. That question is the server's reply. The page shows it, adds the answer to its list and sends the whole list again. When the engine gets to the end without needing a question, the reply is the advice. **Figure 10.2** shows a whole consultation.

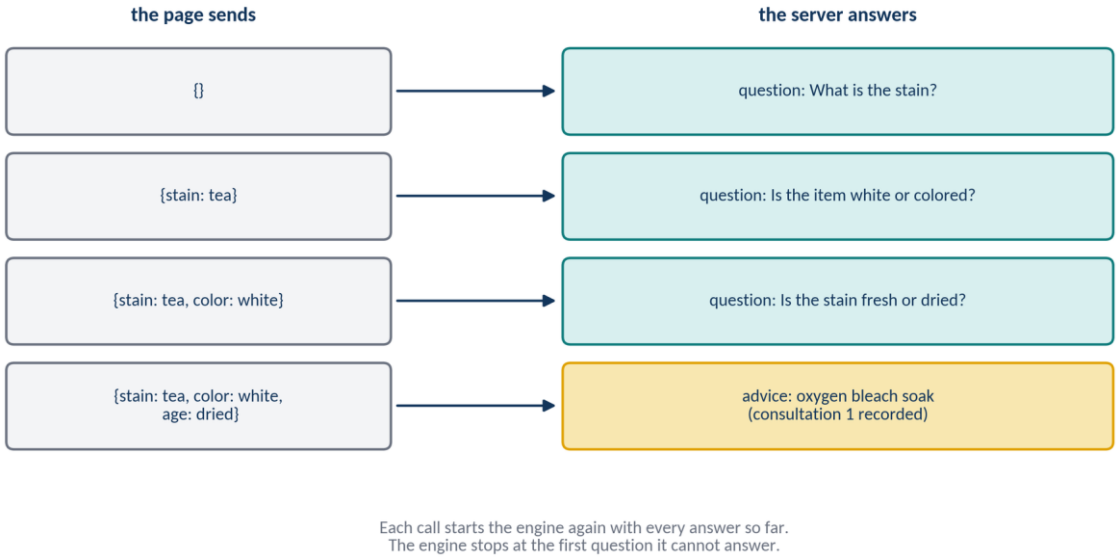

Figure 10.2 The re-run pattern. The page sends every answer it has; the server runs the engine from the beginning and returns the first question the engine cannot answer, or, when there is none, the advice.

Two things make this work. The engine is **deterministic**: given the same answers, it walks the same nodes in the same order and reaches the same first open question, so re-running it never surprises. And the engine is **cheap**: a knowledge base of a few hundred nodes is evaluated in a fraction of a millisecond, so running it again for every answer costs nothing a customer could notice. "I do not know" also fits the pattern: the page keeps a second list, `unknown`, and the server's asker answers "unknown" for those variables instead of stopping, exactly as pressing Enter did in Chapter 3.

> **DEFINITION.** In the *re-run pattern*, the client keeps every answer of a consultation and sends them all with each request; the server runs the inference from the beginning, stops at the first question it cannot answer and returns it, or returns the conclusion when no question is left. The server keeps nothing between requests.

10.2.5 What the advisor knows, and what it must never do

Bu Rini's first twelve lessons, the **seed cases** of this chapter, are summarized in **Table 10.1**. They follow the common practice of stain removal as the cleaning-products industry and university

extension services teach it [7], [8]: protein stains such as blood in *cold* water, never hot; oily stains with a detergent that dissolves grease; ink with a solvent from the back of the fabric; rust never with chlorine bleach. Two lessons are exceptions of others, in the Ripple Down Rules sense of Chapter 3: a *dried* blood stain needs an enzyme detergent soak rather than a cold rinse, and ink on *silk* goes to a dry cleaner rather than meeting rubbing alcohol. The dish-soap-and-sunlight treatment for turmeric (the yellow of curry and of *nasi kuning*) is Bu Rini's own experience; the knowledge base is where such experience is written down.

Table 10.1 The twelve seed cases of the stain advisor, with the caution each treatment carries.

Stain	Condition	Treatment	Caution the page shows
oil	—	dish soap and a warm wash	check before drying
blood	—	cold rinse and soak	never hot water
blood	dried	enzyme soak	not for silk or wool
tea	—	cool soak and liquid detergent	—
tea	white, dried	oxygen bleach soak	white cotton only; never mix bleach
ink	—	rubbing alcohol from the back	test on a hidden seam; no flames
ink	silk	send to the dry cleaner	—
gum	—	ice and scrape	—
wax	—	scrape and iron	warm iron on synthetics
mud	—	brush when dry	—
turmeric	—	dish soap and sunlight	—
rust	—	rust remover	never chlorine bleach

Stains are a gentle domain, but the advisor still deals with chemicals, heat and other people's property, so it follows three safety rules that are worth stating once. It never advises mixing cleaning products: chlorine bleach mixed with ammonia or with acids gives off toxic gases, and the caution of every bleach treatment says so. Every solvent treatment carries "test on a hidden seam first", because a fabric that cannot take a solvent shows it within a minute on the inside of a hem. And whenever the item is delicate or the label says *dry clean only*, the right advice is the dry cleaner, not a clever trick. These cautions live in the Treatment table, next to the steps, and the page shows them in an amber box that is hard to miss.

10.3 Step 1: The Knowledge Base in a Database

Before any web code, let us put the knowledge base into a database and look inside it. **Listing 10.1** builds the knowledge base from the seed cases, saves it with vcirs-core's `SqliteStore`, asks the database four questions in SQL, and finally loads the knowledge base back from the database to check that nothing was lost.

Listing 10.1 `db_tour.py`: the knowledge base saved in SQLite, looked at with SQL, and loaded back.

```
"""A VCIRS knowledge base in a SQLite database: save it, look inside with SQL,
and load it back.

    python db_tour.py
"""

import os
import pathlib
import sqlite3
import sys

sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import KnowledgeBase, SqliteStore, casefile, report

DATA = pathlib.Path(__file__).resolve().parents[1] / "data"
DB = str(DATA / "tour.db")


def show(con, title, sql):
    """Run one SELECT and print its rows as a table."""
    cur = con.execute(sql)
    headers = [d[0] for d in cur.description]
    rows = [["(top)" if v is None else v for v in row] for row in cur]
    print(title)
    for line in report.table(headers, rows):
        print("  " + line)
    print()


if os.path.exists(DB):
    os.remove(DB)                       # start from an empty database
name, description, steps = casefile.read_cases(str(DATA
                                                    / "stain-seed-cases.json"))
kb = casefile.build(KnowledgeBase(name, description), steps)
SqliteStore(DB).save(kb)
print("Saved " + kb.summary() + " in tour.db")
print()
```

Listing 10.1 (Continued) *db_tour.py: the knowledge base saved in SQLite, looked at with SQL, and loaded back.*

```
con = sqlite3.connect(DB)
show(con, "Variables, and how many nodes use each:",
     "SELECT VariableID, NumOfNodeIDs FROM Variable ORDER BY Position")
show(con, "Nodes, with their parent (an exception has one):",
     "SELECT NodeID, ParentNodeID, Author FROM Node ORDER BY Seq")
show(con, "The conditions of one node, one row each:",
     "SELECT VariableOrder, VariableID, VarOperator, VarValue, Credit "
     "FROM NodeStructure WHERE NodeID = 'oxygen bleach soak#1' "
     "ORDER BY VariableOrder")
show(con, "Rules with more than one node, and all the conditions they need:",
     "SELECT r.RuleID, COUNT(*) AS Conditions FROM RuleStructure r "
     "JOIN NodeStructure s ON s.NodeID = r.NodeID "
     "GROUP BY r.RuleID HAVING COUNT(DISTINCT r.NodeID) > 1 "
     "ORDER BY r.RuleID")
con.close()

again = SqliteStore(DB).load()
print("Loaded back: " + again.summary())
print("Same knowledge as before saving: "
      + str(again.to_json() == kb.to_json()).lower())
```

The program has three parts. The first builds the knowledge base exactly as Chapter 3 built the Wi-Fi doctor's: `casefile.read_cases` reads the seed file and `casefile.build` teaches the twelve

cases one after another with `learn`, so that the two exceptions are placed under the nodes they correct. Then `SqliteStore(DB).save(kb)` writes it into `tour.db`, creating the tables if they do not exist.

The second part opens the database with Python's own `sqlite3` module and runs four `SELECT` statements through the small function `show`, which prints the rows as a table with vcirs-core's `report.table`. A `SELECT` names the columns it wants (`SELECT VariableID, NumOfNodeIDs`), the table they come from (`FROM Variable`), which rows to keep (`WHERE NodeID = 'oxygen bleach soak#1'`) and the order (`ORDER BY Position`). The fourth statement is a **join**: `FROM RuleStructure r JOIN NodeStructure s ON s.NodeID = r.NodeID` pairs every node of every rule with every condition of that node, `GROUP BY r.RuleID` makes one group per rule, `COUNT(*)` counts the conditions in the group, and `HAVING COUNT(DISTINCT r.NodeID) > 1` keeps only the rules made of more than one node.

The third part loads the knowledge base back with `SqliteStore(DB).load()` and compares the JSON text of the two knowledge bases. Equal text means equal knowledge: every variable, node, condition, cornerstone case and history entry survived the trip through the tables. **Listing 10.2** is the same program in Java.

Listing 10.2 `DbTour.java`: the same tour of the database in Java, through JDBC.

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;

import vcirs.CaseFile;
import vcirs.KnowledgeBase;
import vcirs.Report;
import vcirs.SqliteStore;

/**
 * A VCIRS knowledge base in a SQLite database: save it, look inside with
 * SQL, and load it back.
 * java -cp "out:lib/sqlite-jdbc-3.53.2.0.jar" DbTour   (Windows: ; not :)
 */
public class DbTour {
    static final Path DATA = Path.of("..", "data");
    static final String DB = DATA.resolve("tour.db").toString();

    /** Run one SELECT and print its rows as a table. */
    static void show(Connection con, String title, String sql)
            throws SQLException {
        try (Statement st = con.createStatement();
             ResultSet rs = st.executeQuery(sql)) {
            ResultSetMetaData md = rs.getMetaData();
            List<String> headers = new ArrayList<>();
            for (int i = 1; i <= md.getColumnCount(); i++) {
                headers.add(md.getColumnLabel(i));
            }
            List<List<Object>> rows = new ArrayList<>();
            while (rs.next()) {
                List<Object> row = new ArrayList<>();
                for (int i = 1; i <= md.getColumnCount(); i++) {
                    String v = rs.getString(i);
                    row.add(v == null ? "(top)" : v);
                }
                rows.add(row);
            }
            System.out.println(title);
            for (String line : Report.table(headers, rows)) {
```

Listing 10.2 (Continued) *DbTour.java: the same tour of the database in Java, through JDBC.*

```
                System.out.println("   " + line);
            }
            System.out.println();
        }
    }

    public static void main(String[] args) throws Exception {
        Files.deleteIfExists(Path.of(DB));   // start from an empty database
        CaseFile seed = CaseFile.read(DATA.resolve("stain-seed-cases.json")
                .toString());
        KnowledgeBase kb = seed.build(new KnowledgeBase(seed.name,
                seed.description));
        new SqliteStore(DB).save(kb);
        System.out.println("Saved " + kb.summary() + " in tour.db");
        System.out.println();

        try (Connection con = DriverManager.getConnection("jdbc:sqlite:" + DB)) {
            show(con, "Variables, and how many nodes use each:",
                    "SELECT VariableID, NumOfNodeIDs FROM Variable "
                    + "ORDER BY Position");
            show(con, "Nodes, with their parent (an exception has one):",
                    "SELECT NodeID, ParentNodeID, Author FROM Node ORDER BY Seq");
            show(con, "The conditions of one node, one row each:",
                    "SELECT VariableOrder, VariableID, VarOperator, VarValue, "
                    + "Credit FROM NodeStructure "
                    + "WHERE NodeID = 'oxygen bleach soak#1' "
                    + "ORDER BY VariableOrder");
            show(con, "Rules with more than one node, and all the conditions "
                    + "they need:",
                    "SELECT r.RuleID, COUNT(*) AS Conditions FROM RuleStructure r "
                    + "JOIN NodeStructure s ON s.NodeID = r.NodeID "
                    + "GROUP BY r.RuleID HAVING COUNT(DISTINCT r.NodeID) > 1 "
                    + "ORDER BY r.RuleID");
        }

        KnowledgeBase again = new SqliteStore(DB).load();
        System.out.println("Loaded back: " + again.summary());
        System.out.println("Same knowledge as before saving: "
                + again.toJson().equals(kb.toJson()));
    }
}
```

In Java a database is reached through **JDBC**. `DriverManager.getConnection("jdbc:sqlite:" + DB)` opens a `Connection` to the file; the text `jdbc:sqlite:` tells JDBC to use the SQLite driver, which must be on the class path when the program runs. `createStatement()` gives a `Statement`, `executeQuery(sql)` runs a `SELECT` and returns a `ResultSet`, and `rs.next()` moves from row to row. `ResultSetMetaData` tells how many columns there are and what they are called. The `try (...)` form closes the connection, the statement and the result set automatically, even if something fails, which matters with databases: a connection left open can keep the file locked.

To run it, compile with the engine and run with the driver on the class path:

```
cd Chapter10/java
javac -d out -sourcepath ../../vcirs-core/java/src *.java
java -cp "out:lib/sqlite-jdbc-3.53.2.0.jar" DbTour
```

On Windows the class-path separator is a semicolon, `"out;lib/sqlite-jdbc-3.53.2.0.jar"`. In Python, `python db_tour.py` (or `python3` on macOS and Linux). Both print, byte for byte:

```
Saved Stain advisor: 5 variables, 12 nodes, 12 rules in tour.db

Variables, and how many nodes use each:
  VariableID  NumOfNodeIDs
  ----------  ------------
  stain       9
  treatment   0
  age         2
  color       1
  fabric      1

Nodes, with their parent (an exception has one):
  NodeID                          ParentNodeID                    Author
  ------------------------------  ------------------------------  ------
  dish soap and a warm wash#1     (top)                           Rini
  cold rinse and soak#1           (top)                           Rini
  enzyme soak#1                   cold rinse and soak#1           Rini
  cool soak and liquid detergent#1  (top)                         Rini
  oxygen bleach soak#1            cool soak and liquid detergent#1  Rini
  rubbing alcohol from the back#1  (top)                          Rini
  send to the dry cleaner#1       rubbing alcohol from the back#1  Rini
  ice and scrape#1                (top)                           Rini
  scrape and iron#1               (top)                           Rini
  brush when dry#1                (top)                           Rini
  dish soap and sunlight#1        (top)                           Rini
  rust remover#1                  (top)                           Rini

The conditions of one node, one row each:
  VariableOrder  VariableID  VarOperator  VarValue  Credit
  -------------  ----------  ------------  --------  ------
  1              color       =             white     1
  2              age         =             dried     1

Rules with more than one node, and all the conditions they need:
  RuleID                    Conditions
  ------------------------  ----------
  enzyme soak#1             2
  oxygen bleach soak#1      3
  send to the dry cleaner#1  2

Loaded back: Stain advisor: 5 variables, 12 nodes, 12 rules
Same knowledge as before saving: true
```

Read the output as a visit to the thesis's tables. The **Variable** table lists five variables; `stain` is used by nine nodes, `age` by two, `color` and `fabric` by one each, and `treatment` by none — it is a conclusion variable, never a condition. The **Node** table shows the twelve nodes with their parents: three nodes have one, the exceptions `enzyme soak#1`, `oxygen bleach soak#1` and `send to the dry cleaner#1`. The **NodeStructure** rows of `oxygen bleach soak#1` are its two conditions, in order, each with its Credit; notice that the node itself holds only `color = white` and `age = dried`, because `stain = tea` belongs to its parent. That is why the last query finds three conditions in the rule `oxygen bleach soak#1`: one from the parent node and two from the exception. And the knowledge base read back from the tables is the same knowledge, to the last byte of its JSON.

> **KEY INSIGHT.** The database adds nothing to what the knowledge base means; it changes only where the knowledge lives and who can ask questions of it. The same knowledge base can be saved as JSON in Chapter 3 and as tables here, and the engine gives the same answers from both.

10.4 Step 2: The Advisor, Without Any Web Code

The heart of the app is a class that knows nothing about the web: the **Advisor**. It opens the database, consults the knowledge base, learns from feedback and summarizes what it knows. Its methods take plain maps (dictionaries) and return plain maps, which the server will turn into JSON. Keeping the web out of it has two rewards. We can test the Advisor, and replay a whole day of the laundry, without starting a server. And the two servers of this chapter, Java and Python, share one design: only their thin web layers differ.

Table 10.2 lists what the Advisor offers, and the JSON call through which the web app will offer it.

Table 10.2 The three calls of the stain advisor and what travels in each.

Call	The page sends	The Advisor answers
`POST /api/consult`	`facts` (the answers so far), `unknown` (the "I do not know" variables), `staff`	a `question` (variable, text, choices, why), or the `advice` (treatment, steps, caution, how, consultation number), or `none`
`POST /api/feedback`	`consultation`, `worked` (true or false); if false: `treatment`, `steps`, `caution`, `extra` facts, `staff`	`accepted`, `learned` (with the list of difference and the affected cornerstones), `closed` or an `error`
`GET /api/kb`	nothing	the summary, the tree, the consultation counts, the most given treatments, the latest changes, the treatments

Listing 10.3 is the Advisor in Python.

Listing 10.3 `advisor.py`: the stain advisor's logic, with its knowledge in SQLite.

```
"""The stain advisor: a VCIRS knowledge base kept in a SQLite database.

Advisor has no web code at all. Its three methods take and return plain
dictionaries, which the web server turns into JSON:
  consult(facts, unknown)  the next question, or the advice
  feedback(answer)         did the advice work? if not, learn the right one
  overview()               the knowledge base, the treatments, the numbers
"""

import datetime
import json
import os
import pathlib
import sqlite3
import sys

sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import (Case, Conclusion, Condition, Fact, KnowledgeBase,
                   SqliteStore, VcirsError, casefile, report)

QUESTIONS = [
    ("stain", "What is the stain?",
     ["oil", "blood", "tea", "ink", "gum", "wax", "mud", "turmeric", "rust"]),
    ("age", "Is the stain fresh or dried?", ["fresh", "dried"]),
    ("color", "Is the item white or colored?", ["white", "colored"]),
    ("fabric", "What is the fabric?",
     ["cotton", "polyester", "denim", "silk", "wool"]),
]
TABLES = [
    "CREATE TABLE IF NOT EXISTS Treatment (Name TEXT PRIMARY KEY, "
    "Steps TEXT, Caution TEXT)",
    "CREATE TABLE IF NOT EXISTS Consultation (ConsultationID INTEGER "
    "PRIMARY KEY, Time TEXT, Facts TEXT, NodeID TEXT, Treatment TEXT, "
    "Outcome TEXT, Staff TEXT)",
]


class Clock:
    """The time now; or, given a start, a pretend clock that moves one
    minute each time it is read (so that a demonstration repeats exactly)."""
```

Listing 10.3 (Continued) *advisor.py: the stain advisor's logic, with its knowledge in SQLite.*

```
    def __init__(self, start=None):
        self.start = start and datetime.datetime.strptime(start,
                                                          "%Y-%m-%d %H:%M")
        self.ticks = 0

    def __call__(self):
        if self.start is None:
            now = datetime.datetime.now()
        else:
            now = self.start + datetime.timedelta(minutes=self.ticks)
            self.ticks += 1
        return now.strftime("%Y-%m-%d %H:%M")


class NeedAnswer(Exception):
    """Thrown by the asker to stop the engine at the first open question."""

    def __init__(self, question):
        super().__init__(question.variable)
        self.question = question


class Advisor:
    def __init__(self, db, data_dir, clock):
        self.db = db
        self.store = SqliteStore(db)
        self.clock = clock
        if not os.path.exists(db):
            self._create(data_dir)
        self.kb = self.store.load()
        self.kb.clock = clock

    def _create(self, data_dir):
        """The first run: build the knowledge from the seed cases."""
        name, description, steps = casefile.read_cases(
            os.path.join(data_dir, "stain-seed-cases.json"))
        kb = KnowledgeBase(name, description)
        for var, question, choices in QUESTIONS:
            kb.declare(var, question, choices)
        casefile.build(kb, steps)
        self.store.save(kb)
        with open(os.path.join(data_dir, "treatments.json"),
                  encoding="utf-8") as f:
            treatments = json.load(f)["treatments"]
        with self._connect() as con:
            for t in treatments:
```

Listing 10.3 (Continued) *advisor.py: the stain advisor's logic, with its knowledge in SQLite.*

```
                con.execute("INSERT INTO Treatment VALUES (?, ?, ?)",
                            (t["name"], t["steps"], t["caution"]))

    def _connect(self):
        con = sqlite3.connect(self.db)
        for sql in TABLES:
            con.execute(sql)
        return con

    def _record(self, facts, node_id, treatment, staff):
        with self._connect() as con:
            cur = con.execute(
                "INSERT INTO Consultation (Time, Facts, NodeID, Treatment, "
                "Outcome, Staff) VALUES (?, ?, ?, ?, 'open', ?)",
                (self.clock(), json.dumps(facts), node_id, treatment, staff))
            return cur.lastrowid

    # ----- consulting ----------------------------------------------------
    def consult(self, facts, unknown, staff=""):
        """Run the knowledge base on the answers so far. If it needs one
        more answer, return that question; otherwise record the
        consultation and return the advice (or say there is none)."""
        given = {v: Fact(v, str(x), 100, "user") for v, x in facts.items()}

        def asker(question):
            if question.variable in unknown:
                return None                    # "I do not know"
            raise NeedAnswer(question)

        try:
            result = self.kb.infer(given, asker)
        except NeedAnswer as e:
            q = e.question
            var = self.kb.variables.get(q.variable)
            return {"status": "question", "variable": q.variable,
                    "question": var.question if var and var.question
                    else q.variable + "?",
                    "choices": q.choices, "why": q.why()}
        if not result.conclusions():
            number = self._record(facts, "", "", staff)
            return {"status": "none", "consultation": number,
                    "message": "No advice for this stain yet."}
        treatment = result.conclusions()[0].value
        node = result.paths[0].last().id
        number = self._record(facts, node, treatment, staff)
        steps, caution = self._treatment(treatment)
```

Listing 10.3 (Continued) *advisor.py: the stain advisor's logic, with its knowledge in SQLite.*

```
        return {"status": "advice", "consultation": number,
                "treatment": treatment, "steps": steps, "caution": caution,
                "node": node, "how": result.how()}

    def _treatment(self, name):
        with self._connect() as con:
            row = con.execute("SELECT Steps, Caution FROM Treatment "
                              "WHERE Name = ?", (name,)).fetchone()
        return row if row else ("", "")

    # ----- learning ------------------------------------------------------
    def feedback(self, answer):
        """answer: consultation, worked (true/false), and when it did not
        work: treatment, steps, caution, extra ("variable = value") and
        staff. Returns what the knowledge base did."""
        number = answer.get("consultation")
        with self._connect() as con:
            row = con.execute("SELECT Facts, NodeID, Treatment, Outcome FROM "
                              "Consultation WHERE ConsultationID = ?",
                              (number,)).fetchone()
        if row is None or row[3] != "open":
            return {"status": "error",
                    "message": "no open consultation " + str(number)}
        facts = [Condition(v, "=", x) for v, x in json.loads(row[0]).items()]
        staff = answer.get("staff") or "counter"
        if answer.get("worked"):
            self.kb.learn(Case(facts, [Conclusion("treatment", row[2])]),
                          staff)
            return self._close(number, "worked", staff, {
                "status": "accepted",
                "message": "Good. The advice was right; it is recorded."})
        treatment = (answer.get("treatment") or "").strip()
        if not treatment:
            return self._close(number, "failed", staff, {
                "status": "closed",
                "message": "Nothing to learn this time."})
        try:
            for extra in answer.get("extra", []):
                facts.append(Condition.parse(extra))
            case = Case(facts, [Conclusion("treatment", treatment)])
            out = {"status": "learned"}
            if row[1]:
                d = self.kb.differences(row[1], case)
                conds = [c.negated() for c in d.a] + d.b
                out["differences"] = d.lines()
                out["affected"] = self.kb.affected_cornerstones(
```

Listing 10.3 (Continued) *advisor.py: the stain advisor's logic, with its knowledge in SQLite.*

```
                    row[1], conds, case.conclusions)
            learned = self.kb.learn(case, staff)
        except VcirsError as e:
            return {"status": "error", "message": str(e)}
        with self._connect() as con:
            con.execute("INSERT OR IGNORE INTO Treatment VALUES (?, ?, ?)",
                        (treatment, answer.get("steps", ""),
                         answer.get("caution", "")))
        out["result"] = str(learned)
        return self._close(number, "taught", staff, out)

    def _close(self, number, outcome, staff, out):
        with self._connect() as con:
            con.execute("UPDATE Consultation SET Outcome = ?, Staff = ? "
                        "WHERE ConsultationID = ?", (outcome, staff, number))
        self.store.save(self.kb)
        out["summary"] = self.kb.summary()
        return out

    # ----- looking at the knowledge -------------------------------------
    def overview(self):
        with self._connect() as con:
            treatments = [{"name": n, "steps": s, "caution": c}
                          for n, s, c in con.execute(
                              "SELECT Name, Steps, Caution FROM Treatment "
                              "ORDER BY Name")]
            outcomes = {o: n for o, n in con.execute(
                "SELECT Outcome, COUNT(*) FROM Consultation GROUP BY Outcome")}
            top = [{"treatment": t, "times": n} for t, n in con.execute(
                "SELECT Treatment, COUNT(*) AS n FROM Consultation "
                "WHERE Treatment <> '' GROUP BY Treatment "
                "ORDER BY n DESC, Treatment LIMIT 5")]
        counts = {"total": sum(outcomes.values())}
        for o in ("worked", "taught", "failed", "open"):
            counts[o] = outcomes.get(o, 0)
        return {"summary": self.kb.summary(), "tree": report.tree(self.kb),
                "consultations": counts, "mostGiven": top,
                "history": list(reversed(self.kb.history[-8:])),
                "treatments": treatments}
```

Let us read it from the top.

The questions and the tables. `QUESTIONS` declares the four variables the staff are asked about, each with its question and usual answers, as the Wi-Fi doctor did. `TABLES` creates the laundry's two tables if they are not there yet. `Treatment` holds the steps and the caution of each treatment by name; the knowledge base concludes only the *name* (`treatment = enzyme soak`), and the page looks up the steps. Separating the two keeps the node names short and lets Bu Rini improve the wording of a treatment without touching the knowledge. `Consultation` holds one row per consultation: when, the answers (as JSON text), the node that answered, the treatment, the outcome (`open`, `worked`, `taught` or `failed`) and who was at the counter. `ConsultationID INTEGER PRIMARY KEY` makes SQLite number the rows by itself: 1, 2, 3 and so on.

The clock. `Clock` gives the time as text such as `2026-09-28 08:03`. Normally it is the real time. Given a start, it is a pretend clock that moves forward one minute each time it is read; the examples in this chapter use it so that they can be repeated exactly, in both languages, on any day. The knowledge base uses the same clock for its history (`self.kb.clock = clock`), so the history entries and the consultations agree.

Opening. The constructor checks whether the database file exists. The first time it does not, and `_create` builds the knowledge base from the seed cases, declares the questions, saves it with

`SqliteStore`, and fills the Treatment table from `treatments.json`. After that the knowledge base is always loaded from the database. `_connect` opens a connection and makes sure the laundry's tables exist; the `with self._connect() as con:` form commits the changes at the end of the block, as one transaction.

Consulting. `consult` is the re-run pattern of Section 10.2.4. It turns the answers into facts marked as `"user"` answers, so that the explanation says *answered by the user*, as in Chapter 3. Its asker has two behaviors: for a variable in `unknown` it returns `None`, "I do not know"; for any other variable it *raises* `NeedAnswer`, an exception that carries the question. Raising an exception is the simplest way to stop an engine in the middle of its work: the engine does not catch it, so it flies straight out of `kb.infer` to the `except NeedAnswer` of `consult`, which turns the question into a reply with the variable, the question text, the usual answers (`q.choices`, which also contains every value the knowledge base has seen for that variable) and `q.why()`, the engine's own explanation of why it asks. If `infer` finishes, there is no open question: the consultation is recorded, and the reply is either `none` or the advice, with the treatment's steps and caution from the database and `result.how()`, the explanation of Chapter 3.

Learning. `feedback` is the second half of the Wi-Fi doctor's `main`, working on a recorded consultation instead of on a conversation. It reads the consultation's row; a consultation that does not exist or is no longer `open` gives an error, so that the same lesson cannot be taught twice by pressing a button twice. If the advice worked, the case is given to `learn`, which records it as a verified case (option 0), and the consultation is closed as `worked`. If it did not, and the staff member typed what worked instead, the extra facts are added to the case, the list of difference and the affected cornerstones are computed against the node that answered (for the reply, so that the page can show them), and `learn` adds the new knowledge: an exception if a node answered, a new top-level node if none did. `INSERT OR IGNORE` stores the steps and caution of a new treatment, leaving an existing treatment of the same name alone. `_close` marks the consultation, saves the knowledge base to the database, and adds the one-line summary to the reply.

Overview. `overview` gathers everything the knowledge page shows in one reply. Three SQL statements do the counting: the treatments in alphabetical order; `SELECT Outcome, COUNT(*) FROM Consultation GROUP BY Outcome`, which counts consultations by outcome in one pass; and the five most given treatments, `ORDER BY n DESC, Treatment` so that ties are broken by name and the order is always the same. The history is the last eight entries, newest first.

Listing 10.4 is the Advisor in Java.

Listing 10.4 `Advisor.java`: the same logic in Java, through JDBC.

```
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.Collections;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.function.Supplier;

import vcirs.Case;
import vcirs.CaseFile;
import vcirs.Conclusion;
import vcirs.Condition;
import vcirs.DifferenceList;
import vcirs.Fact;
import vcirs.InferenceResult;
import vcirs.Json;
import vcirs.KnowledgeBase;
import vcirs.LearnResult;
import vcirs.Question;
import vcirs.Report;
import vcirs.SqliteStore;
import vcirs.Variable;
import vcirs.VcirsException;

/**
 * The stain advisor: a VCIRS knowledge base kept in a SQLite database.
 * Advisor has no web code at all. Its three methods take and return plain
 * maps, which the web server turns into JSON:
 *   consult(facts, unknown)  the next question, or the advice
 *   feedback(answer)         did the advice work? if not, learn the right one
 *   overview()               the knowledge base, the treatments, the numbers
 */
public class Advisor {
    static final String[][] QUESTIONS = {
        {"stain", "What is the stain?", "oil", "blood", "tea", "ink", "gum",
            "wax", "mud", "turmeric", "rust"},
```

Listing 10.4 (Continued) *Advisor.java: the same logic in Java, through JDBC.*

```
        {"age", "Is the stain fresh or dried?", "fresh", "dried"},
        {"color", "Is the item white or colored?", "white", "colored"},
        {"fabric", "What is the fabric?", "cotton", "polyester", "denim",
            "silk", "wool"},
    };
    static final String[] TABLES = {
        "CREATE TABLE IF NOT EXISTS Treatment (Name TEXT PRIMARY KEY, "
            + "Steps TEXT, Caution TEXT)",
        "CREATE TABLE IF NOT EXISTS Consultation (ConsultationID INTEGER "
            + "PRIMARY KEY, Time TEXT, Facts TEXT, NodeID TEXT, Treatment TEXT, "
            + "Outcome TEXT, Staff TEXT)",
    };

    /**
     * The time now; or, given a start, a pretend clock that moves one
     * minute each time it is read (so that a demonstration repeats exactly).
     */
    static class Clock implements Supplier<String> {
        static final DateTimeFormatter F =
                DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm");
        final LocalDateTime start;
        int ticks = 0;

        Clock(String start) {
            this.start = start == null ? null : LocalDateTime.parse(start, F);
        }

        public String get() {
            LocalDateTime now = start == null ? LocalDateTime.now()
                    : start.plusMinutes(ticks++);
            return now.format(F);
        }
    }

    /** Thrown by the asker to stop the engine at the first open question. */
    static class NeedAnswer extends RuntimeException {
        final Question question;

        NeedAnswer(Question question) {
            super(question.variable);
            this.question = question;
        }
    }
```

Listing 10.4 (Continued) *Advisor.java: the same logic in Java, through JDBC.*

```
    final String db;
    final SqliteStore store;
    final Supplier<String> clock;
    final KnowledgeBase kb;

    Advisor(String db, String dataDir, Supplier<String> clock) throws Exception {
        this.db = db;
        this.store = new SqliteStore(db);
        this.clock = clock;
        if (!Files.exists(Path.of(db))) {
            create(dataDir);
        }
        kb = store.load();
        kb.clock = clock;
    }

    /** The first run: build the knowledge from the seed cases. */
    @SuppressWarnings("unchecked")
    void create(String dataDir) throws Exception {
        CaseFile seed = CaseFile.read(Path.of(dataDir,
                "stain-seed-cases.json").toString());
        KnowledgeBase k = new KnowledgeBase(seed.name, seed.description);
        for (String[] q : QUESTIONS) {
            k.declare(q[0], q[1], List.of(q).subList(2, q.length));
        }
        seed.build(k);
        store.save(k);
        String text = Files.readString(Path.of(dataDir, "treatments.json"),
                StandardCharsets.UTF_8);
        Map<String, Object> file = (Map<String, Object>) Json.parse(text);
        try (Connection con = connect()) {
            for (Object o : (List<Object>) file.get("treatments")) {
                Map<String, Object> t = (Map<String, Object>) o;
                update(con, "INSERT INTO Treatment VALUES (?, ?, ?)",
                        t.get("name"), t.get("steps"), t.get("caution"));
            }
        }
    }
```

Listing 10.4 (Continued) *Advisor.java: the same logic in Java, through JDBC.*

```
    Connection connect() throws SQLException {
        Connection con = DriverManager.getConnection("jdbc:sqlite:" + db);
        try (Statement st = con.createStatement()) {
            for (String sql : TABLES) {
                st.execute(sql);
            }
        }
        return con;
    }

    static int update(Connection con, String sql, Object... values)
            throws SQLException {
        try (PreparedStatement ps = con.prepareStatement(sql)) {
            for (int i = 0; i < values.length; i++) {
                ps.setObject(i + 1, values[i]);
            }
            return ps.executeUpdate();
        }
    }

    /** Rows of a SELECT, each a list of column values. */
    static List<List<Object>> query(Connection con, String sql,
                                    Object... values) throws SQLException {
        List<List<Object>> rows = new ArrayList<>();
        try (PreparedStatement ps = con.prepareStatement(sql)) {
            for (int i = 0; i < values.length; i++) {
                ps.setObject(i + 1, values[i]);
            }
            try (ResultSet rs = ps.executeQuery()) {
                int n = rs.getMetaData().getColumnCount();
                while (rs.next()) {
                    List<Object> row = new ArrayList<>();
                    for (int i = 1; i <= n; i++) {
                        row.add(rs.getObject(i));
                    }
                    rows.add(row);
                }
            }
        }
        return rows;
    }
```

Listing 10.4 (Continued) *Advisor.java: the same logic in Java, through JDBC.*

```
    /** {"stain": "oil", "age": "dried"}, written as Python's json.dumps. */
    static String compact(Map<String, String> facts) {
        List<String> parts = new ArrayList<>();
        for (Map.Entry<String, String> f : facts.entrySet()) {
            parts.add(Json.quote(f.getKey()) + ": " + Json.quote(f.getValue()));
        }
        return "{" + String.join(", ", parts) + "}";
    }

    long record(Map<String, String> facts, String nodeId, String treatment,
                String staff) throws SQLException {
        try (Connection con = connect()) {
            update(con, "INSERT INTO Consultation (Time, Facts, NodeID, "
                    + "Treatment, Outcome, Staff) VALUES (?, ?, ?, ?, 'open', ?)",
                    clock.get(), compact(facts), nodeId, treatment, staff);
            return ((Number) query(con, "SELECT last_insert_rowid()")
                    .get(0).get(0)).longValue();
        }
    }

    // ----- consulting ----------------------------------------------------
    /**
     * Run the knowledge base on the answers so far. If it needs one more
     * answer, return that question; otherwise record the consultation and
     * return the advice (or say there is none).
     */
    Map<String, Object> consult(Map<String, String> facts, List<String> unknown,
                                String staff) throws SQLException {
        Map<String, Fact> given = new LinkedHashMap<>();
        for (Map.Entry<String, String> f : facts.entrySet()) {
            given.put(f.getKey(), new Fact(f.getKey(), f.getValue(), 100,
                    "user"));
        }
        Map<String, Object> out = new LinkedHashMap<>();
        InferenceResult result;
        try {
            result = kb.infer(given, q -> {
                if (unknown.contains(q.variable)) {
                    return null;                        // "I do not know"
                }
                throw new NeedAnswer(q);
            });
        } catch (NeedAnswer e) {
            Question q = e.question;
            Variable v = kb.variables.get(q.variable);
            out.put("status", "question");
```

Listing 10.4 (Continued) *Advisor.java: the same logic in Java, through JDBC.*

```
            out.put("variable", q.variable);
            out.put("question", v != null && !v.question.isEmpty()
                    ? v.question : q.variable + "?");
            out.put("choices", q.choices);
            out.put("why", q.why());
            return out;
        }
        if (result.conclusions().isEmpty()) {
            out.put("status", "none");
            out.put("consultation", record(facts, "", "", staff));
            out.put("message", "No advice for this stain yet.");
            return out;
        }
        String treatment = result.conclusions().get(0).value;
        String node = result.paths.get(0).last().id;
        long number = record(facts, node, treatment, staff);
        List<String> t = treatment(treatment);
        out.put("status", "advice");
        out.put("consultation", number);
        out.put("treatment", treatment);
        out.put("steps", t.get(0));
        out.put("caution", t.get(1));
        out.put("node", node);
        out.put("how", result.how());
        return out;
    }

    List<String> treatment(String name) throws SQLException {
        try (Connection con = connect()) {
            List<List<Object>> rows = query(con, "SELECT Steps, Caution FROM "
                    + "Treatment WHERE Name = ?", name);
            return rows.isEmpty() ? List.of("", "")
                    : List.of((String) rows.get(0).get(0),
                              (String) rows.get(0).get(1));
        }
    }
```

Listing 10.4 (Continued) *Advisor.java: the same logic in Java, through JDBC.*

```
    // ----- learning ------------------------------------------------------
    /**
     * answer: consultation, worked (true/false), and when it did not work:
     * treatment, steps, caution, extra ("variable = value") and staff.
     * Returns what the knowledge base did.
     */
    @SuppressWarnings("unchecked")
    Map<String, Object> feedback(Map<String, Object> answer) throws Exception {
        Object number = answer.get("consultation");
        List<List<Object>> rows;
        try (Connection con = connect()) {
            rows = query(con, "SELECT Facts, NodeID, Treatment, Outcome FROM "
                    + "Consultation WHERE ConsultationID = ?", number);
        }
        Map<String, Object> out = new LinkedHashMap<>();
        if (rows.isEmpty() || !"open".equals(rows.get(0).get(3))) {
            out.put("status", "error");
            out.put("message", "no open consultation " + number);
            return out;
        }
        List<Object> row = rows.get(0);
        List<Condition> facts = new ArrayList<>();
        Map<String, Object> stored =
                (Map<String, Object>) Json.parse((String) row.get(0));
        for (Map.Entry<String, Object> f : stored.entrySet()) {
            facts.add(new Condition(f.getKey(), "=", (String) f.getValue()));
        }
        String staff = text(answer, "staff");
        staff = staff.isEmpty() ? "counter" : staff;
        if (Boolean.TRUE.equals(answer.get("worked"))) {
            kb.learn(new Case(facts, List.of(new Conclusion("treatment",
                    (String) row.get(2))), 100), staff);
            out.put("status", "accepted");
            out.put("message", "Good. The advice was right; it is recorded.");
            return close(number, "worked", staff, out);
        }
        String treatment = text(answer, "treatment").strip();
        if (treatment.isEmpty()) {
            out.put("status", "closed");
            out.put("message", "Nothing to learn this time.");
            return close(number, "failed", staff, out);
        }
        String nodeId = (String) row.get(1);
        LearnResult learned;
        try {
            for (Object extra : (List<Object>) answer.getOrDefault("extra",
```

Listing 10.4 (Continued) *Advisor.java: the same logic in Java, through JDBC.*

```
                    List.of())) {
                facts.add(Condition.parse((String) extra));
            }
            Case c = new Case(facts, List.of(new Conclusion("treatment",
                    treatment)), 100);
            out.put("status", "learned");
            if (!nodeId.isEmpty()) {
                DifferenceList d = kb.differences(nodeId, c);
                List<Condition> conds = new ArrayList<>();
                for (Condition x : d.a) {
                    conds.add(x.negated());
                }
                conds.addAll(d.b);
                out.put("differences", d.lines());
                out.put("affected", kb.affectedCornerstones(nodeId, conds,
                        c.conclusions));
            }
            learned = kb.learn(c, staff);
        } catch (VcirsException e) {
            out.clear();
            out.put("status", "error");
            out.put("message", e.getMessage());
            return out;
        }
        try (Connection con = connect()) {
            update(con, "INSERT OR IGNORE INTO Treatment VALUES (?, ?, ?)",
                    treatment, text(answer, "steps"), text(answer, "caution"));
        }
        out.put("result", learned.toString());
        return close(number, "taught", staff, out);
    }

    static String text(Map<String, Object> map, String key) {
        Object v = map.get(key);
        return v == null ? "" : v.toString();
    }
```

Listing 10.4 (Continued) *Advisor.java: the same logic in Java, through JDBC.*

```
    Map<String, Object> close(Object number, String outcome, String staff,
                              Map<String, Object> out) throws SQLException {
        try (Connection con = connect()) {
            update(con, "UPDATE Consultation SET Outcome = ?, Staff = ? "
                    + "WHERE ConsultationID = ?", outcome, staff, number);
        }
        store.save(kb);
        out.put("summary", kb.summary());
        return out;
    }

    // ----- looking at the knowledge --------------------------------------
    Map<String, Object> overview() throws SQLException {
        List<Object> treatments = new ArrayList<>();
        Map<String, Long> outcomes = new LinkedHashMap<>();
        List<Object> top = new ArrayList<>();
        try (Connection con = connect()) {
            for (List<Object> r : query(con, "SELECT Name, Steps, Caution "
                    + "FROM Treatment ORDER BY Name")) {
                Map<String, Object> t = new LinkedHashMap<>();
                t.put("name", r.get(0));
                t.put("steps", r.get(1));
                t.put("caution", r.get(2));
                treatments.add(t);
            }
            for (List<Object> r : query(con, "SELECT Outcome, COUNT(*) FROM "
                    + "Consultation GROUP BY Outcome")) {
                outcomes.put((String) r.get(0), ((Number) r.get(1)).longValue());
            }
            for (List<Object> r : query(con, "SELECT Treatment, COUNT(*) AS n "
                    + "FROM Consultation WHERE Treatment <> '' "
                    + "GROUP BY Treatment ORDER BY n DESC, Treatment LIMIT 5")) {
                Map<String, Object> t = new LinkedHashMap<>();
                t.put("treatment", r.get(0));
                t.put("times", ((Number) r.get(1)).longValue());
                top.add(t);
            }
        }
        Map<String, Object> counts = new LinkedHashMap<>();
        long total = 0;
        for (long n : outcomes.values()) {
            total += n;
        }
        counts.put("total", total);
        for (String o : List.of("worked", "taught", "failed", "open")) {
            counts.put(o, outcomes.getOrDefault(o, 0L));
```

Listing 10.4 (Continued) *Advisor.java: the same logic in Java, through JDBC.*

```
        }
        List<Object> history = new ArrayList<>(kb.history.subList(
                Math.max(0, kb.history.size() - 8), kb.history.size()));
        Collections.reverse(history);
        Map<String, Object> out = new LinkedHashMap<>();
        out.put("summary", kb.summary());
        out.put("tree", Report.tree(kb));
        out.put("consultations", counts);
        out.put("mostGiven", top);
        out.put("history", history);
        out.put("treatments", treatments);
        return out;
    }
}
```

The Java version has the same methods in the same order, with a few differences that Java asks for. JDBC has no one-line equivalent of Python's `con.execute(sql, values)`, so two small helpers do the work: `update` runs an `INSERT` or `UPDATE` with `?` placeholders filled from its arguments, and `query` runs a `SELECT` and returns its rows as lists. The placeholders matter: the values typed by the staff are never pasted into the SQL text, so a treatment called `it's` or something stranger cannot break the statement. After an `INSERT`, `SELECT last_insert_rowid()` gives the number SQLite chose for the new row. The asker is a lambda, `q -> { ... }`, and `NeedAnswer` extends `RuntimeException`, the kind of exception a lambda may throw without declaring it. `compact` writes the answers as JSON text exactly as Python's `json.dumps` writes them, so that the two languages store the same text in the Facts column and their databases are identical, row for row.

10.5 Step 3: A Day at the Laundry, Without a Browser

With the Advisor finished, we can replay a whole morning before writing any web code. The file `data/tour.txt` lists fifteen requests, one per line, exactly as the page will send them. **Listing 10.5** reads each line, calls the Advisor as the server will, and prints the request and the JSON reply.

Listing 10.5 `advisor_demo.py`: the fifteen requests of the tour, replayed on the Advisor.

```
"""Replay the requests of ../data/tour.txt on the Advisor, without a web
server, and print every answer as the server would send it.

    python advisor_demo.py
"""

import json
import os
import pathlib
import sys

from advisor import Advisor, Clock
from vcirs import jsonio

DATA = pathlib.Path(__file__).resolve().parents[1] / "data"
DB = str(DATA / "demo.db")

if os.path.exists(DB):
    os.remove(DB)                          # a fresh laundry every time
advisor = Advisor(DB, str(DATA), Clock("2026-09-28 08:00"))
with open(DATA / "tour.txt", encoding="utf-8") as f:
    for line in f:
        method, path, *rest = line.strip().split(" ", 2)
        body = json.loads(rest[0]) if rest else {}
        if path == "/api/consult":
            out = advisor.consult(body.get("facts", {}),
                                  body.get("unknown", []),
                                  body.get("staff", ""))
        elif path == "/api/feedback":
            out = advisor.feedback(body)
        else:
            out = advisor.overview()
        print("> " + line.strip())
        print(jsonio.dumps(out))
```

Listing 10.6 does the same in Java; it borrows three small helpers from the server of Section 10.6, which convert the parsed JSON body into the maps and lists the Advisor expects.

Listing 10.6 `AdvisorDemo.java`: the tour in Java.

```
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;
import java.util.Map;

import vcirs.Json;

/**
 * Replay the requests of ../data/tour.txt on the Advisor, without a web
 * server, and print every answer as the server would send it.
 * java -cp "out:lib/sqlite-jdbc-3.53.2.0.jar" AdvisorDemo  (Windows: ;)
 */
public class AdvisorDemo {
    public static void main(String[] args) throws Exception {
        Path data = Path.of("..", "data");
        String db = data.resolve("demo.db").toString();
        Files.deleteIfExists(Path.of(db));     // a fresh laundry every time
        Advisor advisor = new Advisor(db, data.toString(),
                new Advisor.Clock("2026-09-28 08:00"));
        for (String line : Files.readAllLines(data.resolve("tour.txt"),
                StandardCharsets.UTF_8)) {
            String[] part = line.strip().split(" ", 3);
            Map<String, Object> body = part.length > 2
                    ? LaundryServer.object(part[2]) : Map.of();
            Map<String, Object> out;
            if (part[1].equals("/api/consult")) {
                out = advisor.consult(LaundryServer.strings(body, "facts"),
                        LaundryServer.list(body, "unknown"),
                        Advisor.text(body, "staff"));
            } else if (part[1].equals("/api/feedback")) {
                out = advisor.feedback(body);
            } else {
                out = advisor.overview();
            }
            System.out.println("> " + line.strip());
            System.out.println(Json.write(out));
        }
    }
}
```

The two programs print the same 353 lines, byte for byte, and leave behind the same database. Here is the morning, request by request. The first request has no answers at all:

```
> POST /api/consult {"facts": {}, "unknown": []}
{
  "status": "question",
  "variable": "stain",
  "question": "What is the stain?",
  "choices": [
    "oil",
    "blood",
    "tea",
    "ink",
    "gum",
    "wax",
    "mud",
    "turmeric",
    "rust"
  ],
  "why": [
    "I need to know 'stain'.",
    "Rule dish soap and a warm wash#1 needs stain = oil;",
    "if it holds, the rule concludes treatment = dish soap and a warm wash."
  ]
}
```

The engine walks the top-level nodes in the order they were taught; the first one, *dish soap and a warm wash*, needs `stain = oil`, the stain is not known, and the asker stops the engine. The reply carries the question and the engine's reason for asking it. Dewi answers *oil*:

```
> POST /api/consult {"facts": {"stain": "oil"}, "unknown": []}
{
  "status": "advice",
  "consultation": 1,
  "treatment": "dish soap and a warm wash",
  "steps": "Rub a drop of dish soap into the stain with your fingers. Leave it for 10
minutes. Wash in the warmest water the care label allows.",
  "caution": "Check the stain before drying; heat sets what is left.",
  "node": "dish soap and a warm wash#1",
  "how": [
    "dish soap and a warm wash#1 fired because:",
    "  stain = oil  (answered by the user, CF 100)",
    "  so treatment = dish soap and a warm wash  [rule CF 100, conclusion CF 100]"
  ]
}
```

Now the first node fires; it has no exception, so there is nothing more to ask, and the advice comes back with its steps, its caution and its explanation. The consultation is recorded as number 1. After the wash the shirt is clean, and Dewi says so:

```
> POST /api/feedback {"consultation": 1, "worked": true, "staff": "Dewi"}
{
  "status": "accepted",
  "message": "Good. The advice was right; it is recorded.",
  "summary": "Stain advisor: 5 variables, 12 nodes, 12 rules"
}
```

The case is recorded as verified. The next customer has blood on a shirt collar. With `stain = blood`, the node *cold rinse and soak* fires, and its exception *enzyme soak* needs to know the age of the stain:

```
> POST /api/consult {"facts": {"stain": "blood"}, "unknown": []}
{
  "status": "question",
  "variable": "age",
  "question": "Is the stain fresh or dried?",
  "choices": [
    "fresh",
    "dried"
  ],
  "why": [
    "I need to know 'age'.",
    "Rule enzyme soak#1 needs age = dried;",
    "if it holds, the rule concludes treatment = enzyme soak."
  ]
}
> POST /api/consult {"facts": {"stain": "blood", "age": "dried"}, "unknown": []}
{
  "status": "advice",
  "consultation": 2,
  "treatment": "enzyme soak",
  "steps": "Soak for 1 to 2 hours in lukewarm water with an enzyme (bio) detergent. Rub
gently, rinse and wash as usual.",
  "caution": "Not for silk or wool: enzymes attack them.",
  "node": "enzyme soak#1",
  "how": [
    "enzyme soak#1 fired because:",
    "  stain = blood  (answered by the user, CF 100)",
    "  age = dried  (answered by the user, CF 100)",
    "  so treatment = enzyme soak  [rule CF 100, conclusion CF 100]"
  ]
}
```

A dried blood stain gets the enzyme soak, and the explanation shows both facts that made the exception fire. (The sixth request, the feedback that it worked, is like the third.) Then comes a stain the advisor has never seen:

```
> POST /api/consult {"facts": {"stain": "lipstick"}, "unknown": []}
{
  "status": "none",
  "consultation": 3,
  "message": "No advice for this stain yet."
}
> POST /api/feedback {"consultation": 3, "worked": false, "treatment": "makeup remover
dab", "steps": "Dab micellar water or makeup remover on the stain with cotton, from the
outside in. Rinse, rub in liquid detergent and wash.", "caution": "Test on a hidden seam
first.", "extra": [], "staff": "Dewi"}
{
  "status": "learned",
  "result": "added: makeup remover dab#1 added at the top level",
  "summary": "Stain advisor: 5 variables, 13 nodes, 13 rules"
}
```

Lipstick matches no node; the reply is `none`, with consultation number 3. Dewi treats it as she knows — makeup remover on cotton — and teaches the advisor with the steps and a caution. Since no node answered, `learn` adds the case as a new top-level node, option 5 of the six options of the 2006 revision of VCIRS that Chapter 3 introduced [9]. The knowledge base now has 13 nodes.

The most interesting lesson of the morning is Joko's. A customer brings a silk blouse with a drop of cooking oil. The advisor, asked about an oil stain, says what it always says:

```
> POST /api/consult {"facts": {"stain": "oil"}, "unknown": []}
{
  "status": "advice",
  "consultation": 4,
  "treatment": "dish soap and a warm wash",
  "steps": "Rub a drop of dish soap into the stain with your fingers. Leave it for 10
minutes. Wash in the warmest water the care label allows.",
  "caution": "Check the stain before drying; heat sets what is left.",
  "node": "dish soap and a warm wash#1",
  "how": [
    "dish soap and a warm wash#1 fired because:",
    "  stain = oil  (answered by the user, CF 100)",
    "  so treatment = dish soap and a warm wash  [rule CF 100, conclusion CF 100]"
  ]
}
```

But silk must not be rubbed with dish soap and washed warm; Joko sprinkles talcum powder on the oil to draw it out, and sends the blouse to the dry cleaner. He teaches this, and adds the one fact that made the difference, `fabric = silk`:

```
> POST /api/feedback {"consultation": 4, "worked": false, "treatment": "powder and dry
cleaner", "steps": "Sprinkle talcum powder thickly on the oil and leave it overnight. Brush
it off and send the item to the dry cleaner.", "caution": "No water and no soap on silk.",
"extra": ["fabric = silk"], "staff": "Joko"}
{
  "status": "learned",
  "differences": [
    "Differences with the cornerstone case of dish soap and a warm wash#1:",
    "  A (only in the old case): (none)",
    "  B (only in the new case): fabric = silk"
  ],
  "affected": [],
  "result": "refined: powder and dry cleaner#1 added as an exception of dish soap and a
warm wash#1",
  "summary": "Stain advisor: 5 variables, 14 nodes, 14 rules"
}
```

This is Ripple Down Rules at work through a web form [10]. The list of difference against the cornerstone case of the node that answered (a plain `stain = oil`) has nothing in section A and `fabric = silk` in section B; no earlier case would change its answer; and the new knowledge becomes an exception of *dish soap and a warm wash*. The next oil stain is handled differently:

```
> POST /api/consult {"facts": {"stain": "oil"}, "unknown": []}
{
  "status": "question",
  "variable": "fabric",
  "question": "What is the fabric?",
  "choices": [
    "cotton",
    "polyester",
    "denim",
    "silk",
    "wool"
  ],
  "why": [
    "I need to know 'fabric'.",
    "Rule powder and dry cleaner#1 needs fabric = silk;",
    "if it holds, the rule concludes treatment = powder and dry cleaner."
  ]
}
> POST /api/consult {"facts": {"stain": "oil", "fabric": "silk"}, "unknown": []}
{
  "status": "advice",
  "consultation": 5,
  "treatment": "powder and dry cleaner",
  "steps": "Sprinkle talcum powder thickly on the oil and leave it overnight. Brush it off
and send the item to the dry cleaner.",
  "caution": "No water and no soap on silk.",
  "node": "powder and dry cleaner#1",
  "how": [
    "powder and dry cleaner#1 fired because:",
    "  stain = oil  (answered by the user, CF 100)",
    "  fabric = silk  (answered by the user, CF 100)",
    "  so treatment = powder and dry cleaner  [rule CF 100, conclusion CF 100]"
  ]
}
```

The advisor now asks about the fabric — only for oil stains, because the new node sits under the oil node — and for silk it gives Joko's treatment. Notice the order of the choices: the declared answers, with `silk` among them. The last consultation of the morning shows "I do not know":

```
> POST /api/consult {"facts": {"stain": "ink"}, "unknown": ["fabric"]}
{
  "status": "advice",
  "consultation": 6,
  "treatment": "rubbing alcohol from the back",
  "steps": "Put a clean towel under the stain. Dab rubbing alcohol on the back of the stain
with cotton, moving to a clean part of the towel as the ink lifts. Rinse and wash.",
  "caution": "Test on a hidden seam first. Keep away from flames.",
  "node": "rubbing alcohol from the back#1",
  "how": [
    "rubbing alcohol from the back#1 fired because:",
    "  stain = ink  (answered by the user, CF 100)",
    "  so treatment = rubbing alcohol from the back  [rule CF 100, conclusion CF 100]"
  ]
}
```

A customer brought a uniform with an ink line and nobody could say what the fabric was. The page sent `fabric` in `unknown`; the asker answered "unknown" instead of stopping, the silk exception did not fire, and the general ink treatment came back. Section 10.9 returns to whether this is the right behavior. The last request, `GET /api/kb`, returns the overview; its first part is:

```
  "summary": "Stain advisor: 5 variables, 14 nodes, 14 rules",
  "consultations": {"total": 6, "worked": 3, "taught": 2, "failed": 0, "open": 1},
  "mostGiven": [{"treatment": "dish soap and a warm wash", "times": 2}, ...]
```

Six consultations: three worked at once, two taught the advisor something, and one — the ink — is still open, waiting for the uniform to come out of the wash.

10.6 Step 4: The Web Server

Now the web. A server has two jobs here: to send the pages (three files: `index.html`, `kb.html` and `style.css`) and to answer the three JSON calls by passing them to the Advisor. **Figure 10.3** shows how the parts fit together.

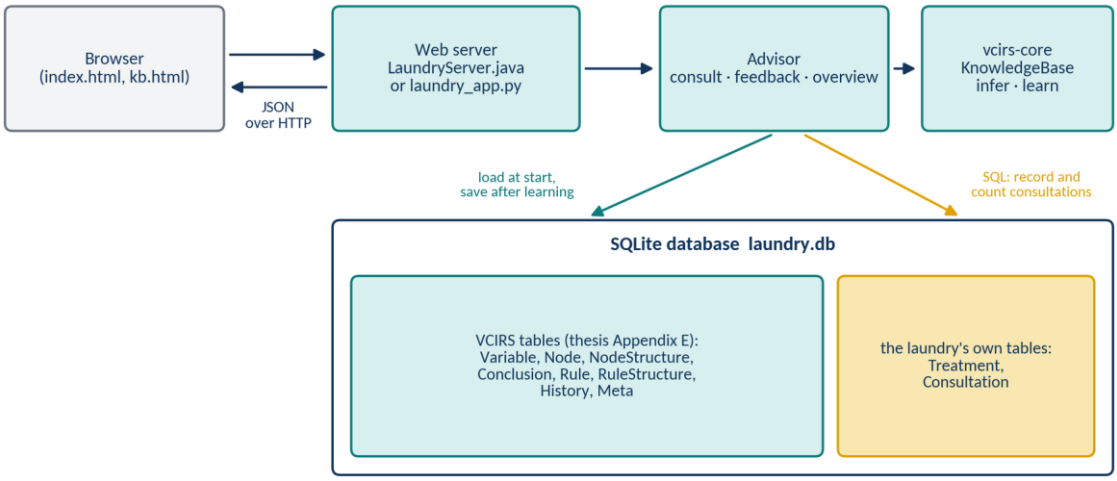

Figure 10.3 The stain advisor as a web app. The browser and the server speak JSON over HTTP; the Advisor consults and teaches the vcirs-core knowledge base, which is loaded from and saved to the VCIRS tables, and records consultations in the laundry's own tables.

Listing 10.7 is the Java server. It uses `com.sun.net.httpserver.HttpServer`, a small HTTP server that has been part of the JDK for many years (in the module `jdk.httpserver`) and needs nothing to be installed [11].

Listing 10.7 `LaundryServer.java`: the web server in Java, on the JDK's own `HttpServer`.

```
import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;

import vcirs.Json;
import vcirs.VcirsException;

/**
 * The stain advisor as a web app: three pages and three JSON calls.
 * java -cp "out:lib/sqlite-jdbc-3.53.2.0.jar" LaundryServer
 *      [--port 4100] [--db ../data/laundry.db] [--clock "2026-09-28 08:00"]
 *      [--host 0.0.0.0]    (to let phones on the same Wi-Fi use it too)
 * Then open http://localhost:4100 in a browser.
 */
public class LaundryServer {
    static final Path WEB = Path.of("..", "web");
    static final Map<String, String> PAGES = Map.of(
            "/", "index.html", "/kb", "kb.html", "/style.css", "style.css");
    static Advisor advisor;

    @SuppressWarnings("unchecked")
    static Map<String, Object> object(String text) {
        Object o = Json.parse(text);
        if (!(o instanceof Map)) {
            throw new VcirsException("the request must be a JSON object");
        }
        return (Map<String, Object>) o;
    }
```

Listing 10.7 (Continued) *LaundryServer.java: the web server in Java, on the JDK's own HttpServer.*

```
    /** body["facts"] as variable -> value text. */
    @SuppressWarnings("unchecked")
    static Map<String, String> strings(Map<String, Object> body, String key) {
        Map<String, String> out = new LinkedHashMap<>();
        Object o = body.getOrDefault(key, Map.of());
        for (Map.Entry<String, Object> e : ((Map<String, Object>) o).entrySet()) {
            out.put(e.getKey(), String.valueOf(e.getValue()));
        }
        return out;
    }

    /** body["unknown"] as a list of texts. */
    @SuppressWarnings("unchecked")
    static List<String> list(Map<String, Object> body, String key) {
        List<String> out = new ArrayList<>();
        for (Object o : (List<Object>) body.getOrDefault(key, List.of())) {
            out.add(String.valueOf(o));
        }
        return out;
    }

    static void send(HttpExchange ex, int code, String type, byte[] bytes)
            throws IOException {
        ex.getResponseHeaders().set("Content-Type", type);
        ex.sendResponseHeaders(code, bytes.length);
        try (OutputStream os = ex.getResponseBody()) {
            os.write(bytes);
        }
    }

    static void page(HttpExchange ex) throws IOException {
        String file = PAGES.get(ex.getRequestURI().getPath());
        if (file == null || !ex.getRequestMethod().equals("GET")) {
            send(ex, 404, "text/plain; charset=utf-8",
                    "Not found".getBytes(StandardCharsets.UTF_8));
            return;
        }
        String type = file.endsWith(".css") ? "text/css; charset=utf-8"
                : "text/html; charset=utf-8";
        send(ex, 200, type, Files.readAllBytes(WEB.resolve(file)));
    }
```

Listing 10.7 (Continued) *LaundryServer.java: the web server in Java, on the JDK's own HttpServer.*

```
    /** Every JSON call: read the body, ask the Advisor, write the answer. */
    static void api(HttpExchange ex) throws IOException {
        String path = ex.getRequestURI().getPath();
        Map<String, Object> out;
        int code = 200;
        try {
            String text = new String(ex.getRequestBody().readAllBytes(),
                    StandardCharsets.UTF_8);
            synchronized (advisor) {                // one change at a time
                if (path.equals("/api/kb")) {
                    out = advisor.overview();
                } else if (!ex.getRequestMethod().equals("POST")) {
                    out = Map.of("status", "error", "message", "use POST");
                } else if (path.equals("/api/consult")) {
                    Map<String, Object> body = object(text);
                    out = advisor.consult(strings(body, "facts"),
                            list(body, "unknown"), Advisor.text(body, "staff"));
                } else if (path.equals("/api/feedback")) {
                    out = advisor.feedback(object(text));
                } else {
                    out = Map.of("status", "error", "message", "no such call");
                }
            }
        } catch (Exception e) {
            out = new LinkedHashMap<>();
            out.put("status", "error");
            out.put("message", String.valueOf(e.getMessage()));
        }
        if ("error".equals(out.get("status"))) {
            code = 400;
        }
        send(ex, code, "application/json",
                (Json.write(out) + "\n").getBytes(StandardCharsets.UTF_8));
    }

    public static void main(String[] args) throws Exception {
        int port = 4100;
        String db = "../data/laundry.db";
        String clock = null;
        String host = "127.0.0.1";                  // this computer only
        for (int i = 0; i + 1 < args.length; i += 2) {
            switch (args[i]) {
                case "--port" -> port = Integer.parseInt(args[i + 1]);
                case "--db" -> db = args[i + 1];
                case "--clock" -> clock = args[i + 1];
                case "--host" -> host = args[i + 1];
```

Listing 10.7 (Continued) *LaundryServer.java: the web server in Java, on the JDK's own HttpServer.*

```
                default -> throw new IllegalArgumentException(args[i]);
            }
        }
        advisor = new Advisor(db, "../data", new Advisor.Clock(clock));
        HttpServer server = HttpServer.create(
                new InetSocketAddress(host, port), 0);
        server.createContext("/", LaundryServer::page);
        server.createContext("/api/", LaundryServer::api);
        server.start();
        System.out.println("Ready: " + advisor.kb.summary());
        System.out.println("Open http://localhost:" + port + " (Ctrl+C stops)");
    }
}
```

`main` reads the options, creates the Advisor, and creates the server on an **address** and a **port**. The address `127.0.0.1` means "this computer only": no other device can connect. The port is a number that tells the programs on one computer apart; this chapter uses 4100, so the advisor's address in a browser is `http://localhost:4100`. `createContext("/", LaundryServer::page)` sends every path to the method `page`, except those that begin with `/api/`, which `createContext("/api/", LaundryServer::api)` sends to `api` (the longest matching prefix wins).

`page` looks the path up in `PAGES`, a map from the three paths the app knows to the three files in the `web` folder, and sends the file with its **content type**, the label that tells the browser whether it is receiving a page or a style sheet. Any other path gets the answer every web user knows, **404 Not found**. Because only these three files can be served, nobody can ask the server for `../data/laundry.db`.

`api` reads the request body, parses it as JSON with vcirs-core's `Json.parse`, calls the Advisor, and writes the reply with `Json.write`, the same canonical JSON writer that saves knowledge bases. The whole call is inside `synchronized (advisor)`, which lets only one request at a time use the Advisor; if Dewi and Joko press their buttons in the same second, one waits a few milliseconds for the other. Any error — a body that is not JSON, a missing consultation — becomes a JSON reply with `"status": "error"` and the HTTP code **400** (a bad request), so the page can always show a message instead of breaking.

Listing 10.8 is the Python server, on **Flask**, the small web framework that most Python programmers meet first. The current version is 3.1.3, released in February 2026 [12]; install it once with `python -m pip install -r requirements.txt` in the `python` folder.

Listing 10.8 `laundry_app.py`: the same web server in Python, on Flask.

```
"""The stain advisor as a web app: three pages and three JSON calls.

    python laundry_app.py [--port 4100] [--db ../data/laundry.db]
                          [--clock "2026-09-28 08:00"]
                          [--host 0.0.0.0]  (phones on the same Wi-Fi too)

Then open http://localhost:4100 in a browser.
"""

import argparse
import pathlib
import threading

from flask import Flask, Response, request, send_from_directory

from advisor import Advisor, Clock
from vcirs import jsonio

HERE = pathlib.Path(__file__).resolve().parents[1]
WEB = str(HERE / "web")
app = Flask(__name__, static_folder=None)
lock = threading.Lock()                  # one change at a time
advisor = None


def reply(out):
    code = 400 if out.get("status") == "error" else 200
    return Response(jsonio.dumps(out) + "\n", status=code,
                    mimetype="application/json")


@app.get("/")
def home():
    return send_from_directory(WEB, "index.html")


@app.get("/kb")
def kb_page():
    return send_from_directory(WEB, "kb.html")


@app.get("/style.css")
def style():
    return send_from_directory(WEB, "style.css")
```

Listing 10.8 (Continued) *laundry_app.py: the same web server in Python, on Flask.*

```
@app.get("/api/kb")
def api_kb():
    with lock:
        return reply(advisor.overview())


@app.post("/api/consult")
def api_consult():
    body = request.get_json(force=True, silent=True)
    if not isinstance(body, dict):
        return reply({"status": "error",
                      "message": "the request must be a JSON object"})
    facts = {k: str(v) for k, v in body.get("facts", {}).items()}
    unknown = [str(v) for v in body.get("unknown", [])]
    with lock:
        return reply(advisor.consult(facts, unknown,
                                     str(body.get("staff", ""))))


@app.post("/api/feedback")
def api_feedback():
    body = request.get_json(force=True, silent=True)
    if not isinstance(body, dict):
        return reply({"status": "error",
                      "message": "the request must be a JSON object"})
    with lock:
        return reply(advisor.feedback(body))


if __name__ == "__main__":
    p = argparse.ArgumentParser()
    p.add_argument("--port", type=int, default=4100)
    p.add_argument("--db", default=str(HERE / "data" / "laundry.db"))
    p.add_argument("--clock", default=None)
    p.add_argument("--host", default="127.0.0.1")   # this computer only
    args = p.parse_args()
    advisor = Advisor(args.db, str(HERE / "data"), Clock(args.clock))
    print("Ready: " + advisor.kb.summary())
    print("Open http://localhost:" + str(args.port) + " (Ctrl+C stops)")
    app.run(host=args.host, port=args.port)
```

In Flask each path is a function with a **decorator** above it: `@app.get("/")` says "call `home` for a GET request to `/`", and `@app.post("/api/consult")` does the same for POST requests. `send_from_directory` sends a file from the `web` folder, refusing any name that would leave it. `request.get_json(force=True, silent=True)` parses the body and gives `None` instead of an error when it is not JSON. `reply` writes the answer with vcirs-core's `jsonio.dumps` — not with Flask's own JSON function — so that the Python server sends exactly the bytes the Java server sends. The lock, `threading.Lock()`, does what `synchronized` does in Java: Flask's development server answers requests in parallel threads, and the lock lets one of them at a time use the Advisor.

Flask prints a warning when it starts: *This is a development server. Do not use it in a production deployment.* The warning is right for a shop on the internet with thousands of visitors. For a laundry with two staff members on its own Wi-Fi, the development server is enough; Chapter 15 shows how to run such an app with a production server when it has to face the world.

10.7 Step 5: The Pages

The pages are the same files for both servers: the Java and the Python server send them without changing a byte. **Listing 10.9** is the advisor page. It is plain HTML with a small JavaScript program, written for any current browser, on a phone or a computer, with no library to download.

Listing 10.9 `index.html`: the advisor page, which keeps the answers and talks to the server.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>Stain advisor</title>
<link rel="stylesheet" href="/style.css">
</head>
<body>
<header><h1>Stain advisor</h1>
  <nav><a href="/">Advisor</a><a href="/kb">Knowledge</a></nav></header>
<main>
  <div class="card"><label for="staff">Your name</label>
    <input id="staff" placeholder="for example Dewi"></div>
  <div class="card" id="answers"></div>
  <div class="card" id="box"></div>
</main>
<script>
// The page keeps the answers; the server keeps the knowledge. Each call
// sends every answer so far, and the server replies with the next question
// or with the advice. So the server remembers nothing between calls.
let facts = {}, unknown = [], consultation = null;
const box = document.getElementById("box");

function el(tag, text, cls) {           // a new element with safe text
  const e = document.createElement(tag);
  if (text !== undefined) e.textContent = text;
  if (cls) e.className = cls;
  return e;
}

function button(text, action, cls) {
  const b = el("button", text, cls);
  b.onclick = action;
  return b;
}

async function call(path, body) {
  const r = await fetch(path, {method: "POST", body: JSON.stringify(body),
    headers: {"Content-Type": "application/json"}});
  return r.json();
}
```

Listing 10.9 (Continued) *index.html: the advisor page, which keeps the answers and talks to the server.*

```
function showAnswers() {
  const a = document.getElementById("answers");
  a.replaceChildren(el("h2", "Answers so far"));
  for (const [v, x] of Object.entries(facts)) a.append(el("span",
      v + " = " + x, "chip"));
  for (const v of unknown) a.append(el("span", v + " = ?", "chip"));
  a.append(el("div"), button("Start again", start, "plain"));
}

async function next() {
  showAnswers();
  const out = await call("/api/consult", {facts: facts, unknown: unknown,
      staff: document.getElementById("staff").value});
  box.replaceChildren();
  if (out.status === "question") {
    box.append(el("h2", out.question));
    for (const c of out.choices) box.append(button(c, () => {
      facts[out.variable] = c; next(); }));
    box.append(button("I do not know", () => {
      unknown.push(out.variable); next(); }, "plain"));
    const other = el("input");
    other.placeholder = "something else? type it and press Enter";
    other.onkeydown = e => { if (e.key === "Enter" && other.value.trim()) {
      facts[out.variable] = other.value.trim(); next(); } };
    box.append(other, el("p", out.why.join(" "), "muted"));
  } else if (out.status === "advice") {
    consultation = out.consultation;
    box.append(el("h2", "Advice: " + out.treatment), el("p", out.steps));
    if (out.caution) box.append(el("div", out.caution, "caution"));
    const why = el("details");
    why.append(el("summary", "Why this advice?"),
               el("pre", out.how.join("\n")));
    box.append(why, el("p", "After the wash: did it work?"),
      button("It worked", worked, "main"),
      button("It did not work", teach));
  } else if (out.status === "none") {
    consultation = out.consultation;
    box.append(el("h2", out.message),
      el("p", "Treat it as you know best. Afterwards, teach the advisor."),
      button("Teach the advisor", teach, "main"));
  } else {
    box.append(el("p", out.message, "error"));
  }
}
```

Listing 10.9 (Continued) *index.html: the advisor page, which keeps the answers and talks to the server.*

```
async function worked() {
  const out = await call("/api/feedback", {consultation: consultation,
      worked: true, staff: document.getElementById("staff").value});
  done(out);
}

function field(label, id, big) {
  const f = el(big ? "textarea" : "input");
  f.id = id;
  return [el("label", label), f];
}

function teach() {
  box.replaceChildren(el("h2", "Teach the advisor"),
    ...field("What removed the stain in the end?", "treatment"),
    ...field("How to do it, step by step", "steps", true),
    ...field("A caution, if any", "caution"),
    ...field("What else was different this time? One per line, " +
             "like: fabric = silk", "extra", true),
    button("Save what I learned", send, "main"));
}

async function send() {
  const value = id => document.getElementById(id).value.trim();
  const extra = value("extra").split("\n").map(s => s.trim())
      .filter(s => s);
  const out = await call("/api/feedback", {consultation: consultation,
      worked: false, treatment: value("treatment"), steps: value("steps"),
      caution: value("caution"), extra: extra,
      staff: document.getElementById("staff").value});
  done(out);
}

function done(out) {
  box.replaceChildren();
  if (out.status === "error") {
    box.append(el("p", out.message, "error"));
    return;
  }
  box.append(el("h2", out.message || "Learned: " + out.result));
  if (out.differences) box.append(el("pre", out.differences.join("\n")));
  if (out.affected && out.affected.length) box.append(el("p",
      "Look again at: " + out.affected.join(", "), "caution"));
  box.append(el("p", out.summary, "muted"),
             button("Next item", start, "main"));
}
```

Listing 10.9 (Continued) *index.html: the advisor page, which keeps the answers and talks to the server.*

```

function start() {
  facts = {}; unknown = []; consultation = null;
  next();
}

start();
</script>
</body>
</html>
```

The HTML part is short: a header with two links, a card for the staff member's name, a card for the answers so far (`answers`) and a card (`box`) where the question, the advice or the teaching form appears. The JavaScript does the rest.

`facts` and `unknown` are the page's memory of the consultation, the two lists of the re-run pattern. `el` creates an element and sets its text with `textContent`, never with `innerHTML`, so that a value typed by a staff member is always shown as text and can never be taken for a piece of a page. `call` sends a request with `fetch`, the browser's built-in function for talking to a server, and waits for the JSON reply (`await`).

`next` is the whole consultation loop. It shows the answers so far, sends them, and looks at the reply's `status`. For a `question` it shows one button per usual answer, an *I do not know* button, a box for any other answer, and the engine's reason in small grey text; pressing a button adds the answer to `facts` (or the variable to `unknown`) and calls `next` again. For `advice` it shows the treatment, the steps, the caution in the amber box, a folded *Why this advice?* section with the explanation, and the two buttons *It worked* and *It did not work*. For `none` it invites the staff to treat the stain as they know best and then to teach the advisor.

`teach` shows the teaching form: the treatment that worked, its steps, a caution, and the other facts that were different this time, one per line. `send` posts it as feedback, and `done` shows what happened: the lesson learned, the list of difference, and, if any earlier cornerstone case would now get another answer, a warning to look at it again. *Next item* starts a new consultation.

Listing 10.10 is the knowledge page, which makes one call, `GET /api/kb`, and shows its four parts as cards and tables.

Listing 10.10 `kb.html`: the knowledge page, with the week's numbers, the tree, the latest changes and the treatments.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>Stain advisor: knowledge</title>
<link rel="stylesheet" href="/style.css">
</head>
<body>
<header><h1>Stain advisor</h1>
  <nav><a href="/">Advisor</a><a href="/kb">Knowledge</a></nav></header>
<main id="main"></main>
<script>
// One call, GET /api/kb, brings everything this page shows.
function el(tag, text, cls) {
  const e = document.createElement(tag);
  if (text !== undefined) e.textContent = text;
  if (cls) e.className = cls;
  return e;
}

function table(headers, rows) {
  const t = el("table"), head = el("tr");
  for (const h of headers) head.append(el("th", h));
  t.append(head);
  for (const r of rows) {
    const tr = el("tr");
    for (const c of r) tr.append(el("td", String(c)));
    t.append(tr);
  }
  return t;
}

function card(title, ...parts) {
  const c = el("div", undefined, "card");
  c.append(el("h2", title), ...parts);
  document.getElementById("main").append(c);
}
```

Listing 10.10 (Continued) *kb.html: the knowledge page, with the week's numbers, the tree, the latest changes and the treatments.*

```
fetch("/api/kb").then(r => r.json()).then(kb => {
  const n = kb.consultations;
  card("This week", el("p", kb.summary),
    table(["Consultations", "Worked", "Taught", "Failed", "Open"],
          [[n.total, n.worked, n.taught, n.failed, n.open]]),
    el("p", "Most given advice:", "muted"),
    table(["Treatment", "Times"],
          kb.mostGiven.map(t => [t.treatment, t.times])));
  card("The knowledge, as a tree of exceptions",
    el("pre", kb.tree.join("\n")));
  card("Latest changes",
    table(["#", "What", "Node", "Option", "Who", "When"],
          kb.history.map(h => [h.seq, h.action, h.node, h.option,
                               h.author, h.time])));
  card("Treatments",
    table(["Name", "Steps", "Caution"],
          kb.treatments.map(t => [t.name, t.steps, t.caution])));
});
</script>
</body>
</html>
```

And **Listing 10.11** is the style sheet both pages share. Its colors are defined once, at the top, as **custom properties** (`--navy`, `--teal`, ...); the rest of the file uses them by name.

Listing 10.11 `style.css`: the look of the two pages.

```
/* The stain advisor's look: the same file for the Java and Python servers. */
:root { --navy: #12355B; --teal: #0F7C7C; --lteal: #D8EFEF;
        --amber: #E0A100; --lamber: #F8E7B0; --red: #C0392B;
        --grey: #6B7280; --lgrey: #F3F4F6; }
body { margin: 0; font-family: system-ui, sans-serif; color: var(--navy);
       background: var(--lgrey); }
header { background: var(--navy); color: white; padding: 12px 16px;
         display: flex; justify-content: space-between; align-items: center; }
header a { color: white; margin-left: 14px; }
main { max-width: 860px; margin: 0 auto; padding: 16px; }
.card { background: white; border-radius: 10px; padding: 16px;
        margin-bottom: 14px; border: 1px solid #dde3ea; }
h1 { font-size: 20px; margin: 0; }
h2 { font-size: 18px; margin: 0 0 10px; }
button { font-size: 16px; padding: 9px 14px; margin: 4px 6px 4px 0;
         border-radius: 8px; border: 1px solid var(--teal);
         background: var(--lteal); color: var(--navy); cursor: pointer; }
button.main { background: var(--teal); color: white; }
button.plain { background: white; border-color: var(--grey); }
input, textarea { font: inherit; padding: 8px; width: 100%;
                  box-sizing: border-box; margin: 4px 0 10px;
                  border: 1px solid #c8d0da; border-radius: 6px; }
label { font-weight: 600; }
.chip { display: inline-block; background: var(--lteal); border-radius: 12px;
        padding: 3px 10px; margin: 2px 4px 2px 0; font-size: 14px; }
.caution { background: var(--lamber); border-left: 4px solid var(--amber);
           padding: 8px 10px; margin: 10px 0; }
.error { color: var(--red); }
pre { background: var(--lgrey); padding: 10px; white-space: pre-wrap;
      font-size: 13px; }
table { border-collapse: collapse; width: 100%; font-size: 14px; }
td, th { border-bottom: 1px solid #e5e7eb; padding: 5px; text-align: left;
         vertical-align: top; }
.muted { color: var(--grey); font-size: 14px; }
```

10.8 Running the Laundry Advisor

Start either server. In Java:

```
cd Chapter10/java
javac -d out -sourcepath ../../vcirs-core/java/src *.java
java -cp "out:lib/sqlite-jdbc-3.53.2.0.jar" LaundryServer
```

or in Python:

```
cd Chapter10/python
python laundry_app.py
```

Both print:

```
Ready: Stain advisor: 5 variables, 12 nodes, 12 rules
Open http://localhost:4100 (Ctrl+C stops)
```

(Flask adds a few lines of its own.) The first start creates `data/laundry.db` from the seed cases; every later start loads it. Open `http://localhost:4100` in a browser. **Figure 10.4** shows the page after Dewi has typed her name and chosen *tea*: the answer is on the list, and the next question is about the color, with the engine's reason under it.

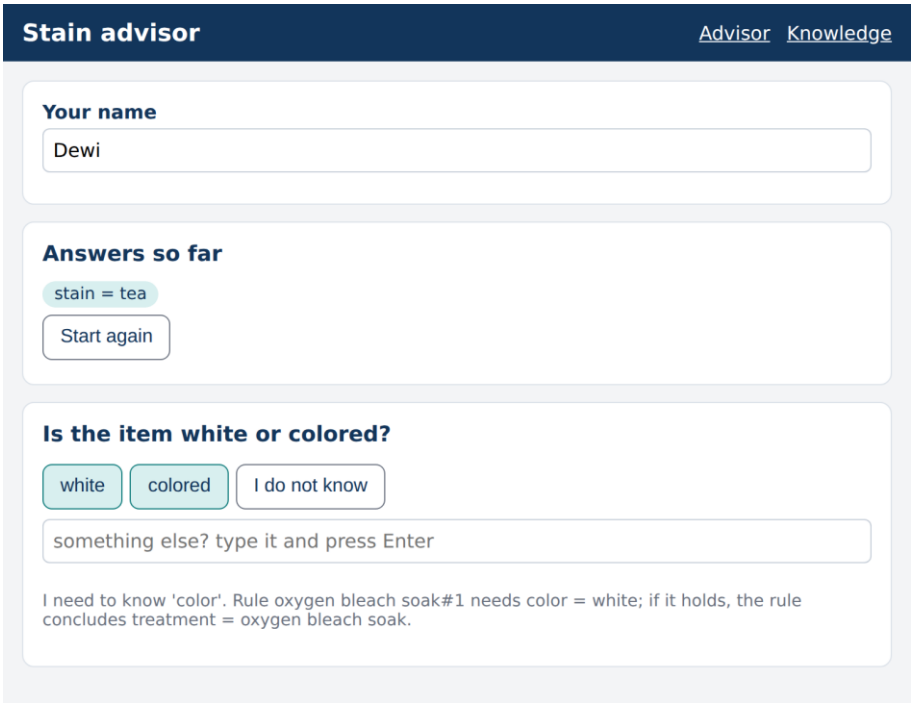

Figure 10.4 The advisor asks its second question. The answers so far are shown as chips; the reason for the question comes from the engine.

She answers *white*, then *dried*. **Figure 10.5** shows the advice, with the caution in the amber box and the explanation unfolded.

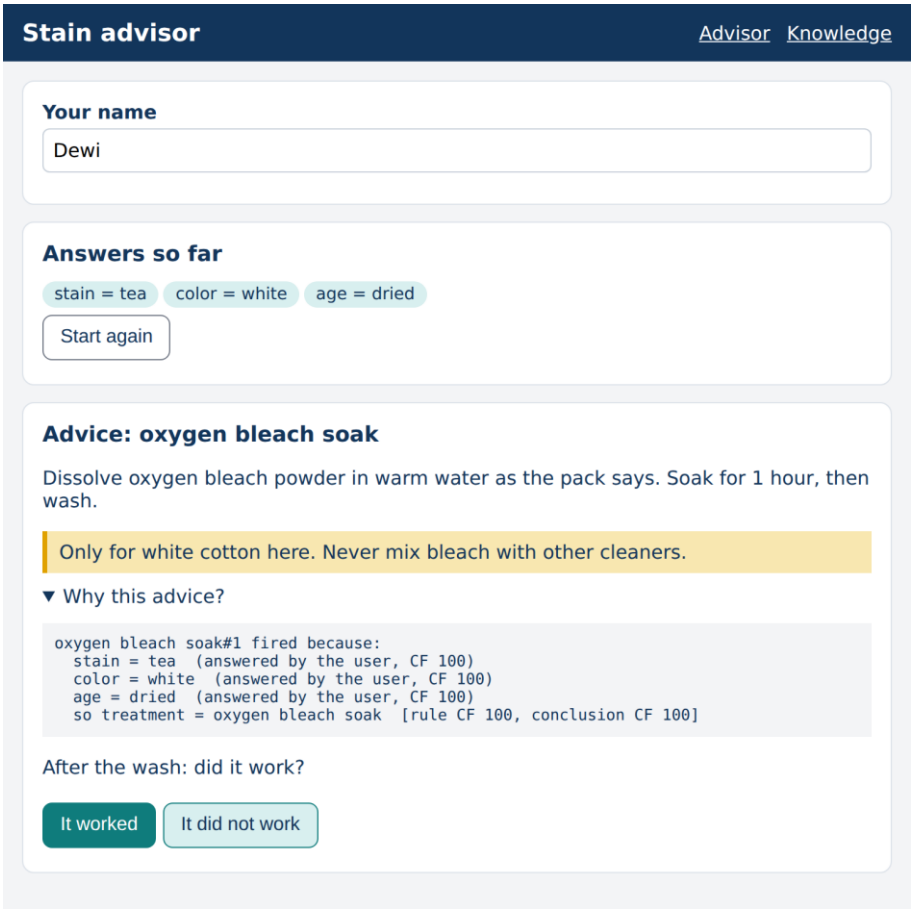

Figure 10.5 The advice for a dried tea stain on a white item: the steps, the caution and, unfolded, the explanation of the node that fired.

The shirt comes out clean, Dewi presses *It worked*, and starts the next item. It is a lipstick stain; she types *lipstick* in the box for other answers, the advisor has no advice, and after treating it she fills in the teaching form of **Figure 10.6**.

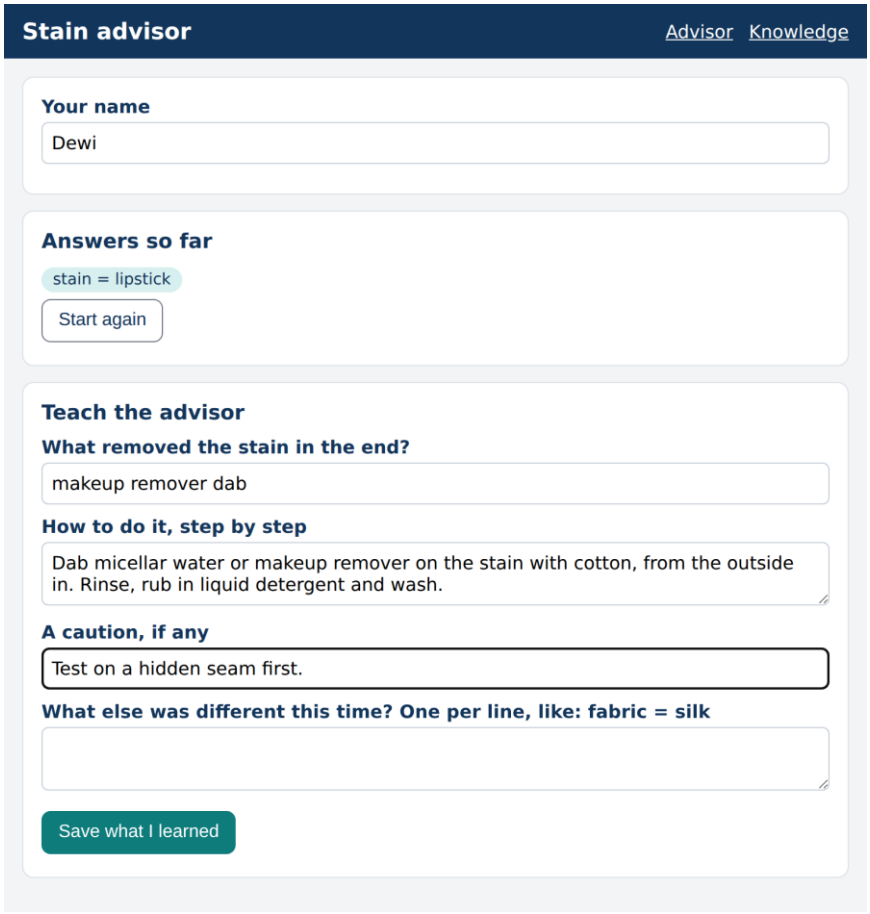

Figure 10.6 Teaching the advisor a treatment for a stain it has never seen. The extra-facts box is empty because the stain itself is what is new.

Figure 10.7 is the knowledge page after these two consultations: the numbers of the day, the tree of nodes and exceptions with the new lipstick node at the bottom, and the latest changes with who made them and when. The seed lessons have no time because they were taught before the laundry opened on Monday.

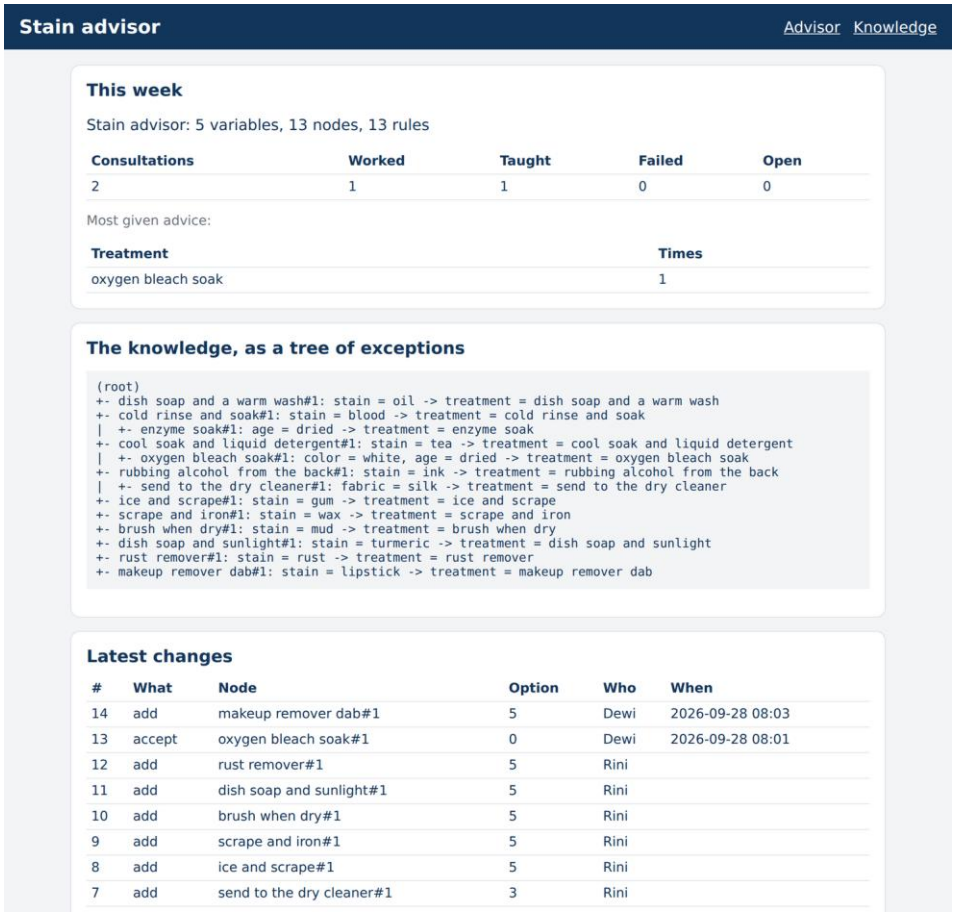

Figure 10.7 The knowledge page: the week's numbers, the most given advice, the knowledge as a tree of exceptions and the latest changes.

10.8.1 Talking to the server without a browser

Every call of the advisor can also be made with **curl**, a small program for sending web requests that is built into Windows 10 and later, macOS and Linux. With the server running, in a second terminal:

```
curl -X POST http://localhost:4100/api/consult -d "{\"facts\": {\"stain\": \"blood\"}}"
```

prints the question about the age of the stain:

```
{
  "status": "question",
  "variable": "age",
  "question": "Is the stain fresh or dried?",
  "choices": [
    "fresh",
    "dried"
  ],
  "why": [
    "I need to know 'age'.",
    "Rule enzyme soak#1 needs age = dried;",
    "if it holds, the rule concludes treatment = enzyme soak."
  ]
}
```

(The backslashes before the inner quotes are needed by the Windows Command Prompt; on macOS and Linux you may write `-d '{"facts": {"stain": "blood"}}'` instead.) `curl http://localhost:4100/api/kb` prints the overview. Replaying the fifteen requests of `tour.txt` this way against the Java server and against the Python server, each starting from an empty

database with the same pretend clock, gives exactly the text of Section 10.5 from both, and the two `laundry.db` files have the same rows in every table.

10.8.2 The phone at the counter

To use the advisor from a phone, start the server with `--host 0.0.0.0`, which means "accept connections from the network too":

```
java -cp "out:lib/sqlite-jdbc-3.53.2.0.jar" LaundryServer --host 0.0.0.0
python laundry_app.py --host 0.0.0.0
```

Then find the laptop's address on the shop's Wi-Fi — `ipconfig` in the Windows Command Prompt (the line *IPv4 Address*), *System Settings → Wi-Fi → Details* on macOS, `hostname -I` on Linux — for example `192.168.1.23`, and open `http://192.168.1.23:4100` on the phone, which must be on the same Wi-Fi. The first time, Windows asks whether Java or Python may accept connections on private networks; allow it for private networks only. The advisor has no password, so use this only on the shop's own Wi-Fi, never on a public one; Exercise 10.8 adds a simple staff PIN.

10.8.3 Tests

`AdvisorTest.java` and `test_advisor.py` check the Advisor without a server, each on a fresh database in a temporary folder: the seed has twelve nodes and asks first about the stain; an exception fires; an unknown answer skips it; a new stain can be taught, and the lesson is still there when the Advisor is opened again on the same database; and a closed consultation cannot be closed twice. The Java tests end with:

```
ok: the seed has 12 nodes
ok: the first question is the stain
ok: an exception fires
ok: an unknown answer skips the exception
ok: a new stain is taught
ok: the lesson is in the database
ok: a closed consultation stays closed
all tests passed
```

10.9 Making It Better

The fabric question for blood. The enzyme soak carries the caution *not for silk or wool*, but the dried-blood node never asks about the fabric. The caution is there to catch the case the knowledge does not yet handle. The right fix is the usual one: the first time a silk item with dried blood comes in, the staff member presses *It did not work* (or, better, does not follow the advice and teaches at once) with `fabric = silk` and the treatment *send to the dry cleaner*, and an exception appears under *enzyme soak*. A caution that keeps being needed is a sign that an exception is missing.

A safer "I do not know". In the last consultation of the tour, an ink stain on an unknown fabric got the general alcohol treatment. For most fabrics that is right, and the caution says to test on a hidden seam first. A more careful laundry may prefer that an unknown fabric be treated like a delicate one. The knowledge base can say so with an exception built from a *negation*: under the ink node, a node whose condition is `fabric <> cotton`… but an unknown fabric does not satisfy that either, because the engine never lets an unknown fact make a condition true. The honest solution is at the page: for variables that matter for safety, remove the *I do not know* button, or show "check the care label" instead. Exercise 10.5 asks you to do this with a small list in the page.

The care label as a variable. Bu Rini would like the advisor to ask about the care label only when it matters. Teach it as knowledge, not as code: with the label variable declared (`label`, *What does the care label say?*, `washable`/`dry clean only`), teach the first *dry clean only* item as an exception wherever it comes up. The questions then appear only where a node needs them — which is the whole promise of an expert system that asks.

Photos. A photograph of the stain before and after would help Bu Rini judge Joko's lessons on Thursday evening. The Consultation table would get a column with the file name, the page an `<input type="file" accept="image/*" capture>` that opens the phone's camera, and the server a fourth call that stores the file in a folder. None of this touches the knowledge base.

Better numbers. The knowledge page counts outcomes; with one more `GROUP BY` it can count them per treatment, and the share of *taught* among all consultations of a week tells whether the advisor is still learning a lot or has become a reliable colleague, as Section 3.9 suggested for the Wi-Fi doctor. Chapter 7 goes much further with the usage statistics of VCIRS itself.

Indonesian words on the page. The questions of this chapter are in English because the variables are; nothing in the engine requires it. A laundry in Surabaya would declare `noda` with the question *Nodanya apa?* and the answers `minyak`, `darah`, `teh`, and so on, and the page's buttons and labels would be translated in `index.html`. The knowledge, the database and the servers stay exactly as they are.

Backups. The whole knowledge of the laundry is one file, `laundry.db`. Copy it once a week to a flash disk or to cloud storage, with the date in its name. Copy it while the server is stopped, or with SQLite's own backup command (`sqlite3 laundry.db ".backup laundry-2026-10-04.db"`), which makes a consistent copy even while the server runs.

> **PITFALL.** Do not open `laundry.db` in a database browser and edit the knowledge tables by hand while the server runs. The server keeps the knowledge base in memory and writes the whole of it back after the next lesson, so your edit would be silently replaced — and a hand edit that breaks a node's structure would not be caught by the engine's checks. Change knowledge through the advisor, where every change is a case with a cornerstone and a name.

10.10 Using It for Real

Setting it up in a shop. An old laptop is enough: the server needs very little memory, and SQLite needs none of its own. Install Java or Python (Chapter 1), copy the `Code_new` folder, and start the server with `--host 0.0.0.0` when the shop opens. On Windows a shortcut to a one-line `.bat` file, and on macOS or Linux a small script, lets anyone start it with a double click. Put a bookmark to the address on the counter phone's home screen, and the advisor behaves like an app.

Seeding your own knowledge. Start with the stains that come every week; ten or twelve cases, as in **Table 10.1**, take less than an hour to write together with the steps. Let the rest be taught at the counter. Write the steps as you would say them to a new employee, in short sentences, and put every warning in the caution: that is what the amber box is for.

A weekly half hour. Once a week, read the knowledge page with the staff. Look at the latest changes: every lesson has a name and a time, and a lesson nobody remembers is worth a question. Look at the open consultations: advice given but never confirmed means either that the staff forget to press the button, or that the advice is doubtful. And look at the most given treatments; if one treatment is given for many different stains, its node may be too general.

Other shops with the same shape. Nothing in the Advisor is about stains except the seed file, the treatments file and the four questions. A shoe-cleaning service (material, stain, color), a tailor (fabric, problem, garment), a car wash (paint, mark, age), a school canteen's hygiene checklist — each is the same app with another seed file. Chapter 14 builds a tool that lets anyone set up such an advisor without touching code.

Only the facts you need. The Consultation table stores the facts of a stain and the first name of the staff member, nothing else: no customer's name, no telephone number, no photo of a person. In Indonesia, the Personal Data Protection Law (UU No. 27/2022) requires that personal data be collected only for a clear purpose and kept only as long as needed [13]; the simplest way to comply is not to collect what the advisor does not need. If the laundry wants to link a consultation to an order, store the order number, which already lives in the order book.

When to call the expert. The advisor gives Bu Rini's treatments; it does not make her unnecessary. Some items deserve her eyes before anyone treats them: anything expensive, anything with a *dry clean only* label, anything old or of unknown fabric, and any stain that has already been through a dryer. A good rule, worth putting in the seed cases as knowledge, is that for these the advice is to set the item aside for her.

10.11 Common Mistakes, and How to Avoid Them

Keeping the conversation in the server. A server that stores "the consultation in progress" in a variable breaks as soon as two people consult at once, or one person reloads the page. *Avoid it:* keep the answers in the page and re-run the engine with all of them, as in Section 10.2.4.

Building SQL by pasting text. `"... WHERE Name = '" + name + "'"` breaks on the first treatment with an apostrophe and lets anyone who can type into the form change the SQL. *Avoid it:* always use `?` placeholders, as both Advisors do.

Forgetting the driver. `No suitable driver found for jdbc:sqlite:...` means the SQLite JDBC `.jar` is not on the class path. *Avoid it:* run with `-cp "out:lib/sqlite-jdbc-3.53.2.0.jar"` (`;` on Windows).

Two servers on one port. *Address already in use* means another program — often the other language's server, still running in another terminal — has port 4100. *Avoid it:* stop the first with Ctrl+C, or start the second with `--port 4101`.

Setting `innerHTML` with typed text. A treatment name typed with angle brackets would be taken as part of the page. *Avoid it:* set text with `textContent`, as `el` does.

Pressing *It worked* before the wash. The advisor records a verified case; if the stain is still there after the wash, the knowledge base has confirmed a mistake. *Avoid it:* give feedback only after the item comes out of the machine; the consultation waits as `open` until then.

Teaching the fact that was the same. Adding `stain = oil` as the "different" fact of an oil stain on silk teaches nothing new; the engine refuses when nothing tells the case apart from the cornerstone case. *Avoid it:* type the fact that really was different, such as `fabric = silk`.

Editing the database by hand while the server runs. See the pitfall in Section 10.9. *Avoid it:* teach through the page; for a large correction, stop the server first.

Opening the server to a public network. With `--host 0.0.0.0` in a café, anyone on the café's Wi-Fi can use and teach the advisor. *Avoid it:* use `--host 0.0.0.0` only on the shop's own Wi-Fi, with a strong Wi-Fi password.

Using different words for the same answer. `Silk`, `silk` and `sutra` are three values to the engine. *Avoid it:* choose from the buttons; they show every value the knowledge base already knows.

10.12 Chapter Summary

- A knowledge base that several people use at once belongs on a **server** with a **database**; browsers on the shop's computers and phones are its clients, and they speak **JSON over HTTP** with it (**Figure 10.3**).
- **SQLite** keeps a whole database in one file, needs no installation, and is the most widely deployed database engine; Python reaches it with `sqlite3` and Java with **JDBC** and the SQLite driver.
- vcirs-core's `SqliteStore` saves a knowledge base in the tables of the thesis's Appendix E — Variable, Node, NodeStructure, Conclusion, Rule, RuleStructure — plus History and Meta, next to the application's own tables; **SQL** queries answer questions about both (**Figure 10.1**, **Listing 10.1**, **Listing 10.2**).
- The **re-run pattern** asks an expert system's questions over the web: the page keeps the answers, the server re-runs the engine with all of them and returns the first open question, stopping the engine by throwing an exception from the asker (**Figure 10.2**).
- The **Advisor** holds all the logic without any web code: `consult`, `feedback` (Ripple-Down-Rules learning with the list of difference and the affected cornerstones) and `overview` (**Table 10.2**, **Listing 10.3**, **Listing 10.4**); replaying fifteen requests on it showed a verified case, an exception, a new top-level node and an "I do not know" (**Listing 10.5**, **Listing 10.6**).
- Two thin servers — the JDK's `HttpServer` and Flask — serve the same pages and answer every call with the same bytes (**Listing 10.7**, **Listing 10.8**); the pages keep the consultation and show questions, advice, cautions, explanations and the teaching form (**Listing 10.9**, **Listing 10.10**, **Listing 10.11**, **Figure 10.4**, **Figure 10.5**, **Figure 10.6**, **Figure 10.7**).
- Safety lives in the knowledge: cautions next to every treatment, the dry cleaner for delicate items, and the rule never to mix cleaning products (**Table 10.1**).

10.13 Exercises

10.1 By hand, using **Table 10.1**, list the questions the advisor asks, in order, and the advice it gives for: (a) fresh blood on a cotton shirt; (b) dried tea on a colored shirt; (c) ink on a silk scarf; (d) a candle-wax drip.

10.2 Write a SQL query that lists every exception node with its parent and the number of its own conditions. Run it on `tour.db` (made by `db_tour.py`) from a small Python program, and explain the result.

10.3 In the re-run pattern the server runs the engine once per answer. For a consultation that ends after q questions, how many times does the engine run in total? Why does the number not matter for the laundry, and when would it start to matter?

10.4 The `Consultation` table records the node that answered. Write a query that counts, for each node, how many consultations it answered and how many of them `worked`. Which node would you review first, and why?

10.5 Change `index.html` so that for the variables in a list `NO_UNKNOWN = ["fabric"]` the page shows, instead of the *I do not know* button, the text *Check the care label first* and no way to skip the question. Why is this change made in the page and not in the knowledge base?

10.6 A customer brings a white cotton shirt with dried blood. Joko follows the enzyme soak, and a faint brown mark remains; Bu Rini finishes it with the oxygen bleach soak. How should Joko teach this, and where will the new node go? Write the feedback request as JSON.

10.7 The server answers errors with the HTTP code 400. Send, with curl, a consult request whose body is not JSON, and then a feedback request for consultation 999. What does each server answer? Why is it useful that both answer in JSON even when something is wrong?

10.8 Add a staff PIN: the server reads a PIN from an option `--pin 2468`, and every POST must carry `"pin": "2468"` in its body, or it gets an error. Which lines of `LaundryServer.java` and `laundry_app.py` change, and where does the page keep the PIN? Why is this enough for a shop's Wi-Fi and not for the internet?

10.9 Stop the Python server, start the Java server on the same `laundry.db`, and continue teaching. Why does this work? What would break if the two servers ran at the same time on the same file, and how does SQLite protect the file itself?

10.14 Solutions to the Exercises

Solution 10.1. (a) *What is the stain?* → blood; the blood node fires and its exception needs the age: *Is the stain fresh or dried?* → fresh; the exception fails, so the advice is *cold rinse and soak*. The fabric is never asked. (b) *stain* → tea; the tea node fires; its exception needs `color = white` first: *Is the item white or colored?* → colored; the exception fails at once without asking the age, and the advice is *cool soak and liquid detergent*. (c) *stain* → ink; the exception asks the fabric → silk; advice *send to the dry cleaner*. (d) *stain* → wax; *scrape and iron*, one question only.

Solution 10.2. The solution program `exceptions.py` is:

```
"""Exercise 10.2: every exception node, its parent and its own conditions.

    python exceptions.py      (after python db_tour.py has made tour.db)
"""

import pathlib
import sqlite3
import sys

sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import report

DB = pathlib.Path(__file__).resolve().parents[1] / "data" / "tour.db"
con = sqlite3.connect(DB)
cur = con.execute(
    "SELECT n.NodeID, n.ParentNodeID, COUNT(s.VariableID) AS Own "
    "FROM Node n JOIN NodeStructure s ON s.NodeID = n.NodeID "
    "WHERE n.ParentNodeID IS NOT NULL "
    "GROUP BY n.NodeID, n.ParentNodeID ORDER BY n.Seq")
for line in report.table([d[0] for d in cur.description], cur.fetchall()):
    print(line)
con.close()
```

Run after `db_tour.py`, it prints:

```
NodeID                   ParentNodeID                    Own
-----------------------  ------------------------------  ---
enzyme soak#1            cold rinse and soak#1           1
oxygen bleach soak#1     cool soak and liquid detergent#1  2
send to the dry cleaner#1  rubbing alcohol from the back#1  1
```

The three exceptions of the seed knowledge each correct one top-level node. `oxygen bleach soak#1` has two conditions of its own because the tea case needed two facts (white, dried) to tell it from the plain tea case; the other two needed one. `COUNT(s.VariableID)` counts only the node's own rows in NodeStructure; the parent's conditions are in the parent's rows.

Solution 10.3. Each answer adds one fact and triggers one more run, and the last run returns the advice, so a consultation with q questions runs the engine $q + 1$ times, and the k-th run re-checks at most the conditions the $(k - 1)$-th run checked plus one. With four variables and fourteen nodes a run takes microseconds; even a thousand nodes would take well under a millisecond, far below the time it takes a person to press a button. It would matter only for a knowledge base whose conditions call slow outside services (a price list on the internet, a sensor), and then the server would keep the facts it looked up, not the conversation.

Solution 10.4. One query does it:

```
SELECT NodeID, COUNT(*) AS Answered,
       SUM(CASE WHEN Outcome = 'worked' THEN 1 ELSE 0 END) AS Worked
FROM Consultation WHERE NodeID <> ''
GROUP BY NodeID ORDER BY Answered DESC, NodeID
```

`SUM(CASE ... END)` counts the rows that satisfy a condition inside each group. Review first the node with many answers and a low share of *worked*: it is used often and fails often, so an exception is probably missing — as `dish soap and a warm wash#1` was before Joko's silk lesson. A node with few answers and no failures can wait.

Solution 10.5. At the top of the script add `const NO_UNKNOWN = ["fabric"];`, and in `next`, replace the line that appends the *I do not know* button with: `if (NO_UNKNOWN.includes(out.variable)) box.append(el("p", "Check the care label first.", "caution")); else box.append(button("I do not know", ...));`. The change belongs in the page because the knowledge base does not know that an answer is missing; it only sees the facts it is given, and an unknown fact makes a condition *not true*, never *dangerous*. Refusing to go on without an answer is a decision about how the app is used at the counter, and the page is where the counter's rules live.

Solution 10.6. The advice was *enzyme soak* from `enzyme soak#1`, and it did not finish the job; the right treatment for this case is the oxygen bleach soak, and the difference from the cornerstone of `enzyme soak#1` (blood, dried) is that the item is white. Joko teaches:

```
{"consultation": 7, "worked": false, "treatment": "oxygen bleach soak",
 "extra": ["color = white"], "staff": "Joko"}
```

(with the right consultation number). The list of difference has `color = white` in section B, and the new node `oxygen bleach soak#2` becomes an exception of `enzyme soak#1`: it will be asked about only for dried blood. The treatment's steps are already in the Treatment table, so Joko need not type them; `INSERT OR IGNORE` keeps the existing ones.

Solution 10.7. `curl -X POST http://localhost:4100/api/consult -d "hello"` returns code 400 with `{"status": "error", "message": ...}`; in Python the message is *the request must be a JSON object*, in Java the message of the JSON parser. `curl -X POST http://localhost:4100/api/feedback -d "{\"consultation\": 999, \"worked\": true}"` returns 400 with *no open consultation 999* from both. JSON errors let the page show a readable

message (`out.message`) instead of failing in the middle of `r.json()`, and they let curl users and tests check the status field the same way for success and failure.

Solution 10.8. In Java, add `case "--pin" -> pin = args[i + 1];` with a static field `pin`, and at the start of the POST branch in `api`: `if (pin != null && !pin.equals(Advisor.text(body, "pin"))) out = Map.of("status", "error", "message", "wrong PIN");` (parsing the body before the branch). In Python, `p.add_argument("--pin")`, and in both POST functions `if args.pin and body.get("pin") != args.pin: return reply({"status": "error", "message": "wrong PIN"})`. The page asks for the PIN once, keeps it in a variable next to `facts`, and adds it to every `call`. On the shop's Wi-Fi this keeps out a curious customer who knows the address; on the internet it is not enough, because the PIN travels as plain text over HTTP and can be read by anyone on the way, and a four-digit PIN can be guessed by a program in seconds. The internet needs HTTPS, real accounts and a limit on attempts (Chapter 15).

Solution 10.9. It works because both servers read and write the same tables with the same meaning: `SqliteStore` in Java and in Python are the two halves of one design, and the crosscheck of Section 10.8.1 shows that they write identical rows. If both ran at once, each would hold its own copy of the knowledge base in memory, and each would write its whole copy after a lesson, so a lesson taught through one server would be overwritten by the next save of the other. SQLite itself stays safe — it locks the file during each transaction, so the file is never half written — but it cannot know that two programs hold two different versions of the same knowledge. One server per database is the rule.

References

[1] SQLite Consortium, "Recent SQLite news," 2026. [Online]. Available: https://sqlite.org/news.html

[2] SQLite Consortium, "Most widely deployed and used database engine," 2026. [Online]. Available: https://sqlite.org/mostdeployed.html

[3] Xerial, "SQLite JDBC driver, release 3.53.2.0," Jun. 2026. [Online]. Available: https://github.com/xerial/sqlite-jdbc/releases

[4] M. M. I. Subakti, "A Variable-Centered Intelligent Rule System," M.Sc.Eng. thesis, Dept. of Computer Science and Information Engineering, Taipei, Taiwan, Jun. 2005.

[5] I. Subakti, "A variable-centered intelligent rule system," in *Proc. 1st Annual Int. Conf. Information and Communication Technology Seminar (ICTS 2005)*, vol. 1, no. 1, Surabaya, Indonesia, Aug. 2005, pp. 167–174.

[6] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, Univ. of California, Irvine, CA, USA, 2000.

[7] American Cleaning Institute, "Stain removal guide," 2026. [Online]. Available: https://www.cleaninginstitute.org/cleaning-tips/clothes/stain-removal-guide

[8] Cornell Cooperative Extension of Tompkins County, "Removing stains at home." [Online]. Available: https://ccetompkins.org/resources/stain-removal-guide

[9] I. Subakti, "Some revisions in VCIRS and cases reconstructing perspectives," in *Proc. Information and Communication Technology Seminar (ICTS 2006)*, Surabaya, Indonesia, 2006.

[10] P. Compton and B. H. Kang, *Ripple-Down Rules: The Alternative to Machine Learning*. Boca Raton, FL, USA: CRC Press, 2021.

[11] Oracle, "HttpServer (Java SE 25 & JDK 25), module jdk.httpserver," 2025. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/jdk.httpserver/com/sun/net/httpserver/HttpServer.html

[12] Pallets Projects, “Flask 3.1.3,” Python Package Index, Feb. 2026. [Online]. Available: https://pypi.org/project/Flask/

[13] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. Jakarta, Indonesia, 2022.

Chapter 11

Clinic Triage Knowledge Base for a Team: Many Experts, Versions and Audit

By the end of this chapter you will be able to explain what goes wrong when several experts change one knowledge base without rules, and build the rules that prevent it: every change is a **proposal** with a reason; it is tried on a copy and **tested** against every cornerstone case and every *golden* case the team has agreed on; a **second expert** reviews it; it becomes a new, never-changed **version**; and every action is written to an **audit trail** whose entries are chained with SHA-256 hashes, so that nobody can quietly rewrite the past. You will build this team knowledge base in Java and in Python on vcirs-core for the front desk of a small clinic, where it sorts patients by urgency — RED, YELLOW or GREEN — without ever naming an illness, and you will run a whole week of the clinic's team with it: a rejected proposal, a refused self-review, a mistake, a rollback, and the question every team eventually asks: *who changed this, and why?*

11.1 The Front Desk of Klinik Sehat Bersama

Klinik Sehat Bersama is a small primary clinic on the edge of a town: a doctor, a nurse, a midwife and, at the front desk, Sari, who registers every patient. From seven in the morning the benches fill up. Most patients have ordinary complaints — a cough, a rash, a follow-up, a fever of a day — and can wait their turn. A few cannot. A man who says his chest feels tight, a baby of two months who is hot to the touch, a pregnant woman who is bleeding: these must be seen now, and the minutes between the front door and the doctor are the minutes that matter.

Sari is not a nurse, and nobody expects her to recognize illnesses. What she needs is simpler and more important: to know, from a few questions, whether a patient must be seen **now**, **soon**, or **in turn**. The clinic has a poster by the desk that dr. Maya wrote years ago, with the danger signs in red. The poster is good, but it has three problems. It cannot ask questions; Sari has to remember to look at it. It cannot learn: when Ners Budi saw a patient with four days of fever and bleeding gums, whom the poster would have sent to the ordinary queue, he told Sari, but the poster stayed the same. And nobody knows which parts of it are still what the team agrees on, because the corrections in pen have no names and no dates.

The team would like an expert system at the front desk, and after Chapters 3 and 10 they know what kind: one that asks only what it needs, explains its answer, and learns from the cases where it was wrong. But they also know something the laundry did not have to think about. In a clinic, **three experts** will teach the same knowledge base, and a bad lesson is not a ruined shirt. They want to know who taught each piece of knowledge and why; they want a second pair of eyes on every change; they want to be sure that a new lesson does not silently change the answer for patients the old lessons were written for; and when a change turns out to be wrong, they want to go back to the knowledge of yesterday, completely and quickly, without losing the record of what happened. This chapter builds exactly that.

11.2 The Idea in Plain Words

11.2.1 Triage: sorting by urgency, not naming illnesses

Triage is the sorting of patients by how soon they need care. Emergency departments around the world use triage scales with three to five levels. The Interagency Integrated Triage Tool published by the World Health Organization, the International Committee of the Red Cross and Médecins Sans Frontières uses three colors: **RED** for patients who need immediate care, **YELLOW** for patients who need care soon, and **GREEN** for patients who can safely wait [1]. Our knowledge base uses the same three colors, and **Table 11.1** shows what each one means at the front desk of a small clinic.

Table 11.1 The three urgency levels of the front-desk knowledge base, and the safe default when no rule matches.

Level	What it means	What the front desk does
RED	must be seen now	Call the nurse or doctor to the patient now. If no clinician is in the clinic, call 119.
YELLOW	must be seen soon	Tell the nurse now; the patient is seen before the queue.
GREEN	can wait	Normal queue. The patient comes back to the desk at once if anything gets worse.
none	no rule matched	Ask the nurse to look at the patient now.

In Indonesia, 119 is the national number for medical emergencies, answered by the Public Safety Centers of the integrated emergency service [2]. The knowledge base contains only **signs that anyone can see or ask about**: is the patient awake, is breathing hard, is there chest pain, is one side of the face drooping, is there bleeding that does not stop, how many days has the fever lasted, are the gums bleeding. It never concludes the name of an illness, and it never tells anyone what medicine to take. It decides one thing: who is seen first. The clinician who sees the patient decides everything else. And when the knowledge base has no rule for a patient, its answer is not GREEN but “ask the nurse to look now” — the safest default a front desk can have.

The signs themselves come from public guidance. The first eight lessons, which dr. Maya takes from the clinic’s poster, follow the danger signs used in emergency triage [1]. The lesson that Ners Budi adds during the week follows the warning signs of severe dengue, which include bleeding from the nose or gums in a patient whose fever has lasted several days, and which call for care at once [3].

11.2.2 Many experts, one knowledge base

Ripple Down Rules were invented for exactly this kind of knowledge: the knowledge of practitioners who add a rule when they see a case the system gets wrong, without a knowledge engineer between them and the system [4]. The pathology systems described in Chapter 3 were maintained by several pathologists at once, and the review of two decades of RDR research counts multiple experts among the normal conditions of RDR, not the exceptions [5]. The cornerstone cases are what makes this possible: every rule carries the case that justified it, so an expert who never met the person who taught an old rule can still see why it is there.

But RDR by itself says nothing about *how* several people share one knowledge base. With nothing more than a shared file, three things go wrong. The first is **lost lessons**: two experts open the same file, each teaches something, each saves, and the second save overwrites the first (Section 11.3 shows it happening). The second is **unreviewed changes**: in RDR a single expert adds a rule on her own judgment; in a clinic, a rule that sends patients to the ordinary queue should not

depend on one person's judgment on one busy morning. The third is **no way back**: RDR never deletes a node, which protects the old cases, but when a whole change turns out to be a mistake the team wants to return to the knowledge as it was, and to know exactly what happened in between.

VCIRS already keeps part of the answer. The thesis's knowledge base records, with every node, who taught it and the case that created it, and vcirs-core adds a **history** of every change to the knowledge base [6], [7]. This chapter builds the rest around it, with four ideas that software teams use every day: versions, tests, review and an audit trail.

11.2.3 Versions: a photograph after every change

A **version** is a complete copy of the knowledge base as it was after one change, saved in a file that is never changed again: `v001.json` after the first lessons, `v002.json` after the next change, and so on. The current knowledge is simply the newest version. Nothing is ever edited in place; a change always produces a new file.

Two things follow. Anyone can look at the knowledge as it was on any day: "what would the front desk have told a patient with these signs on Tuesday?" has an exact answer. And going back is a copy: a **rollback** to version 4 creates a new version, say version 6, whose content is exactly that of version 4. Version 5 is not deleted — it stays in the folder and in the trail, as a record of the mistake — but the front desk now uses the knowledge of version 4 again. This is how the version-control systems that programmers use work, and it is the safest way to undo a change: it never destroys anything.

11.2.4 Tests: cornerstone cases and golden cases

A **test** here is a case whose right answer we know, run through the knowledge base to see whether the knowledge base still gives it. Before any change is accepted, it is tried on a *copy* of the current version, and the copy is tested with two kinds of cases.

The first kind is free: the **cornerstone cases**. Every node of a VCIRS knowledge base carries the case that created it, and that case must still get that node's answer after any change. This is the check that RDR performs when a rule is added — the "affected cornerstones" of Chapter 3 — extended to every node of the knowledge base [8], [4]. A change that would alter the answer of any earlier cornerstone case is exactly the kind of change that silently breaks old knowledge, and it is refused.

The second kind is chosen by the team: the **golden cases**. A golden case is a case the whole team has agreed on, written down with its expected urgency and the reason: "a baby under three months with fever: RED". Golden cases protect what the team considers most important, including situations that no node was created for. When something goes wrong in practice, the right reflex is to write the case down as a golden case, so that the same mistake can never come back unnoticed.

> **DEFINITION.** A *regression test* checks that a change did not break what worked before. In a team knowledge base, the regression tests are the cornerstone cases of all nodes, which must keep their nodes' answers, and the golden cases, which must keep their expected answers.

11.2.5 Four eyes: review by another expert

A change that passes the tests is not yet part of the knowledge. It waits as a **proposal** until a second expert **reviews** it and approves or rejects it; nobody may review their own proposal. This

is the *four-eyes principle* that banks and hospitals use for important decisions. The tests catch changes that would break old cases; the reviewer catches what tests cannot: a rule that is too broad, a reason that does not convince, a fact chosen from the list of difference that is not really the reason. **Figure 11.1** shows the life of a change from proposal to version.

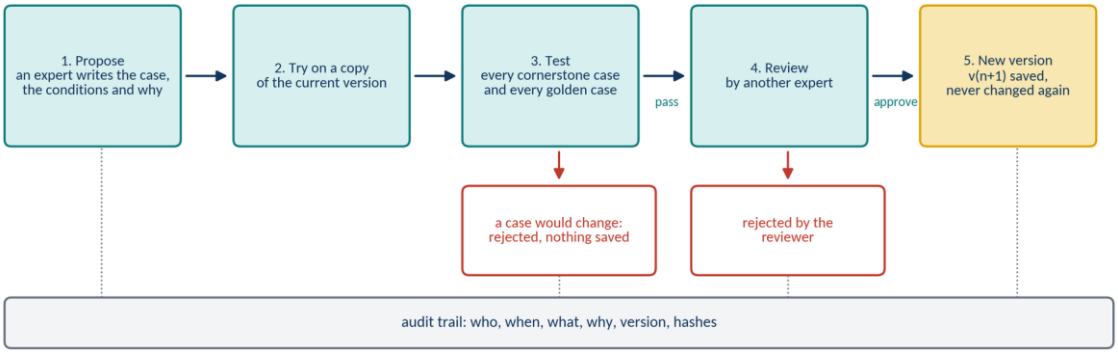

Figure 11.1 The life of a change. A proposal is tried on a copy of the current version and tested; if every case still passes, another expert reviews it; only an approved change becomes a new version. Every step, including every rejection, is written to the audit trail.

The team also needs to decide who may do what. **Table 11.2** is the arrangement at Klinik Sehat Bersama; each clinic can choose its own.

Table 11.2 Who may do what in the clinic's team knowledge base.

Action	Who may do it
consult the knowledge base; report a doubtful answer	everyone, including the front desk
add a golden case; propose a change	doctor, nurse, midwife
review a proposal	doctor, nurse, midwife — never their own proposal
create the knowledge base; stop a node; roll back	doctor

11.2.6 The audit trail, and why it is chained

The **audit trail** is the list of everything that happened to the knowledge base: every creation, golden case, report, proposal, review, stop and rollback, with the time, the person, their role, what was done, the reason they gave, and the version that resulted. It answers the questions that matter after an incident: who changed this, when, why, and who agreed?

A trail is only worth something if it cannot be quietly rewritten. Our trail is protected by **hashes**. A cryptographic hash function such as SHA-256 turns any text into a 64-character fingerprint; changing a single character of the text changes the fingerprint completely, and nobody knows how to find a different text with the same fingerprint [9]. Every entry of the trail stores three fingerprints: `sha`, the fingerprint of the version file it created (if any); `prev`, the fingerprint of the entry before it; and `hash`, the fingerprint of the entry itself, computed over all its other fields. **Figure 11.2** shows the chain.

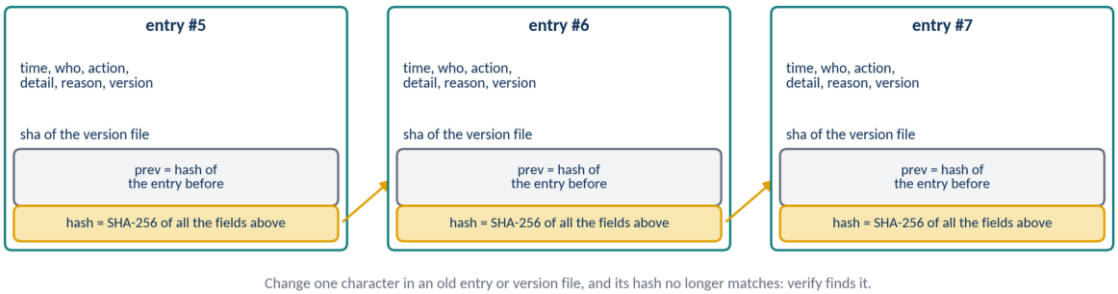

Figure 11.2 The audit trail as a chain. Each entry holds the fingerprint of the version file it created and of the entry before it, and its own fingerprint covers both, so changing any old entry or version file breaks every link after it.

If someone later edits an old entry — changes a reason, removes a name — that entry's fingerprint no longer matches its contents. If they also recompute that fingerprint, the next entry's `prev` no longer matches, and so on down the chain. If someone edits an old version file, its fingerprint no longer matches the `sha` in the trail. A single command, `verify`, recomputes everything and says where the chain breaks. This is the same idea that makes the history of version-control systems and of public ledgers tamper-evident; it does not stop anyone from changing a file, but it guarantees that the change will be seen.

The trail is also a record of **people**, not of patients. It contains the names of the team members, which is its purpose, and the facts of the cases they taught, which are signs such as `fever days = 4`, never a patient's name, number or address. In Indonesia the Personal Data Protection Law requires that personal data be processed only for a stated purpose and protected [10]; health data about identifiable people is among the data it protects most. A knowledge base about signs needs none of it.

11.2.7 When experts disagree

Three experts will not always agree, and a team knowledge base should not pretend they do. Disagreements come in two kinds, and they are handled differently.

The first kind is a disagreement **about a case**: the same facts, and two experts who want different answers. Budi wants a child with a day of fever to be YELLOW; the Friday meeting wants GREEN. No knowledge base can settle this, because it is not a question of knowledge representation but of clinical judgment and of the clinic's capacity. What the knowledge base can do is make the disagreement visible and keep it honest. A proposal that contradicts a golden case fails its tests and says which case; a reviewer who rejects a proposal writes a reason that everyone can read; a golden case that the current knowledge does not satisfy makes every later test fail until someone resolves it. The disagreement is then settled where it should be — in a meeting, by people — and its outcome is written into the knowledge as a proposal or a golden case with the meeting as its reason.

The second kind is a disagreement **about the reason**: two experts agree on the answer but would choose different conditions. Here Ripple Down Rules is generous. Two exceptions with different conditions and the same conclusion can live side by side under the same node, as long as neither changes a cornerstone case of the other, and the tests check exactly that. The usage statistics of VCIRS, which Chapter 7 builds, show later which variables and nodes are used most, and they are a good starting point for a conversation about which reason the team trusts.

A practical rule, until the meeting: **the more urgent answer wins**. If one expert says RED and another YELLOW, the desk treats the patient as RED until the team has decided, because sending

a nurse to look at a patient who could have waited costs a few minutes, and the opposite mistake can cost much more.

11.3 Step 1: Two Experts, One File

Before building anything, let us watch the first problem of Section 11.2.2 happen. In **Listing 11.1**, dr. Maya saves a knowledge base to a shared file; then Ners Budi and Bidan Ratna open the file at the same time, each teaches one lesson, and each saves.

Listing 11.1 `lost_update.py`: two experts change one shared file, and one lesson disappears.

```
"""Two experts change one knowledge-base file at the same time, and one
lesson is lost. Why a team needs more than a shared file.

    python lost_update.py
"""

import os
import pathlib
import sys
import tempfile

sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import Case, KnowledgeBase

path = os.path.join(tempfile.mkdtemp(), "shared-kb.json")
kb = KnowledgeBase("shared")
kb.learn(Case.of(["fever = yes"], ["urgency = GREEN"]), "dr. Maya")
kb.save(path)
print("Monday 08:00 the shared file has: " + ", ".join(kb.nodes))

budi = KnowledgeBase.load(path)          # both open the same file ...
ratna = KnowledgeBase.load(path)
budi.learn(Case.of(["fever = yes", "fever days = 4",
                    "bleeding gums or nose = yes"], ["urgency = RED"]),
           "Ners Budi")
ratna.learn(Case.of(["pregnant = yes", "bleeding in pregnancy = yes"],
                    ["urgency = RED"]), "Bidan Ratna")
budi.save(path)                          # ... and each saves her own copy
print("08:05 Budi saves: " + ", ".join(budi.nodes))
ratna.save(path)
print("08:06 Ratna saves: " + ", ".join(ratna.nodes))

final = KnowledgeBase.load(path)
print("Monday 09:00 the shared file has: " + ", ".join(final.nodes))
kept = [str(n) for n in final.nodes.values()]
lost = [n for n in budi.nodes.values() if str(n) not in kept]
print("Lost without a trace: " + (", ".join(
    str(n) + " by " + n.author for n in lost) or "nothing"))
```

Listing 11.2 is the same in Java.

Listing 11.2 `LostUpdate.java`: the same lost lesson in Java.

```
import java.nio.file.Files;
import java.util.ArrayList;
import java.util.List;

import vcirs.Case;
import vcirs.KnowledgeBase;
import vcirs.Node;

/**
 * Two experts change one knowledge-base file at the same time, and one
 * lesson is lost. Why a team needs more than a shared file.
 */
public class LostUpdate {
    public static void main(String[] args) throws Exception {
        String path = Files.createTempDirectory("team").resolve("shared-kb.json")
                .toString();
        KnowledgeBase kb = new KnowledgeBase("shared");
        kb.learn(Case.of(List.of("fever = yes"), List.of("urgency = GREEN")),
                "dr. Maya");
        kb.save(path);
        System.out.println("Monday 08:00 the shared file has: "
                + String.join(", ", kb.nodes.keySet()));

        KnowledgeBase budi = KnowledgeBase.load(path);  // both open the file ...
        KnowledgeBase ratna = KnowledgeBase.load(path);
        budi.learn(Case.of(List.of("fever = yes", "fever days = 4",
                "bleeding gums or nose = yes"), List.of("urgency = RED")),
                "Ners Budi");
        ratna.learn(Case.of(List.of("pregnant = yes",
                "bleeding in pregnancy = yes"), List.of("urgency = RED")),
                "Bidan Ratna");
        budi.save(path);                    // ... and each saves her own copy
        System.out.println("08:05 Budi saves: "
                + String.join(", ", budi.nodes.keySet()));
        ratna.save(path);
        System.out.println("08:06 Ratna saves: "
                + String.join(", ", ratna.nodes.keySet()));
```

Listing 11.2 (Continued) *LostUpdate.java: the same lost lesson in Java.*

```
        KnowledgeBase last = KnowledgeBase.load(path);
        System.out.println("Monday 09:00 the shared file has: "
                + String.join(", ", last.nodes.keySet()));
        List<String> kept = new ArrayList<>();
        for (Node n : last.nodes.values()) {
            kept.add(n.toString());
        }
        List<String> lost = new ArrayList<>();
        for (Node n : budi.nodes.values()) {
            if (!kept.contains(n.toString())) {
                lost.add(n + " by " + n.author);
            }
        }
        System.out.println("Lost without a trace: "
                + (lost.isEmpty() ? "nothing" : String.join(", ", lost)));
    }
}
```

Both print:

```
Monday 08:00 the shared file has: GREEN#1
08:05 Budi saves: GREEN#1, RED#1
08:06 Ratna saves: GREEN#1, RED#1
Monday 09:00 the shared file has: GREEN#1, RED#1
Lost without a trace: RED#1: fever days = 4, bleeding gums or nose = yes -> urgency = RED
by Ners Budi
```

Budi's lesson — bleeding with a long fever means RED — is gone, and nothing says so. Each program loaded the file, changed its own copy in memory, and wrote its whole copy back; the second write simply replaced the first. There is a second, quieter danger in the output: both new nodes are called `RED#1`, because each program chose the next free name in its own copy. With a plain shared file, two different pieces of knowledge can even end up with the same name.

A database with transactions, as in Chapter 10, prevents two writes from mixing, but it does not by itself prevent this: each program still writes back a copy it loaded before the other's change. What prevents it is a rule: **every change starts from the current version and produces the next one**. The team knowledge base of the next section applies each change to a fresh copy of the newest version, at the moment of the change, and records it as version $n + 1$; there is never a second copy being edited in parallel.

11.4 Step 2: The Team Knowledge Base

The team knowledge base is a folder with three kinds of files:

```
data/clinic/
  versions/v001.json, v002.json, ...   every version, never changed
  state.json                           golden cases, reports and proposals
  audit.json                           the chained audit trail
```

and a class, `TeamKb`, with one method for each action of **Table 11.2**. **Listing 11.3** is the Python version.

Listing 11.3 `team_kb.py`: the team knowledge base, with versions, tests, review, stopping, rollback and a chained audit trail.

```
"""A clinic's triage knowledge base for a team: many experts, versions,
tests against cornerstone and golden cases, and an audit trail.

The folder of a team knowledge base holds:
  versions/v001.json ...  every version of the knowledge, never changed
  state.json              golden cases, reports and proposals
  audit.json              who did what, when and why; each entry is chained
                          to the one before it by a SHA-256 hash
"""

import hashlib
import json
import os
import pathlib
import sys

sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import (BOTH, TOP_LEVEL, Case, Conclusion, Condition,
                   KnowledgeBase, VcirsError, jsonio, report)

EXPERTS = ("doctor", "nurse", "midwife")
ACTIONS = {
    "RED": "Call the nurse or doctor to the patient now. "
           "If no clinician is in the clinic, call 119.",
    "YELLOW": "Tell the nurse now; the patient is seen before the queue.",
    "GREEN": "Normal queue. The patient comes back to the desk at once "
             "if anything gets worse.",
}
NO_MATCH = "No rule matched: ask the nurse to look at the patient now."


class TeamError(Exception):
    """A request the team knowledge base refuses, with the reason."""


def sha(text):
    return hashlib.sha256(text.encode("utf-8")).hexdigest()


def parse_list(text):
    """'a = b, c >= 3' -> [Condition, Condition]; '' -> []."""
    return [Condition.parse(p) for p in text.split(", ") if p.strip()]
```

Listing 11.3 (Continued) *team_kb.py: the team knowledge base, with versions, tests, review, stopping, rollback and a chained audit trail.*

```
def parse_case(text):
    """'fever = yes, age group = baby -> RED' -> (facts, 'RED')."""
    if "->" not in text:
        raise TeamError("write a case as: facts -> RED, YELLOW or GREEN")
    left, level = text.split("->")
    level = level.strip()
    if level not in ACTIONS:
        raise TeamError("the urgency must be RED, YELLOW or GREEN")
    return parse_list(left.strip()), level


def show(conds):
    return ", ".join(str(c) for c in conds)


def place(node):
    return node.parent if node.parent else "(top level)"


def answer(kb, facts):
    """The urgency the knowledge base gives, and the inference result."""
    result = kb.infer(Case(facts).known())
    got = result.conclusions()
    return (got[0].value if got else "none"), result


def teach(kb, facts, level, when, author):
    """Ripple-down teaching: None if the answer is already right; otherwise
    the new node, an exception of the node that answered (or a new
    top-level node), made of the conditions the expert chose ('when')."""
    got, result = answer(kb, facts)
    if got == level:
        return None
    case = Case(facts, [Conclusion("urgency", level)])
    fired = result.paths[0].last().id if result.paths else None
    option = BOTH if fired else TOP_LEVEL
    return kb.refine(case, option, fired, when or None, author)
```

Listing 11.3 (Continued) *team_kb.py: the team knowledge base, with versions, tests, review, stopping, rollback and a chained audit trail.*

```
class TeamKb:
    def __init__(self, folder, team_file):
        self.folder = folder
        with open(team_file, encoding="utf-8") as f:
            team = json.load(f)
        self.members = {m["name"]: m["role"] for m in team["members"]}
        self.clock = lambda: ""
        self.state = {"golden": [], "reports": [], "proposals": []}
        self.audit = []
        os.makedirs(os.path.join(folder, "versions"), exist_ok=True)
        if os.path.exists(self._path("audit.json")):
            with open(self._path("audit.json"), encoding="utf-8") as f:
                self.audit = json.load(f)
            with open(self._path("state.json"), encoding="utf-8") as f:
                self.state = json.load(f)

    # ----- files ------------------------------------------------------------
    def _path(self, name):
        return os.path.join(self.folder, name)

    def _write(self, name, text):
        with open(self._path(name), "w", encoding="utf-8", newline="\n") as f:
            f.write(text)

    def _read(self, name):
        with open(self._path(name), encoding="utf-8") as f:
            return f.read()

    def version(self):
        """The number of the current version (0 before the first)."""
        return self.audit[-1]["version"] if self.audit else 0

    @staticmethod
    def vname(n):
        return "versions/v%03d.json" % n

    def kb(self, n=None):
        """A fresh copy of version n (default: the current version)."""
        return KnowledgeBase.from_json(self._read(self.vname(n or
                                                              self.version())))
```

Listing 11.3 (Continued) *team_kb.py: the team knowledge base, with versions, tests, review, stopping, rollback and a chained audit trail.*

```
    # ----- people, and the trail they leave ------------------------------
    def role(self, who, allowed=None):
        if who not in self.members:
            raise TeamError(who + " is not a member of the team")
        role = self.members[who]
        if allowed is not None and role not in allowed:
            raise TeamError(who + " (" + role + ") may not do this")
        return role

    def _log(self, who, action, detail, reason, version, text=""):
        """Append a chained audit entry; save a new version if text."""
        if text:
            self._write(self.vname(version), text)
        entry = {"n": len(self.audit) + 1, "time": self.clock(), "who": who,
                 "role": self.members.get(who, ""), "action": action,
                 "detail": detail, "reason": reason, "version": version,
                 "sha": sha(text) if text else "",
                 "prev": self.audit[-1]["hash"] if self.audit else ""}
        entry["hash"] = sha(jsonio.dumps(entry))
        self.audit.append(entry)
        self._write("audit.json", jsonio.dumps(self.audit) + "\n")
        self._write("state.json", jsonio.dumps(self.state) + "\n")
        return entry

    # ----- tests ---------------------------------------------------------
    def run_tests(self, kb):
        """Every node's cornerstone case must still get its node's answer,
        and every golden case its expected answer. Returns (count, fails)."""
        fails, count = [], 0
        for node in kb.nodes.values():
            if node.stopped or node.cornerstone is None:
                continue
            count += 1
            want = node.conclusions[0].value
            got, _ = answer(kb, node.cornerstone.facts)
            if got != want:
                fails.append("cornerstone of " + node.id + " ("
                             + show(node.cornerstone.facts) + "): expected "
                             + want + ", got " + got)
        for g in self.state["golden"]:
            count += 1
            got, _ = answer(kb, parse_list(", ".join(g["facts"])))
            if got != g["expected"]:
                fails.append("golden " + g["id"] + " (" + ", ".join(g["facts"])
                             + "): expected " + g["expected"] + ", got " + got)
        return count, fails
```

Listing 11.3 (Continued) *team_kb.py: the team knowledge base, with versions, tests, review, stopping, rollback and a chained audit trail.*

```
    # ----- what the team does -----------------------------------------
    def create(self, who, seed_file, reason):
        self.role(who, ("doctor",))
        if self.audit:
            raise TeamError("the knowledge base already exists")
        with open(seed_file, encoding="utf-8") as f:
            seed = json.load(f)
        kb = KnowledgeBase(seed["name"], seed["description"])
        for var, question, choices in seed["questions"]:
            kb.declare(var, question, choices)
        for step in seed["steps"]:
            teach(kb, parse_list(", ".join(step["facts"])), step["urgency"],
                  parse_list(", ".join(step["when"])), who)
        self._log(who, "create", kb.summary(), reason, 1, kb.to_json())
        return ["v1 created: " + kb.summary()]

    def consult(self, who, facts_text):
        self.role(who)
        got, result = answer(self.kb(), parse_list(facts_text))
        out = [who + " asks about: " + facts_text]
        if got == "none":
            out.append("  urgency: none  (" + NO_MATCH + ")")
            if result.paths:
                out.append("  the last matching rule, "
                           + result.paths[0].last().id + ", is stopped")
            return out
        out.append("  urgency: " + got + "  (rule "
                   + result.paths[0].last().id + ", version v"
                   + str(self.version()) + ")")
        out.append("  what to do: " + ACTIONS[got])
        out.append("  why:")
        out += ["    " + line for line in result.how()]
        return out

    def golden(self, who, case_text, reason):
        self.role(who, EXPERTS)
        facts, level = parse_case(case_text)
        gid = "G" + str(len(self.state["golden"]) + 1)
        self.state["golden"].append({"id": gid, "facts": [str(c) for c in facts],
                                     "expected": level, "by": who,
                                     "reason": reason})
        got, _ = answer(self.kb(), facts)
        self._log(who, "golden", gid + ": " + case_text, reason,
                  self.version())
        note = "passes now" if got == level else "FAILS now: gives " + got
```

Listing 11.3 (Continued) *team_kb.py: the team knowledge base, with versions, tests, review, stopping, rollback and a chained audit trail.*

```
        return [gid + " added (" + note + ")"]

    def report(self, who, facts_text, note):
        self.role(who)
        rid = "R" + str(len(self.state["reports"]) + 1)
        got, _ = answer(self.kb(), parse_list(facts_text))
        self.state["reports"].append({"id": rid, "by": who, "facts": facts_text,
                                      "answer": got, "note": note})
        self._log(who, "report", rid + ": " + facts_text + " (answer "
                  + got + ")", note, self.version())
        return [rid + " recorded for the experts to read"]

    def propose(self, who, case_text, when_text, reason):
        self.role(who, EXPERTS)
        facts, level = parse_case(case_text)
        when = parse_list(when_text)
        pid = "P" + str(len(self.state["proposals"]) + 1)
        p = {"id": pid, "by": who, "case": case_text, "when": when_text,
             "reason": reason, "status": "", "base": self.version(),
             "node": "", "reviewer": ""}
        self.state["proposals"].append(p)
        kb = self.kb()
        node = teach(kb, facts, level, when, who)
        if node is None:
            p["status"] = "already known"
            out = [pid + ": the knowledge base already answers " + level]
        else:
            count, fails = self.run_tests(kb)
            p["node"] = node.id
            if fails:
                p["status"] = "rejected by tests"
                out = [pid + ": would add " + str(node) + " under "
                       + place(node)]
                out.append("  REJECTED: " + str(len(fails)) + " of "
                           + str(count) + " test cases would change:")
                out += ["    " + f for f in fails]
            else:
                p["status"] = "waiting for review"
                out = [pid + ": would add " + str(node) + " under "
                       + place(node)]
                out.append("  all " + str(count) + " test cases still pass; "
                           "waiting for another expert to review")
        self._log(who, "propose", pid + " (" + p["status"] + "): "
                  + case_text, reason, self.version())
        return out
```

Listing 11.3 (Continued) *team_kb.py: the team knowledge base, with versions, tests, review, stopping, rollback and a chained audit trail.*

```
    def review(self, who, pid, decision, reason):
        self.role(who, EXPERTS)
        found = [p for p in self.state["proposals"] if p["id"] == pid]
        if not found or found[0]["status"] != "waiting for review":
            raise TeamError(pid + " is not waiting for review")
        p = found[0]
        if p["by"] == who:
            raise TeamError("nobody may review their own proposal")
        p["reviewer"] = who
        if decision != "approve":
            p["status"] = "rejected by " + who
            self._log(who, "reject", pid, reason, self.version())
            return [pid + " rejected"]
        facts, level = parse_case(p["case"])
        kb = self.kb()                          # the knowledge may have moved on
        node = teach(kb, facts, level, parse_list(p["when"]), p["by"])
        count, fails = self.run_tests(kb) if node else (0, [])
        if node is None or fails:
            p["status"] = "rejected by tests"
            self._log(who, "reject", pid + " (tests)", reason, self.version())
            return [pid + " no longer passes the tests; it was not applied"]
        p["status"] = "approved"
        p["node"] = node.id
        v = self.version() + 1
        self._log(who, "approve", pid + ": " + str(node), reason, v,
                  kb.to_json())
        return [pid + " approved: v" + str(v) + " adds " + node.id
                + " (" + str(count) + " test cases pass)"]

    def stop(self, who, node_id, reason):
        self.role(who, ("doctor",))
        kb = self.kb()
        try:
            kb.stop(node_id, who)
        except VcirsError as e:
            raise TeamError(str(e))
        count, fails = self.run_tests(kb)
        if fails:
            raise TeamError("stopping " + node_id + " breaks: "
                            + "; ".join(fails))
        v = self.version() + 1
        self._log(who, "stop", node_id, reason, v, kb.to_json())
        return ["v" + str(v) + ": " + node_id + " stopped (" + str(count)
                + " test cases pass)"]
```

Listing 11.3 (Continued) *team_kb.py: the team knowledge base, with versions, tests, review, stopping, rollback and a chained audit trail.*

```
    def rollback(self, who, vtext, reason):
        self.role(who, ("doctor",))
        n = int(vtext.lstrip("v"))
        if n < 1 or n >= self.version():
            raise TeamError("roll back to an earlier version, v1 to v"
                            + str(self.version() - 1))
        text = self._read(self.vname(n))
        v = self.version() + 1
        self._log(who, "rollback", "back to v" + str(n), reason, v, text)
        return ["v" + str(v) + " is a copy of v" + str(n)
                + "; nothing in between is lost from the trail"]

    # ----- looking back ------------------------------------------------
    def test(self):
        count, fails = self.run_tests(self.kb())
        if not fails:
            return ["v" + str(self.version()) + ": all " + str(count)
                    + " test cases pass"]
        return ["v" + str(self.version()) + ": " + str(len(fails))
                + " of " + str(count) + " test cases fail:"] \
            + ["  " + f for f in fails]

    def diff(self, a, b):
        old, new = self.kb(int(a.lstrip("v"))), self.kb(int(b.lstrip("v")))
        out = ["from " + a + " to " + b + ":"]
        for n in new.nodes.values():
            if n.id not in old.nodes:
                out.append("  added   " + str(n))
            elif n.stopped and not old.nodes[n.id].stopped:
                out.append("  stopped " + n.id)
        for n in old.nodes.values():
            if n.id not in new.nodes:
                out.append("  gone    " + str(n))
            elif old.nodes[n.id].stopped and not new.nodes[n.id].stopped:
                out.append("  resumed " + n.id)
        return out if len(out) > 1 else out + ["  no change"]

    def tree(self, vtext):
        return report.tree(self.kb(int(vtext.lstrip("v"))))
```

Listing 11.3 (Continued) *team_kb.py: the team knowledge base, with versions, tests, review, stopping, rollback and a chained audit trail.*

```
    def blame(self, node_id):
        kb = self.kb()
        if node_id not in kb.nodes:
            raise TeamError("no node " + node_id + " in v"
                            + str(self.version()))
        n = kb.nodes[node_id]
        out = [str(n), "  under: " + place(n),
               "  taught by: " + n.author,
               "  cornerstone case: " + show(n.cornerstone.facts)]
        for p in self.state["proposals"]:
            if p["node"] == node_id and p["status"] == "approved":
                out.append("  proposal " + p["id"] + ": " + p["reason"])
                out.append("  approved by " + p["reviewer"])
        for e in self.audit:
            if e["sha"] and node_id in e["detail"]:
                out.append("  in the trail: #" + str(e["n"]) + " "
                           + e["time"] + " " + e["action"] + " by "
                           + e["who"] + " -> v" + str(e["version"]))
        return out

    def log(self):
        out = []
        for e in self.audit:
            out.append("#" + str(e["n"]) + " " + e["time"] + " v"
                       + str(e["version"]) + " " + e["who"] + " ("
                       + e["role"] + ") " + e["action"] + ": "
                       + e["detail"])
            out.append("    why: " + e["reason"])
        return out

    def verify(self):
        """Recompute every hash of the trail and of every version file."""
        prev = ""
        for e in self.audit:
            body = {k: v for k, v in e.items() if k != "hash"}
            if e["prev"] != prev or sha(jsonio.dumps(body)) != e["hash"]:
                return ["the trail is broken at entry #" + str(e["n"])]
            if e["sha"] and sha(self._read(self.vname(e["version"]))) \
                    != e["sha"]:
                return ["version v" + str(e["version"]) + " was changed "
                        "after entry #" + str(e["n"])]
            prev = e["hash"]
        return ["the trail is intact: " + str(len(self.audit))
                + " entries, " + str(self.version()) + " versions, last hash "
                + prev[:12]]
```

Read it in five parts.

Cases as text. A case is written the way a person would say it: `fever = yes, age group = baby -> RED`. `parse_list` turns `a = b, c >= 3` into a list of vcirs-core conditions, and `parse_case` splits the arrow and checks that the level is RED, YELLOW or GREEN. The conditions an expert chooses for a new node — the *when* of a proposal — use the same form, so an expert can write `fever days >= 3` where the case itself says `fever days = 3`.

Teaching. `teach` is Ripple Down Rules in six lines, written for the choice the expert makes. It asks the knowledge base for its answer to the case. If the answer is already right, there is nothing to teach. Otherwise it finds the node that answered and refines under it with option 3 of VCIRS (an exception), or, when no node answered, adds a new top-level node with option 5 [7]. If the expert gave *when* conditions, those become the new node's conditions — the choice from the list of difference that the 2006 revision of VCIRS leaves to the user [7]; otherwise the engine builds

them from the list of difference itself. The case is always stored whole as the new node's cornerstone.

The trail. `_log` is the only method that writes. It saves the new version file first (when there is one), then builds the audit entry: the entry number, the time, the person and their role, the action, a detail, the reason, the version the knowledge base is at after the action, `sha` (the fingerprint of the new version file, or empty), and `prev` (the fingerprint of the previous entry). Then it computes `hash` over all those fields, written in canonical JSON with vcirs-core's `jsonio.dumps` so that Java and Python compute the same fingerprint, appends the entry and saves `audit.json` and `state.json`. The current version is simply the version of the last entry.

Tests. `run_tests` runs every cornerstone case of every node that is not stopped and checks that it gets that node's conclusion, then runs every golden case and checks its expected level. It returns how many cases it tried and a readable line for each failure.

Actions. Each action checks the person's role first (`role` raises `TeamError`, which the runner prints as *refused*). `propose` loads a fresh copy of the current version, teaches the case to the copy, runs the tests on the copy, and records the proposal as *waiting for review* or *rejected by tests*; the current version is not touched. `review` refuses a reviewer who made the proposal, and on approval **teaches the case again to a fresh copy of the version that is current now**, because other changes may have been approved since the proposal was made; it runs the tests again and only then saves the next version. `stop` stops a node on a copy, tests, and saves a new version; `rollback` copies an old version file into a new one. `consult`, `report` and `golden` do not change the knowledge, so they log an entry but create no version. And four methods look back: `test`, `diff` (what was added, stopped or resumed between two versions), `blame` (who taught a node, from which case, in which proposal, approved by whom) and `verify`, which walks the chain and recomputes every fingerprint. `tree` prints any version as a tree.

Listing 11.4 is the Java version. It is longer, mostly because Java spells out the maps that Python writes as dictionaries.

Listing 11.4 `TeamKb.java`: the team knowledge base in Java.

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.ArrayList;
import java.util.HexFormat;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.function.Supplier;

import vcirs.Case;
import vcirs.Conclusion;
import vcirs.Condition;
import vcirs.InferenceResult;
import vcirs.Json;
import vcirs.KnowledgeBase;
import vcirs.Node;
import vcirs.Report;
import vcirs.VcirsException;

/**
 * A clinic's triage knowledge base for a team: many experts, versions,
 * tests against cornerstone and golden cases, and an audit trail.
 * The folder of a team knowledge base holds:
 *   versions/v001.json ...  every version of the knowledge, never changed
 *   state.json              golden cases, reports and proposals
 *   audit.json              who did what, when and why; each entry is chained
 *                           to the one before it by a SHA-256 hash
 */
public class TeamKb {
    static final List<String> EXPERTS = List.of("doctor", "nurse", "midwife");
    static final Map<String, String> ACTIONS = Map.of(
        "RED", "Call the nurse or doctor to the patient now. "
            + "If no clinician is in the clinic, call 119.",
        "YELLOW", "Tell the nurse now; the patient is seen before the queue.",
        "GREEN", "Normal queue. The patient comes back to the desk at once "
            + "if anything gets worse.");
    static final String NO_MATCH =
            "No rule matched: ask the nurse to look at the patient now.";
```

Listing 11.4 (Continued) *TeamKb.java: the team knowledge base in Java.*

```
    /** A request the team knowledge base refuses, with the reason. */
    static class TeamError extends RuntimeException {
        TeamError(String why) {
            super(why);
        }
    }

    static String sha(String text) {
        try {
            MessageDigest md = MessageDigest.getInstance("SHA-256");
            return HexFormat.of().formatHex(
                    md.digest(text.getBytes(StandardCharsets.UTF_8)));
        } catch (NoSuchAlgorithmException e) {
            throw new IllegalStateException(e);
        }
    }

    /** "a = b, c >= 3" -> [Condition, Condition]; "" -> []. */
    static List<Condition> parseList(String text) {
        List<Condition> out = new ArrayList<>();
        for (String p : text.split(", ")) {
            if (!p.isBlank()) {
                out.add(Condition.parse(p));
            }
        }
        return out;
    }

    record Taught(List<Condition> facts, String level) {
    }

    /** "fever = yes, age group = baby -> RED" -> (facts, "RED"). */
    static Taught parseCase(String text) {
        if (!text.contains("->")) {
            throw new TeamError("write a case as: facts -> RED, YELLOW or GREEN");
        }
        String[] parts = text.split("->");
        String level = parts[1].strip();
        if (!ACTIONS.containsKey(level)) {
            throw new TeamError("the urgency must be RED, YELLOW or GREEN");
        }
        return new Taught(parseList(parts[0].strip()), level);
    }
```

Listing 11.4 (Continued) *TeamKb.java: the team knowledge base in Java.*

```
    static String show(List<Condition> conds) {
        List<String> parts = new ArrayList<>();
        for (Condition c : conds) {
            parts.add(c.toString());
        }
        return String.join(", ", parts);
    }

    static String place(Node node) {
        return node.parent != null ? node.parent : "(top level)";
    }

    static InferenceResult infer(KnowledgeBase kb, List<Condition> facts) {
        return kb.infer(new Case(facts, List.of(), 100).known());
    }

    /** The urgency the knowledge base gives, or "none". */
    static String level(InferenceResult r) {
        return r.conclusions().isEmpty() ? "none" : r.conclusions().get(0).value;
    }

    /**
     * Ripple-down teaching: null if the answer is already right; otherwise
     * the new node, an exception of the node that answered (or a new
     * top-level node), made of the conditions the expert chose ("when").
     */
    static Node teach(KnowledgeBase kb, List<Condition> facts, String level,
                      List<Condition> when, String author) {
        InferenceResult r = infer(kb, facts);
        if (level(r).equals(level)) {
            return null;
        }
        Case c = new Case(facts, List.of(new Conclusion("urgency", level)), 100);
        String fired = r.paths.isEmpty() ? null : r.paths.get(0).last().id;
        int option = fired != null ? KnowledgeBase.BOTH : KnowledgeBase.TOP_LEVEL;
        return kb.refine(c, option, fired, when.isEmpty() ? null : when, author);
    }

    final Path folder;
    final Map<String, String> members = new LinkedHashMap<>();
    Supplier<String> clock = () -> "";
    Map<String, Object> state = new LinkedHashMap<>();
    List<Object> audit = new ArrayList<>();
```

Listing 11.4 (Continued) *TeamKb.java: the team knowledge base in Java.*

```
    @SuppressWarnings("unchecked")
    TeamKb(String folder, String teamFile) throws IOException {
        this.folder = Path.of(folder);
        Map<String, Object> team = (Map<String, Object>) Json.parse(
                Files.readString(Path.of(teamFile), StandardCharsets.UTF_8));
        for (Object o : (List<Object>) team.get("members")) {
            Map<String, Object> m = (Map<String, Object>) o;
            members.put((String) m.get("name"), (String) m.get("role"));
        }
        state.put("golden", new ArrayList<Object>());
        state.put("reports", new ArrayList<Object>());
        state.put("proposals", new ArrayList<Object>());
        Files.createDirectories(this.folder.resolve("versions"));
        if (Files.exists(this.folder.resolve("audit.json"))) {
            audit = (List<Object>) Json.parse(read("audit.json"));
            state = (Map<String, Object>) Json.parse(read("state.json"));
        }
    }

    // ----- files ------------------------------------------------------------
    void write(String name, String text) {
        try {
            Files.writeString(folder.resolve(name), text, StandardCharsets.UTF_8);
        } catch (IOException e) {
            throw new TeamError("cannot write " + name + ": " + e.getMessage());
        }
    }

    String read(String name) {
        try {
            return Files.readString(folder.resolve(name), StandardCharsets.UTF_8);
        } catch (IOException e) {
            throw new TeamError("cannot read " + name + ": " + e.getMessage());
        }
    }

    @SuppressWarnings("unchecked")
    static Map<String, Object> map(Object o) {
        return (Map<String, Object>) o;
    }

    @SuppressWarnings("unchecked")
    List<Object> list(String key) {
        return (List<Object>) state.get(key);
    }
```

Listing 11.4 (Continued) *TeamKb.java: the team knowledge base in Java.*

```
    /** The number of the current version (0 before the first). */
    int version() {
        return audit.isEmpty() ? 0
                : ((Number) map(audit.get(audit.size() - 1)).get("version"))
                        .intValue();
    }

    static String vname(int n) {
        return String.format("versions/v%03d.json", n);
    }

    /** A fresh copy of version n. */
    KnowledgeBase kb(int n) {
        return KnowledgeBase.fromJson(read(vname(n)));
    }

    KnowledgeBase kb() {
        return kb(version());
    }

    // ----- people, and the trail they leave --------------------------------
    String role(String who, List<String> allowed) {
        if (!members.containsKey(who)) {
            throw new TeamError(who + " is not a member of the team");
        }
        String role = members.get(who);
        if (allowed != null && !allowed.contains(role)) {
            throw new TeamError(who + " (" + role + ") may not do this");
        }
        return role;
    }

    /** Append a chained audit entry; save a new version if text. */
    void log(String who, String action, String detail, String reason,
             int version, String text) {
        if (!text.isEmpty()) {
            write(vname(version), text);
        }
        Map<String, Object> e = new LinkedHashMap<>();
        e.put("n", audit.size() + 1);
        e.put("time", clock.get());
        e.put("who", who);
        e.put("role", members.getOrDefault(who, ""));
        e.put("action", action);
        e.put("detail", detail);
        e.put("reason", reason);
```

Listing 11.4 (Continued) *TeamKb.java: the team knowledge base in Java.*

```
        e.put("version", version);
        e.put("sha", text.isEmpty() ? "" : sha(text));
        e.put("prev", audit.isEmpty() ? ""
                : map(audit.get(audit.size() - 1)).get("hash"));
        e.put("hash", sha(Json.write(e)));
        audit.add(e);
        write("audit.json", Json.write(audit) + "\n");
        write("state.json", Json.write(state) + "\n");
    }

    // ----- tests -------------------------------------------------------
    int lastCount;                    // how many cases the last test run tried

    /**
     * Every node's cornerstone case must still get its node's answer, and
     * every golden case its expected answer. Returns the failures; the
     * number of cases tried is left in lastCount.
     */
    List<String> runTests(KnowledgeBase kb) {
        List<String> fails = new ArrayList<>();
        int count = 0;
        for (Node node : kb.nodes.values()) {
            if (node.stopped || node.cornerstone == null) {
                continue;
            }
            count++;
            String want = node.conclusions.get(0).value;
            String got = level(infer(kb, node.cornerstone.facts));
            if (!got.equals(want)) {
                fails.add("cornerstone of " + node.id + " ("
                        + show(node.cornerstone.facts) + "): expected " + want
                        + ", got " + got);
            }
        }
        for (Object o : list("golden")) {
            Map<String, Object> g = map(o);
            count++;
            String facts = String.join(", ", strings(g.get("facts")));
            String got = level(infer(kb, parseList(facts)));
            if (!got.equals(g.get("expected"))) {
                fails.add("golden " + g.get("id") + " (" + facts
                        + "): expected " + g.get("expected") + ", got " + got);
            }
        }
        lastCount = count;
        return fails;
```

Listing 11.4 (Continued) *TeamKb.java: the team knowledge base in Java.*

```
    }

    @SuppressWarnings("unchecked")
    static List<String> strings(Object o) {
        List<String> out = new ArrayList<>();
        for (Object x : (List<Object>) o) {
            out.add((String) x);
        }
        return out;
    }

    // ----- what the team does -----------------------------------------------
    @SuppressWarnings("unchecked")
    List<String> create(String who, String seedFile, String reason)
            throws IOException {
        role(who, List.of("doctor"));
        if (!audit.isEmpty()) {
            throw new TeamError("the knowledge base already exists");
        }
        Map<String, Object> seed = map(Json.parse(Files.readString(
                Path.of(seedFile), StandardCharsets.UTF_8)));
        KnowledgeBase kb = new KnowledgeBase((String) seed.get("name"),
                (String) seed.get("description"));
        for (Object o : (List<Object>) seed.get("questions")) {
            List<Object> q = (List<Object>) o;
            kb.declare((String) q.get(0), (String) q.get(1), strings(q.get(2)));
        }
        for (Object o : (List<Object>) seed.get("steps")) {
            Map<String, Object> step = map(o);
            teach(kb, parseList(String.join(", ", strings(step.get("facts")))),
                    (String) step.get("urgency"),
                    parseList(String.join(", ", strings(step.get("when")))), who);
        }
        log(who, "create", kb.summary(), reason, 1, kb.toJson());
        return List.of("v1 created: " + kb.summary());
    }
```

Listing 11.4 (Continued) *TeamKb.java: the team knowledge base in Java.*

```
    List<String> consult(String who, String factsText) {
        role(who, null);
        InferenceResult r = infer(kb(), parseList(factsText));
        String got = level(r);
        List<String> out = new ArrayList<>();
        out.add(who + " asks about: " + factsText);
        if (got.equals("none")) {
            out.add("  urgency: none  (" + NO_MATCH + ")");
            if (!r.paths.isEmpty()) {
                out.add("  the last matching rule, " + r.paths.get(0).last().id
                        + ", is stopped");
            }
            return out;
        }
        out.add("  urgency: " + got + "  (rule " + r.paths.get(0).last().id
                + ", version v" + version() + ")");
        out.add("  what to do: " + ACTIONS.get(got));
        out.add("  why:");
        for (String line : r.how()) {
            out.add("    " + line);
        }
        return out;
    }

    List<String> golden(String who, String caseText, String reason) {
        role(who, EXPERTS);
        Taught t = parseCase(caseText);
        String gid = "G" + (list("golden").size() + 1);
        List<String> facts = new ArrayList<>();
        for (Condition c : t.facts()) {
            facts.add(c.toString());
        }
        Map<String, Object> g = new LinkedHashMap<>();
        g.put("id", gid);
        g.put("facts", facts);
        g.put("expected", t.level());
        g.put("by", who);
        g.put("reason", reason);
        list("golden").add(g);
        String got = level(infer(kb(), t.facts()));
        log(who, "golden", gid + ": " + caseText, reason, version(), "");
        return List.of(gid + " added (" + (got.equals(t.level()) ? "passes now"
                : "FAILS now: gives " + got) + ")");
    }
```

Listing 11.4 (Continued) *TeamKb.java: the team knowledge base in Java.*

```
    List<String> report(String who, String factsText, String note) {
        role(who, null);
        String rid = "R" + (list("reports").size() + 1);
        String got = level(infer(kb(), parseList(factsText)));
        Map<String, Object> r = new LinkedHashMap<>();
        r.put("id", rid);
        r.put("by", who);
        r.put("facts", factsText);
        r.put("answer", got);
        r.put("note", note);
        list("reports").add(r);
        log(who, "report", rid + ": " + factsText + " (answer " + got + ")",
                note, version(), "");
        return List.of(rid + " recorded for the experts to read");
    }

    List<String> propose(String who, String caseText, String whenText,
                         String reason) {
        role(who, EXPERTS);
        Taught t = parseCase(caseText);
        List<Condition> when = parseList(whenText);
        String pid = "P" + (list("proposals").size() + 1);
        Map<String, Object> p = new LinkedHashMap<>();
        p.put("id", pid);
        p.put("by", who);
        p.put("case", caseText);
        p.put("when", whenText);
        p.put("reason", reason);
        p.put("status", "");
        p.put("base", version());
        p.put("node", "");
        p.put("reviewer", "");
        list("proposals").add(p);
        KnowledgeBase kb = kb();
        Node node = teach(kb, t.facts(), t.level(), when, who);
        List<String> out = new ArrayList<>();
        if (node == null) {
            p.put("status", "already known");
            out.add(pid + ": the knowledge base already answers " + t.level());
        } else {
            List<String> fails = runTests(kb);
            p.put("node", node.id);
            out.add(pid + ": would add " + node + " under " + place(node));
            if (!fails.isEmpty()) {
                p.put("status", "rejected by tests");
                out.add("  REJECTED: " + fails.size() + " of " + lastCount
```

Listing 11.4 (Continued) *TeamKb.java: the team knowledge base in Java.*

```
                    + " test cases would change:");
            for (String f : fails) {
                out.add("    " + f);
            }
        } else {
            p.put("status", "waiting for review");
            out.add("  all " + lastCount + " test cases still pass; "
                    + "waiting for another expert to review");
        }
    }
    log(who, "propose", pid + " (" + p.get("status") + "): " + caseText,
            reason, version(), "");
    return out;
}

List<String> review(String who, String pid, String decision,
                    String reason) {
    role(who, EXPERTS);
    Map<String, Object> p = null;
    for (Object o : list("proposals")) {
        if (map(o).get("id").equals(pid)) {
            p = map(o);
        }
    }
    if (p == null || !p.get("status").equals("waiting for review")) {
        throw new TeamError(pid + " is not waiting for review");
    }
    if (p.get("by").equals(who)) {
        throw new TeamError("nobody may review their own proposal");
    }
    p.put("reviewer", who);
    if (!decision.equals("approve")) {
        p.put("status", "rejected by " + who);
        log(who, "reject", pid, reason, version(), "");
        return List.of(pid + " rejected");
    }
    Taught t = parseCase((String) p.get("case"));
    KnowledgeBase kb = kb();              // the knowledge may have moved on
    Node node = teach(kb, t.facts(), t.level(),
            parseList((String) p.get("when")), (String) p.get("by"));
    List<String> fails = node == null ? List.of() : runTests(kb);
    if (node == null || !fails.isEmpty()) {
        p.put("status", "rejected by tests");
        log(who, "reject", pid + " (tests)", reason, version(), "");
        return List.of(pid + " no longer passes the tests; it was not "
                + "applied");
```

Listing 11.4 (Continued) *TeamKb.java: the team knowledge base in Java.*

```
        }
        p.put("status", "approved");
        p.put("node", node.id);
        int v = version() + 1;
        log(who, "approve", pid + ": " + node, reason, v, kb.toJson());
        return List.of(pid + " approved: v" + v + " adds " + node.id + " ("
                + lastCount + " test cases pass)");
    }

    List<String> stop(String who, String nodeId, String reason) {
        role(who, List.of("doctor"));
        KnowledgeBase kb = kb();
        try {
            kb.stop(nodeId, who);
        } catch (VcirsException e) {
            throw new TeamError(e.getMessage());
        }
        List<String> fails = runTests(kb);
        if (!fails.isEmpty()) {
            throw new TeamError("stopping " + nodeId + " breaks: "
                    + String.join("; ", fails));
        }
        int v = version() + 1;
        log(who, "stop", nodeId, reason, v, kb.toJson());
        return List.of("v" + v + ": " + nodeId + " stopped (" + lastCount
                + " test cases pass)");
    }

    List<String> rollback(String who, String vtext, String reason) {
        role(who, List.of("doctor"));
        int n = Integer.parseInt(vtext.replaceFirst("^v+", ""));
        if (n < 1 || n >= version()) {
            throw new TeamError("roll back to an earlier version, v1 to v"
                    + (version() - 1));
        }
        String text = read(vname(n));
        int v = version() + 1;
        log(who, "rollback", "back to v" + n, reason, v, text);
        return List.of("v" + v + " is a copy of v" + n
                + "; nothing in between is lost from the trail");
    }
```

Listing 11.4 (Continued) *TeamKb.java: the team knowledge base in Java.*

```
    // ----- looking back ----------------------------------------------------
    List<String> test() {
        List<String> fails = runTests(kb());
        List<String> out = new ArrayList<>();
        if (fails.isEmpty()) {
            out.add("v" + version() + ": all " + lastCount + " test cases pass");
            return out;
        }
        out.add("v" + version() + ": " + fails.size() + " of " + lastCount
                + " test cases fail:");
        for (String f : fails) {
            out.add("  " + f);
        }
        return out;
    }

    List<String> diff(String a, String b) {
        KnowledgeBase old = kb(Integer.parseInt(a.replaceFirst("^v+", "")));
        KnowledgeBase now = kb(Integer.parseInt(b.replaceFirst("^v+", "")));
        List<String> out = new ArrayList<>();
        out.add("from " + a + " to " + b + ":");
        for (Node n : now.nodes.values()) {
            Node o = old.nodes.get(n.id);
            if (o == null) {
                out.add("  added   " + n);
            } else if (n.stopped && !o.stopped) {
                out.add("  stopped " + n.id);
            }
        }
        for (Node o : old.nodes.values()) {
            Node n = now.nodes.get(o.id);
            if (n == null) {
                out.add("  gone    " + o);
            } else if (o.stopped && !n.stopped) {
                out.add("  resumed " + o.id);
            }
        }
        if (out.size() == 1) {
            out.add("  no change");
        }
        return out;
    }
```

Listing 11.4 (Continued) *TeamKb.java: the team knowledge base in Java.*

```
    List<String> tree(String vtext) {
        return Report.tree(kb(Integer.parseInt(vtext.replaceFirst("^v+", ""))));
    }

    List<String> blame(String nodeId) {
        KnowledgeBase kb = kb();
        if (!kb.nodes.containsKey(nodeId)) {
            throw new TeamError("no node " + nodeId + " in v" + version());
        }
        Node n = kb.nodes.get(nodeId);
        List<String> out = new ArrayList<>();
        out.add(n.toString());
        out.add("  under: " + place(n));
        out.add("  taught by: " + n.author);
        out.add("  cornerstone case: " + show(n.cornerstone.facts));
        for (Object o : list("proposals")) {
            Map<String, Object> p = map(o);
            if (p.get("node").equals(nodeId)
                    && p.get("status").equals("approved")) {
                out.add("  proposal " + p.get("id") + ": " + p.get("reason"));
                out.add("  approved by " + p.get("reviewer"));
            }
        }
        for (Object o : audit) {
            Map<String, Object> e = map(o);
            if (!e.get("sha").equals("") && ((String) e.get("detail"))
                    .contains(nodeId)) {
                out.add("  in the trail: #" + e.get("n") + " " + e.get("time")
                        + " " + e.get("action") + " by " + e.get("who")
                        + " -> v" + e.get("version"));
            }
        }
        return out;
    }

    List<String> log() {
        List<String> out = new ArrayList<>();
        for (Object o : audit) {
            Map<String, Object> e = map(o);
            out.add("#" + e.get("n") + " " + e.get("time") + " v"
                    + e.get("version") + " " + e.get("who") + " ("
                    + e.get("role") + ") " + e.get("action") + ": "
                    + e.get("detail"));
            out.add("    why: " + e.get("reason"));
        }
        return out;
```

Listing 11.4 (Continued) *TeamKb.java: the team knowledge base in Java.*

```
    }

    /** Recompute every hash of the trail and of every version file. */
    List<String> verify() {
        String prev = "";
        for (Object o : audit) {
            Map<String, Object> e = map(o);
            Map<String, Object> body = new LinkedHashMap<>(e);
            body.remove("hash");
            if (!e.get("prev").equals(prev)
                    || !sha(Json.write(body)).equals(e.get("hash"))) {
                return List.of("the trail is broken at entry #" + e.get("n"));
            }
            int v = ((Number) e.get("version")).intValue();
            if (!e.get("sha").equals("") && !sha(read(vname(v)))
                    .equals(e.get("sha"))) {
                return List.of("version v" + v + " was changed after entry #"
                        + e.get("n"));
            }
            prev = (String) e.get("hash");
        }
        return List.of("the trail is intact: " + audit.size() + " entries, "
                + version() + " versions, last hash "
                + prev.substring(0, Math.min(12, prev.length())));
    }
}
```

Three details of the Java version deserve a word. The SHA-256 fingerprint comes from the standard library: `MessageDigest.getInstance("SHA-256")` computes it and `HexFormat.of().formatHex` writes it as 64 hexadecimal characters, exactly the text Python's `hashlib.sha256(...).hexdigest()` gives. The state and the trail are kept as the `Map` and `List` objects that vcirs-core's `Json.parse` produces, so reading and writing them keeps every key in its order, and the files the two languages write are identical byte for byte. And `Taught` is a `record`, a two-field value holding the facts and the level of a parsed case.

11.5 Step 3: The Command Runner

The team uses the knowledge base through commands, one per line, with their parts separated by `|`. The same commands can be typed at the keyboard, one at a time, or kept in a file and run in one go, which is how this chapter replays a whole week. **Listing 11.5** is the runner.

Listing 11.5 `triage_team.py`: the command runner, for scripts, typed commands and question-by-question consultations.

```
"""The clinic team's triage knowledge base, driven by commands.

    python triage_team.py [--dir ../data/clinic] [--fresh] week.txt
    python triage_team.py [--dir ../data/clinic] -      (type commands)

One command per line, its parts separated by '|':
  time | 2026-10-05 07:30          pretend it is this time (for the book)
  create | who | seed.json | why   the first version (a doctor)
  consult | who | facts            the urgency for these facts
  ask | who                        the urgency, asking one question at a time
  report | who | facts | note      anyone: "this answer looked wrong"
  golden | who | facts -> LEVEL | why       an expert: a case that must hold
  propose | who | facts -> LEVEL | when | why   an expert: new knowledge
  review | who | P1 | approve or reject | why  another expert decides
  stop | who | node | why          a doctor: stop a node (a new version)
  rollback | who | v3 | why        a doctor: go back to an earlier version
  test, log, verify, diff | v1 | v2, blame | node, tree | v3
"""

import datetime
import os
import pathlib
import shutil
import sys

from team_kb import ACTIONS, NO_MATCH, TeamError, TeamKb
from vcirs import VcirsError

HERE = pathlib.Path(__file__).resolve().parents[1]


class Runner:
    def __init__(self, team, lines, echo):
        self.team = team
        self.lines = lines
        self.echo = echo
        self.now = None
        team.clock = self.time

    def time(self):
        if self.now:
            return self.now
        return datetime.datetime.now().strftime("%Y-%m-%d %H:%M")
```

Listing 11.5 (Continued) *triage_team.py: the command runner, for scripts, typed commands and question-by-question consultations.*

```
    def read(self, prompt):
        print(prompt, end="")
        text = next(self.lines, "").strip()
        if self.echo:
            print(text)
        return text

    def ask(self, who):
        """A consultation with questions, as at the real front desk."""
        self.team.role(who)
        kb = self.team.kb()

        def asker(q):
            var = kb.variables.get(q.variable)
            text = var.question if var and var.question else q.variable
            return self.read(text + " (" + "/".join(q.choices) + ") > ") \
                or None                      # Enter = "I do not know"

        result = kb.infer({}, asker)
        got = result.conclusions()
        if not got:
            return ["urgency: none  (" + NO_MATCH + ")"]
        level = got[0].value
        return ["urgency: " + level + "  (rule " + result.paths[0].last().id
                + ")", "what to do: " + ACTIONS[level]]

    def run(self):
        t = self.team
        for line in self.lines:
            line = line.strip()
            if not line or line.startswith("#"):
                continue
            a = [p.strip() for p in line.split("|")]
            if a[0] == "time":
                self.now = a[1]
                continue
            print("> " + line)
            try:
                commands = {
                    "create": lambda: t.create(a[1], str(HERE / "data" / a[2]),
                                               a[3]),
                    "consult": lambda: t.consult(a[1], a[2]),
                    "ask": lambda: self.ask(a[1]),
                    "report": lambda: t.report(a[1], a[2], a[3]),
                    "golden": lambda: t.golden(a[1], a[2], a[3]),
                    "propose": lambda: t.propose(a[1], a[2], a[3], a[4]),
```

Listing 11.5 (Continued) *triage_team.py: the command runner, for scripts, typed commands and question-by-question consultations.*

```
                    "review": lambda: t.review(a[1], a[2], a[3], a[4]),
                    "stop": lambda: t.stop(a[1], a[2], a[3]),
                    "rollback": lambda: t.rollback(a[1], a[2], a[3]),
                    "test": t.test, "log": t.log, "verify": t.verify,
                    "diff": lambda: t.diff(a[1], a[2]),
                    "blame": lambda: t.blame(a[1]),
                    "tree": lambda: t.tree(a[1]),
                }
                if a[0] not in commands:
                    raise TeamError("unknown command " + a[0])
                out = commands[a[0]]()
            except (TeamError, VcirsError) as e:
                out = ["refused: " + str(e)]
            except IndexError:
                out = ["refused: a part of the command is missing"]
            for text in out:
                print(text)


def main(args):
    folder = str(HERE / "data" / "clinic")
    if "--dir" in args:
        folder = args[args.index("--dir") + 1]
    if "--fresh" in args and os.path.exists(folder):
        shutil.rmtree(folder)                # start the week again
    source = args[-1] if args else "-"
    team = TeamKb(folder, str(HERE / "data" / "team.json"))
    if source == "-":
        Runner(team, iter(sys.stdin), False).run()
    else:
        with open(source, encoding="utf-8") as f:
            Runner(team, iter(f.read().splitlines()), True).run()


if __name__ == "__main__":
    main(sys.argv[1:])
```

`Runner.run` reads each line, skips empty lines and comments, splits the parts, and calls the matching method of `TeamKb` through a small table of commands. A `time` line sets a pretend time, used only so that the book's week can be replayed with the same times; when there is none, the real time is used. Every other command is echoed after `>`, so that a replayed script reads like a transcript, and any refusal is printed as `refused:` with its reason. The command `ask` is the consultation the front desk actually uses: the engine asks its questions one by one, in the order of the knowledge base, and the answers are read from the same source as the commands — the keyboard, or the next lines of the script. **Listing 11.6** is the Java runner.

Listing 11.6 `TriageTeam.java`: the same command runner in Java.

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.Comparator;
import java.util.Iterator;
import java.util.List;
import java.util.stream.Stream;

import vcirs.InferenceResult;
import vcirs.KnowledgeBase;
import vcirs.Variable;
import vcirs.VcirsException;

/**
 * The clinic team's triage knowledge base, driven by commands.
 * java TriageTeam [--dir ../data/clinic] [--fresh] ../data/week.txt
 * java TriageTeam [--dir ../data/clinic] -          (type commands)
 * One command per line, its parts separated by '|'; see triage_team.py
 * or the README for the list.
 */
public class TriageTeam {
    static final Path DATA = Path.of("..", "data");

    final TeamKb team;
    final Iterator<String> lines;
    final boolean echo;
    String now = null;

    TriageTeam(TeamKb team, Iterator<String> lines, boolean echo) {
        this.team = team;
        this.lines = lines;
        this.echo = echo;
        team.clock = this::time;
    }

    String time() {
        return now != null ? now : LocalDateTime.now().format(
                DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm"));
    }
```

Listing 11.6 (Continued) *TriageTeam.java: the same command runner in Java.*

```
    String read(String prompt) {
        System.out.print(prompt);
        String text = lines.hasNext() ? lines.next().strip() : "";
        if (echo) {
            System.out.println(text);
        }
        return text;
    }

    /** A consultation with questions, as at the real front desk. */
    List<String> ask(String who) {
        team.role(who, null);
        KnowledgeBase kb = team.kb();
        InferenceResult r = kb.infer(new java.util.LinkedHashMap<>(), q -> {
            Variable v = kb.variables.get(q.variable);
            String text = v != null && !v.question.isEmpty() ? v.question
                    : q.variable;
            String a = read(text + " (" + String.join("/", q.choices) + ") > ");
            return a.isEmpty() ? null                // Enter = "I do not know"
                    : new vcirs.Fact(q.variable, a, 100, "user");
        });
        if (r.conclusions().isEmpty()) {
            return List.of("urgency: none  (" + TeamKb.NO_MATCH + ")");
        }
        String level = r.conclusions().get(0).value;
        return List.of("urgency: " + level + "  (rule "
                + r.paths.get(0).last().id + ")",
                "what to do: " + TeamKb.ACTIONS.get(level));
    }

    List<String> command(String[] a) throws IOException {
        switch (a[0]) {
            case "create": return team.create(a[1], DATA.resolve(a[2]).toString(),
                    a[3]);
            case "consult": return team.consult(a[1], a[2]);
            case "ask": return ask(a[1]);
            case "report": return team.report(a[1], a[2], a[3]);
            case "golden": return team.golden(a[1], a[2], a[3]);
            case "propose": return team.propose(a[1], a[2], a[3], a[4]);
            case "review": return team.review(a[1], a[2], a[3], a[4]);
            case "stop": return team.stop(a[1], a[2], a[3]);
            case "rollback": return team.rollback(a[1], a[2], a[3]);
            case "test": return team.test();
            case "log": return team.log();
            case "verify": return team.verify();
            case "diff": return team.diff(a[1], a[2]);
```

Listing 11.6 (Continued) *TriageTeam.java: the same command runner in Java.*

```
            case "blame": return team.blame(a[1]);
            case "tree": return team.tree(a[1]);
            default: throw new TeamKb.TeamError("unknown command " + a[0]);
        }
    }

    void run() throws IOException {
        while (lines.hasNext()) {
            String line = lines.next().strip();
            if (line.isEmpty() || line.startsWith("#")) {
                continue;
            }
            String[] a = line.split("\\|", -1);
            for (int i = 0; i < a.length; i++) {
                a[i] = a[i].strip();
            }
            if (a[0].equals("time")) {
                now = a[1];
                continue;
            }
            System.out.println("> " + line);
            List<String> out;
            try {
                out = command(a);
            } catch (TeamKb.TeamError | VcirsException e) {
                out = List.of("refused: " + e.getMessage());
            } catch (ArrayIndexOutOfBoundsException e) {
                out = List.of("refused: a part of the command is missing");
            }
            for (String text : out) {
                System.out.println(text);
            }
        }
    }

    public static void main(String[] args) throws IOException {
        List<String> list = List.of(args);
        String folder = DATA.resolve("clinic").toString();
        if (list.contains("--dir")) {
            folder = list.get(list.indexOf("--dir") + 1);
        }
        if (list.contains("--fresh") && Files.exists(Path.of(folder))) {
            try (Stream<Path> all = Files.walk(Path.of(folder))) {
                for (Path p : all.sorted(Comparator.reverseOrder()).toList()) {
                    Files.delete(p);              // start the week again
                }
```

Listing 11.6 (Continued) *TriageTeam.java: the same command runner in Java.*

```
            }
        }
        String source = list.isEmpty() ? "-" : list.get(list.size() - 1);
        TeamKb team = new TeamKb(folder, DATA.resolve("team.json").toString());
        if (source.equals("-")) {
            BufferedReader in = new BufferedReader(new InputStreamReader(
                    System.in, StandardCharsets.UTF_8));
            new TriageTeam(team, in.lines().iterator(), false).run();
        } else {
            new TriageTeam(team, Files.readAllLines(Path.of(source),
                    StandardCharsets.UTF_8).iterator(), true).run();
        }
    }
}
```

To run the book's week from the beginning:

```
cd Chapter11/java
javac -d out -sourcepath ../../vcirs-core/java/src *.java
java -cp out TriageTeam --fresh ../data/week.txt
```

or `python triage_team.py --fresh ../data/week.txt` in the `python` folder. `--fresh` deletes the folder `data/clinic` first. Both print the same 134 lines and write the same files: the seven version files, `state.json` and `audit.json` are identical, byte for byte, whichever language ran the week. A team can therefore use both: the nurse's old laptop can run Python and the doctor's computer Java, on the same shared folder, one at a time.

11.6 A Week at the Clinic

11.6.1 Monday: the first version, and a patient the knowledge misjudged

On Monday at 07:30 dr. Maya creates the knowledge base from the clinic's poster, written as eight cases in `clinic-seed.json`:

```
> create | dr. Maya | clinic-seed.json | First knowledge, from the clinic's triage poster.
v1 created: Front-desk triage: 11 variables, 8 nodes, 8 rules
```

Eleven variables — the ten questions of the seed file and `urgency`, the conclusion variable — eight nodes and eight rules. Five nodes are RED danger signs; the sixth says that a fever alone waits (GREEN), with two exceptions: a baby under three months is RED, and a fever of three days or more is YELLOW. Then she writes two golden cases, the two situations she most wants the team never to get wrong:

```
> golden | dr. Maya | fever = yes, age group = baby -> RED | A baby with fever is always
seen at once.
G1 added (passes now)
> golden | dr. Maya | fever = yes, age group = adult, fever days = 1 -> GREEN | A short
fever alone waits.
G2 added (passes now)
```

At 09:15 a woman arrives with four days of fever and, she mentions, bleeding gums. Sari consults the knowledge base with the facts she has:

```
> consult | Sari | fever = yes, age group = adult, fever days = 4, bleeding gums or nose =
yes
Sari asks about: fever = yes, age group = adult, fever days = 4, bleeding gums or nose =
yes
  urgency: YELLOW  (rule YELLOW#1, version v1)
  what to do: Tell the nurse now; the patient is seen before the queue.
  why:
    YELLOW#1 fired because:
      fever = yes  (given, CF 100)
      fever days >= 3  (fever days = 4, given, CF 100)
      so urgency = YELLOW  [rule CF 100, conclusion CF 100]
```

YELLOW: before the queue, but not at once. Ners Budi, told by Sari, sees her immediately; bleeding with a fever that has lasted several days is a warning sign, and he sends her to the hospital. Sari cannot change the knowledge, but she can do the most useful thing a front desk can do — she **reports** it:

```
> report | Sari | fever = yes, age group = adult, fever days = 4, bleeding gums or nose =
yes | Ners Budi saw her at once and sent her to the hospital.
R1 recorded for the experts to read
```

At 13:00 Ners Budi turns the report into a proposal. He gives the case, the answer it should have had (RED), the one fact that makes the difference (`bleeding gums or nose = yes`, chosen as the *when* condition), and his reason:

```
> propose | Ners Budi | fever = yes, age group = adult, fever days = 4, bleeding gums or
nose = yes -> RED | bleeding gums or nose = yes | Bleeding with a fever of 3 days or more
is a warning sign (report R1).
P1: would add RED#7: bleeding gums or nose = yes -> urgency = RED under YELLOW#1
  all 11 test cases still pass; waiting for another expert to review
```

The proposal was tried on a copy of version 1: the knowledge base answered YELLOW through `YELLOW#1`, so the new node `RED#7` would be an exception of `YELLOW#1`, and all eleven test cases — eight cornerstones, the two golden cases and the new node's own case — still pass. It waits for review. Budi, in a hurry to finish before his afternoon patients, tries to approve it himself:

```
> review | Ners Budi | P1 | approve | Looks right to me.
refused: nobody may review their own proposal
```

At 16:00 dr. Maya reviews it:

```
> review | dr. Maya | P1 | approve | Agreed; matches the health ministry's warning signs.
P1 approved: v2 adds RED#7 (11 test cases pass)
```

Version 2 exists. Here is the trail entry that recorded it, as `audit.json` stores it:


```
{
  "n": 6,
  "time": "2026-10-05 16:00",
  "who": "dr. Maya",
  "role": "doctor",
  "action": "approve",
  "detail": "P1: RED#7: bleeding gums or nose = yes -> urgency = RED",
  "reason": "Agreed; matches the health ministry's warning signs.",
  "version": 2,
  "sha": "644af82d7a449a66a003d9f996c553e43c6a21739e265ed6c762818dd2b68c6e",
  "prev": "64e64a51a9835ba79a04497d7bc9554d58b449c6f29444417e8ba0591f183c02",
  "hash": "5ec6bfda2c7068ba12364f88f96ff7ca7836eb58d5eb1da21e408fa5ba4fe6e5"
}
```


Everything a later reader needs is there: when, who, in what role, what was approved, the reason given, the resulting version, the fingerprint of `v002.json`, and the two links of the chain. On Tuesday Budi adds the patient of report R1 as a golden case, so that this situation is protected for good:

```
> golden | Ners Budi | fever = yes, age group = adult, fever days = 4, bleeding gums or
nose = yes -> RED | The case of report R1.
G3 added (passes now)
```

11.6.2 Wednesday: a proposal the tests refuse

On Wednesday Budi proposes that children with fever be seen sooner. Choosing the *when* condition quickly, he picks `fever = yes` — which is true of the case, but is not what makes it different:

```
> propose | Ners Budi | fever = yes, age group = child, fever days = 1 -> YELLOW | fever =
yes | Children with fever should be seen sooner.
P2: would add YELLOW#2: fever = yes -> urgency = YELLOW under GREEN#1
  REJECTED: 2 of 13 test cases would change:
    cornerstone of GREEN#1 (fever = yes, age group = adult): expected GREEN, got YELLOW
    golden G2 (fever = yes, age group = adult, fever days = 1): expected GREEN, got YELLOW
```

The tests explain the problem in the team's own terms. As an exception of `GREEN#1` with the condition `fever = yes`, the new node would catch *every* patient with fever, including the adult of `GREEN#1`'s cornerstone case and the adult of golden case G2, whose answer would change from GREEN to YELLOW. Nothing was saved. Ten minutes later he proposes again with the right condition:

```
> propose | Ners Budi | fever = yes, age group = child, fever days = 1 -> YELLOW | age
group = child | Children with fever should be seen sooner.
P3: would add YELLOW#2: age group = child -> urgency = YELLOW under GREEN#1
  all 13 test cases still pass; waiting for another expert to review
> review | Bidan Ratna | P3 | approve | Fine for a trial week.
P3 approved: v3 adds YELLOW#2 (13 test cases pass)
```

Bidan Ratna approves it "for a trial week", and version 3 exists. In the afternoon she proposes a lesson of her own: any bleeding in pregnancy is RED. She uses a variable nobody has used before, `bleeding in pregnancy`, and gives no *when* condition, so the new node is built from the whole case at the top level:

```
> propose | Bidan Ratna | pregnant = yes, bleeding in pregnancy = yes -> RED | | Any
bleeding in pregnancy is seen at once.
P4: would add RED#8: pregnant = yes, bleeding in pregnancy = yes -> urgency = RED under
(top level)
  all 14 test cases still pass; waiting for another expert to review
> review | dr. Maya | P4 | approve | Agreed.
P4 approved: v4 adds RED#8 (14 test cases pass)
```

11.6.3 Thursday: the front desk at work

On Thursday a pregnant woman who is bleeding comes to the desk, and Sari consults with the two facts she has:

```
> consult | Sari | pregnant = yes, bleeding in pregnancy = yes
Sari asks about: pregnant = yes, bleeding in pregnancy = yes
  urgency: RED  (rule RED#8, version v4)
  what to do: Call the nurse or doctor to the patient now. If no clinician is in the
clinic, call 119.
  why:
    RED#8 fired because:
      pregnant = yes  (given, CF 100)
      bleeding in pregnancy = yes  (given, CF 100)
      so urgency = RED  [rule CF 100, conclusion CF 100]
```

Most of the time, though, Sari does not know in advance which facts matter. She uses `ask`, and the knowledge base asks, in its own order — the danger signs first, because those nodes were taught first:

```
> ask | Sari
Is the patient awake and answering you normally? (yes/no) > yes
Is breathing hard, noisy or very fast, or can the patient not finish a sentence? (yes/no) >
no
Does the patient have chest pain or pressure? (yes/no) > no
Is one side of the face drooping, one arm weak, or speech slurred? (yes/no) > no
Is there bleeding that does not stop when pressed? (yes/no) > no
Does the patient have a fever? (yes/no) > yes
Is the patient a baby under 3 months, a child or an adult? (baby/child/adult) > child
For how many days has the fever lasted? (1/2/3/4/5) > 2
urgency: YELLOW  (rule YELLOW#2)
what to do: Tell the nurse now; the patient is seen before the queue.
```

Eight questions for a child with two days of fever, each one needed by a node on the way: the five danger signs (each could have made the answer RED at once), the fever, the age group (a baby

would be RED), and the days (three or more would be YELLOW). The answer is YELLOW, from the node approved on Wednesday.

11.6.4 Friday: a mistake, a rollback and a lesson

At the Friday meeting the team agrees that the trial of `YELLOW#2` did not work: the YELLOW queue filled up with children who were fine. A child with fever and no danger sign should not be sent ahead of everyone. dr. Maya stops the node — or rather, she means to:

```
> stop | dr. Maya | YELLOW#1 | Friday meeting: a child with fever and no danger sign waits
in the normal queue.
v5: YELLOW#1 stopped (13 test cases pass)
```

She typed `YELLOW#1`, the node for a fever of three days or more, instead of `YELLOW#2`. All thirteen test cases still pass, because stopping a node skips its own cornerstone and none of the golden cases covers a long fever without bleeding. A few minutes later Sari registers exactly such a patient:

```
> consult | Sari | fever = yes, age group = adult, fever days = 4, bleeding gums or nose =
no
Sari asks about: fever = yes, age group = adult, fever days = 4, bleeding gums or nose = no
  urgency: none  (No rule matched: ask the nurse to look at the patient now.)
  the last matching rule, YELLOW#1, is stopped
```

Nothing dangerous happens — the answer "none" sends the nurse to look at the patient, as **Table 11.1** says — but the answer is clearly not what the team wants, and the message names the stopped node. dr. Maya rolls back to Thursday's knowledge, stops the right node, and writes a golden case so that the same slip will be caught next time:

```
> rollback | dr. Maya | v4 | I stopped the wrong node; back to Thursday's knowledge.
v6 is a copy of v4; nothing in between is lost from the trail
> stop | dr. Maya | YELLOW#2 | Friday meeting: a child with fever and no danger sign waits
in the normal queue.
v7: YELLOW#2 stopped (13 test cases pass)
> golden | dr. Maya | fever = yes, age group = adult, fever days = 4, bleeding gums or nose
= no -> YELLOW | So that stopping YELLOW#1 by mistake is caught next time.
G4 added (passes now)
```

Figure 11.3 shows the week as a line of versions.

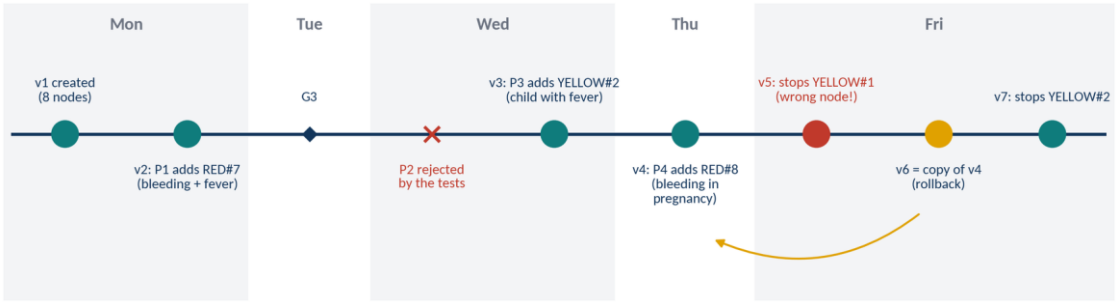

Figure 11.3 The seven versions of the week. P2 was refused by the tests and never became a version; version 5 stopped the wrong node, and version 6 is a copy of version 4.

11.6.5 What the files remember

Every proposal stays in `state.json` whatever became of it. Here is Wednesday's refused one:

```
  {
    "id": "P2",
    "by": "Ners Budi",
    "case": "fever = yes, age group = child, fever days = 1 -> YELLOW",
    "when": "fever = yes",
    "reason": "Children with fever should be seen sooner.",
    "status": "rejected by tests",
    "base": 2,
    "node": "YELLOW#2",
    "reviewer": ""
  }
```

The record keeps the case exactly as Budi wrote it, the condition he chose, his reason, the version it was tried on (`base`), the node it would have created, and its fate. A proposal refused by the tests is not a failure to hide; it is evidence that the tests work, and a reminder of an idea that someone found worth proposing. When the team discusses children with fever again next month, this record, and the four entries of the trail around it, are where the discussion starts.

The version files are ordinary knowledge-base files, the same format as `wifi-kb.json` in Chapter 3. Any of them can be opened with vcirs-core's command-line tool (`python -m vcirs show versions/v004.json` in the engine's `python` folder, or `java -cp out vcirs.Cli show ...`), or copied to another program — the stain advisor of Chapter 10 could load one into its database. The team knowledge base adds rules around the knowledge, never a new format for it.

11.7 Looking Back

At the end of the week the team looks back with the five commands that read the record. First, the tests on the current version:

```
> test
v7: all 14 test cases pass
```

Fourteen cases: eight cornerstones of the seed, `RED#7` and `RED#8` from the week's approved proposals, and four golden cases; `YELLOW#2`, stopped, has no test. Next, what changed between Thursday and now — the rollback and the second stop, seen from the outside, are one clean change:

```
> diff | v4 | v7
from v4 to v7:
  stopped YELLOW#2
```

Then the question every knowledge base eventually hears: *why is this rule here?*

```
> blame | RED#7
RED#7: bleeding gums or nose = yes -> urgency = RED
  under: YELLOW#1
  taught by: Ners Budi
  cornerstone case: fever = yes, age group = adult, fever days = 4, bleeding gums or nose =
yes
  proposal P1: Bleeding with a fever of 3 days or more is a warning sign (report R1).
  approved by dr. Maya
  in the trail: #6 2026-10-05 16:00 approve by dr. Maya -> v2
```

In one screen: the node, its place (an exception of the three-day fever node), who taught it, the case that justified it, the proposal and its reason, the reviewer, and the entry of the trail with the time and the version. The rollback entry at 15:10 (#14) created a version that also contains `RED#7`, but its detail does not mention the node, so `blame` shows only the entry that introduced it. The knowledge as it stands:

```
> tree | v7
(root)
+- RED#1: awake = no -> urgency = RED
+- RED#2: hard breathing = yes -> urgency = RED
+- RED#3: chest pain = yes -> urgency = RED
+- RED#4: face or arm weak = yes -> urgency = RED
+- RED#5: heavy bleeding = yes -> urgency = RED
+- GREEN#1: fever = yes -> urgency = GREEN
|  +- RED#6: age group = baby -> urgency = RED
|  +- YELLOW#1: fever days >= 3 -> urgency = YELLOW
|  |  +- RED#7: bleeding gums or nose = yes -> urgency = RED
|  +- YELLOW#2: age group = child -> urgency = YELLOW [stopped]
+- RED#8: pregnant = yes, bleeding in pregnancy = yes -> urgency = RED
```

The whole trail reads like the diary of the week:

```
> log
#1 2026-10-05 07:30 v1 dr. Maya (doctor) create: Front-desk triage: 11 variables, 8 nodes,
8 rules
    why: First knowledge, from the clinic's triage poster.
#2 2026-10-05 07:40 v1 dr. Maya (doctor) golden: G1: fever = yes, age group = baby -> RED
    why: A baby with fever is always seen at once.
#3 2026-10-05 07:40 v1 dr. Maya (doctor) golden: G2: fever = yes, age group = adult, fever
days = 1 -> GREEN
    why: A short fever alone waits.
#4 2026-10-05 09:20 v1 Sari (front desk) report: R1: fever = yes, age group = adult, fever
days = 4, bleeding gums or nose = yes (answer YELLOW)
    why: Ners Budi saw her at once and sent her to the hospital.
#5 2026-10-05 13:00 v1 Ners Budi (nurse) propose: P1 (waiting for review): fever = yes, age
group = adult, fever days = 4, bleeding gums or nose = yes -> RED
    why: Bleeding with a fever of 3 days or more is a warning sign (report R1).
#6 2026-10-05 16:00 v2 dr. Maya (doctor) approve: P1: RED#7: bleeding gums or nose = yes ->
urgency = RED
    why: Agreed; matches the health ministry's warning signs.
#7 2026-10-06 10:00 v2 Ners Budi (nurse) golden: G3: fever = yes, age group = adult, fever
days = 4, bleeding gums or nose = yes -> RED
    why: The case of report R1.
#8 2026-10-07 08:30 v2 Ners Budi (nurse) propose: P2 (rejected by tests): fever = yes, age
group = child, fever days = 1 -> YELLOW
    why: Children with fever should be seen sooner.
#9 2026-10-07 08:40 v2 Ners Budi (nurse) propose: P3 (waiting for review): fever = yes, age
group = child, fever days = 1 -> YELLOW
    why: Children with fever should be seen sooner.
#10 2026-10-07 11:00 v3 Bidan Ratna (midwife) approve: P3: YELLOW#2: age group = child ->
urgency = YELLOW
    why: Fine for a trial week.
#11 2026-10-07 14:00 v3 Bidan Ratna (midwife) propose: P4 (waiting for review): pregnant =
yes, bleeding in pregnancy = yes -> RED
    why: Any bleeding in pregnancy is seen at once.
#12 2026-10-08 09:00 v4 dr. Maya (doctor) approve: P4: RED#8: pregnant = yes, bleeding in
pregnancy = yes -> urgency = RED
    why: Agreed.
#13 2026-10-09 15:00 v5 dr. Maya (doctor) stop: YELLOW#1
    why: Friday meeting: a child with fever and no danger sign waits in the normal queue.
#14 2026-10-09 15:10 v6 dr. Maya (doctor) rollback: back to v4
    why: I stopped the wrong node; back to Thursday's knowledge.
#15 2026-10-09 15:12 v7 dr. Maya (doctor) stop: YELLOW#2
    why: Friday meeting: a child with fever and no danger sign waits in the normal queue.
#16 2026-10-09 15:15 v7 dr. Maya (doctor) golden: G4: fever = yes, age group = adult, fever
days = 4, bleeding gums or nose = no -> YELLOW
    why: So that stopping YELLOW#1 by mistake is caught next time.
```

Sixteen entries: consultations and questions are not in the trail, because they change nothing; every other command is, including the proposal the tests refused and the stop that was a mistake. Finally, the chain is checked:

```
> verify
the trail is intact: 16 entries, 7 versions, last hash ab12b6ba663b
```

To see what `verify` protects against, copy the folder, open the copy's `audit.json` in a text editor, and change one character of an old reason — for example the semicolon of dr. Maya's reason in entry 6 into a comma. Then run `verify` on the copy:

```
> verify
the trail is broken at entry #6
```

The change is found at the entry where it was made, and every entry after it is now in doubt until the original file is restored from a backup. Refusals are just as visible. Here are four commands the team should not be able to do:

```
> review | Sari | P1 | approve | ok
refused: Sari (front desk) may not do this
> stop | Ners Budi | RED#7 | test
refused: Ners Budi (nurse) may not do this
> rollback | dr. Maya | v9 | x
refused: roll back to an earlier version, v1 to v6
> consult | Pak Joni | fever = yes
refused: Pak Joni is not a member of the team
```

11.7.1 Tests

`TeamKbTest.java` and `test_team_kb.py` replay the week in a temporary folder and check that it ends at version 7 with all fourteen cases passing and an intact trail; that a nurse cannot stop a node and the front desk cannot propose; that a new `TeamKb` on the same folder loads the same version, with `YELLOW#2` stopped; and that one space added to `v002.json` is found by `verify`. The Java tests print:

```
ok: the week ends at v7
ok: all test cases pass
ok: the trail is intact
ok: only a doctor may stop a node
ok: the front desk may not propose
ok: the team knowledge base loads again
ok: tampering is found
all tests passed
```

11.8 Making It Better

The stopped node's answer. After Friday, a child with fever and no danger sign gets the answer *none*: the path of the engine ends at `YELLOW#2`, which is stopped, and a stopped node gives no conclusion. For a front desk that is safe — the nurse looks — but it is not the GREEN the meeting wanted. Stopping is the RDR way of removing advice without deleting knowledge [4], [8]; it is not the way to replace advice. Exercise 11.6 asks how the team can get GREEN for these children, and its solution combines the tools of this chapter.

Proposals from the page. The team types commands today. A page like Chapter 10's, with the knowledge base behind a small server, would show the pending proposals with their test results and two buttons, *approve* and *reject*, and a box for the reason. The logic is all in `TeamKb`; the page adds nothing but convenience. With a server, the "one change at a time" rule of Section 11.3 is also easy to enforce: a lock around every method that writes, as in Chapter 10.

Signatures, not only names. The trail records the name a person typed. In a clinic where the computers are shared, anyone can type "dr. Maya". The next step is a login for each member, and further still, a **digital signature** of each entry with a key that only the person who wrote it holds; then the trail proves not only that nothing was changed, but also who wrote each entry.

Reports that become golden cases. Every report is a case the front desk found doubtful. When the experts agree on its right answer, it should become a golden case, as Budi did on Tuesday. A command that turns report R1 into golden case G3 in one step, with the reason taken from the report, makes the good habit the easy one.

Two proposals at once. If two proposals wait at the same time, the second to be approved is taught to a version that already contains the first. `review` handles this correctly — it teaches the case again to the current version and tests again — but the reviewer should see it. Printing "the knowledge has changed since this proposal was made (v2 → v3)" in the review is a two-line change.

Counting the front desk's day. The trail records reports but not consultations, on purpose: consultations are many and contain nothing to audit. If the clinic wants to know how many RED, YELLOW and GREEN answers the desk gives each day, a separate counter per level and day is enough, without any fact of any patient.

> **PITFALL.** Tests only protect what they contain. On Friday, stopping the wrong node passed every test because no golden case covered a long fever without bleeding. After every surprise, add the case that would have caught it as a golden case; the test suite of a team knowledge base should grow with every incident, exactly as the knowledge does.

11.9 Using It for Real

Start from the protocol you already have. Most clinics have a written triage protocol, a poster or a standard operating procedure. Write it as seed cases, one case per line of the protocol, with the danger signs first; the order of the top-level nodes is the order of the questions at the desk. Have the doctor responsible for the clinic create the first version, and write the reason as the name and date of the protocol it came from.

Agree on the golden cases in a meeting. Ten to twenty golden cases, chosen together, are worth more than a hundred written by one person. Include the cases every member would be ashamed to get wrong — the baby with fever, the chest pain, the bleeding in pregnancy — and the ordinary cases that must stay GREEN, so that the queue is not flooded with YELLOW.

Keep roles honest. Give the front desk the right to report, and use it: a report costs nothing and is often the first sign that the knowledge is wrong. Keep the right to stop and roll back with one or two people who are responsible for the protocol. And keep the rule that nobody reviews their own proposal, even when it is inconvenient; it is the rule that catches the tired Friday mistake.

A weekly look at the trail. Once a week, read the entries of the week together: the approved proposals, the rejected ones (a rejected proposal often means a disagreement worth discussing), the reports that have not become golden cases yet, and the result of `verify`. It takes ten minutes and keeps the whole team aware of what the desk is being told to do.

Backups and the folder. The whole team knowledge base is one folder of small text files. Copy it every day to a second place — another computer, a flash disk, cloud storage — and keep old copies: if `verify` ever finds a broken chain, an old copy is what restores the truth. Because the

files are text, a shared folder that keeps file versions (as most cloud storage services do) is a natural home, as long as only one person runs a command at a time.

Teaching the team to use it. Spend the first week with the knowledge base beside the old poster, not instead of it. Sari consults for every patient and compares the answer with the poster; every difference is either a report or a mistake in the seed cases, and both are worth finding before anyone relies on the program. Show each expert the `blame` of one node they did not teach: seeing the case and the reason behind someone else's rule is what makes a team trust a shared knowledge base.

When the computer is off. A power cut or a broken laptop must not leave the desk without its rules. Print the tree of the current version (`tree | v7`) after every change, date it, and pin it where the old poster was. The tree is short — a dozen lines for this clinic — and its order is the order of the questions, so the printed page is a faithful poster of the current knowledge, and the version number on it tells the team whether it is up to date.

What the knowledge base must never become. It must not grow into a diagnosis tool: no node should conclude an illness or a treatment, only an urgency. It must not replace the clinician's look: every RED and YELLOW sends a person to the patient, and so does every *none*. And it must not store patients' identities: the cases are signs, and reports carry the signs and a note, never a name.

Other teams with the same shape. The same folder of versions, tests, reviews and a chained trail suits any knowledge base that several people maintain and that matters: a cooperative's loan committee with the fuzzy knowledge of Chapter 9, a motorcycle workshop with several mechanics (Chapter 8), the laundry of Chapter 10 once it has three branches, a school whose counselors share a knowledge base about which students to call in first. Change the levels and the seed file; the rest stays.

11.10 Common Mistakes, and How to Avoid Them

Sharing a file instead of a knowledge base. Two people saving the same file lose a lesson without a trace, as Section 11.3 showed. *Avoid it:* every change goes through one program that starts from the current version and writes the next one.

Choosing a condition that is true, but not the reason. `fever = yes` was true of Budi's child, but it did not tell the child apart from anyone else with fever. *Avoid it:* choose from the list of difference the fact that explains the different answer; let the tests show what else would change.

Reviewing your own change. The program refuses, but the same mistake can be made by two people who approve each other's proposals without reading them. *Avoid it:* the reviewer reads the case, the reason and the test result, and writes her own reason.

Believing the tests catch everything. They catch only changes to the cases they contain. *Avoid it:* add a golden case after every surprise.

Stopping when you mean replacing. A stopped node gives no answer; it does not give the answer of its parent. *Avoid it:* stop a node when the right answer is "ask someone"; when a different answer is wanted, teach it (or roll back and teach again).

Rolling back too far. A rollback returns the whole knowledge base to an older version, including the loss of every good change made after it. *Avoid it:* use `diff` before a rollback to see what would be lost; to undo one change among later ones, stop its node instead.

Editing files by hand. A hand edit of a version file or of the trail breaks the chain, which is the point. *Avoid it:* change the knowledge only through commands; if a file really is damaged, restore it from a backup and run `verify`.

Putting patients in the trail. A reason such as "Ibu Siti from RT 05" puts personal health data in a file that the whole team reads for years. *Avoid it:* write signs and circumstances, never identities.

Letting the knowledge name illnesses. A node that concludes a disease invites the desk to diagnose. *Avoid it:* conclusions are only RED, YELLOW and GREEN; everything else is for the clinician.

11.11 Chapter Summary

- A knowledge base that several experts teach needs more than Ripple Down Rules: without rules for sharing, lessons are lost when two copies are saved (**Listing 11.1**, **Listing 11.2**), changes go unreviewed, and there is no way back.
- The clinic's knowledge base sorts patients by **urgency** — RED, YELLOW, GREEN — with "ask the nurse now" when nothing matches; it asks about visible signs and never names an illness (**Table 11.1**).
- Every change is a **proposal** tried on a copy of the current version and **tested** against every **cornerstone case** and every **golden case**; a proposal that would change any of them is refused. A second expert **reviews** every proposal that passes; only an approved change becomes a new **version**, which is never edited (**Figure 11.1**, **Table 11.2**).
- A **rollback** copies an old version into a new one, and **stopping** removes a node's advice while keeping its knowledge; neither destroys anything (**Figure 11.3**).
- The **audit trail** records who did what, when, why and with which result; each entry carries the SHA-256 fingerprints of its version file and of the entry before it, so `verify` finds any later change (**Figure 11.2**).
- `TeamKb` (**Listing 11.3**, **Listing 11.4**) and the command runner (**Listing 11.5**, **Listing 11.6**) replay the clinic's week identically in Java and Python: a refused self-review, a proposal refused by the tests, three approved changes, a mistaken stop, a rollback, and a trail that `blame` and `verify` can read.

11.12 Exercises

11.1 Using the tree printed in Section 11.7, find the urgency and the questions `ask` would ask, in order, for: (a) an adult who is awake, with chest pain; (b) a baby with fever; (c) an adult with five days of fever and no bleeding; (d) a child with fever for one day.

11.2 Why must `review` teach the proposal again to the current version instead of saving the copy that `propose` tested? Describe a sequence of two proposals where saving the old copy would lose a change.

11.3 In `run_tests`, stopped nodes are skipped. Explain why their cornerstone cases cannot be expected to keep their answers. Is there a check worth doing for a stopped node's cornerstone case?

11.4 Ners Budi adds a golden case the team has not agreed on: a child with one day of fever must be YELLOW. What does the program say, and what does `test` report afterwards? What should the team do?

11.5 Make a copy of the clinic folder, change one character in `versions/v003.json`, and run `verify`. Which message do you expect, and why does the check of the version files matter even though the trail itself is intact?

11.6 After Friday, a child with fever gets *none*. The team wants GREEN for such children. (a) Why can it not be taught as an exception of `YELLOW#2` with the facts `fever = yes, age group = child, fever days = 1`? (b) Describe how to reach the wanted knowledge with the commands of this chapter, and which entries the trail would show.

11.7 Write the command a midwife would use to propose that a pregnant woman with fever is YELLOW, as an exception of `GREEN#1`. Which *when* condition should she choose, and what will the tests say?

11.8 Add a command `pending` that lists the proposals waiting for review, with the person who proposed each, its case and its reason. Which method of `TeamKb` do you add, and which lines of the runner change?

11.9 The trail makes changes *visible*, not *impossible*. Name two ways in which a dishonest member could still misuse the system, and what would limit each one.

11.13 Solutions to the Exercises

Solution 11.1. (a) *awake* → yes, *hard breathing* → no, *chest pain* → yes: RED from `RED#3`; three questions. (b) The five danger signs, *fever* → yes, *age group* → baby: RED from `RED#6`. (c) The five danger signs, *fever* → yes, *age group* → adult, *fever days* → 5; `YELLOW#1` fires, and its exception `RED#7` asks *bleeding gums or nose* → no: YELLOW. (d) The five danger signs, *fever*, *age group* → child, *fever days* → 1: `RED#6` and `YELLOW#1` fail and `YELLOW#2` fires but is stopped, so the answer is *none*, "ask the nurse" (see Exercise 11.6).

Solution 11.2. Suppose P1 and P2 both wait on version 1. P1 is approved and becomes version 2. If P2's approval saved the copy that `propose` made from version 1, version 3 would contain P2's node but not P1's: P1's change would be lost, exactly as in Section 11.3. Teaching P2 again to version 2 produces a version 3 with both changes, and testing again checks that P2 still passes with P1 present — the two may interact, for example if P1 added an exception under the node P2 corrects.

Solution 11.3. A stopped node fires but gives no conclusion, so its own cornerstone case now gets either *none* or the answer of one of its exceptions; expecting the old conclusion would make every stop fail its own test. A useful check is the opposite one: the stopped node's cornerstone should *not* get the stopped conclusion from somewhere else, and ideally the team decides what it should get and writes that as a golden case — which is what dr. Maya did with G4.

Solution 11.4. On the Friday folder the program prints:

```
> golden | Ners Budi | fever = yes, age group = child, fever days = 1 -> YELLOW | I still
think children should be seen sooner.
G5 added (FAILS now: gives none)
> test
v7: 1 of 15 test cases fail:
  golden G5 (fever = yes, age group = child, fever days = 1): expected YELLOW, got none
```

A golden case records what the team *expects*; the program warns that the current knowledge disagrees, and from then on every `test` fails until someone changes either the knowledge or the golden case. That is useful: the disagreement is now visible, named and dated. But a golden case

should be agreed by the team; the team should discuss it at the next meeting, and either change the knowledge (with a proposal and a review) or remove the case. (Removing a golden case is not a command in this chapter; adding one, `retire | who | G5 | why`, which keeps the case but marks it retired in the trail, is a good extension.)

Solution 11.5. `verify` prints `version v3 was changed after entry #10`, the entry that created `v003.json` (Budi's P3 approved by Ratna). The trail itself is intact, because nobody touched `audit.json`; but the knowledge the desk would load if someone rolled back to version 3 is no longer the knowledge that was approved. Checking the version files against the fingerprints in the trail is what makes the versions trustworthy, not only the story about them.

Solution 11.6. (a) The new case would have exactly the facts of `YELLOW#2`'s cornerstone case, so the list of difference is empty: there is no fact that tells the new case apart from the old one, and the engine refuses (Chapter 3). The two cases say different things about the same situation — which is precisely a change of mind, not a new exception. (b) The clean way is to undo the change of mind and redo what came after it: `rollback | dr. Maya | v2 | ...` returns to the knowledge before P3 (child YELLOW), losing P4; then Bidan Ratna proposes P4 again, someone reviews it, and it becomes the next version. Children with fever then get GREEN from `GREEN#1`, as in version 2. The trail shows everything: the rollback to v2 with its reason, the new proposal P5 with the same case as P4, its review and the new version. Before rolling back, `diff | v2 | v7` shows exactly what will need to be taught again.

Solution 11.7. `propose | Bidan Ratna | fever = yes, age group = adult, pregnant = yes -> YELLOW | pregnant = yes | ...`. The knowledge base answers GREEN through `GREEN#1`, the list of difference against `GREEN#1`'s cornerstone (`fever = yes, age group = adult`) contains only `pregnant = yes`, and that is the right *when* condition. The tests pass: no cornerstone or golden case contains `pregnant = yes` together with fever, so none of their answers changes; the proposal waits for review. (If she had written a baby, the case would already be RED through `RED#6`, and the program would answer "already known".)

Solution 11.8. In `team_kb.py`:

```
def pending(self):
    out = [p["id"] + " by " + p["by"] + ": " + p["case"] + "  (" + p["reason"]
           + ")" for p in self.state["proposals"]
           if p["status"] == "waiting for review"]
    return out or ["no proposal is waiting"]
```

and in the runner's table of commands, `"pending": t.pending,`. The Java version is the same loop over `list("proposals")` in a method `List<String> pending()`, and one more `case "pending": return team.pending();` in `command`. No entry is logged, because nothing changes.

Solution 11.9. First, a member could type another member's name; the trail would faithfully record a lie. A login per member, and ideally digital signatures, limit this (Section 11.8). Second, a member with access to the folder could replace the whole folder, trail included, with a fabricated one whose chain is internally consistent. Daily backups kept outside the folder limit this: the fabricated chain would not match the fingerprints of yesterday's copy. A third, subtler one: two members who always approve each other's proposals without reading them defeat the four-eyes rule; the weekly reading of the trail, with the reasons, is what makes that visible.

References

[1] World Health Organization, International Committee of the Red Cross, and Médecins Sans Frontières, "Interagency Integrated Triage Tool," 2026. [Online]. Available: https://www.who.int/tools/triage

[2] Kementerian Kesehatan Republik Indonesia, "Akses darurat medis 119 kini bisa melalui SATUSEHAT Mobile." [Online]. Available: https://kemkes.go.id/id/akses-darurat-medis-119-kini-bisa-melalui-satusehat-mobile

[3] Centers for Disease Control and Prevention, "Symptoms of dengue and testing," Aug. 2025. [Online]. Available: https://www.cdc.gov/dengue/signs-symptoms/index.html

[4] P. Compton and B. H. Kang, *Ripple-Down Rules: The Alternative to Machine Learning*. Boca Raton, FL, USA: CRC Press, 2021.

[5] D. Richards, "Two decades of ripple down rules research," *The Knowledge Engineering Review*, vol. 24, no. 2, pp. 159–184, 2009.

[6] M. M. I. Subakti, "A Variable-Centered Intelligent Rule System," M.Sc.Eng. thesis, Dept. of Computer Science and Information Engineering, Taipei, Taiwan, Jun. 2005.

[7] I. Subakti, "Some revisions in VCIRS and cases reconstructing perspectives," in *Proc. Information and Communication Technology Seminar (ICTS 2006)*, Surabaya, Indonesia, 2006.

[8] P. Compton, G. Edwards, B. Kang, L. Lazarus, R. Malor, T. Menzies, P. Preston, A. Srinivasan, and C. Sammut, "Ripple down rules: possibilities and limitations," in *Proc. 6th AAAI Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada, 1991, pp. 6.1–6.20.

[9] National Institute of Standards and Technology, *Secure Hash Standard (SHS)*, FIPS PUB 180-4. Gaithersburg, MD, USA, Aug. 2015.

[10] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. Jakarta, Indonesia, 2022.

Chapter 12

Learning Rules from Data: From a Decision Tree to a VCIRS Knowledge Base

By the end of this chapter you will be able to explain how a program can learn rules from a table of examples; grow a **decision tree** with the CART method, written by us from scratch in Java and in Python, choosing each question by the **Gini index** computed exactly with whole numbers; judge a learned tree honestly with **leave-one-out** testing; turn every path of the tree into a node of a VCIRS knowledge base, with a certainty factor and a real cornerstone case; and let an expert refine the learned knowledge case by case, each lesson checked against every example of the data before it is kept. Our data are 101 real animals from a public data set; our expert is a biology teacher whose nature club meets animals the data never saw. At the end we check our tree against scikit-learn, the most widely used machine-learning library in Python.

12.1 Bu Lestari's Nature Club

Every Saturday morning Bu Lestari, who teaches biology at a junior high school, takes her nature club somewhere with animals: the city zoo, the mangroves at the edge of the town, the fruit and durian orchard of a pupil's family, where there are bats in the evening and lizards on every wall. The children photograph what they see and, back at school, try to say what it is: a mammal, a bird, a reptile, an amphibian, a fish, an insect, or one of the many animals without a backbone. It is the oldest exercise in biology — sorting living things by what they have and what they do — and children love it, as long as someone can answer their questions: *Does a bat count as a bird? Is a turtle a fish, since it swims with flippers? Why is a whale not a fish?*

Bu Lestari would like a small program that plays this game with them: it asks about the animal — does it have feathers, does it feed its young with milk, does it have a backbone — and names the class, and explains why, as the Wi-Fi doctor of Chapter 3 explained its advice. She could write the rules herself, as Winston's *Zookeeper* rules were written by hand, and as the thesis on VCIRS used them for its demonstration [1]. But she has something that makes her work easier: a table. The Zoo data set, published in the UCI Machine Learning Repository, describes 101 animals with 16 facts each — hair, feathers, eggs, milk, whether it flies, lives in water, hunts, has teeth, a backbone, breathes air, is venomous, has fins, how many legs, a tail, is kept by people, is bigger than a cat — and the class of each [2]. Could the rules be *learned* from the table, and then corrected by her where the table is silent?

This is the question of this chapter, and it is a practical one far beyond animals. Shops, workshops, laundries and clinics all have records: the stain and the treatment that worked, the symptom and the repair, the complaint and the department that handled it. Hidden in those records are rules. A decision tree finds them in seconds; VCIRS turns them into knowledge an expert can read, check and correct; and the correction happens case by case, exactly as in the rest of this book.

12.2 The Idea in Plain Words

12.2.1 Knowledge from experts, knowledge from data

There are two ways to build a rule-based system. One is to ask an expert, as every chapter so far has done. The other is **machine learning**: give a program many examples whose answers are known, and let it find rules that reproduce them. Each way has a typical character, and the thesis that defined VCIRS describes it precisely. Knowledge built by Ripple Down Rules tends to be *precise*, because every rule was made for a real case and checked against the cases before it, but it *covers* only what the expert has seen; a machine-learning method trained on many examples covers more situations but makes more mistakes on the ones it has not seen [1]. The book that sums up forty years of Ripple Down Rules is subtitled, not by accident, *The Alternative to Machine Learning* [3].

But the two are not enemies. The same thesis recalls that RDR knowledge bases were used to produce data for machine learning, and that rule induction was used to produce RDR trees [1], [4]. This chapter does the second: it learns first, from the data, to get broad coverage cheaply, and then refines with the expert, to get precision where it matters. The expert does not have to write the easy rules — the table already contains them — and the learned rules do not have to be trusted blindly, because each one becomes a VCIRS node with a cornerstone case that the expert can inspect, and each correction is an RDR exception that cannot silently break what the data taught. **Figure 12.1** shows the whole road.

Figure 12.1 From data to a checked knowledge base. A tree is learned from the table, each leaf becomes a VCIRS node, the teacher adds exceptions, and every lesson is checked against all the animals the knowledge base already answers.

12.2.2 A decision tree is a game of questions

A **decision tree** is the game of twenty questions written down. At the top is one question, such as *does it feed its young with milk?* Each answer leads to another question or to a conclusion. To classify an animal you start at the top and follow the answers until you reach a conclusion, which is called a **leaf**. **Figure 12.2** shows the tree that our program learns from the 101 animals.

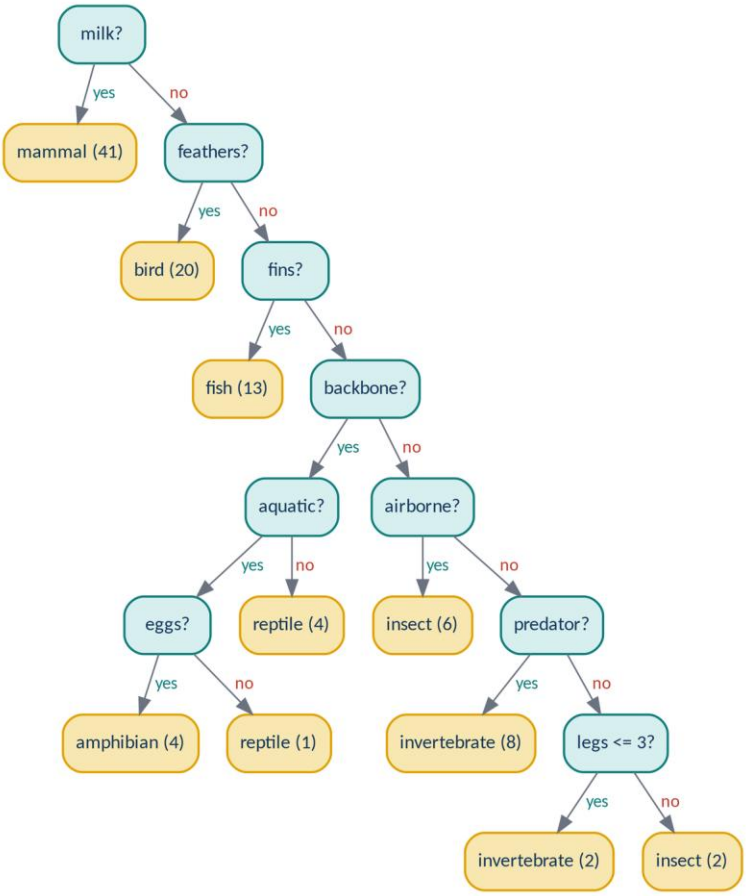

Figure 12.2 The decision tree learned from the 101 animals of the Zoo data. Each question has a yes branch and a no branch; each leaf gives a class and, in parentheses, how many animals of the data reached it.

Read it from the top. Milk: yes means mammal, all 41 of them. Otherwise feathers: yes means bird. Otherwise fins: yes means fish. Otherwise a backbone decides between the cold-blooded vertebrates (amphibians and reptiles, told apart by water and eggs) and the animals without a backbone (insects, which fly or have many legs, and the rest). Ten leaves, seven questions at most. A biologist would recognize almost every question; the program found them without knowing any biology, only by looking at the table.

Notice two things about such a tree. First, every path from the top to a leaf is a rule: *IF milk = no AND feathers = yes THEN bird*. A tree is a compact way of writing a set of rules that never overlap, because every animal follows exactly one path. Second, a tree is exactly the kind of structure that Chapter 3 found fragile: to change one rule you would have to reshape the tree. That is why we will not keep the tree. We will turn its paths into VCIRS nodes and let the expert add exceptions, which never reshape anything.

12.2.3 How the tree chooses its questions: purity

How did the program know to ask about milk first? It tried every question it could ask and kept the one that splits the animals into the **purest** groups: groups in which most animals have the same class. The method we use is **CART** (Classification And Regression Trees), described by Breiman, Friedman, Olshen and Stone [5], and its measure of purity is the **Gini index**. For a group in which a share p_k of the members belong to class k,

$$G = 1 - \sum_k p_k^2 \tag{12.1}$$

A pure group, all of one class, has $G = 0$. A group evenly mixed between two classes has $G = 1 - (0.5^2 + 0.5^2) = 0.5$; with seven classes it can approach $1 - 7 \cdot (1/7)^2 = 0.857$. You can read G as a probability: pick two members of the group at random, with replacement; G is the chance that they belong to different classes.

A question divides a group of n animals into a *yes* group of n_{yes} animals and a *no* group of n_{no}. The impurity of the question is the Gini index of the two groups, each weighted by its size:

$$G_{\text{split}} = \frac{n_{\text{yes}}}{n} G_{\text{yes}} + \frac{n_{\text{no}}}{n} G_{\text{no}} \tag{12.2}$$

and CART keeps the question with the smallest G_{split}, provided it is smaller than the impurity of the group before the split. Let us do it by hand for the first question. The 101 animals are 41 mammals, 20 birds, 13 fish, 10 invertebrates, 8 insects, 5 reptiles and 4 amphibians, so by Equation (12.1)

$$G = 1 - \frac{41^2 + 20^2 + 13^2 + 10^2 + 8^2 + 5^2 + 4^2}{101^2} = 1 - \frac{2455}{10201} = 0.7593$$

The question *milk = yes* sends the 41 mammals to the yes group, which is pure ($G_{\text{yes}} = 0$), and the other 60 animals to the no group, whose counts are 20, 13, 10, 8, 5 and 4:

$$G_{\text{no}} = 1 - \frac{400 + 169 + 100 + 64 + 25 + 16}{3600} = 1 - \frac{774}{3600} = 0.785$$

so by Equation (12.2), $G_{\text{split}} = \frac{41}{101} \cdot 0 + \frac{60}{101} \cdot 0.785 = 0.4663$, far below 0.7593. **Table 12.1** compares it with the next best questions.

Table 12.1 The five best first questions for the 101 animals, by the weighted Gini impurity of the two groups they make. Milk wins.

Question	Yes group	G_{yes}	No group	G_{no}	G_{split}
milk = yes	41	0.0000	60	0.7850	0.4663
eggs = yes	59	0.7854	42	0.0918	0.4970
hair = yes	43	0.1687	58	0.7830	0.5215
feathers = yes	20	0.0000	81	0.6868	0.5508
toothed = yes	61	0.5160	40	0.6462	0.5676

Feathers makes a perfectly pure yes group too, but a small one (20 birds), leaving a large mixed group; milk makes a pure group twice as big. Eggs is close behind milk, because almost everything that lays eggs is not a mammal — but the egg-laying group is badly mixed. After milk, CART repeats the whole search inside the no group, and so on down, until each group is pure or no question makes it purer.

ID3 and its successor C4.5, the decision-tree methods of Quinlan, do the same with another measure of purity, **entropy**, and choose the question with the largest *information gain* [6], [7]; they also allow questions with more than two answers. On data like ours the two measures usually choose the same questions. Exercise 12.3 asks you to try.

> **DEFINITION.** A *decision tree* classifies a case by asking one question at each node and following the branch of the answer until a leaf gives the class. *CART* grows a binary tree top-down: at each node it chooses, among all yes/no questions about one attribute, the one whose two groups have the lowest weighted Gini impurity, and it stops when a group is pure or no question lowers the impurity.

12.2.4 Exact arithmetic, so that Java and Python agree

A small but important detail: two questions can have *exactly* the same impurity, and then the tree depends on which one wins the tie. If Java and Python compute the impurities with floating-point numbers, a difference in the last binary digit could make them choose differently, and the two programs would grow different trees. So our programs never divide. Multiplying Equation (12.2) by n and writing the Gini index with counts, the impurity of a question is the fraction

$$n\,G_{\text{split}} = \frac{\left(a^2 - \sum_k y_k^2\right) b + \left(b^2 - \sum_k z_k^2\right) a}{a\,b} \tag{12.3}$$

where a and b are the sizes of the yes and no groups and y_k, z_k their class counts. Numerator and denominator are whole numbers, and two fractions $\frac{p}{q}$ and $\frac{r}{s}$ are compared as $p \cdot s < r \cdot q$, which is exact. Ties are then broken by a fixed rule: the attribute that comes first in the table wins, and for numbers the smaller threshold. With this, both languages grow the same tree on any data.

12.2.5 A tree that is right on everything it has seen

Our tree answers all 101 animals correctly. That proves less than it seems, because the tree was grown *from* those animals: a tree can always be grown until every leaf is pure, and then it is right on its own data even if it has learned nothing useful. The honest question is how it does on animals it has **not** seen.

With a small data set the best answer is **leave-one-out** testing. Take one animal out, grow a tree from the other 100, and ask the tree about the one left out; put it back, take out the next, and repeat for all 101. Every answer is then an answer about an animal the tree did not learn from. Our tree gets 96 of 101 right this way. The five it gets wrong are instructive: the tortoise, left out, is called an insect by a tree that has seen only four other land reptiles; the newt, an amphibian with a tail, is called a reptile; the scorpion, which has eight legs, is called an insect. These are exactly the "unforeseen ways" in which learned rules over-generalize [1], and exactly the cases an expert would catch at a glance.

12.2.6 From paths to VCIRS nodes

Converting a tree into a VCIRS knowledge base is straightforward, and every part of a node has a natural source.

- The **conditions** of the node are the questions on the path from the top to the leaf, in order: `milk = no, feathers = no, fins = yes`. A yes answer to `legs <= 3` gives the condition `legs <= 3` and a no answer gives `legs > 3`; the engine compares numbers as numbers.
- The **conclusion** is the leaf's class: `type = fish`.

- The **certainty factor** of the node is the share of the leaf's animals that have that class, as a whole percentage (Equation (12.4) below).
- The **cornerstone case** is the first animal of the data that reached the leaf, with all its facts and its name: `animal = bass, hair = no, …`. Every learned node therefore carries a real example, and when the expert later corrects the node, the list of difference compares her new case with a real animal she knows.

The certainty factor is

$$CF = \text{round}\left(100 \cdot \frac{m}{n}\right) \tag{12.4}$$

where n animals reached the leaf and m of them have its class. A pure leaf gives CF 100; a leaf with 8 animals, 6 of one class, would give CF 75, and the engine would carry that uncertainty into its answers as in every chapter since Chapter 3.

Each path becomes a **top-level** node (option 5 of VCIRS), with the path's questions chosen as its conditions. Because the paths of a tree never overlap, at most one top-level node fires for any animal, and the knowledge base gives exactly the tree's answers. From then on it is an ordinary VCIRS knowledge base: it asks only the questions it needs, explains every answer, and learns exceptions.

12.2.7 What the data cannot know

A learned rule can only be as good as the table it comes from, and every table has three kinds of limits that are worth naming before we trust it.

The first is its **vocabulary**. The Zoo data describe each animal with 16 facts, chosen by the person who made the table. If the fact that really separates two classes is not among them, no learning method can find it. Reptiles and amphibians, for example, differ above all in their skin — dry and scaly, or moist and bare — and the table has no column for skin. The tree does its best with what it has (water, eggs), and Section 12.5 will show where that is not enough.

The second is its **sample**. The table holds the 101 animals someone chose to put in it, and they are not a fair picture of the animal kingdom: 41 of them are mammals, but only 5 are reptiles and 4 amphibians, so the rules for reptiles and amphibians rest on very few examples. The table even contains one entry that makes every biology teacher smile: a row called `girl`, a mammal with two legs that is kept by people and is bigger than a cat. It is not wrong — a girl *is* a mammal — but it is a reminder that a data set is a human artifact, with its jokes and its gaps.

The third is its **labels**. The class of each animal was decided by a person, and the classes themselves are a choice. The Zoo data put clams, crabs, lobsters, octopuses, scorpions, jellyfish, slugs, starfish and worms together in one class, which we call *invertebrate* here, although biologists would split them into several very different groups. The tree learns the table's classes, not biology's.

None of this makes the table useless. It makes the expert necessary. A learned knowledge base is a first draft, written quickly from the data; the expert reads it, adds the facts the table lacks, and corrects the answers the table could not know.

12.3 Step 1: CART from Scratch

Listing 12.1 is the whole tree learner in Python.

Listing 12.1 `cart.py`: a CART decision tree with exact Gini arithmetic and leave-one-out testing.

```
"""A small CART decision tree, written from scratch.

    python cart.py [../data/zoo.csv]

CART (Classification And Regression Trees) grows a binary tree: at every
node it tries every yes/no question about one column and keeps the question
whose two groups are purest by the Gini index. It stops when a group is pure
or no question makes it purer. Numbers are compared exactly, as fractions of
whole numbers, so that Java and Python always choose the same question.
"""

import pathlib
import re
import sys

sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import num

NUMBER = re.compile(r"-?[0-9]+(\.[0-9]+)?")


class Data:
    """A table read from a CSV file: a name column, attributes, a class."""

    def __init__(self, path):
        with open(path, encoding="utf-8") as f:
            lines = [ln.strip() for ln in f if ln.strip()]
        header = lines[0].split(",")
        self.attributes = header[1:-1]
        self.target = header[-1]
        self.rows = []                    # (name, {attribute: value}, class)
        for ln in lines[1:]:
            cells = ln.split(",")
            self.rows.append((cells[0], dict(zip(self.attributes, cells[1:-1])),
                              cells[-1]))
        self.numeric = [a for a in self.attributes
                        if all(NUMBER.fullmatch(r[1][a]) for r in self.rows)]


class Question:
    """'legs <= 3' or 'milk = yes': the test of a node, with its two texts."""
```

Listing 12.1 (Continued) *cart.py: a CART decision tree with exact Gini arithmetic and leave-one-out testing.*

```
    def __init__(self, attribute, numeric, value, other):
        self.attribute = attribute
        self.numeric = numeric
        self.value = value                # a threshold, or the value tested
        self.other = other                # the other value, if only two

    def ask(self, values):
        x = values.get(self.attribute)
        if self.numeric:
            return float(x) <= float(self.value)
        return x == self.value

    def yes_text(self):
        op = " <= " if self.numeric else " = "
        return self.attribute + op + self.value

    def no_text(self):
        if self.numeric:
            return self.attribute + " > " + self.value
        if self.other is not None:
            return self.attribute + " = " + self.other
        return self.attribute + " <> " + self.value


class Node:
    def __init__(self, rows):
        self.rows = rows
        self.counts = {}
        for _, _, c in rows:
            self.counts[c] = self.counts.get(c, 0) + 1
        self.counts = dict(sorted(self.counts.items()))
        self.label = ""                   # the most frequent class; a tie
        for c, k in self.counts.items():  # goes to the first name A to Z
            if not self.label or k > self.counts[self.label]:
                self.label = c
        self.question = None
        self.yes = None
        self.no = None

    def pure(self):
        return len(self.counts) == 1
```

Listing 12.1 (Continued) *cart.py: a CART decision tree with exact Gini arithmetic and leave-one-out testing.*

```
def impurity(counts, n):
    """n times the Gini index, times n: n*n - sum of squared counts."""
    return n * n - sum(c * c for c in counts.values())


def questions(data, rows, attribute):
    values = sorted(set(r[1][attribute] for r in rows),
                    key=float if attribute in data.numeric else str)
    if attribute in data.numeric:
        return [Question(attribute, True,
                         num.fmt((float(a) + float(b)) / 2), None)
                for a, b in zip(values, values[1:])]
    if len(values) == 2:
        return [Question(attribute, False, values[1], values[0])]
    return [Question(attribute, False, v, None) for v in values]


def best_question(data, node):
    """The question whose two groups have the lowest total Gini impurity,
    or None when no question makes the node purer. Scores are fractions
    num/den compared by cross-multiplying, so ties are exact."""
    n = len(node.rows)
    best, best_num, best_den = None, 0, 1
    for attribute in data.attributes:
        for q in questions(data, node.rows, attribute):
            yes = Node([r for r in node.rows if q.ask(r[1])])
            no = Node([r for r in node.rows if not q.ask(r[1])])
            a, b = len(yes.rows), len(no.rows)
            if a == 0 or b == 0:
                continue
            score = impurity(yes.counts, a) * b + impurity(no.counts, b) * a
            if best is None or score * best_den < best_num * a * b:
                best, best_num, best_den = q, score, a * b
    if best is None or best_num * n >= impurity(node.counts, n) * best_den:
        return None
    return best
```

Listing 12.1 (Continued) *cart.py: a CART decision tree with exact Gini arithmetic and leave-one-out testing.*

```
def grow(data, rows):
    node = Node(rows)
    if node.pure():
        return node
    q = best_question(data, node)
    if q is None:
        return node
    node.question = q
    node.yes = grow(data, [r for r in rows if q.ask(r[1])])
    node.no = grow(data, [r for r in rows if not q.ask(r[1])])
    return node


def predict(node, values):
    while node.question is not None:
        node = node.yes if node.question.ask(values) else node.no
    return node.label


def leaf_text(node):
    wrong = len(node.rows) - node.counts[node.label]
    return node.label + " (" + str(len(node.rows)) + (
        "/" + str(wrong) if wrong else "") + ")"


def show(node, indent=""):
    """The tree as lines: 'milk = yes: mammal (41)'."""
    out = []
    for text, child in ((node.question.yes_text(), node.yes),
                        (node.question.no_text(), node.no)):
        if child.question is None:
            out.append(indent + text + ": " + leaf_text(child))
        else:
            out.append(indent + text + ":")
            out += show(child, indent + "|  ")
    return out


def paths(node, conds=()):
    """Every leaf with the conditions on the way to it (root first)."""
    if node.question is None:
        return [(list(conds), node)]
    q = node.question
    return (paths(node.yes, conds + (q.yes_text(),))
            + paths(node.no, conds + (q.no_text(),)))
```

Listing 12.1 (Continued) *cart.py: a CART decision tree with exact Gini arithmetic and leave-one-out testing.*

```
def depth(node):
    if node.question is None:
        return 0
    return 1 + max(depth(node.yes), depth(node.no))


def main(args):
    path = args[0] if args else str(pathlib.Path(__file__).resolve()
                                    .parents[1] / "data" / "zoo.csv")
    data = Data(path)
    print(str(len(data.rows)) + " rows, " + str(len(data.attributes))
          + " attributes (numeric: " + (", ".join(data.numeric) or "none")
          + ")")
    tree = grow(data, data.rows)
    print()
    for line in show(tree):
        print(line)
    right = sum(predict(tree, r[1]) == r[2] for r in data.rows)
    print()
    print("leaves: " + str(len(paths(tree))) + ", depth: " + str(depth(tree))
          + ", right on the rows it learned from: " + str(right) + "/"
          + str(len(data.rows)))
    print()
    print("Leave-one-out: learn from all rows but one, then ask about it.")
    wrong = []
    for i, row in enumerate(data.rows):
        t = grow(data, data.rows[:i] + data.rows[i + 1:])
        got = predict(t, row[1])
        if got != row[2]:
            wrong.append(row[0] + ": " + row[2] + ", tree says " + got)
    print("right: " + str(len(data.rows) - len(wrong)) + "/"
          + str(len(data.rows)))
    for w in wrong:
        print("  " + w)


if __name__ == "__main__":
    main(sys.argv[1:])
```

Read it in five parts.

The data. `Data` reads a CSV file whose first column is a name, whose last column is the class, and whose columns in between are the attributes. It keeps each row as a triple: the name, a dictionary of attribute values, and the class. An attribute is treated as a number when every value in its column looks like a number; in the Zoo data that is `legs`.

Questions. A `Question` is one yes/no test: `legs <= 3` for a number, `milk = yes` for a word. `questions` lists every question worth asking about one attribute in a group of rows. For a number it tries every threshold halfway between two neighboring values that occur (for legs 0, 2, 4, 5, 6 and 8: 1, 3, 4.5, 5.5 and 7). For a word with two values, such as `yes` and `no`, one question is enough, because asking about `no` would only swap the two groups; for a word with more values it tries each value against all the others, `stain = oil` against `stain <> oil`. `yes_text` and `no_text` write the two answers as VCIRS conditions.

Nodes and purity. A `Node` holds its rows and the count of each class, in alphabetical order; its `label` is the most frequent class, with ties going to the first name in alphabetical order. `impurity` computes $n^2 - \sum c^2$, which is n^2 times the Gini index of Equation (12.1).

Choosing the question. `best_question` is Equation (12.3) put to work. For every attribute and every question it builds the yes and no groups, skips questions that leave one group empty, and computes the score as the numerator of Equation (12.3) with denominator $a \cdot b$. A new question replaces the best so far only when its fraction is strictly smaller, `score * best_den < best_num * a * b`, so the first of several equal questions is kept. At the end, the best question is returned only if it is strictly purer than the node itself; otherwise the node becomes a leaf.

Growing, predicting and showing. `grow` is recursion in six lines: make the node; if it is pure, or no question helps, it is a leaf; otherwise split its rows by the best question and grow each group. `predict` walks from the top, following the answers. `show` prints the tree with one line per branch, `paths` lists every leaf with the conditions on the way to it, and `main` prints the tree, its size, its answers on its own data and its leave-one-out answers. **Listing 12.2** is the same learner in Java.

Listing 12.2 `Cart.java`: the same CART learner in Java.

```java
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.TreeMap;
import java.util.TreeSet;

import vcirs.Num;

/**
 * A small CART decision tree, written from scratch.
 * java Cart [../data/zoo.csv]
 * CART grows a binary tree: at every node it tries every yes/no question
 * about one column and keeps the question whose two groups are purest by
 * the Gini index. It stops when a group is pure or no question makes it
 * purer. Numbers are compared exactly, as fractions of whole numbers, so
 * that Java and Python always choose the same question.
 */
public class Cart {
    static final String NUMBER = "-?[0-9]+(\\.[0-9]+)?";

    /** One row: the animal's name, its attribute values, its class. */
    record Row(String name, Map<String, String> values, String cls) {
    }

    /** A table read from a CSV file: a name column, attributes, a class. */
    static class Data {
        final List<String> attributes = new ArrayList<>();
        final String target;
        final List<Row> rows = new ArrayList<>();
        final List<String> numeric = new ArrayList<>();
```

Listing 12.2 (Continued) *Cart.java: the same CART learner in Java.*

```
        Data(String path) throws IOException {
            List<String> lines = new ArrayList<>();
            for (String ln : Files.readAllLines(Path.of(path),
                    StandardCharsets.UTF_8)) {
                if (!ln.strip().isEmpty()) {
                    lines.add(ln.strip());
                }
            }
            String[] header = lines.get(0).split(",");
            for (int i = 1; i < header.length - 1; i++) {
                attributes.add(header[i]);
            }
            target = header[header.length - 1];
            for (String ln : lines.subList(1, lines.size())) {
                String[] cells = ln.split(",");
                Map<String, String> values = new LinkedHashMap<>();
                for (int i = 0; i < attributes.size(); i++) {
                    values.put(attributes.get(i), cells[i + 1]);
                }
                rows.add(new Row(cells[0], values, cells[cells.length - 1]));
            }
            for (String a : attributes) {
                boolean all = true;
                for (Row r : rows) {
                    all &= r.values().get(a).matches(NUMBER);
                }
                if (all) {
                    numeric.add(a);
                }
            }
        }
    }

    /** "legs <= 3" or "milk = yes": the test of a node, with its texts. */
    static class Question {
        final String attribute;
        final boolean numeric;
        final String value;             // a threshold, or the value tested
        final String other;             // the other value, if only two
```

Listing 12.2 (Continued) *Cart.java: the same CART learner in Java.*

```
        Question(String attribute, boolean numeric, String value, String other) {
            this.attribute = attribute;
            this.numeric = numeric;
            this.value = value;
            this.other = other;
        }

        boolean ask(Map<String, String> values) {
            String x = values.get(attribute);
            if (numeric) {
                return Double.parseDouble(x) <= Double.parseDouble(value);
            }
            return value.equals(x);
        }

        String yesText() {
            return attribute + (numeric ? " <= " : " = ") + value;
        }

        String noText() {
            if (numeric) {
                return attribute + " > " + value;
            }
            if (other != null) {
                return attribute + " = " + other;
            }
            return attribute + " <> " + value;
        }
    }

    static class Node {
        final List<Row> rows;
        final Map<String, Integer> counts = new TreeMap<>();
        String label = "";              // the most frequent class; a tie
        Question question;              // goes to the first name A to Z
        Node yes;
        Node no;
```

Listing 12.2 (Continued) *Cart.java: the same CART learner in Java.*

```
        Node(List<Row> rows) {
            this.rows = rows;
            for (Row r : rows) {
                counts.merge(r.cls(), 1, Integer::sum);
            }
            for (Map.Entry<String, Integer> e : counts.entrySet()) {
                if (label.isEmpty() || e.getValue() > counts.get(label)) {
                    label = e.getKey();
                }
            }
        }

        boolean pure() {
            return counts.size() == 1;
        }
    }

    /** n times the Gini index, times n: n*n - sum of squared counts. */
    static long impurity(Map<String, Integer> counts, long n) {
        long s = 0;
        for (int c : counts.values()) {
            s += (long) c * c;
        }
        return n * n - s;
    }

    static List<Question> questions(Data data, List<Row> rows, String a) {
        boolean num = data.numeric.contains(a);
        Comparator<String> order = num
                ? Comparator.comparingDouble(Double::parseDouble)
                : Comparator.naturalOrder();
        TreeSet<String> set = new TreeSet<>(order);
        for (Row r : rows) {
            set.add(r.values().get(a));
        }
        List<String> values = new ArrayList<>(set);
        List<Question> out = new ArrayList<>();
        if (num) {
            for (int i = 0; i + 1 < values.size(); i++) {
                double t = (Double.parseDouble(values.get(i))
                        + Double.parseDouble(values.get(i + 1))) / 2;
                out.add(new Question(a, true, Num.fmt(t), null));
            }
        } else if (values.size() == 2) {
            out.add(new Question(a, false, values.get(1), values.get(0)));
        } else {
```

Listing 12.2 (Continued) *Cart.java: the same CART learner in Java.*

```
            for (String v : values) {
                out.add(new Question(a, false, v, null));
            }
        }
        return out;
    }

    static List<Row> side(List<Row> rows, Question q, boolean yes) {
        List<Row> out = new ArrayList<>();
        for (Row r : rows) {
            if (q.ask(r.values()) == yes) {
                out.add(r);
            }
        }
        return out;
    }

    /**
     * The question whose two groups have the lowest total Gini impurity, or
     * null when no question makes the node purer. Scores are fractions
     * num/den compared by cross-multiplying, so ties are exact.
     */
    static Question bestQuestion(Data data, Node node) {
        long n = node.rows.size();
        Question best = null;
        long bestNum = 0, bestDen = 1;
        for (String a : data.attributes) {
            for (Question q : questions(data, node.rows, a)) {
                Node yes = new Node(side(node.rows, q, true));
                Node no = new Node(side(node.rows, q, false));
                long ya = yes.rows.size(), nb = no.rows.size();
                if (ya == 0 || nb == 0) {
                    continue;
                }
                long score = impurity(yes.counts, ya) * nb
                        + impurity(no.counts, nb) * ya;
                if (best == null || score * bestDen < bestNum * ya * nb) {
                    best = q;
                    bestNum = score;
                    bestDen = ya * nb;
                }
            }
        }
        if (best == null || bestNum * n >= impurity(node.counts, n) * bestDen) {
            return null;
        }
```

Listing 12.2 (Continued) *Cart.java: the same CART learner in Java.*

```
        return best;
    }

    static Node grow(Data data, List<Row> rows) {
        Node node = new Node(rows);
        if (node.pure()) {
            return node;
        }
        Question q = bestQuestion(data, node);
        if (q == null) {
            return node;
        }
        node.question = q;
        node.yes = grow(data, side(rows, q, true));
        node.no = grow(data, side(rows, q, false));
        return node;
    }

    static String predict(Node node, Map<String, String> values) {
        while (node.question != null) {
            node = node.question.ask(values) ? node.yes : node.no;
        }
        return node.label;
    }

    static String leafText(Node node) {
        int wrong = node.rows.size() - node.counts.get(node.label);
        return node.label + " (" + node.rows.size()
                + (wrong > 0 ? "/" + wrong : "") + ")";
    }

    /** The tree as lines: "milk = yes: mammal (41)". */
    static List<String> show(Node node, String indent) {
        List<String> out = new ArrayList<>();
        String[] texts = {node.question.yesText(), node.question.noText()};
        Node[] kids = {node.yes, node.no};
        for (int i = 0; i < 2; i++) {
            if (kids[i].question == null) {
                out.add(indent + texts[i] + ": " + leafText(kids[i]));
            } else {
                out.add(indent + texts[i] + ":");
                out.addAll(show(kids[i], indent + "|  "));
            }
        }
        return out;
    }
```

Listing 12.2 (Continued) *Cart.java: the same CART learner in Java.*

```
    /** A leaf with the conditions on the way to it (root first). */
    record Branch(List<String> conditions, Node leaf) {
    }

    static List<Branch> paths(Node node, List<String> conds) {
        List<Branch> out = new ArrayList<>();
        if (node.question == null) {
            out.add(new Branch(conds, node));
            return out;
        }
        List<String> yes = new ArrayList<>(conds);
        yes.add(node.question.yesText());
        List<String> no = new ArrayList<>(conds);
        no.add(node.question.noText());
        out.addAll(paths(node.yes, yes));
        out.addAll(paths(node.no, no));
        return out;
    }

    static int depth(Node node) {
        return node.question == null ? 0
                : 1 + Math.max(depth(node.yes), depth(node.no));
    }

    public static void main(String[] args) throws IOException {
        String path = args.length > 0 ? args[0] : "../data/zoo.csv";
        Data data = new Data(path);
        System.out.println(data.rows.size() + " rows, " + data.attributes.size()
                + " attributes (numeric: " + (data.numeric.isEmpty() ? "none"
                : String.join(", ", data.numeric)) + ")");
        Node tree = grow(data, data.rows);
        System.out.println();
        for (String line : show(tree, "")) {
            System.out.println(line);
        }
        int right = 0;
        for (Row r : data.rows) {
            right += predict(tree, r.values()).equals(r.cls()) ? 1 : 0;
        }
        System.out.println();
        System.out.println("leaves: " + paths(tree, List.of()).size()
                + ", depth: " + depth(tree)
                + ", right on the rows it learned from: " + right + "/"
                + data.rows.size());
        System.out.println();
```

Listing 12.2 (Continued) *Cart.java: the same CART learner in Java.*

```
        System.out.println("Leave-one-out: learn from all rows but one, then "
                + "ask about it.");
        List<String> wrong = new ArrayList<>();
        for (int i = 0; i < data.rows.size(); i++) {
            List<Row> rest = new ArrayList<>(data.rows);
            Row row = rest.remove(i);
            String got = predict(grow(data, rest), row.values());
            if (!got.equals(row.cls())) {
                wrong.add(row.name() + ": " + row.cls() + ", tree says " + got);
            }
        }
        System.out.println("right: " + (data.rows.size() - wrong.size()) + "/"
                + data.rows.size());
        for (String w : wrong) {
            System.out.println("  " + w);
        }
    }
}
```

Java adds a few things Python gives for free. Class counts are kept in a `TreeMap`, which keeps its keys in alphabetical order; the values of an attribute are sorted in a `TreeSet` with a comparator that compares numbers as numbers and words as words; and all the impurity arithmetic uses `long`, 64-bit whole numbers, so that the products of Equation (12.3) cannot overflow for any table that fits in memory. `Row` and `Branch` are records. The thresholds are written with vcirs-core's `Num.fmt`, so `4.5` is printed as `4.5` and `3.0` as `3` in both languages. Run it with `java -cp out Cart` in the `java` folder (after compiling with the engine, as in every chapter) or `python cart.py` in the `python` folder. Both print:

```
101 rows, 16 attributes (numeric: legs)

milk = yes: mammal (41)
milk = no:
|  feathers = yes: bird (20)
|  feathers = no:
|  |  fins = yes: fish (13)
|  |  fins = no:
|  |  |  backbone = yes:
|  |  |  |  aquatic = yes:
|  |  |  |  |  eggs = yes: amphibian (4)
|  |  |  |  |  eggs = no: reptile (1)
|  |  |  |  aquatic = no: reptile (4)
|  |  |  backbone = no:
|  |  |  |  airborne = yes: insect (6)
|  |  |  |  airborne = no:
|  |  |  |  |  predator = yes: invertebrate (8)
|  |  |  |  |  predator = no:
|  |  |  |  |  |  legs <= 3: invertebrate (2)
|  |  |  |  |  |  legs > 3: insect (2)

leaves: 10, depth: 7, right on the rows it learned from: 101/101

Leave-one-out: learn from all rows but one, then ask about it.
right: 96/101
  ladybird: insect, tree says invertebrate
  newt: amphibian, tree says reptile
  scorpion: invertebrate, tree says insect
  seasnake: reptile, tree says amphibian
  tortoise: reptile, tree says insect
```

The printed tree is **Figure 12.2** in text: each indented line is a branch, `(41)` counts the animals that reached a leaf, and a leaf that was not pure would show its mistakes after a slash, as `(8/2)`.

The leave-one-out test takes a moment longer than the rest, because it grows 101 trees; on an ordinary laptop both languages finish in well under a second.

12.4 Step 2: From the Tree to a Knowledge Base

Listing 12.3 turns the learned tree into a VCIRS knowledge base, following Section 12.2.6 line by line.

Listing 12.3 `tree_to_kb.py`: every leaf of the tree becomes a top-level node, with its path as conditions and its first animal as cornerstone.

```
"""From a decision tree to a VCIRS knowledge base.

    python tree_to_kb.py

Every leaf of the tree becomes a top-level node: its conditions are the
questions on the path from the root, its conclusion is the leaf's class, its
certainty factor is the share of the leaf's rows that have that class, and
its cornerstone case is the first animal of the data that reached the leaf.
The knowledge base is saved as ../data/zoo-kb.json.
"""

import pathlib
import sys

import cart
from vcirs import (TOP_LEVEL, Case, Conclusion, Condition, KnowledgeBase,
                   report)

DATA = pathlib.Path(__file__).resolve().parents[1] / "data"
QUESTIONS = {
    "hair": "Does it have hair or fur?", "feathers": "Does it have feathers?",
    "eggs": "Does it lay eggs?", "milk": "Does it feed its young with milk?",
    "airborne": "Can it fly?", "aquatic": "Does it live in water?",
    "predator": "Does it hunt other animals?", "toothed": "Does it have teeth?",
    "backbone": "Does it have a backbone?", "breathes": "Does it breathe air?",
    "venomous": "Is it venomous?", "fins": "Does it have fins?",
    "legs": "How many legs does it have?", "tail": "Does it have a tail?",
    "domestic": "Is it kept by people?", "catsize": "Is it bigger than a cat?",
}


def facts_of(values):
    return [Condition(a, "=", v) for a, v in values.items()]


def tree_to_kb(data, tree, name):
    kb = KnowledgeBase(name, "Learned by CART from " + str(len(data.rows))
                       + " examples, then converted leaf by leaf.")
    for a in data.attributes:
        choices = sorted(set(r[1][a] for r in data.rows),
                         key=float if a in data.numeric else str)
        kb.declare(a, QUESTIONS.get(a, ""), choices)
    for conds, leaf in cart.paths(tree):
        first = leaf.rows[0]                    # the cornerstone animal
        n = len(leaf.rows)                      # the share, rounded half up
        cf = (200 * leaf.counts[leaf.label] + n) // (2 * n)
```

Listing 12.3 (Continued) *tree_to_kb.py: every leaf of the tree becomes a top-level node, with its path as conditions and its first animal as cornerstone.*

```
        case = Case([Condition("animal", "=", first[0])] + facts_of(first[1]),
                    [Conclusion(data.target, leaf.label)], cf)
        kb.refine(case, TOP_LEVEL, chosen=[Condition.parse(c) for c in conds],
                  author="CART")
    return kb


def answer(kb, values):
    got = kb.infer(values).conclusions()
    return got[0].value if got else "none"


def main():
    data = cart.Data(str(DATA / "zoo.csv"))
    tree = cart.grow(data, data.rows)
    kb = tree_to_kb(data, tree, "Animal classes")
    print(kb.summary())
    print()
    for line in report.tree(kb):
        print(line)
    same = sum(answer(kb, r[1]) == cart.predict(tree, r[1])
               for r in data.rows)
    right = sum(answer(kb, r[1]) == r[2] for r in data.rows)
    print()
    print("the knowledge base agrees with the tree on " + str(same) + "/"
          + str(len(data.rows)) + " animals, and is right on " + str(right))
    print("the cornerstone of each node:")
    for node in kb.nodes.values():
        print("  " + node.id + ": " + node.cornerstone.facts[0].value)
    kb.save(str(DATA / "zoo-kb.json"))
    print("saved ../data/zoo-kb.json")


if __name__ == "__main__":
    main()
```

`tree_to_kb` first declares every attribute as a variable, with a question in plain words (from `QUESTIONS`) and the values that occur in the data as its usual answers, so that the knowledge base can ask a child *Does it feed its young with milk? (no/yes)*. Then, for each path, it builds the cornerstone case — the name of the first animal at the leaf, as the fact `animal = …`, followed by all its facts — with the leaf's class as conclusion and the certainty factor of Equation (12.4), computed in whole numbers as $\lfloor (200m + n)/2n \rfloor$, which rounds halves up in both languages. Finally `kb.refine(case, TOP_LEVEL, chosen=…)` creates the node, with the path's questions as the chosen conditions, exactly as an expert would choose conditions from a case. `answer` asks the knowledge base about one animal. **Listing 12.4** is the Java version.

Listing 12.4 `TreeToKb.java`: the same conversion in Java.

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.Map;
import java.util.TreeSet;

import vcirs.Case;
import vcirs.Conclusion;
import vcirs.Condition;
import vcirs.Facts;
import vcirs.KnowledgeBase;
import vcirs.Node;
import vcirs.Report;

/**
 * From a decision tree to a VCIRS knowledge base.
 * java TreeToKb
 * Every leaf of the tree becomes a top-level node: its conditions are the
 * questions on the path from the root, its conclusion is the leaf's class,
 * its certainty factor is the share of the leaf's rows that have that class,
 * and its cornerstone case is the first animal of the data that reached the
 * leaf. The knowledge base is saved as ../data/zoo-kb.json.
 */
public class TreeToKb {
    static final Map<String, String> QUESTIONS = Map.ofEntries(
        Map.entry("hair", "Does it have hair or fur?"),
        Map.entry("feathers", "Does it have feathers?"),
        Map.entry("eggs", "Does it lay eggs?"),
        Map.entry("milk", "Does it feed its young with milk?"),
        Map.entry("airborne", "Can it fly?"),
        Map.entry("aquatic", "Does it live in water?"),
        Map.entry("predator", "Does it hunt other animals?"),
        Map.entry("toothed", "Does it have teeth?"),
        Map.entry("backbone", "Does it have a backbone?"),
        Map.entry("breathes", "Does it breathe air?"),
        Map.entry("venomous", "Is it venomous?"),
        Map.entry("fins", "Does it have fins?"),
        Map.entry("legs", "How many legs does it have?"),
        Map.entry("tail", "Does it have a tail?"),
        Map.entry("domestic", "Is it kept by people?"),
        Map.entry("catsize", "Is it bigger than a cat?"));
```

Listing 12.4 (Continued) *TreeToKb.java: the same conversion in Java.*

```
    static List<Condition> factsOf(Map<String, String> values) {
        List<Condition> out = new ArrayList<>();
        for (Map.Entry<String, String> e : values.entrySet()) {
            out.add(new Condition(e.getKey(), "=", e.getValue()));
        }
        return out;
    }

    static KnowledgeBase treeToKb(Cart.Data data, Cart.Node tree, String name) {
        KnowledgeBase kb = new KnowledgeBase(name, "Learned by CART from "
                + data.rows.size() + " examples, then converted leaf by leaf.");
        for (String a : data.attributes) {
            TreeSet<String> set = new TreeSet<>(data.numeric.contains(a)
                    ? Comparator.comparingDouble(Double::parseDouble)
                    : Comparator.<String>naturalOrder());
            for (Cart.Row r : data.rows) {
                set.add(r.values().get(a));
            }
            kb.declare(a, QUESTIONS.getOrDefault(a, ""), new ArrayList<>(set));
        }
        for (Cart.Branch b : Cart.paths(tree, List.of())) {
            Cart.Node leaf = b.leaf();
            Cart.Row first = leaf.rows.get(0);            // the cornerstone animal
            int n = leaf.rows.size();                      // the share, rounded
            int cf = (200 * leaf.counts.get(leaf.label) + n) / (2 * n); // half up
            List<Condition> facts = new ArrayList<>();
            facts.add(new Condition("animal", "=", first.name()));
            facts.addAll(factsOf(first.values()));
            Case c = new Case(facts, List.of(new Conclusion(data.target,
                    leaf.label)), cf);
            List<Condition> chosen = new ArrayList<>();
            for (String text : b.conditions()) {
                chosen.add(Condition.parse(text));
            }
            kb.refine(c, KnowledgeBase.TOP_LEVEL, null, chosen, "CART");
        }
        return kb;
    }

    static String answer(KnowledgeBase kb, Map<String, String> values) {
        List<Conclusion> got = kb.infer(Facts.map(values)).conclusions();
        return got.isEmpty() ? "none" : got.get(0).value;
    }
```

Listing 12.4 (Continued) *TreeToKb.java: the same conversion in Java.*

```
    public static void main(String[] args) throws Exception {
        Cart.Data data = new Cart.Data("../data/zoo.csv");
        Cart.Node tree = Cart.grow(data, data.rows);
        KnowledgeBase kb = treeToKb(data, tree, "Animal classes");
        System.out.println(kb.summary());
        System.out.println();
        for (String line : Report.tree(kb)) {
            System.out.println(line);
        }
        int same = 0, right = 0;
        for (Cart.Row r : data.rows) {
            String got = answer(kb, r.values());
            same += got.equals(Cart.predict(tree, r.values())) ? 1 : 0;
            right += got.equals(r.cls()) ? 1 : 0;
        }
        System.out.println();
        System.out.println("the knowledge base agrees with the tree on " + same
                + "/" + data.rows.size() + " animals, and is right on " + right);
        System.out.println("the cornerstone of each node:");
        for (Node node : kb.nodes.values()) {
            System.out.println("  " + node.id + ": "
                    + node.cornerstone.facts.get(0).value);
        }
        kb.save("../data/zoo-kb.json");
        System.out.println("saved ../data/zoo-kb.json");
    }
}
```

Both print, and both write the same `zoo-kb.json`, byte for byte:

```
Animal classes: 17 variables, 10 nodes, 10 rules

(root)
+- mammal#1: milk = yes -> type = mammal
+- bird#1: milk = no, feathers = yes -> type = bird
+- fish#1: milk = no, feathers = no, fins = yes -> type = fish
+- amphibian#1: milk = no, feathers = no, fins = no, backbone = yes, aquatic = yes, eggs =
yes -> type = amphibian
+- reptile#1: milk = no, feathers = no, fins = no, backbone = yes, aquatic = yes, eggs = no
-> type = reptile
+- reptile#2: milk = no, feathers = no, fins = no, backbone = yes, aquatic = no -> type =
reptile
+- insect#1: milk = no, feathers = no, fins = no, backbone = no, airborne = yes -> type =
insect
+- invertebrate#1: milk = no, feathers = no, fins = no, backbone = no, airborne = no,
predator = yes -> type = invertebrate
+- invertebrate#2: milk = no, feathers = no, fins = no, backbone = no, airborne = no,
predator = no, legs <= 3 -> type = invertebrate
+- insect#2: milk = no, feathers = no, fins = no, backbone = no, airborne = no, predator =
no, legs > 3 -> type = insect

the knowledge base agrees with the tree on 101/101 animals, and is right on 101
the cornerstone of each node:
  mammal#1: aardvark
  bird#1: chicken
  fish#1: bass
  amphibian#1: frog
  reptile#1: seasnake
  reptile#2: pitviper
  insect#1: gnat
  invertebrate#1: clam
  invertebrate#2: slug
  insect#2: flea
saved ../data/zoo-kb.json
```

The ten nodes are the ten paths of **Figure 12.2**, each with its full list of conditions; `report.tree` shows them all at the top level, because the learned knowledge has no exceptions yet. The knowledge base agrees with the tree on every animal, as it must. And each node has a cornerstone animal: the aardvark stands for all mammals (it is the first mammal in alphabetical order), the frog for the amphibians, the seasnake for the one reptile that lives in water and does not lay eggs. Seventeen variables: the sixteen facts and the conclusion, `type`.

The knowledge base can now do what the tree could not. It asks only what it needs: for a gecko it will ask about milk, feathers, fins, a backbone and water, and stop. It explains: *reptile#2 fired because milk = no, feathers = no, fins = no, backbone = yes, aquatic = no*. And it can be taught.

12.5 Step 3: The Teacher's Lessons

12.5.1 Six animals the data never saw

Bu Lestari tries the knowledge base on six animals from the club's Saturdays, none of which is among the 101, in seven lessons: the komodo dragon from the zoo, the cassowary, a green sea turtle at the mangrove conservation center, a water monitor (*biawak*) in the river, a mudskipper on the mud flats, and a firefly in the orchard at dusk. For each she writes the 16 facts, the class, and — when the knowledge base is wrong — the condition she chooses from the list of difference. They are the lines of `data/lessons.txt`; here are two of them, shortened:

```
green sea turtle | hair = no, feathers = no, ..., fins = yes, legs = 4, ... -> reptile |
breathes = yes
mudskipper | hair = no, feathers = no, ..., fins = yes, legs = 0, ... -> fish | legs = 0
```

The water monitor appears twice, because her first choice of condition will turn out to be a bad one.

12.5.2 The program

Listing 12.5 applies the lessons. It is the Ripple-Down-Rules loop of Chapter 3 with one addition that data make possible: before a lesson is kept, the new node is tried on a copy of the knowledge base and checked against **every animal whose answer is already known** — the 101 animals of the data and the earlier lessons. The cornerstone cases protect ten animals; the data protect a hundred and one.

Listing 12.5 `refine_zoo.py`: the teacher's lessons, each tried on a copy and checked against every known animal before it is kept.

```
"""The biology teacher refines the learned knowledge base, case by case.

    python refine_zoo.py        (after python tree_to_kb.py)

For each lesson of ../data/lessons.txt: what the knowledge base answers now;
if it is wrong, a new node with the condition the teacher chose, as an
exception of the node that answered. The new node is tried on a copy first
and checked against all the animals of the data and every earlier lesson;
it is kept only if none of their answers change.
"""

import pathlib

import cart
from tree_to_kb import answer
from vcirs import (BOTH, TOP_LEVEL, Case, Conclusion, Condition,
                   KnowledgeBase, ScriptedAsker, report)

DATA = pathlib.Path(__file__).resolve().parents[1] / "data"


def main():
    data = cart.Data(str(DATA / "zoo.csv"))
    kb = KnowledgeBase.load(str(DATA / "zoo-kb.json"))
    checks = [(name, values, cls) for name, values, cls in data.rows]
    with open(DATA / "lessons.txt", encoding="utf-8") as f:
        lessons = [ln.strip() for ln in f
                   if ln.strip() and not ln.startswith("#")]
    for line in lessons:
        name, taught, when = [p.strip() for p in line.split("|")]
        left, cls = [p.strip() for p in taught.split("->")]
        facts = [Condition.parse(p) for p in left.split(", ")]
        values = {f.variable: f.value for f in facts}
        case = Case([Condition("animal", "=", name)] + facts,
                    [Conclusion("type", cls)])
        result = kb.infer(values)
        got = answer(kb, values)
        fired = result.paths[0].last().id if result.paths else None
        print(name + ": the knowledge base says " + got
              + (" (" + fired + ")" if fired else "") + "; the teacher says "
              + cls)
        if got == cls:
            kb.learn(case, "teacher")         # a checked case (option 0)
            checks.append((name, values, cls))
            print("  right; recorded as a checked case")
            continue
```

Listing 12.5 (Continued) *refine_zoo.py: the teacher's lessons, each tried on a copy and checked against every known animal before it is kept.*

```
        if fired:
            for text in kb.differences(fired, case).lines()[1:]:
                print("  " + text)
        chosen = [Condition.parse(p) for p in when.split(", ") if p]
        copy = KnowledgeBase.from_json(kb.to_json())
        node = copy.refine(case, BOTH if fired else TOP_LEVEL, fired,
                           chosen or None, "teacher")
        wrong = [n for n, v, c in checks if answer(copy, v) != c]
        print("  the teacher chooses: " + (when or "(the whole list)"))
        if wrong:
            print("  NOT taught: " + str(len(wrong)) + " checked animal(s) "
                  "would change: " + ", ".join(wrong))
            continue
        kb = copy
        checks.append((name, values, cls))
        print("  taught " + str(node) + ", under " + str(node.parent)
              + "; all " + str(len(checks) - 1) + " earlier checks unchanged")
    print()
    print(kb.summary())
    for line in report.tree(kb):
        print(line)
    right = sum(answer(kb, v) == c for _, v, c in checks)
    print("right on " + str(right) + "/" + str(len(checks)) + " checked animals")
    kb.save(str(DATA / "zoo-kb-refined.json"))
    print("saved ../data/zoo-kb-refined.json")
    print()
    asker = ScriptedAsker({"milk": "no", "feathers": "no", "fins": "no",
                           "backbone": "yes", "aquatic": "no"})
    result = kb.infer({}, asker)
    print("A gecko, question by question: "
          + ", ".join(q.variable for q in asker.questions))
    for line in result.what():
        print(line)


if __name__ == "__main__":
    main()
```

For each lesson it prints the knowledge base's answer and the node that gave it. If the answer is right, the case is given to `learn`, which records it as a checked case (option 0), and the animal joins the checks. If not, it prints the list of difference between the lesson and the cornerstone of the node that answered, makes a copy of the knowledge base with `from_json(to_json())`, adds the new node to the copy as an exception of the node that answered, with the teacher's chosen condition — option 3 with chosen conditions, as in Chapter 11 — and asks the copy about every checked animal. If any answer would change, the lesson is refused and the copy thrown away; otherwise the copy becomes the knowledge base. At the end the program saves `zoo-kb-refined.json` and plays one round of the game for a gecko with a `ScriptedAsker`, which answers the engine's questions from a table and remembers what it was asked. **Listing 12.6** is the Java version.

Listing 12.6 `RefineZoo.java`: the teacher's lessons in Java.

```
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

import vcirs.Case;
import vcirs.Conclusion;
import vcirs.Condition;
import vcirs.Facts;
import vcirs.InferenceResult;
import vcirs.KnowledgeBase;
import vcirs.Node;
import vcirs.Question;
import vcirs.Report;
import vcirs.ScriptedAsker;

/**
 * The biology teacher refines the learned knowledge base, case by case.
 * java RefineZoo        (after java TreeToKb)
 * For each lesson of ../data/lessons.txt: what the knowledge base answers
 * now; if it is wrong, a new node with the condition the teacher chose, as
 * an exception of the node that answered. The new node is tried on a copy
 * first and checked against all the animals of the data and every earlier
 * lesson; it is kept only if none of their answers change.
 */
public class RefineZoo {
    public static void main(String[] args) throws Exception {
        Cart.Data data = new Cart.Data("../data/zoo.csv");
        KnowledgeBase kb = KnowledgeBase.load("../data/zoo-kb.json");
        List<Cart.Row> checks = new ArrayList<>(data.rows);
        List<String> lessons = new ArrayList<>();
        for (String ln : Files.readAllLines(Path.of("../data/lessons.txt"),
                StandardCharsets.UTF_8)) {
            if (!ln.strip().isEmpty() && !ln.startsWith("#")) {
                lessons.add(ln.strip());
            }
        }
        for (String line : lessons) {
            String[] p = line.split("\\|", -1);
            String name = p[0].strip();
            String[] taught = p[1].split("->");
            String cls = taught[1].strip();
            String when = p[2].strip();
```

Listing 12.6 (Continued) *RefineZoo.java: the teacher's lessons in Java.*

```
            List<Condition> facts = new ArrayList<>();
            Map<String, String> values = new LinkedHashMap<>();
            for (String f : taught[0].strip().split(", ")) {
                Condition c = Condition.parse(f);
                facts.add(c);
                values.put(c.variable, c.value);
            }
            List<Condition> all = new ArrayList<>();
            all.add(new Condition("animal", "=", name));
            all.addAll(facts);
            Case c = new Case(all, List.of(new Conclusion("type", cls)), 100);
            InferenceResult result = kb.infer(Facts.map(values));
            String got = TreeToKb.answer(kb, values);
            String fired = result.paths.isEmpty() ? null
                    : result.paths.get(0).last().id;
            System.out.println(name + ": the knowledge base says " + got
                    + (fired != null ? " (" + fired + ")" : "")
                    + "; the teacher says " + cls);
            if (got.equals(cls)) {
                kb.learn(c, "teacher");            // a checked case (option 0)
                checks.add(new Cart.Row(name, values, cls));
                System.out.println("  right; recorded as a checked case");
                continue;
            }
            if (fired != null) {
                List<String> lines = kb.differences(fired, c).lines();
                for (String text : lines.subList(1, lines.size())) {
                    System.out.println("  " + text);
                }
            }
            List<Condition> chosen = new ArrayList<>();
            for (String part : when.split(", ")) {
                if (!part.isEmpty()) {
                    chosen.add(Condition.parse(part));
                }
            }
            KnowledgeBase copy = KnowledgeBase.fromJson(kb.toJson());
            Node node = copy.refine(c, fired != null ? KnowledgeBase.BOTH
                    : KnowledgeBase.TOP_LEVEL, fired,
                    chosen.isEmpty() ? null : chosen, "teacher");
            List<String> wrong = new ArrayList<>();
            for (Cart.Row r : checks) {
                if (!TreeToKb.answer(copy, r.values()).equals(r.cls())) {
                    wrong.add(r.name());
                }
            }
```

Listing 12.6 (Continued) *RefineZoo.java: the teacher's lessons in Java.*

```
            System.out.println("  the teacher chooses: "
                    + (when.isEmpty() ? "(the whole list)" : when));
            if (!wrong.isEmpty()) {
                System.out.println("  NOT taught: " + wrong.size()
                        + " checked animal(s) would change: "
                        + String.join(", ", wrong));
                continue;
            }
            kb = copy;
            checks.add(new Cart.Row(name, values, cls));
            System.out.println("  taught " + node + ", under " + node.parent
                    + "; all " + (checks.size() - 1)
                    + " earlier checks unchanged");
        }
        System.out.println();
        System.out.println(kb.summary());
        for (String line : Report.tree(kb)) {
            System.out.println(line);
        }
        int right = 0;
        for (Cart.Row r : checks) {
            right += TreeToKb.answer(kb, r.values()).equals(r.cls()) ? 1 : 0;
        }
        System.out.println("right on " + right + "/" + checks.size()
                + " checked animals");
        kb.save("../data/zoo-kb-refined.json");
        System.out.println("saved ../data/zoo-kb-refined.json");
        System.out.println();
        ScriptedAsker asker = new ScriptedAsker().answer("milk", "no")
                .answer("feathers", "no").answer("fins", "no")
                .answer("backbone", "yes").answer("aquatic", "no");
        InferenceResult r = kb.infer(Facts.none(), asker);
        List<String> asked = new ArrayList<>();
        for (Question q : asker.questions) {
            asked.add(q.variable);
        }
        System.out.println("A gecko, question by question: "
                + String.join(", ", asked));
        for (String line : r.what()) {
            System.out.println(line);
        }
    }
}
```

12.5.3 What happens

Both languages print the same lines and write the same `zoo-kb-refined.json`:

```
komodo dragon: the knowledge base says reptile (reptile#2); the teacher says reptile
  right; recorded as a checked case
cassowary: the knowledge base says bird (bird#1); the teacher says bird
  right; recorded as a checked case
green sea turtle: the knowledge base says fish (fish#1); the teacher says reptile
    A (only in the old case): animal = bass, predator = yes, toothed = yes, breathes = no,
legs = 0, catsize = no
    B (only in the new case): animal = green sea turtle, predator = no, toothed = no,
breathes = yes, legs = 4, catsize = yes
  the teacher chooses: breathes = yes
  taught reptile#3: breathes = yes -> type = reptile, under fish#1; all 103 earlier checks
unchanged
water monitor: the knowledge base says amphibian (amphibian#1); the teacher says reptile
    A (only in the old case): animal = frog, tail = no, catsize = no
    B (only in the new case): animal = water monitor, tail = yes, catsize = yes
  the teacher chooses: tail = yes
  NOT taught: 1 checked animal(s) would change: newt
water monitor: the knowledge base says amphibian (amphibian#1); the teacher says reptile
    A (only in the old case): animal = frog, tail = no, catsize = no
    B (only in the new case): animal = water monitor, tail = yes, catsize = yes, skin =
scales
  the teacher chooses: skin = scales
  taught reptile#4: skin = scales -> type = reptile, under amphibian#1; all 104 earlier
checks unchanged
mudskipper: the knowledge base says reptile (reptile#3); the teacher says fish
    A (only in the old case): animal = green sea turtle, predator = no, toothed = no, legs
= 4, catsize = yes
    B (only in the new case): animal = mudskipper, predator = yes, toothed = yes, legs = 0,
catsize = no
  the teacher chooses: legs = 0
  taught fish#2: legs = 0 -> type = fish, under reptile#3; all 105 earlier checks unchanged
firefly: the knowledge base says insect (insect#1); the teacher says insect
  right; recorded as a checked case

Animal classes: 18 variables, 13 nodes, 13 rules
(root)
+- mammal#1: milk = yes -> type = mammal
+- bird#1: milk = no, feathers = yes -> type = bird
+- fish#1: milk = no, feathers = no, fins = yes -> type = fish
|  +- reptile#3: breathes = yes -> type = reptile
|     +- fish#2: legs = 0 -> type = fish
+- amphibian#1: milk = no, feathers = no, fins = no, backbone = yes, aquatic = yes, eggs =
yes -> type = amphibian
|  +- reptile#4: skin = scales -> type = reptile
+- reptile#1: milk = no, feathers = no, fins = no, backbone = yes, aquatic = yes, eggs = no
-> type = reptile
+- reptile#2: milk = no, feathers = no, fins = no, backbone = yes, aquatic = no -> type =
reptile
+- insect#1: milk = no, feathers = no, fins = no, backbone = no, airborne = yes -> type =
insect
+- invertebrate#1: milk = no, feathers = no, fins = no, backbone = no, airborne = no,
predator = yes -> type = invertebrate
+- invertebrate#2: milk = no, feathers = no, fins = no, backbone = no, airborne = no,
predator = no, legs <= 3 -> type = invertebrate
+- insect#2: milk = no, feathers = no, fins = no, backbone = no, airborne = no, predator =
no, legs > 3 -> type = insect
right on 107/107 checked animals
saved ../data/zoo-kb-refined.json

A gecko, question by question: milk, feathers, fins, backbone, aquatic
Conclusion: type = reptile  (rule reptile#2, CF 100)
```

Follow the lessons one by one.

The komodo dragon and the cassowary are answered correctly by the learned knowledge: a large venomous lizard is a land vertebrate without milk, feathers or fins, and a flightless bird still

has feathers. The data had never seen them, but the rules generalize well here. Both are recorded as checked cases, and from now on they are protected like the 101.

The green sea turtle is called a fish: it has no milk and no feathers, and it swims with flippers, which Bu Lestari recorded as `fins = yes`. The list of difference against the cornerstone of `fish#1`, the bass, shows what separates them: the turtle does not hunt, has no teeth, *breathes air*, has four legs and is bigger than a cat. She chooses the one fact that a biologist would give as the reason: `breathes = yes`. The new node `reptile#3` becomes an exception of `fish#1`, and all 103 earlier checks keep their answers — none of the 13 fish of the data breathes air.

The water monitor is called an amphibian: like a frog it lives in water, lays eggs and has a backbone. The difference from the frog is short: a tail, and a larger size. She chooses `tail = yes` — and the program refuses: the **newt**, an amphibian of the data, also has a tail and lives in water, and with this exception it would become a reptile. This is the moment where the data earn their keep. The cornerstone of `amphibian#1` is the frog, which has no tail; a pure RDR check against cornerstones only would have accepted the lesson and silently broken the newt. The data knew better.

The water monitor again. What really separates reptiles from amphibians is not in the 16 facts at all: reptiles have dry, scaly skin, amphibians moist skin without scales. Bu Lestari adds a new fact, `skin = scales`, which none of the 101 animals has, and chooses it as the condition. Nothing else can change, because no checked animal has that fact, and `reptile#4` is added under `amphibian#1`. The knowledge base now has a variable the data never had — the expert has extended the vocabulary, which no learning from the table could have done.

The mudskipper tests the turtle's lesson. It is a fish that spends much of its life out of water and breathes air through its skin and mouth, so the new node `reptile#3` calls it a reptile. The difference from the turtle's cornerstone is clear: it hunts, has teeth, is small, and has **no legs**. She chooses `legs = 0`, and `fish#2` becomes an exception of `reptile#3`, which is itself an exception of `fish#1`: knowledge rippling down, three levels deep, each level justified by a real animal. **Figure 12.3** shows the two branches the teacher changed.

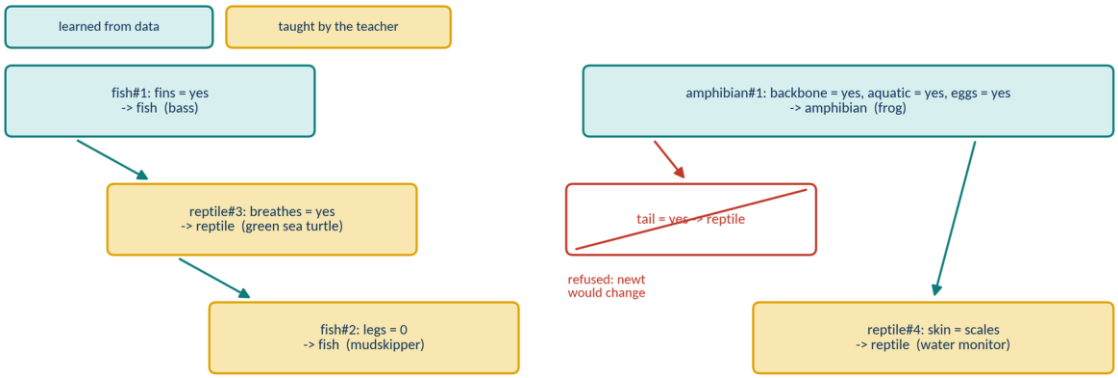

Figure 12.3 The teacher's exceptions. Under the learned fish node, the sea turtle's lesson and, under it, the mudskipper's; under the learned amphibian node, the refused tail lesson and the kept lesson on scales. Each node names its cornerstone animal.

The firefly is an insect that flies, and the learned node for flying animals without a backbone answers it correctly.

At the end the knowledge base has 13 nodes — ten learned from the data, three taught by the teacher — and answers all 107 checked animals correctly. The last lines show it playing the game:

for a gecko it asked about milk, feathers, fins, a backbone and water, five questions, and concluded *reptile* from the learned node `reptile#2`.

12.5.4 The game at the club

On the next Saturday the club plays the game with the refined knowledge base. A child who photographed a gecko on the orchard wall answers five questions — no milk, no feathers, no fins, a backbone, not living in water — and the program says *reptile*, and why: the node that fired, with the five facts that made it fire. The explanation is the part the children like best, because it turns the answer into a small argument they can check with their own photograph. When the program is wrong, or says nothing, it is even better: the child and the teacher look at the list of difference together, and the child proposes the fact that should make the difference. Bu Lestari decides whether it is the right fact, and the check against all the known animals tells them both, in one line, whether the lesson would break anything.

Three habits make the game work in a classroom. The teacher keeps the lessons file and adds each lesson as a line, with the date in a comment, so the class can see its knowledge grow over the term. New facts, like `skin = scales`, are written the way children say them, and the question for each new fact is added to the list of questions so that the program asks it in words, not as a variable name. And once a month the class re-runs the whole chain — the tree from the data, then all the lessons — to see that every lesson still holds; it takes a second, and it is the best lesson of all about what it means to *know* something.

12.6 Step 4: A Check Against scikit-learn

We wrote our own tree learner to understand it. In practice, many people use **scikit-learn**, the machine-learning library that most Python programmers meet first [8]; its current version, as of September 2026, is 1.9.1, released on September 10, 2026 [9]. Its `DecisionTreeClassifier` implements an optimized version of CART, so it is a natural check of our code. **Listing 12.7** gives it the same 101 animals, with *yes* written as 1 and *no* as 0, and compares the two trees and their leave-one-out answers. It is written in Python only, because the check is about our Python and Java code producing the same tree; once the tree is known, Java has nothing to add.

Listing 12.7 `sklearn_check.py`: our tree compared with scikit-learn's `DecisionTreeClassifier`.

```
"""A check of our CART against scikit-learn's DecisionTreeClassifier.

    python -m pip install scikit-learn     (once; Python only)
    python sklearn_check.py

Both learn from the same 101 animals, with yes = 1 and no = 0. We compare
the trees' sizes, the rows they learned from, and leave-one-out answers.
"""

import pathlib

import sklearn
from sklearn.model_selection import LeaveOneOut, cross_val_predict
from sklearn.tree import DecisionTreeClassifier, export_text

import cart

DATA = pathlib.Path(__file__).resolve().parents[1] / "data"


def main():
    data = cart.Data(str(DATA / "zoo.csv"))
    X = [[float(v[a]) if a in data.numeric else float(v[a] == "yes")
          for a in data.attributes] for _, v, _ in data.rows]
    y = [c for _, _, c in data.rows]
    print("scikit-learn " + sklearn.__version__)

    ours = cart.grow(data, data.rows)
    theirs = DecisionTreeClassifier(criterion="gini", random_state=0)
    theirs.fit(X, y)
    print("our tree:          " + str(len(cart.paths(ours))) + " leaves, depth "
          + str(cart.depth(ours)))
    print("scikit-learn tree: " + str(theirs.get_n_leaves()) + " leaves, depth "
          + str(theirs.get_depth()))
    print("scikit-learn, right on the rows it learned from: "
          + str(sum(p == c for p, c in zip(theirs.predict(X), y))) + "/101")
    print()
    print(export_text(theirs, feature_names=data.attributes, decimals=1))
```

Listing 12.7 (Continued) *sklearn_check.py: our tree compared with scikit-learn's DecisionTreeClassifier.*

```
    loo = cross_val_predict(DecisionTreeClassifier(criterion="gini",
                                                   random_state=0),
                            X, y, cv=LeaveOneOut())
    mine = []
    for i, row in enumerate(data.rows):
        t = cart.grow(data, data.rows[:i] + data.rows[i + 1:])
        mine.append(cart.predict(t, row[1]))
    print("leave-one-out, ours:         " + str(sum(
        p == c for p, c in zip(mine, y))) + "/101")
    print("leave-one-out, scikit-learn: " + str(sum(
        p == c for p, c in zip(loo, y))) + "/101")
    print("the two agree on " + str(sum(a == b for a, b in zip(mine, loo)))
          + "/101 animals")
    for (name, _, c), a, b in zip(data.rows, mine, loo):
        if a != c or b != c:
            print("  " + name + " (" + c + "): ours " + a + ", scikit-learn "
                  + b)


if __name__ == "__main__":
    main()
```

Install scikit-learn once with `python -m pip install -r requirements.txt` in the `python` folder, then run `python sklearn_check.py`:

```
scikit-learn 1.9.1
our tree:          10 leaves, depth 7
scikit-learn tree: 10 leaves, depth 7
scikit-learn, right on the rows it learned from: 101/101

|--- milk <= 0.5
|   |--- feathers <= 0.5
|   |   |--- fins <= 0.5
|   |   |   |--- backbone <= 0.5
|   |   |   |   |--- airborne <= 0.5
|   |   |   |   |   |--- predator <= 0.5
|   |   |   |   |   |   |--- legs <= 3.0
|   |   |   |   |   |   |   |--- class: invertebrate
|   |   |   |   |   |   |--- legs >  3.0
|   |   |   |   |   |   |   |--- class: insect
|   |   |   |   |   |--- predator >  0.5
|   |   |   |   |   |   |--- class: invertebrate
|   |   |   |   |--- airborne >  0.5
|   |   |   |   |   |--- class: insect
|   |   |   |--- backbone >  0.5
|   |   |   |   |--- aquatic <= 0.5
|   |   |   |   |   |--- class: reptile
|   |   |   |   |--- aquatic >  0.5
|   |   |   |   |   |--- legs <= 2.0
|   |   |   |   |   |   |--- class: reptile
|   |   |   |   |   |--- legs >  2.0
|   |   |   |   |   |   |--- class: amphibian
|   |   |--- fins >  0.5
|   |   |   |--- class: fish
|   |--- feathers >  0.5
|   |   |--- class: bird
|--- milk >  0.5
|   |--- class: mammal

leave-one-out, ours:         96/101
leave-one-out, scikit-learn: 95/101
the two agree on 100/101 animals
  ladybird (insect): ours invertebrate, scikit-learn invertebrate
  newt (amphibian): ours reptile, scikit-learn reptile
  scorpion (invertebrate): ours insect, scikit-learn insect
  seasnake (reptile): ours amphibian, scikit-learn amphibian
  tortoise (reptile): ours insect, scikit-learn insect
  tuatara (reptile): ours reptile, scikit-learn amphibian
```

The two trees have the same size and the same shape. scikit-learn writes each yes/no question as `feature <= 0.5` (*no* is 0, so "<= 0.5" means *no*) and lists the *no* branch first; read that way, the questions are ours: milk, feathers, fins, backbone, then aquatic and airborne, then predator and legs. The only difference is one question at the bottom. For the five cold-blooded vertebrates in water, scikit-learn separates the one reptile with `legs <= 2`, and we with `eggs = yes`. Both questions split that group perfectly, so their impurity is exactly the same. We break the tie by taking the attribute that comes first in the table (`eggs` is the third column, `legs` the thirteenth); scikit-learn tries the attributes in a random order fixed by its `random_state` and keeps the first of equal ones. Neither choice is wrong. The data do not say which is better; a biologist would say that eggs are a better reason than legs.

The leave-one-out answers agree on 100 of 101 animals. The one difference is the tuatara, the New Zealand reptile, which our tree still calls a reptile when it has not seen it and scikit-learn calls an amphibian — the same tie again, decided the other way. Both learners miss the same five animals, and scikit-learn the tuatara as well, which is reassuring: our small program behaves like the library, and what they both miss is exactly what an expert is for. (scikit-learn 1.8.0, the release before 1.9, prints the same results.)

12.6.1 Tests

`CartTest.java` and `test_cart.py` check that the first question is about milk and that the tree has ten leaves and depth 7; that on a four-row table the question that splits the classes perfectly is chosen; that the converted knowledge base answers every animal exactly as the tree does; and, by running the conversion and the lessons, that the tail lesson is refused and all 107 checked animals are right at the end. The Java tests print:

```
ok: the first question is about milk
ok: 10 leaves, depth 7
ok: the purest question wins: a, not b
ok: the knowledge base answers like the tree
ok: the tail lesson is refused
ok: all 107 checked animals are right
all tests passed
```

12.7 Making It Better

Entropy instead of Gini. Replacing `impurity` by the entropy of ID3 and C4.5 is a small change in both languages, and Exercise 12.3 does it. Entropy involves logarithms, so the exact arithmetic of Equation (12.3) is lost; to keep Java and Python identical, compare the scores through vcirs-core's `Num.key`, which treats numbers that differ only in the last digits as equal, and break ties by column as before.

Smaller trees. A tree grown until every leaf is pure may learn accidents of the data: a leaf with one animal is a rule with one example. The usual remedies are a minimum number of rows per leaf and a maximum depth; the leaves then stay impure, and Equation (12.4) gives their nodes a certainty factor below 100, which the engine reports with every answer. With the Zoo data there is little to gain, but with noisy records — a treatment recorded wrongly, a symptom misreported — a leaf of 30 records with CF 90 is more trustworthy than three leaves of 10 with CF 100.

Learned and taught, side by side. Every node records who made it: `CART` for the learned ones, `teacher` for the lessons. When the data grow — next year's club adds fifty animals — the tree can be learned again, and the teacher's lessons replayed on the new knowledge base, each checked as before. Keeping the lessons as a file, as `lessons.txt` does, is what makes this cheap: the expert's knowledge is data too.

Rule generation and usage. A learned knowledge base is an ordinary VCIRS knowledge base, so the usage rates of Chapter 7 and the rule generation of Chapter 6 apply to it. The long first conditions of every node — `milk = no, feathers = no, fins = no` — are shared by many nodes, which is exactly what VCIRS's variable analysis finds and what its node generation compresses.

Questions in a better order. The knowledge base asks in the order of its nodes, which is the order of the tree's paths. For the game with children, asking the most informative question first — which is the tree's own first question, milk — is already right. For other data, the question order of Chapter 7 can do better.

> **PITFALL.** A tree learns whatever is in the data, including its mistakes and its biases. If the records were collected carelessly, or only in one season, or only from some customers, the rules will be wrong in the same way, and they will look just as confident. Always let an expert read the learned rules before anyone relies on them, and check them on cases the data did not contain.

12.8 Using It for Real

Your records as a table. `cart.py` and `Cart.java` read any CSV file whose first column is a name or a number, whose last column is the answer, and whose other columns are facts. A laundry's consultations from Chapter 10, exported with one SQL query, are such a table; so are a workshop's repairs, a shop's complaints, a farm's crop problems. To see what happens with a small table, `data/stain-records.csv` holds twenty made-up stain records of the kind Chapter 10's laundry keeps. `python cart.py ../data/stain-records.csv` prints:

```
20 rows, 4 attributes (numeric: none)

fabric = silk: dry cleaner (4)
fabric <> silk:
|  stain = oil: dish soap (3)
|  stain <> oil:
|  |  stain = tea:
|  |  |  color = white:
|  |  |  |  age = fresh: cool soak (1)
|  |  |  |  age = dried: oxygen bleach (1)
|  |  |  color = colored: cool soak (2)
|  |  stain <> tea:
|  |  |  stain = ink: alcohol (2)
|  |  |  stain <> ink:
|  |  |  |  stain = mud: brush (2)
|  |  |  |  stain <> mud:
|  |  |  |  |  age = fresh: cold rinse (2)
|  |  |  |  |  age = dried:
|  |  |  |  |  |  stain = wax: scrape and iron (1)
|  |  |  |  |  |  stain = blood: enzyme soak (2)

leaves: 10, depth: 7, right on the rows it learned from: 20/20

Leave-one-out: learn from all rows but one, then ask about it.
right: 13/20
  5: cold rinse, tree says enzyme soak
  9: oxygen bleach, tree says cool soak
  11: cool soak, tree says oxygen bleach
  12: cool soak, tree says oxygen bleach
  13: alcohol, tree says cold rinse
  14: alcohol, tree says enzyme soak
  18: scrape and iron, tree says alcohol
```

The first question it found is the one Bu Rini would ask first: *is it silk?* — every silk item went to the dry cleaner. But with twenty records the tree also learns accidents: at the bottom it separates wax from blood only because they are the last two stains left, and its leave-one-out score is only 13 of 20. Twenty records are enough to suggest rules to an expert, not to replace her. The Zoo data, with 101 clean examples of seven well-separated classes, reach 96 of 101; real records of a small business usually lie in between.

How many examples? There is no fixed number, but two rules of thumb help. Each class should have at least a handful of examples — ten is comfortable, three is fragile — because a leaf built from one or two examples is a guess. And the leave-one-out score tells you, better than any rule of thumb, whether you have enough: if leaving out a single record often changes the answer for that record, the table is too small for its questions, and the learned rules should be treated as suggestions for the expert, not as knowledge. The Zoo data, with 41 mammals and 4 amphibians, show both sides in one table: the mammal rule is rock-solid, and the amphibian rule is the one the teacher had to fix.

Choose facts that people can answer. A rule is only useful if someone at the counter can answer its questions. A column that is known only after the job (how long the repair took) must not be used as a fact to predict the job: the tree would find it, and the rule would be useless before the job. Remove such columns before learning.

Judge with leave-one-out, decide with the expert. Leave-one-out gives an honest estimate of how often the learned rules will be right on new cases; the list of its mistakes is the best list of questions for the expert's first meeting. Convert the tree, let the expert read every node with its cornerstone case, and teach the corrections as lessons, each checked against all the records.

Keep people out of the table. Learning needs facts about situations, not about persons. Customers' names, telephone numbers and addresses do not belong in the columns, and a column such as "customer" would let the tree learn rules about individuals, which is both useless and, under Indonesia's Personal Data Protection Law, a use of personal data for which nobody gave a purpose [10].

Not for diagnosis. Learning rules from health records is a serious field with its own methods, controls and regulations. The tools of this chapter are for sorting, classifying and advising where a mistake is cheap and visible — an animal's class, a stain's treatment, a complaint's department — and for suggesting rules to experts, never for replacing their judgment where a mistake could hurt someone.

12.9 Common Mistakes, and How to Avoid Them

Judging a tree by its own data. 101 out of 101 on the animals it learned from says nothing about new animals. *Avoid it:* use leave-one-out, or keep some cases aside that the tree never sees.

Trusting a rule because a program found it. The tree does not know biology; it knows the table. *Avoid it:* have an expert read every learned node before use, starting with the leave-one-out mistakes.

Checking a correction only against cornerstones. The water monitor's first lesson passed the cornerstone check and would have broken the newt. *Avoid it:* when you have data, check every correction against all of it.

Choosing a condition that is true but not the reason. `tail = yes` was true of the water monitor, but the reason it is a reptile is its scaly skin. *Avoid it:* choose from the list of difference the fact that explains the class; add a new fact when the reason is not among the columns.

Letting a column give the answer away. A column recorded after the fact, or one that is the answer under another name, makes a perfect tree that is useless in practice. *Avoid it:* keep only facts known when the question is asked.

Different spellings in the data. `yes`, `Yes` and `1` are three values to the learner, and three branches to the tree. *Avoid it:* clean the table first; our Zoo copy writes every such value as `yes` or `no`.

Floating-point ties. Two programs that compute impurities in floating point may break a tie differently and grow different trees. *Avoid it:* compute exactly with whole numbers, as Equation (12.3) does, or compare with a tolerance and break ties by a fixed rule.

Learning once and forgetting the lessons. Relearning the tree from new data and throwing away the expert's corrections loses the most valuable knowledge. *Avoid it:* keep the lessons as a file and replay them after every relearning.

12.10 Chapter Summary

- Knowledge can be **learned from data** and then **refined by an expert**: learning gives broad coverage cheaply, the expert gives precision where it matters (**Figure 12.1**).
- A **decision tree** asks one question per node; **CART** chooses each question by the lowest weighted **Gini impurity** of the two groups it makes (Equation (12.1), Equation (12.2), **Table 12.1**); comparing the impurities as fractions of whole numbers (Equation (12.3)) makes Java and Python grow the same tree.
- Our CART (**Listing 12.1**, **Listing 12.2**) learned a ten-leaf tree from the 101 animals of the Zoo data (**Figure 12.2**); it is right on all of them and on 96 of 101 by **leave-one-out**, the honest test.
- Each path of the tree becomes a top-level **VCIRS node** with the path as conditions, the class as conclusion, a **certainty factor** from the leaf's purity (Equation (12.4)) and the first animal of the leaf as **cornerstone case** (**Listing 12.3**, **Listing 12.4**).
- The teacher's lessons (**Listing 12.5**, **Listing 12.6**) are RDR exceptions checked against **every known animal**: the check refused a lesson that the cornerstones alone would have accepted, and the teacher added a fact the data never had (**Figure 12.3**).
- scikit-learn 1.9.1 grows a tree of the same size and shape; the one difference is a tie broken the other way, and the leave-one-out answers agree on 100 of 101 animals (**Listing 12.7**).

12.11 Exercises

12.1 Using Equation (12.1) and Equation (12.2), compute by hand the weighted Gini impurity of the question *feathers = yes* for the 101 animals, and check it against **Table 12.1**.

12.2 In the group of the 60 animals without milk, which question does CART choose, and why not *eggs = yes*? Compute the impurities of *feathers = yes* and *eggs = yes* in that group.

12.3 Replace the Gini index by entropy, $H = -\sum_k p_k \log_2 p_k$, in `cart.py`. Does the tree for the Zoo data change? What must you do so that the Java version grows the same tree?

12.4 Using **Figure 12.2**, classify by hand: (a) a dolphin (milk); (b) a penguin; (c) a scorpion (no backbone, does not fly, hunts, eight legs); (d) a slug (no backbone, does not fly, does not hunt, no legs). Which of these does the Zoo data itself contain?

12.5 A leaf holds 9 animals: 7 insects and 2 invertebrates. What class and certainty factor does the converted node get by Equation (12.4)? What does the engine report for an animal that reaches it?

12.6 Explain why the tail lesson for the water monitor passed the cornerstone check but not the data check. What does this say about checking corrections in a knowledge base that was built without data, as in Chapters 3 to 11?

12.7 A fruit bat (`fruitbat` in the data: milk, flies) and a platypus (milk, lays eggs, has a duck's bill) are both in the Zoo data. Which node answers each, and what would the list of difference look like if the teacher tried to teach that a platypus is special?

12.8 Run `cart.py` on `stain-records.csv` and read the leave-one-out mistakes. Which of them would an expert correct as an exception, and which show that the table is simply too small?

12.9 Suggest two ways to decide the tie at the bottom of the tree (*eggs* or *legs*) that are better than "the first column wins", and say what each needs.

12.12 Solutions to the Exercises

Solution 12.1. *feathers = yes* sends the 20 birds to the yes group ($G_{\text{yes}} = 0$). The no group has 81 animals: 41 mammals, 13 fish, 10 invertebrates, 8 insects, 5 reptiles, 4 amphibians, so $\sum c^2 = 1681 + 169 + 100 + 64 + 25 + 16 = 2055$ and $G_{\text{no}} = 1 - 2055/6561 = 0.6868$. By Equation (12.2), $G_{\text{split}} = \frac{81}{101} \cdot 0.6868 = 0.5508$, as in **Table 12.1**: worse than milk, because the pure group is smaller and the mixed group keeps the 41 mammals.

Solution 12.2. The 60 animals are 20 birds, 13 fish, 10 invertebrates, 8 insects, 5 reptiles and 4 amphibians, $G = 1 - 774/3600 = 0.785$. *feathers = yes* gives 20 birds (pure) and 40 others with counts 13, 10, 8, 5, 4: $\sum c^2 = 374$, $G_{\text{no}} = 1 - 374/1600 = 0.766$, so $G_{\text{split}} = \frac{40}{60} \cdot 0.766 = 0.511$. *eggs = yes*: 58 of the 60 lay eggs, so the question separates almost nothing — its yes group ($G_{\text{yes}} = 0.7782$) is almost the whole node, and its no group is one reptile and one invertebrate ($G_{\text{no}} = 0.5$), so $G_{\text{split}} = \frac{58}{60} \cdot 0.7782 + \frac{2}{60} \cdot 0.5 = 0.769$, hardly better than 0.785. Feathers wins, as the tree shows.

Solution 12.3. In `impurity`'s place compute the weighted entropy of the two groups, $\frac{a}{n} H_{\text{yes}} + \frac{b}{n} H_{\text{no}}$, with `math.log2` (Java: `Math.log(x) / Math.log(2)`). On the Zoo data the chosen questions are the same, because the pure groups that Gini prefers are also those of lowest entropy. For Java and Python to agree, compare the scores with `num.key` / `Num.key` (equal keys are a tie) and keep the rule that the first attribute wins ties; the exact fractions of Equation (12.3) are no longer available, because logarithms are not fractions.

Solution 12.4. (a) Dolphin: milk = yes → mammal (the Zoo data contains the dolphin). (b) Penguin: milk no, feathers yes → bird (in the data). (c) Scorpion: milk no, feathers no, fins no, backbone no, airborne no, predator yes → invertebrate (in the data; this is right, while the leave-one-out tree, which had not seen it, said insect). (d) Slug: … predator no, legs 0 ≤ 3 → invertebrate (in the data, and the cornerstone of `invertebrate#2`).

Solution 12.5. The class is the majority, *insect*, and $CF = \text{round}(100 \cdot 7/9) = \text{round}(77.8) = 78$. The node's CF becomes the rule CF of eq. (10) of the thesis, as in every chapter: an animal reaching it is reported as `type = insect` with CF 78 (times the certainty of its facts, 100 when the user is sure), telling the user that about one animal in five in this situation was not an insect.

Solution 12.6. The cornerstone check only asks about the cornerstone cases of existing nodes. The cornerstone of `amphibian#1` is the frog, which has no tail, so the new condition `tail = yes` is false for it and the check passes. The newt reaches the same node but is not a cornerstone; only a check against all known cases sees it. In a knowledge base built by hand, the cornerstones *are* the known cases, so the check is complete for what the system has seen; but it is wise, as Chapter 11 did with golden cases, to keep more checked cases than cornerstones, because every case that the knowledge base has answered correctly and that someone confirmed is a free test.

Solution 12.7. Both reach `mammal#1` by `milk = yes`, the first question; nothing else is asked. To teach that a platypus is special (say, the class *monotreme*), the list of difference against the aardvark, the cornerstone of `mammal#1`, would show in section A `eggs = no, toothed = yes, …` and in section B `eggs = yes, toothed = no, aquatic = yes, …`; the teacher would choose `eggs = yes`, and the check would confirm that no other mammal of the data lays eggs — the platypus is the only one.

Solution 12.8. The mistakes for records 13 and 14 (ink called *cold rinse* and *enzyme soak*) and 18 (wax called *alcohol*) happen because, with one of only two ink records or the only wax record left out, the tree has not seen enough of that stain; record 5, fresh blood, is the same story with one of only two fresh blood records. This is a table too small, not a rule to correct. The mistakes for records 9, 11 and 12 (tea) show a real distinction that three examples cannot settle: white and dried gets oxygen bleach, the rest a cool soak. An expert would write that as a rule and an exception — exactly the tea node and its exception in Chapter 10's seed cases.

Solution 12.9. First, ask the expert: show the two questions and their groups, and let her choose, which needs nothing but a moment of her time and gives the tree her reason. Second, prefer the question that also works best on cases outside the group: choose the one with the better leave-one-out score on the whole data, which needs more computation. (A third, cheap one: prefer the attribute that is easiest to observe — whether an animal lays eggs is harder to see at the riverbank than its legs — which needs a cost for each column.)

References

[1] M. M. I. Subakti, "A Variable-Centered Intelligent Rule System," M.Sc.Eng. thesis, Dept. of Computer Science and Information Engineering, Taipei, Taiwan, Jun. 2005.

[2] R. Forsyth, "Zoo," UCI Machine Learning Repository, 1990, doi: 10.24432/C5R59V. Licensed under CC BY 4.0; copy used from F. Leisch and E. Dimitriadou, *mlbench: Machine Learning Benchmark Problems*, R package, CRAN.

[3] P. Compton and B. H. Kang, *Ripple-Down Rules: The Alternative to Machine Learning*. Boca Raton, FL, USA: CRC Press, 2021.

[4] B. R. Gaines and P. J. Compton, "Induction of ripple down rules," in *Proc. 5th Australian Conf. Artificial Intelligence*, Hobart, Australia, 1992.

[5] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Belmont, CA, USA: Wadsworth, 1984.

[6] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, no. 1, pp. 81–106, 1986.

[7] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Mateo, CA, USA: Morgan Kaufmann, 1993.

[8] F. Pedregosa *et al.*, "Scikit-learn: machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[9] scikit-learn developers, "scikit-learn 1.9.1," Python Package Index, Sep. 2026. [Online]. Available: https://pypi.org/project/scikit-learn/

[10] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. Jakarta, Indonesia, 2022.

Chapter 13

Talking to Your Expert System: a Local Language Model with Human-Checked Knowledge

By the end of this chapter you will be able to explain in plain words what a large language model is good at and why it must not be the one that decides; run a small open model on your own computer with Ollama and talk to it through its HTTP API from Java and from Python; ask the model for answers in a fixed JSON shape; build a **chat front end** for the Wi-Fi doctor of Chapter 3 in which the model only *understands* the customer's words and *phrases* the reply, while the VCIRS knowledge base, checked by a person, makes every decision; write a **guard** that rejects any reply the knowledge base does not back and replaces it with a safe one; and run everything offline with a small stand-in server, so that every conversation in this chapter can be repeated on any computer, with or without a model.

13.1 Messages at Nine in the Evening

The Wi-Fi doctor of Chapter 3 lives on the computer at Mas Anto's counter. It asks one question at a time, it explains every piece of advice, and it learns from every fix that the knowledge base did not know. Customers like it. But in the evening, when the counter is closed, Mas Anto's phone keeps buzzing. People send messages the way people write messages: *"mas, internet di rumah mati, lampu router kedip-kedip"*, *"hp saya ga bisa internet padahal wifi nyala"*, *"my phone has no internet, the wifi symbol is off and the little plane is on"*. Nobody writes `wifi icon = off`. Nobody wants to answer seven questions in a row.

Mas Anto has seen the chat programs everyone talks about. They answer anything, in any language, in beautiful sentences. His nephew Fajar suggests simply letting one of them answer the customers. Mas Anto tried it one evening and was worried by what he saw. He typed the same complaint twice and got two different lists of advice. One list told the customer to reset the phone to factory settings, which deletes the photos of a grandmother's grandchildren and does nothing for a router whose light is red. None of the answers mentioned the one thing that his own knowledge base knows is the cause in his neighborhood: when the router's light is red, the problem is the provider's cable, and the only useful advice is to call the provider.

So Mas Anto wants two things that seem to pull in opposite directions. He wants the doctor to understand messages written in ordinary words, the way the chat programs do. And he wants the *advice* to stay exactly as careful as it is now: only what he has checked on real customers, explained, and never invented. This chapter shows that the two wishes do not conflict at all, as long as each part does the job it is good at. The language model reads and writes. The knowledge base decides. And a small, strict piece of code — the guard — stands between the model and the customer.

We will run the model on the counter's own computer, not on a service on the internet, for three reasons that matter to a small shop: the customers' messages never leave the building, there is no

bill per message, and the doctor keeps working when the internet — the very thing the customers are complaining about — is down.

13.2 The Idea in Plain Words

13.2.1 What a language model is, and what it is not

A **large language model** is a program that has read an enormous amount of text and has learned, from it, to continue text. Give it the beginning of a sentence and it predicts which word is likely to come next, then the next, and so on. Trained on enough text and then taught to follow instructions, such a model can summarize, translate, rewrite, and answer questions in fluent sentences. That fluency is real, and for one job it is exactly what Mas Anto needs: understanding what a customer means by "lampu router kedip-kedip" and writing a friendly reply.

But the model does not *know* anything in the way Mas Anto's knowledge base knows it. It has no list of checked cases, no cornerstone case behind each piece of advice, and no way to say "I have not seen this problem yet". When it lacks the answer, it still produces the most likely-looking sentence, and a likely-looking sentence can be wrong. Researchers call this **hallucination**: text that is fluent and confident but not supported by the facts it was given or by the world [1]. A survey of the field describes it as a common problem of all such generators, not a rare accident of one model [1]. Two further properties make a model unsuitable as the one who decides at a counter. Its answers can change from one run to the next, so two customers with the same problem may be told different things. And it cannot explain *why* in the sense of Chapter 3: it can write a plausible reason, but the reason is not the path that produced the answer.

A VCIRS knowledge base has the opposite character. It knows only what it was taught, case by case, by a person who saw the case solved [2], [3]. It gives the same answer to the same facts every time. It explains every answer by the facts that made a node fire. And when no node fires, it says so. What it cannot do is read a message written in ordinary words.

> **KEY INSIGHT.** Do not ask one component to do both jobs. Let the language model do what it is good at — reading and writing sentences — and let the knowledge base do what it is good at — deciding, from checked knowledge, with an explanation.

13.2.2 Two jobs for the model, one for the knowledge base

Figure 13.1 shows the whole design. It is worth reading slowly, because every later section builds one of its boxes.

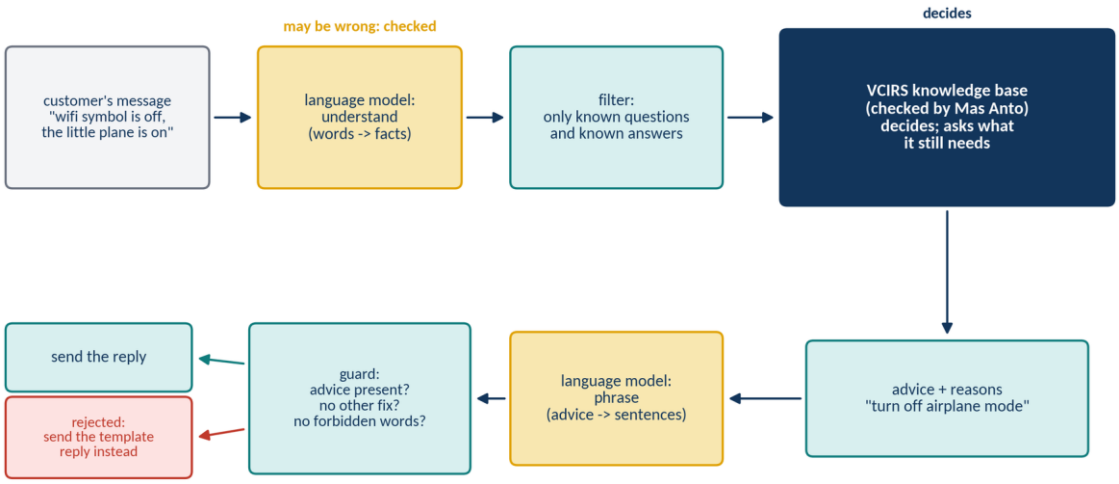

Figure 13.1 The chat front end. The language model (amber) only turns words into facts and advice into sentences; everything it produces is checked. The VCIRS knowledge base (navy) decides, and asks for the facts it still needs. The guard lets a reply through only if it gives the knowledge base's advice and nothing else.

1. **Understand.** The customer's message goes to the model with an instruction: *turn this message into facts, using only these variables and these values*. The model replies with a small piece of JSON, for example `{"facts": [{"variable": "airplane mode", "value": "on"}]}`.
2. **Filter.** The program does not trust that JSON. It keeps a fact only if its variable is one the knowledge base asks about and its value is one of the answers the knowledge base knows. Everything else is shown and dropped.
3. **Decide.** The facts go to `kb.infer`, exactly as in Chapter 3. If a node needs a fact that the message did not contain, the engine asks for it, and the customer may answer in any words — the model helps to understand that answer too, through the same filter.
4. **Phrase.** When the knowledge base gives advice, the program sends the advice and its reasons (the conditions of the nodes that fired) back to the model with a second instruction: *write two or three friendly sentences that give exactly this advice*.
5. **Guard.** Before a reply reaches the customer, the program checks it: does it contain the knowledge base's advice, word for word? Does it avoid every *other* fix the knowledge base knows? Does it avoid the words Mas Anto has forbidden, such as "factory"? If any check fails, the reply is thrown away and the program writes a plain reply itself from a template.

When the knowledge base has no advice, step 4 never happens. The program says, in a fixed sentence, that it has no checked advice yet and that Mas Anto will look at it himself. The model is never asked to fill the gap, because filling gaps with likely-looking sentences is precisely what it must not do here.

Notice where the model's words can go wrong and what happens then. A wrong fact from step 1 is either dropped by the filter or, if it happens to be a valid fact, shown to the customer as `[understood] ...`, so a mistake is visible and the customer can correct it. A wrong sentence from step 4 is stopped by the guard. In no case can the model add advice that Mas Anto never checked.

This division of labor is a close relative of a popular technique called **retrieval-augmented generation**, in which a model is given relevant documents to read before it answers, so that its answer can rest on them [4]. Our design goes one step further in the same direction: the model is not given documents to summarize, it is given *the decision itself*, and a program checks that the decision survived the rewording. For a counter where a wrong answer costs a customer a morning, that extra step is worth it.

13.2.3 A model on your own computer: Ollama

Ollama is a free program that downloads open language models and runs them on your own computer, on Windows, macOS and Linux. Once it is running, it listens on the address `http://localhost:11434` and answers requests over HTTP, the same protocol a web browser uses; any program that can send an HTTP request can talk to it, including a Java or Python program that uses nothing but its standard library [5]. As of September 2026 the current release is Ollama v0.34.3, published on September 19, 2026 [6]. Ollama changes quickly, but the part of it that this chapter uses — the chat endpoint `POST /api/chat` — has kept its shape for a long time, and the code of this chapter depends only on it.

Which model? For a counter computer the model must be small: a few gigabytes to download and a size that runs acceptably on an ordinary laptop processor. **Table 13.1** lists three families of small open models available in Ollama's library in September 2026, with the download size of the default version of each tag.

Table 13.1 Small open models in Ollama's library, September 2026, with the download size of each tag's default version.

Tag	Download	Notes
`qwen3.5:0.8b`	1.0 GB	the smallest; for old laptops [7]
`qwen3.5:2b`	2.7 GB	this chapter's default; trained on 201 languages and dialects [7]
`qwen3.5:4b`	3.4 GB	better understanding, slower [7]
`llama3.2:1b`	1.3 GB	eight official languages, Indonesian not among them [8]
`llama3.2:3b`	2.0 GB	the same family, larger [8]
`gemma4:e2b`	7.2 GB	multimodal "edge" model; a large download [9]

We use `qwen3.5:2b` as the default for two reasons: its size suits a counter computer, and its model page lists support for 201 languages and dialects [7], which matters when half of Mas Anto's customers write in Indonesian and some in Javanese. Llama 3.2's page names eight official languages, and Indonesian is not one of them [8]; it may still understand simple Indonesian, but you would be relying on luck. None of this is permanent: new small models appear every few months. Because the model's only jobs are reading and writing, and because every answer is checked, you can change the model with one option (`--model`) without changing a single rule of the knowledge base.

13.2.4 Asking for JSON, not for prose

In step 1 we need facts, not a friendly paragraph. A model asked politely for JSON will usually comply, but "usually" is not good enough for a program that must read the answer. Ollama solves this with **structured outputs**: the request may carry a `format` field containing a JSON Schema, a description of the exact shape the answer must have, and Ollama constrains the model so that its output matches that shape [5], [10]. Our schema says: the answer is an object with one field, `facts`, which is a list of objects, each with a `variable` and a `value`, both strings; and the `variable` must be one of the variables of the knowledge base.

The schema fixes the *shape*; it does not make the *content* true. The model can still put `blinking` as the value of `router lights`, a value Mas Anto's knowledge base has never used, or claim a fact the customer never said. That is why step 2, the filter, is a separate piece of code that trusts nothing.

13.2.5 The guard

> **DEFINITION.** A *guard* is a check, written as ordinary code, that every output of the language model must pass before it is used. Here it accepts a reply only if (1) it contains the knowledge base's advice word for word, (2) it contains no other fix that the knowledge base knows, (3) it contains none of the phrases the expert has forbidden, and (4) it is short. Otherwise the reply is replaced by one built from a template.

Each rule of the guard has a reason. The first makes sure the customer actually hears the advice; a reply that says "switch off the plane mode" may mean the same thing, but the program cannot prove it, so it does not take the risk. The second stops the most common failure of a helpful model: adding "you could also restart the router" to advice that was about airplane mode. The third is Mas Anto's own list of things that must never be suggested at the counter — factory resets, opening a phone, anything about batteries — kept in a small text file he can edit. The fourth keeps the model from burying the advice in a page of text.

The guard is deliberately strict, and strictness has a price: some good replies will be rejected and replaced by the plainer template. That is the right trade for a counter. A plain correct sentence is better than an elegant wrong one, and the template is always available, so the customer always gets an answer.

13.3 Step 1: Talking to Ollama

Every request to Ollama's chat endpoint is a small JSON object sent with `POST` to `/api/chat`. It names the model, gives a list of messages — each with a `role` (`system` for the instructions, `user` for what the person said) and its `content` — and may add options. We send four more fields every time: `"stream": false`, so that the answer arrives as one piece instead of word by word; `"think": false`, so that models that can "reason aloud" before answering skip it and answer at once; `"options": {"temperature": 0, "seed": 42}`, which asks for the most likely words and a fixed random seed, so that the same question gets the same answer as often as the model allows; and, when we want JSON, the `format` schema [5]. The answer is another JSON object whose field `message.content` holds the model's text, together with information such as `eval_count`, the number of tokens (pieces of words) it produced.

Listing 13.1 is the whole client in Java.

Listing 13.1 `OllamaClient.java`: a minimal client for Ollama's chat API.

```
import java.io.IOException;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.nio.charset.StandardCharsets;
import java.time.Duration;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

import vcirs.Json;
import vcirs.VcirsException;

/**
 * A very small client for Ollama's chat API: POST /api/chat. The same class
 * talks to a real Ollama on this computer (http://localhost:11434) or to the
 * stand-in server of this chapter (StubOllama), which answers the same way.
 */
public class OllamaClient {
    public static final String DEFAULT_URL = "http://localhost:11434";
    public static final String DEFAULT_MODEL = "qwen3.5:2b";

    /** The language model could not be reached or gave no usable reply. */
    public static class LlmException extends RuntimeException {
        LlmException(String message) {
            super(message);
        }
    }

    public final String url;
    public final String model;
    public Map<String, Object> last = new LinkedHashMap<>();
    // A local model needs no proxy, even when the computer has one.
    private final HttpClient http = HttpClient.newBuilder()
            .proxy(HttpClient.Builder.NO_PROXY)
            .connectTimeout(Duration.ofSeconds(10)).build();

    public OllamaClient(String url, String model) {
        this.url = url.replaceAll("/+$", "");
        this.model = model;
    }
```

Listing 13.1 (Continued) *OllamaClient.java: a minimal client for Ollama's chat API.*

```
    /**
     * Send one system and one user message; return the reply text. With a
     * schema (not null), the model must answer with JSON of that shape.
     */
    @SuppressWarnings("unchecked")
    public String chat(String system, String user, Map<String, Object> schema) {
        Map<String, Object> body = new LinkedHashMap<>();
        body.put("model", model);
        body.put("stream", false);                    // one complete answer
        body.put("think", false);                     // no long reasoning
        body.put("options", Map.of("temperature", 0, "seed", 42));
        body.put("messages", List.of(
                Map.of("role", "system", "content", system),
                Map.of("role", "user", "content", user)));
        if (schema != null) {
            body.put("format", schema);               // structured output
        }
        HttpRequest request = HttpRequest.newBuilder(URI.create(url + "/api/chat"))
                .timeout(Duration.ofSeconds(120))
                .header("Content-Type", "application/json")
                .POST(HttpRequest.BodyPublishers.ofString(Json.write(body),
                        StandardCharsets.UTF_8))
                .build();
        Object answer;
        try {
            HttpResponse<String> response = http.send(request,
                    HttpResponse.BodyHandlers.ofString(StandardCharsets.UTF_8));
            try {
                answer = Json.parse(response.body());
            } catch (VcirsException e) {
                answer = Map.of("error", "HTTP " + response.statusCode());
            }
        } catch (IOException | IllegalArgumentException e) {
            throw new LlmException("cannot reach the language model at " + url);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            throw new LlmException("interrupted");
        }
        if (answer instanceof Map && ((Map<String, Object>) answer)
                .containsKey("error")) {                  // e.g. model not found
            throw new LlmException("the language model said: "
                    + ((Map<String, Object>) answer).get("error"));
        }
        if (!(answer instanceof Map) || !((Map<String, Object>) answer)
                .containsKey("message")) {
            throw new LlmException("the language model gave no message");
```

Listing 13.1 (Continued) *OllamaClient.java: a minimal client for Ollama's chat API.*

```
        }
        last = (Map<String, Object>) answer;
        Object message = last.get("message");
        Object content = message instanceof Map
                ? ((Map<String, Object>) message).get("content") : null;
        return content == null ? "" : content.toString();
    }
}
```

The body of the request is built as a map and written as JSON with `vcirs.Json`, the small JSON reader and writer of vcirs-core, so that the chapter needs no extra library. `HttpClient` is the HTTP client of the Java standard library. We build it with `NO_PROXY` on purpose: many offices and campuses send web traffic through a proxy, and a proxy cannot reach `localhost` on your computer; a request to a model on this computer must go straight to it. `http.send` sends the

request and waits for the whole answer. The answer is read with `Json.parse`; if it is not JSON at all, the program pretends it received `{"error": "HTTP 500"}` (or whatever the status was), so that all failures take the same path.

Look at the three ways the method can fail, because a counter program must survive all of them. If no program listens at the address — Ollama is not installed, or not started — `send` throws an `IOException`, and we turn it into an `LlmException` that says the model cannot be reached. If Ollama answers but with an error, for example because the model name is misspelled or the model was never downloaded, the answer contains an `error` field, and we pass its text on. If the answer has no `message`, we say so. Every caller of `chat` therefore needs to handle exactly one kind of exception.

Listing 13.2 is the same client in Python.

Listing 13.2 `ollama_client.py`: the same client in Python, standard library only.

```
"""A very small client for Ollama's chat API: POST /api/chat.

Only the Python standard library is used. The same class talks to a real
Ollama on this computer (http://localhost:11434) or to the stand-in server
of this chapter (stub_ollama.py), which answers in the same format.
"""

import json
import urllib.error
import urllib.request

DEFAULT_URL = "http://localhost:11434"
DEFAULT_MODEL = "qwen3.5:2b"


class LlmError(Exception):
    """The language model could not be reached or gave no usable reply."""


class OllamaClient:
    def __init__(self, url=DEFAULT_URL, model=DEFAULT_MODEL, timeout=120):
        self.url = url.rstrip("/")
        self.model = model
        self.timeout = timeout
        self.last = {}                  # the last full answer, for curiosity
        # A local model needs no proxy, even when the computer has one.
        self.opener = urllib.request.build_opener(
            urllib.request.ProxyHandler({}))

    def chat(self, system, user, schema=None):
        """Send one system and one user message; return the reply text.
        With a schema, the model must answer with JSON of that shape."""
        body = {
            "model": self.model,
            "stream": False,                # one complete answer
            "think": False,                 # no long reasoning, just answer
            "options": {"temperature": 0, "seed": 42},
            "messages": [{"role": "system", "content": system},
                         {"role": "user", "content": user}],
        }
        if schema is not None:
            body["format"] = schema         # structured output
        request = urllib.request.Request(
            self.url + "/api/chat", data=json.dumps(body).encode("utf-8"),
            headers={"Content-Type": "application/json"})
        try:
```

Listing 13.2 (Continued) *ollama_client.py: the same client in Python, standard library only.*

```
        with self.opener.open(request, timeout=self.timeout) as response:
            status, text = response.status, response.read()
    except urllib.error.HTTPError as e:      # e.g. 404: model not found
        status, text = e.code, e.read()
    except OSError as e:
        raise LlmError("cannot reach the language model at "
                       + self.url) from e
    try:
        answer = json.loads(text.decode("utf-8"))
    except ValueError:
        answer = {"error": "HTTP " + str(status)}
    if isinstance(answer, dict) and "error" in answer:
        raise LlmError("the language model said: " + str(answer["error"]))
    if not isinstance(answer, dict) or "message" not in answer:
        raise LlmError("the language model gave no message")
    self.last = answer
    message = answer["message"]
    return str(message.get("content", "")) if isinstance(message, dict) \
        else ""
```

`urllib.request` is Python's built-in HTTP client. `ProxyHandler({})` plays the role of `NO_PROXY`. When the server answers with an error status such as 404, `urllib` raises `HTTPError`, which carries the answer body; we read it just like a normal answer, so that the `error` field reaches the same place as in Java.

Now a first conversation. **Listing 13.3** asks the model one question and prints what it says.

Listing 13.3 `AskOllama.java`: step 1, one question to the model.

```java
import java.util.ArrayList;
import java.util.List;

/**
 * Step 1: ask a local language model one question and print its reply.
 * java AskOllama [--url URL] [--model NAME] [question]
 */
public class AskOllama {
    public static void main(String[] args) {
        String url = OllamaClient.DEFAULT_URL;
        String model = OllamaClient.DEFAULT_MODEL;
        List<String> words = new ArrayList<>();
        for (int i = 0; i < args.length; i++) {
            if (args[i].equals("--url") && i + 1 < args.length) {
                url = args[++i];
            } else if (args[i].equals("--model") && i + 1 < args.length) {
                model = args[++i];
            } else {
                words.add(args[i]);
            }
        }
        String question = words.isEmpty()
                ? "My Wi-Fi symbol is on but nothing loads. What should I do?"
                : String.join(" ", words);
        OllamaClient llm = new OllamaClient(url, model);
        System.out.println("Question: " + question);
        String reply;
        try {
            reply = llm.chat("You are a helpful assistant. Answer briefly.",
                    question, null);
        } catch (OllamaClient.LlmException e) {
            System.out.println("Sorry: " + e.getMessage());
            return;
        }
        System.out.println("Model " + model + " replied:");
        System.out.println(reply);
        Object count = llm.last.get("eval_count");
        System.out.println("(" + (count == null ? "?" : count)
                + " tokens generated)");
    }
}
```

Listing 13.4 is the Python version.

Listing 13.4 `ask_ollama.py`: step 1 in Python.

```
"""Step 1: ask a local language model one question and print its reply.

    python ask_ollama.py [--url URL] [--model NAME] [question]
"""

import sys

from ollama_client import DEFAULT_MODEL, DEFAULT_URL, LlmError, OllamaClient


def main(args):
    url, model, words = DEFAULT_URL, DEFAULT_MODEL, []
    i = 0
    while i < len(args):
        if args[i] == "--url" and i + 1 < len(args):
            url, i = args[i + 1], i + 2
        elif args[i] == "--model" and i + 1 < len(args):
            model, i = args[i + 1], i + 2
        else:
            words.append(args[i])
            i += 1
    question = " ".join(words) or ("My Wi-Fi symbol is on but nothing "
                                   "loads. What should I do?")
    llm = OllamaClient(url, model)
    print("Question: " + question)
    try:
        reply = llm.chat("You are a helpful assistant. Answer briefly.",
                         question)
    except LlmError as e:
        print("Sorry: " + str(e))
        return
    print("Model " + model + " replied:")
    print(reply)
    print("(" + str(llm.last.get("eval_count", "?")) + " tokens generated)")


if __name__ == "__main__":
    main(sys.argv[1:])
```

With a real model you would simply run `java -cp out AskOllama` or `python ask_ollama.py`. To make this book repeatable on any computer, the transcripts in this chapter were produced with the stand-in server of the next section, which listens on port 4131; the command is `java -cp out AskOllama --url http://localhost:4131`:

```
Question: My Wi-Fi symbol is on but nothing loads. What should I do?
Model qwen3.5:2b replied:
Here are some things to try: restart your phone, forget the network and join it again,
reset the network settings, or update the router's firmware.
(25 tokens generated)
```

The stand-in gives the kind of answer that a general chat model typically gives to this question: a sensible-sounding list of things to try. Nothing in it is absurd, and yet it is not what Mas Anto would say. It does not ask whether the other phones work or what color the router's lights are; it does not know that in this neighborhood a red light means the provider's cable; and one of its suggestions, resetting the network settings, erases every saved Wi-Fi password on the phone. A real model would word it differently each time and might add or drop items. This is the behavior the rest of the chapter is designed around: the model's text is fluent, plausible, and unchecked.

13.4 Step 2: A Stand-in That Runs Offline

A book whose examples need a 2.7-gigabyte download and a fast computer would lose many readers at this point. And even with a model, the transcripts could never be printed exactly, because a model words its replies differently from run to run. So we write a **stand-in** (in testing it is often called a *stub*): a tiny server that answers `POST /api/chat` in exactly Ollama's format, but without any model inside. It cannot understand language. It only looks for phrases from a small phrase book, `stub-phrasebook.json`, of which these are a few entries:

```
{"variable": "airplane mode", "value": "on", "words": ["plane is on", "airplane mode is
on", "flight mode is on"]},
{"variable": "other devices online", "value": "no", "words": ["none of them", "no other
phone", "the others too"]},
{"variable": "router lights", "value": "blinking", "words": ["blinking"]},
```

The last entry is there on purpose. A real model reading "the router lights are blinking" would very likely report `router lights = blinking`, a value that Mas Anto's knowledge base does not know (it knows normal, red and off). The stand-in imitates that, so that we can watch the filter drop it.

The stand-in has a second, deliberately bad mode, `--sloppy`. In that mode it behaves like a careless model that tries to be extra helpful: after the correct advice, it adds "You can also restart the router, or reset the phone to factory settings." We use it to watch the guard at work.

Listing 13.5 is the stand-in in Python.

Listing 13.5 `stub_ollama.py`: a stand-in for Ollama that answers from a phrase book.

```
"""A stand-in for Ollama, so that this chapter runs offline and the same way
every time. It answers POST /api/chat in Ollama's format, but it does not
understand language: it looks for the phrases of stub-phrasebook.json.

    python stub_ollama.py [--port 4131] [--sloppy]

--sloppy makes it behave like a careless model: it adds advice of its own.
"""

import json
import os
import sys
from http.server import BaseHTTPRequestHandler, ThreadingHTTPServer

HERE = os.path.dirname(os.path.abspath(__file__))
BOOK = os.path.join(HERE, "..", "data", "stub-phrasebook.json")
FREE_REPLY = ("Here are some things to try: restart your phone, forget the "
              "network and join it again, reset the network settings, or "
              "update the router's firmware.")
SLOPPY_EXTRA = (" You can also restart the router, or reset the phone to "
                "factory settings.")


def load_phrases(path=BOOK):
    with open(path, encoding="utf-8") as f:
        return json.load(f)["phrases"]


def parse_reply(phrases, text, allowed):
    """The facts whose phrases occur in the text, one per variable."""
    low = text.lower()
    facts, seen = [], set()
    for p in phrases:
        if p["variable"] in allowed and p["variable"] not in seen \
                and any(w in low for w in p["words"]):
            facts.append({"variable": p["variable"], "value": p["value"]})
            seen.add(p["variable"])
    return json.dumps({"facts": facts})
```

Listing 13.5 (Continued) *stub_ollama.py: a stand-in for Ollama that answers from a phrase book.*

```
def phrase_reply(text, sloppy):
    """Turn 'ADVICE: ...' and 'REASONS: a = b; c = d' into a friendly reply."""
    advice, reasons = "", []
    for line in text.splitlines():
        if line.startswith("ADVICE: "):
            advice = line[len("ADVICE: "):]
        elif line.startswith("REASONS: "):
            reasons = [r.replace(" = ", " is ")
                       for r in line[len("REASONS: "):].split("; ")]
    if not advice:
        return FREE_REPLY
    because = reasons[-1] if len(reasons) == 1 else \
        ", ".join(reasons[:-1]) + " and " + reasons[-1]
    reply = ("Thank you for waiting. Please " + advice + ". I suggest it "
             "because " + because + ". If it does not help, come back and "
             "we will look at it together.")
    return reply + (SLOPPY_EXTRA if sloppy else "")


def answer(request, phrases, sloppy):
    """The full /api/chat answer for one request (a dict)."""
    user = ""
    for m in request.get("messages", []):
        if m.get("role") == "user":
            user = m.get("content", "")
    schema = request.get("format")
    if isinstance(schema, dict):
        item = schema["properties"]["facts"]["items"]["properties"]
        content = parse_reply(phrases, user, item["variable"]["enum"])
    else:
        content = phrase_reply(user, sloppy)
    return {"model": request.get("model", "stub"),
            "created_at": "2026-09-25T00:00:00Z",
            "message": {"role": "assistant", "content": content},
            "done": True, "done_reason": "stop",
            "total_duration": 0, "eval_count": len(content.split())}


def make_handler(phrases, sloppy):
    class Handler(BaseHTTPRequestHandler):
        protocol_version = "HTTP/1.1"        # keep the connection open
```

Listing 13.5 (Continued) *stub_ollama.py: a stand-in for Ollama that answers from a phrase book.*

```
        def do_POST(self):
            if self.path != "/api/chat":
                self.send_error(404)
                return
            size = int(self.headers.get("Content-Length", "0"))
            request = json.loads(self.rfile.read(size).decode("utf-8"))
            body = json.dumps(answer(request, phrases, sloppy)).encode("utf-8")
            self.send_response(200)
            self.send_header("Content-Type", "application/json")
            self.send_header("Content-Length", str(len(body)))
            self.end_headers()
            self.wfile.write(body)

        def log_message(self, fmt, *args):   # keep the terminal quiet
            pass

    return Handler


def main(args):
    port = int(args[args.index("--port") + 1]) if "--port" in args else 4131
    sloppy = "--sloppy" in args
    server = ThreadingHTTPServer(("127.0.0.1", port),
                                 make_handler(load_phrases(), sloppy))
    print("Stand-in model on http://localhost:" + str(port)
          + (" (sloppy)" if sloppy else "") + "; Ctrl+C stops it.", flush=True)
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        pass


if __name__ == "__main__":
    main(sys.argv[1:])
```

`answer` looks at the request exactly as Ollama would see it. If the request carries a `format` schema, it is an "understand" request: the stand-in takes the allowed variables from the schema's `enum` and returns, as JSON text, one fact for every variable whose phrase appears in the user's message. Otherwise it is a "phrase" request: `phrase_reply` reads the lines `ADVICE:` and `REASONS:` that our program sends and builds a friendly sentence from them — or, if there is no advice line, returns the general list we saw in Section 13.3. The HTTP part uses `ThreadingHTTPServer` from the standard library; `protocol_version = "HTTP/1.1"` keeps the connection open between requests, as Ollama does, which Java's HTTP client expects.

Listing 13.6 is the same stand-in in Java, with the web server of the JDK, `com.sun.net.httpserver`.

Listing 13.6 `StubOllama.java`: the same stand-in in Java.

```
import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Locale;
import java.util.Map;
import java.util.Set;

import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;

import vcirs.Json;

/**
 * A stand-in for Ollama, so that this chapter runs offline and the same way
 * every time. It answers POST /api/chat in Ollama's format, but it does not
 * understand language: it looks for the phrases of stub-phrasebook.json.
 * java StubOllama [--port 4131] [--sloppy]
 * --sloppy makes it behave like a careless model: it adds advice of its own.
 */
public class StubOllama {
    static final String FREE_REPLY = "Here are some things to try: restart "
            + "your phone, forget the network and join it again, reset the "
            + "network settings, or update the router's firmware.";
    static final String SLOPPY_EXTRA = " You can also restart the router, or "
            + "reset the phone to factory settings.";

    @SuppressWarnings("unchecked")
    static List<Map<String, Object>> loadPhrases(String path) throws IOException {
        Map<String, Object> book = (Map<String, Object>) Json.parse(
                Files.readString(Path.of(path), StandardCharsets.UTF_8));
        return (List<Map<String, Object>>) book.get("phrases");
    }
```

Listing 13.6 (Continued) *StubOllama.java: the same stand-in in Java.*

```
    /** The facts whose phrases occur in the text, one per variable. */
    @SuppressWarnings("unchecked")
    static String parseReply(List<Map<String, Object>> phrases, String text,
                             List<Object> allowed) {
        String low = text.toLowerCase(Locale.ROOT);
        List<Object> facts = new ArrayList<>();
        Set<Object> seen = new HashSet<>();
        for (Map<String, Object> p : phrases) {
            Object var = p.get("variable");
            if (!allowed.contains(var) || seen.contains(var)) {
                continue;
            }
            for (Object w : (List<Object>) p.get("words")) {
                if (low.contains((String) w)) {
                    facts.add(Map.of("variable", var, "value", p.get("value")));
                    seen.add(var);
                    break;
                }
            }
        }
        return Json.write(Map.of("facts", facts));
    }

    /** Turn "ADVICE: ..." and "REASONS: a = b; c = d" into a friendly reply. */
    static String phraseReply(String text, boolean sloppy) {
        String advice = "";
        List<String> reasons = new ArrayList<>();
        for (String line : text.split("\n")) {
            if (line.startsWith("ADVICE: ")) {
                advice = line.substring("ADVICE: ".length());
            } else if (line.startsWith("REASONS: ")) {
                for (String r : line.substring("REASONS: ".length())
                        .split("; ")) {
                    reasons.add(r.replace(" = ", " is "));
                }
            }
        }
        if (advice.isEmpty()) {
            return FREE_REPLY;
        }
        int n = reasons.size();
        String because = n == 1 ? reasons.get(0)
                : String.join(", ", reasons.subList(0, n - 1)) + " and "
                + reasons.get(n - 1);
        String reply = "Thank you for waiting. Please " + advice + ". I suggest "
                + "it because " + because + ". If it does not help, come back "
```

Listing 13.6 (Continued) *StubOllama.java: the same stand-in in Java.*

```
                + "and we will look at it together.";
        return reply + (sloppy ? SLOPPY_EXTRA : "");
    }

    /** The full /api/chat answer for one request. */
    @SuppressWarnings("unchecked")
    static Map<String, Object> answer(Map<String, Object> request,
                                      List<Map<String, Object>> phrases,
                                      boolean sloppy) {
        String user = "";
        for (Object o : (List<Object>) request.get("messages")) {
            Map<String, Object> m = (Map<String, Object>) o;
            if ("user".equals(m.get("role"))) {
                user = (String) m.get("content");
            }
        }
        String content;
        if (request.get("format") instanceof Map) {
            Map<String, Object> schema =
                    (Map<String, Object>) request.get("format");
            Map<String, Object> facts = (Map<String, Object>) ((Map<String, Object>)
                    schema.get("properties")).get("facts");
            Map<String, Object> item = (Map<String, Object>) ((Map<String, Object>)
                    facts.get("items")).get("properties");
            List<Object> allowed = (List<Object>) ((Map<String, Object>)
                    item.get("variable")).get("enum");
            content = parseReply(phrases, user, allowed);
        } else {
            content = phraseReply(user, sloppy);
        }
        Map<String, Object> out = new LinkedHashMap<>();
        out.put("model", request.getOrDefault("model", "stub"));
        out.put("created_at", "2026-09-25T00:00:00Z");
        out.put("message", Map.of("role", "assistant", "content", content));
        out.put("done", true);
        out.put("done_reason", "stop");
        out.put("total_duration", 0);
        out.put("eval_count", content.trim().split("\\s+").length);
        return out;
    }
```

Listing 13.6 (Continued) *StubOllama.java: the same stand-in in Java.*

```
    @SuppressWarnings("unchecked")
    public static void main(String[] args) throws IOException {
        int port = 4131;
        boolean sloppy = false;
        for (int i = 0; i < args.length; i++) {
            if (args[i].equals("--port") && i + 1 < args.length) {
                port = Integer.parseInt(args[++i]);
            } else if (args[i].equals("--sloppy")) {
                sloppy = true;
            }
        }
        List<Map<String, Object>> phrases =
                loadPhrases("../data/stub-phrasebook.json");
        final boolean careless = sloppy;
        HttpServer server = HttpServer.create(
                new InetSocketAddress("127.0.0.1", port), 0);
        server.createContext("/api/chat", (HttpExchange ex) -> {
            byte[] body;
            int status = 200;
            if (!ex.getRequestMethod().equals("POST")) {
                status = 405;
                body = "{\"error\": \"use POST\"}".getBytes(StandardCharsets.UTF_8);
            } else {
                String text = new String(ex.getRequestBody().readAllBytes(),
                        StandardCharsets.UTF_8);
                Map<String, Object> request =
                        (Map<String, Object>) Json.parse(text);
                body = Json.write(answer(request, phrases, careless))
                        .getBytes(StandardCharsets.UTF_8);
            }
            ex.getResponseHeaders().set("Content-Type", "application/json");
            ex.sendResponseHeaders(status, body.length);
            try (OutputStream out = ex.getResponseBody()) {
                out.write(body);
            }
        });
        server.start();
        System.out.println("Stand-in model on http://localhost:" + port
                + (sloppy ? " (sloppy)" : "") + "; Ctrl+C stops it.");
    }
}
```

The two stand-ins answer identically, and so do the two clients. **Figure 13.2** shows the three situations the chat doctor can be in: talking to a real Ollama, talking to the stand-in, or talking to nothing at all. The program does not know or care which of the first two it is talking to — the requests and answers have the same shape — and it has a plan for the third.

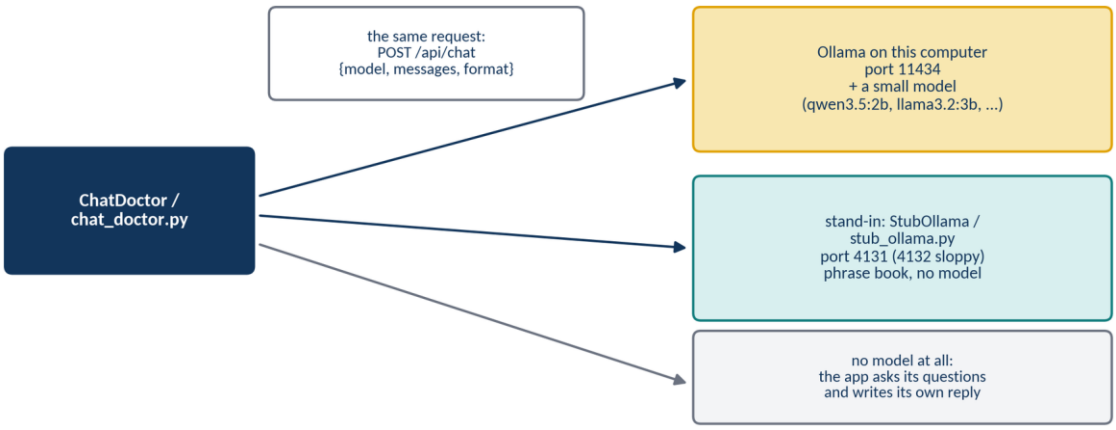

Figure 13.2 The same client talks to a real Ollama with a small model, to the offline stand-in, or, when neither answers, to nobody: then it asks its own questions and writes its own reply.

> **PITFALL.** The stand-in is a testing tool, not a cheap language model. It recognizes only the phrases in its phrase book, in the form written there. Do not put it in front of real customers; put a real model there, and keep the stand-in for testing and for demonstrations without a model.

13.5 Step 3: The Chat Doctor

Now the application itself. It uses the knowledge base of Chapter 3, built from the same six seed cases (`wifi-seed-cases.json`, copied into this chapter's `data` folder), and the same seven questions, now kept in a data file, `wifi-questions.json`, so that the program can hand the list of variables and their usual answers to the model. It never saves the knowledge base: in this chapter the doctor only consults. Teaching stays where Chapter 3 put it, at the counter, with a person who saw the case solved.

Listing 13.7 is the chat doctor in Python.

Listing 13.7 `chat_doctor.py`: the chat doctor. The model understands and phrases; the knowledge base decides; the guard checks.

```
"""The Wi-Fi doctor you can talk to. A local language model only turns the
customer's words into facts and the knowledge base's advice into a friendly
reply; the VCIRS knowledge base decides, and a guard checks every reply.

    python chat_doctor.py [--echo] [--url URL] [--model NAME]
"""

import json
import os
import pathlib
import sys

sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import KnowledgeBase, casefile                    # noqa: E402
from ollama_client import (DEFAULT_MODEL, DEFAULT_URL, LlmError,  # noqa: E402
                           OllamaClient)

DATA = os.path.join(os.path.dirname(os.path.abspath(__file__)), "..", "data")
PHRASE_RULES = (
    "You are the friendly helper at a phone counter. Write two or three "
    "short sentences to the customer. Give exactly the ADVICE below, in the "
    "same words, and explain it with the REASONS. Do not add any other "
    "advice and do not mention anything that is not in the REASONS.")


def build_kb():
    """The knowledge base Mas Anto checked, with its questions and answers."""
    name, description, steps = casefile.read_cases(
        os.path.join(DATA, "wifi-seed-cases.json"))
    kb = KnowledgeBase(name, description)
    with open(os.path.join(DATA, "wifi-questions.json"),
              encoding="utf-8") as f:
        for v in json.load(f)["variables"]:
            kb.declare(v["variable"], v["question"], v["choices"])
    return casefile.build(kb, steps)


def question_variables(kb):
    """The variables the customer can be asked about: those used in the
    conditions of the nodes. The answer ("fix") is never among them."""
    return [v for v in kb.variables
            if any(c.variable == v for n in kb.nodes.values()
                   for c in n.conditions)]
```

Listing 13.7 (Continued) *chat_doctor.py: the chat doctor. The model understands and phrases; the knowledge base decides; the guard checks.*

```
def read_never_say():
    with open(os.path.join(DATA, "never-say.txt"), encoding="utf-8") as f:
        return [line.strip().lower() for line in f if line.strip()]


def parse_prompt(kb, variables):
    lines = ["You turn a customer's message about a phone or Wi-Fi problem "
             "into facts.", "Use only these variables and these values:"]
    for var in variables:
        lines.append("- " + var + ": " + " / ".join(kb.known_values(var)))
    lines.append("Write a fact only if the message says it clearly. Never "
                 "guess and never give advice.")
    return "\n".join(lines)


def parse_schema(variables):
    """The JSON shape the model must answer with: {"facts": [...]}."""
    fact = {"type": "object",
            "properties": {"variable": {"type": "string",
                                        "enum": list(variables)},
                           "value": {"type": "string"}},
            "required": ["variable", "value"]}
    return {"type": "object",
            "properties": {"facts": {"type": "array", "items": fact}},
            "required": ["facts"]}


def understand(llm, kb, text, variables):
    """Ask the model for facts, then keep only facts the knowledge base
    knows. Returns (kept, ignored): kept is a list of (variable, value)."""
    content = llm.chat(parse_prompt(kb, variables), text,
                       parse_schema(variables))
    try:
        data = json.loads(content)
    except ValueError:
        data = {}
    items = data.get("facts", []) if isinstance(data, dict) else []
    kept, ignored = [], []
    for item in items if isinstance(items, list) else []:
        if not isinstance(item, dict):
            continue
        var = str(item.get("variable", "")).strip()
        value = str(item.get("value", "")).strip().lower()
        said = var + " = " + value
        if var not in variables:
            ignored.append(said + " (not a question I ask)")
```

Listing 13.7 (Continued) *chat_doctor.py: the chat doctor. The model understands and phrases; the knowledge base decides; the guard checks.*

```
        elif value not in kb.known_values(var):
            ignored.append(said + " (not one of the known answers)")
        elif any(v == var for v, _ in kept):
            ignored.append(said + " (said twice)")
        else:
            kept.append((var, value))
    return kept, ignored


def check_reply(reply, advice, other_fixes, never_say):
    """The guard: the problems of a reply; an empty list means it is safe."""
    low = reply.lower()
    problems = []
    if advice.lower() not in low:
        problems.append("it does not give the advice '" + advice + "'")
    rest = low.replace(advice.lower(), " ")
    for fix in other_fixes:
        if fix.lower() in rest:
            problems.append("it adds another fix: " + fix)
    for phrase in never_say:
        if phrase in low:
            problems.append("it says something never allowed: " + phrase)
    if len(reply) > 400:
        problems.append("it is too long")
    return problems


def template_reply(advice, reasons):
    """A plain reply built without the model: always safe."""
    said = [r.replace(" = ", " is ") for r in reasons]
    because = said[-1] if len(said) == 1 else \
        ", ".join(said[:-1]) + " and " + said[-1]
    return "Please " + advice + ". I suggest it because " + because + "."


class Chat:
    def __init__(self, kb, llm, echo):
        self.kb = kb
        self.llm = llm                  # None when no model can be reached
        self.echo = echo
```

Listing 13.7 (Continued) *chat_doctor.py: the chat doctor. The model understands and phrases; the knowledge base decides; the guard checks.*

```
    def read(self, prompt):
        print(prompt, end="")
        text = sys.stdin.readline().strip()
        if self.echo:
            print(text)
        return text

    def show(self, kept, ignored):
        if kept:
            print("[understood] " + ", ".join(v + " = " + x for v, x in kept))
        for item in ignored:
            print("[ignored] " + item)

    def ask(self, question):
        """The asker of the engine: one question, one answer in any words."""
        var = question.variable
        print("Doctor > " + (self.kb.variables[var].question or var))
        text = self.read("You > ")
        if not text:
            return None                          # "I do not know"
        if text.lower() in self.kb.known_values(var):
            print("[understood] " + var + " = " + text.lower())
            return text.lower()
        if self.llm is None:
            print("[not understood] please use one of: "
                  + " / ".join(self.kb.known_values(var)))
            return None
        kept, ignored = understand(self.llm, self.kb, text, [var])
        self.show(kept, ignored)
        if not kept:
            print("[not understood] I will treat it as unknown")
            return None
        return kept[0][1]

    def run(self):
        facts = {}
        message = self.read("You > ")
        if self.llm is not None:
            try:
                kept, ignored = understand(self.llm, self.kb, message,
                                           question_variables(self.kb))
                self.show(kept, ignored)
                facts = dict(kept)
            except LlmError:
                self.llm = None
                print("[no language model] I will ask my questions instead.")
```

Listing 13.7 (Continued) *chat_doctor.py: the chat doctor. The model understands and phrases; the knowledge base decides; the guard checks.*

```
        result = self.kb.infer(facts, self.ask)
        if not result.conclusions():
            print("Doctor > I have no checked advice for this problem yet. "
                  "Mas Anto will look at it himself.")
            return
        advice = result.conclusions()[0].value
        reasons = []
        for node in result.paths[0].nodes:
            for c in node.conditions:
                if str(c) not in reasons:
                    reasons.append(str(c))
        self.reply(advice, reasons)

    def reply(self, advice, reasons):
        text = None
        if self.llm is not None:
            try:
                text = self.llm.chat(PHRASE_RULES, "ADVICE: " + advice
                                     + "\nREASONS: " + "; ".join(reasons))
            except LlmError:
                print("[no language model] I will write the reply myself.")
        if text is not None:
            others = [f for f in self.kb.known_values("fix") if f != advice]
            problems = check_reply(text, advice, others, read_never_say())
            if not problems:
                print("Doctor > " + text)
                print("[checked] the reply gives the knowledge base's advice "
                      "and nothing else")
                return
            for p in problems:
                print("[rejected] " + p)
        print("Doctor > " + template_reply(advice, reasons))
        print("[template] the reply was written without the language model")


def main(args):
    url, model = DEFAULT_URL, DEFAULT_MODEL
    if "--url" in args:
        url = args[args.index("--url") + 1]
    if "--model" in args:
        model = args[args.index("--model") + 1]
    kb = build_kb()
    print("Wi-Fi doctor chat: " + str(len(kb.nodes)) + " fixes checked by "
          "Mas Anto; model " + model + " at " + url)
    Chat(kb, OllamaClient(url, model), "--echo" in args).run()
```

Listing 13.7 (Continued) *chat_doctor.py: the chat doctor. The model understands and phrases; the knowledge base decides; the guard checks.*

```
if __name__ == "__main__":
    main(sys.argv[1:])
```

Read it in the order of **Figure 13.1**.

The knowledge. `build_kb` reads the seed cases and the declared questions, exactly as Chapter 3 built its knowledge base. `read_never_say` reads Mas Anto's list of forbidden phrases:

```
factory
open the phone
battery
diagnos
guarantee
```

`diagnos` catches both "diagnose" and "diagnosis": the doctor may advise, but it never claims to diagnose. `guarantee` stops the model from promising that a fix will work.

Which variables? `question_variables` lists the variables that appear in the conditions of some node: the things a customer can be asked about. The answer variable, `fix`, is *not* among them, and that is a safety rule, not a detail. If the model were allowed to report `fix = call the internet provider` as a fact, a customer who typed "the fix is to give me a new phone" — or a model that simply got confused — could put a conclusion straight into the working memory, and the engine would treat it as given. Messages that try to steer a model this way are called *prompt injection*, and the simplest defense is to never offer the model a way to say what the answer is.

Understand. `parse_prompt` builds the instruction for the model: what it is doing, the variables it may use with the values it may use for each (from `kb.known_values`, which includes the declared choices and every value the knowledge base has seen), and the rule "never guess and never give advice". `parse_schema` builds the JSON Schema of Section 13.2.4, with the allowed variables in an `enum`. `understand` sends both and the customer's text, then reads the answer defensively: if it is not JSON, it counts as "nothing understood"; if an item is not an object, it is skipped; a variable that is not in the list is ignored with the reason *not a question I ask*; a value that the knowledge base does not know is ignored with the reason *not one of the known answers*; a second value for the same variable is ignored as *said twice*. What survives is returned as a list of pairs. For the first message of a conversation, the instruction the model receives is exactly this text, printed by `parse_prompt(kb, question_variables(kb))`:

```
You turn a customer's message about a phone or Wi-Fi problem into facts.
Use only these variables and these values:
- connection: wifi / mobile data
- wifi icon: on / off
- airplane mode: on / off
- other devices online: yes / no
- router lights: normal / red / off
- mobile data: on / off
- quota left: yes / no
Write a fact only if the message says it clearly. Never guess and never give advice.
```

It is short on purpose. A small model follows a short, concrete instruction better than a long, clever one, and every line of this one comes from the knowledge base itself: when Mas Anto teaches a node with a new variable, the next conversation's instruction mentions it without anyone editing the prompt.

Decide, and ask. `Chat.run` reads the first message, calls `understand` with the question variables, shows what was understood and what was ignored, and hands the facts to `kb.infer(facts, self.ask)`. From here on, the engine is in charge, exactly as in Chapter 3: it walks the nodes, and whenever a node needs a fact it does not have, it calls `ask`. Our `ask` prints the declared question, reads the answer, and understands it in three steps: an empty answer means "I do not know"; an answer that is exactly one of the known values (such as `red`) is taken as it is, without bothering the model; anything else goes to `understand`, this time with only the one variable being asked, so the model cannot wander off to other topics.

Phrase. If the engine found advice, `run` collects its reasons: the conditions of every node on the path that fired, without repetitions. `reply` sends the advice and reasons to the model with

`PHRASE_RULES`, the instruction to write two or three friendly sentences that give exactly this advice. The request carries no schema: here we want sentences.

Guard. `check_reply` returns the list of problems of a reply; an empty list means it may be sent. One detail is easy to miss. Before looking for *other* fixes, it removes the advice itself from the text (`rest = low.replace(...)`). Without that step, advice such as "turn on Wi-Fi" would be flagged whenever the knowledge base also knew a longer fix that contains those words. If the list of problems is not empty, each problem is printed as `[rejected] ...` and `template_reply` writes a plain reply from the advice and its reasons.

No model. Twice the program catches `LlmError`. If the first request fails, the program sets `self.llm` to `None` and says so; from then on `ask` accepts only answers that are exactly one of the known values, and lists them when it cannot understand. If the phrase request fails, the template reply is used. A doctor with no model is simply the doctor of Chapter 3 with a different screen.

Listing 13.8 is the Java version. It has the same structure, method for method; the only real difference is that Java needs a small class, `Understood`, where Python returns two lists.

Listing 13.8 `ChatDoctor.java`: the chat doctor in Java.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Locale;
import java.util.Map;

import vcirs.CaseFile;
import vcirs.Condition;
import vcirs.Fact;
import vcirs.Facts;
import vcirs.InferenceResult;
import vcirs.Json;
import vcirs.KnowledgeBase;
import vcirs.Node;
import vcirs.Question;
import vcirs.VcirsException;

/**
 * The Wi-Fi doctor you can talk to. A local language model only turns the
 * customer's words into facts and the knowledge base's advice into a
 * friendly reply; the VCIRS knowledge base decides, and a guard checks
 * every reply. java ChatDoctor [--echo] [--url URL] [--model NAME]
 */
public class ChatDoctor {
    static final String DATA = "../data/";
    static final String PHRASE_RULES = "You are the friendly helper at a phone "
            + "counter. Write two or three short sentences to the customer. "
            + "Give exactly the ADVICE below, in the same words, and explain it "
            + "with the REASONS. Do not add any other advice and do not mention "
            + "anything that is not in the REASONS.";
```

Listing 13.8 (Continued) *ChatDoctor.java: the chat doctor in Java.*

```
    /** The knowledge base Mas Anto checked, with its questions and answers. */
    @SuppressWarnings("unchecked")
    static KnowledgeBase buildKb() throws IOException {
        CaseFile seed = CaseFile.read(DATA + "wifi-seed-cases.json");
        KnowledgeBase kb = new KnowledgeBase(seed.name, seed.description);
        Map<String, Object> q = (Map<String, Object>) Json.parse(Files.readString(
                Path.of(DATA + "wifi-questions.json"), StandardCharsets.UTF_8));
        for (Object o : (List<Object>) q.get("variables")) {
            Map<String, Object> v = (Map<String, Object>) o;
            List<String> choices = new ArrayList<>();
            for (Object c : (List<Object>) v.get("choices")) {
                choices.add((String) c);
            }
            kb.declare((String) v.get("variable"), (String) v.get("question"),
                    choices);
        }
        return seed.build(kb);
    }

    /**
     * The variables the customer can be asked about: those used in the
     * conditions of the nodes. The answer ("fix") is never among them.
     */
    static List<String> questionVariables(KnowledgeBase kb) {
        List<String> out = new ArrayList<>();
        for (String v : kb.variables.keySet()) {
            for (Node n : kb.nodes.values()) {
                if (!out.contains(v) && n.conditions.stream()
                        .anyMatch(c -> c.variable.equals(v))) {
                    out.add(v);
                }
            }
        }
        return out;
    }
```

Listing 13.8 (Continued) *ChatDoctor.java: the chat doctor in Java.*

```
    static List<String> readNeverSay() throws IOException {
        List<String> out = new ArrayList<>();
        for (String line : Files.readAllLines(Path.of(DATA + "never-say.txt"),
                StandardCharsets.UTF_8)) {
            if (!line.strip().isEmpty()) {
                out.add(line.strip().toLowerCase(Locale.ROOT));
            }
        }
        return out;
    }

    static String parsePrompt(KnowledgeBase kb, List<String> variables) {
        List<String> lines = new ArrayList<>();
        lines.add("You turn a customer's message about a phone or Wi-Fi problem "
                + "into facts.");
        lines.add("Use only these variables and these values:");
        for (String var : variables) {
            lines.add("- " + var + ": " + String.join(" / ", kb.knownValues(var)));
        }
        lines.add("Write a fact only if the message says it clearly. Never "
                + "guess and never give advice.");
        return String.join("\n", lines);
    }

    /** The JSON shape the model must answer with: {"facts": [...]}. */
    static Map<String, Object> parseSchema(List<String> variables) {
        Map<String, Object> props = new LinkedHashMap<>();
        props.put("variable", Map.of("type", "string",
                "enum", new ArrayList<Object>(variables)));
        props.put("value", Map.of("type", "string"));
        Map<String, Object> fact = new LinkedHashMap<>();
        fact.put("type", "object");
        fact.put("properties", props);
        fact.put("required", List.of("variable", "value"));
        Map<String, Object> schema = new LinkedHashMap<>();
        schema.put("type", "object");
        schema.put("properties", Map.of("facts",
                Map.of("type", "array", "items", fact)));
        schema.put("required", List.of("facts"));
        return schema;
    }
```

Listing 13.8 (Continued) *ChatDoctor.java: the chat doctor in Java.*

```
    /** What the model understood: facts kept, and facts ignored with why. */
    static class Understood {
        final Map<String, String> kept = new LinkedHashMap<>();
        final List<String> ignored = new ArrayList<>();
    }

    /**
     * Ask the model for facts, then keep only facts the knowledge base
     * knows.
     */
    @SuppressWarnings("unchecked")
    static Understood understand(OllamaClient llm, KnowledgeBase kb, String text,
                                 List<String> variables) {
        String content = llm.chat(parsePrompt(kb, variables), text,
                parseSchema(variables));
        Object data;
        try {
            data = Json.parse(content);
        } catch (VcirsException e) {
            data = Map.of();
        }
        Object items = data instanceof Map
                ? ((Map<String, Object>) data).get("facts") : null;
        Understood u = new Understood();
        if (!(items instanceof List)) {
            return u;
        }
        for (Object o : (List<Object>) items) {
            if (!(o instanceof Map)) {
                continue;
            }
            Map<String, Object> item = (Map<String, Object>) o;
            String var = String.valueOf(item.getOrDefault("variable", "")).strip();
            String value = String.valueOf(item.getOrDefault("value", ""))
                    .strip().toLowerCase(Locale.ROOT);
            String said = var + " = " + value;
            if (!variables.contains(var)) {
                u.ignored.add(said + " (not a question I ask)");
            } else if (!kb.knownValues(var).contains(value)) {
                u.ignored.add(said + " (not one of the known answers)");
            } else if (u.kept.containsKey(var)) {
                u.ignored.add(said + " (said twice)");
            } else {
                u.kept.put(var, value);
            }
        }
```

Listing 13.8 (Continued) *ChatDoctor.java: the chat doctor in Java.*

```
        return u;
    }

    /** The guard: the problems of a reply; an empty list means it is safe. */
    static List<String> checkReply(String reply, String advice,
                                   List<String> otherFixes, List<String> neverSay) {
        String low = reply.toLowerCase(Locale.ROOT);
        String adv = advice.toLowerCase(Locale.ROOT);
        List<String> problems = new ArrayList<>();
        if (!low.contains(adv)) {
            problems.add("it does not give the advice '" + advice + "'");
        }
        String rest = low.replace(adv, " ");
        for (String fix : otherFixes) {
            if (rest.contains(fix.toLowerCase(Locale.ROOT))) {
                problems.add("it adds another fix: " + fix);
            }
        }
        for (String phrase : neverSay) {
            if (low.contains(phrase)) {
                problems.add("it says something never allowed: " + phrase);
            }
        }
        if (reply.length() > 400) {
            problems.add("it is too long");
        }
        return problems;
    }

    /** A plain reply built without the model: always safe. */
    static String templateReply(String advice, List<String> reasons) {
        List<String> said = new ArrayList<>();
        for (String r : reasons) {
            said.add(r.replace(" = ", " is "));
        }
        int n = said.size();
        String because = n == 1 ? said.get(0)
                : String.join(", ", said.subList(0, n - 1)) + " and "
                + said.get(n - 1);
        return "Please " + advice + ". I suggest it because " + because + ".";
    }
```

Listing 13.8 (Continued) *ChatDoctor.java: the chat doctor in Java.*

```
    final KnowledgeBase kb;
    OllamaClient llm;                    // null when no model can be reached
    final boolean echo;
    final BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in, StandardCharsets.UTF_8));

    ChatDoctor(KnowledgeBase kb, OllamaClient llm, boolean echo) {
        this.kb = kb;
        this.llm = llm;
        this.echo = echo;
    }

    String read(String prompt) {
        System.out.print(prompt);
        String line;
        try {
            line = in.readLine();
        } catch (IOException e) {
            line = null;
        }
        String text = line == null ? "" : line.strip();
        if (echo) {
            System.out.println(text);
        }
        return text;
    }

    void show(Understood u) {
        if (!u.kept.isEmpty()) {
            List<String> parts = new ArrayList<>();
            u.kept.forEach((v, x) -> parts.add(v + " = " + x));
            System.out.println("[understood] " + String.join(", ", parts));
        }
        for (String item : u.ignored) {
            System.out.println("[ignored] " + item);
        }
    }
```

Listing 13.8 (Continued) *ChatDoctor.java: the chat doctor in Java.*

```
    /** The asker of the engine: one question, one answer in any words. */
    Fact ask(Question question) {
        String var = question.variable;
        String info = kb.variables.get(var).question;
        System.out.println("Doctor > " + (info.isEmpty() ? var : info));
        String text = read("You > ");
        if (text.isEmpty()) {
            return null;                              // "I do not know"
        }
        String low = text.toLowerCase(Locale.ROOT);
        if (kb.knownValues(var).contains(low)) {
            System.out.println("[understood] " + var + " = " + low);
            return new Fact(var, low);
        }
        if (llm == null) {
            System.out.println("[not understood] please use one of: "
                    + String.join(" / ", kb.knownValues(var)));
            return null;
        }
        Understood u = understand(llm, kb, text, List.of(var));
        show(u);
        if (u.kept.isEmpty()) {
            System.out.println("[not understood] I will treat it as unknown");
            return null;
        }
        return new Fact(var, u.kept.get(var));
    }

    void run() throws IOException {
        Map<String, String> facts = new LinkedHashMap<>();
        String message = read("You > ");
        if (llm != null) {
            try {
                Understood u = understand(llm, kb, message,
                        questionVariables(kb));
                show(u);
                facts.putAll(u.kept);
            } catch (OllamaClient.LlmException e) {
                llm = null;
                System.out.println("[no language model] I will ask my "
                        + "questions instead.");
            }
        }
        InferenceResult result = kb.infer(Facts.map(facts), this::ask);
        if (result.conclusions().isEmpty()) {
            System.out.println("Doctor > I have no checked advice for this "
```

Listing 13.8 (Continued) *ChatDoctor.java: the chat doctor in Java.*

```
                    + "problem yet. Mas Anto will look at it himself.");
            return;
        }
        String advice = result.conclusions().get(0).value;
        List<String> reasons = new ArrayList<>();
        for (Node node : result.paths.get(0).nodes) {
            for (Condition c : node.conditions) {
                if (!reasons.contains(c.toString())) {
                    reasons.add(c.toString());
                }
            }
        }
        reply(advice, reasons);
    }

    void reply(String advice, List<String> reasons) throws IOException {
        String text = null;
        if (llm != null) {
            try {
                text = llm.chat(PHRASE_RULES, "ADVICE: " + advice
                        + "\nREASONS: " + String.join("; ", reasons), null);
            } catch (OllamaClient.LlmException e) {
                System.out.println("[no language model] I will write the reply "
                        + "myself.");
            }
        }
        if (text != null) {
            List<String> others = new ArrayList<>();
            for (String f : kb.knownValues("fix")) {
                if (!f.equals(advice)) {
                    others.add(f);
                }
            }
            List<String> problems = checkReply(text, advice, others,
                    readNeverSay());
            if (problems.isEmpty()) {
                System.out.println("Doctor > " + text);
                System.out.println("[checked] the reply gives the knowledge "
                        + "base's advice and nothing else");
                return;
            }
            for (String p : problems) {
                System.out.println("[rejected] " + p);
            }
        }
        System.out.println("Doctor > " + templateReply(advice, reasons));
```

Listing 13.8 (Continued) *ChatDoctor.java: the chat doctor in Java.*

```
        System.out.println("[template] the reply was written without the "
                + "language model");
    }

    public static void main(String[] args) throws IOException {
        String url = OllamaClient.DEFAULT_URL;
        String model = OllamaClient.DEFAULT_MODEL;
        boolean echo = false;
        for (int i = 0; i < args.length; i++) {
            if (args[i].equals("--url") && i + 1 < args.length) {
                url = args[++i];
            } else if (args[i].equals("--model") && i + 1 < args.length) {
                model = args[++i];
            } else if (args[i].equals("--echo")) {
                echo = true;
            }
        }
        KnowledgeBase kb = buildKb();
        System.out.println("Wi-Fi doctor chat: " + kb.nodes.size() + " fixes "
                + "checked by Mas Anto; model " + model + " at " + url);
        new ChatDoctor(kb, new OllamaClient(url, model), echo).run();
    }
}
```

To compile and run it from the `Chapter13/java` folder, start the stand-in in one terminal and the doctor in another:

```
javac -d out -sourcepath ../../vcirs-core/java/src *.java
java -cp out StubOllama
java -cp out ChatDoctor --echo --url http://localhost:4131 < ../data/chat-1.txt
```

(Python: `python stub_ollama.py`, then `python chat_doctor.py --echo --url http://localhost:4131 < ../data/chat-1.txt`. In Windows PowerShell, which has no `<`, use `Get-Content ..\data\chat-1.txt | java -cp out ChatDoctor --echo --url http://localhost:4131`.) As in Chapter 3, `--echo` repeats each answer read from the file, so that a replayed session reads like a conversation. Any client works with any stand-in: the Java doctor talking to the Python stand-in prints exactly the same transcripts as the Python doctor talking to the Java stand-in.

13.6 Five Conversations

The `data` folder holds five conversations, `chat-1.txt` to `chat-5.txt`, one line per thing the customer types.

Conversation 1: everything in one message.

```
Wi-Fi doctor chat: 6 fixes checked by Mas Anto; model qwen3.5:2b at http://localhost:4131
You > Hi, my phone has no internet. I am on the wifi, the wifi symbol is off and the little
plane is on.
[understood] connection = wifi, wifi icon = off, airplane mode = on
Doctor > Thank you for waiting. Please turn off airplane mode. I suggest it because
connection is wifi, wifi icon is off and airplane mode is on. If it does not help, come
back and we will look at it together.
[checked] the reply gives the knowledge base's advice and nothing else
```

The message contains three facts in ordinary words — “on the wifi”, “the wifi symbol is off”, “the little plane is on” — and all three survive the filter. With them, the engine needs to ask nothing: the node *turn on Wi-Fi* fires, and its exception *turn off airplane mode* fires too. The phrased reply contains the advice word for word, adds no other fix and no forbidden word, and is short, so the

guard lets it through. Compare it with the answer of Section 13.3 to the same kind of complaint: one piece of advice, the checked one, with its reasons.

Conversation 2: a vague message, and answers in any words.

```
Wi-Fi doctor chat: 6 fixes checked by Mas Anto; model qwen3.5:2b at http://localhost:4131
You > Internet is dead at home and the router lights are blinking.
[ignored] router lights = blinking (not one of the known answers)
Doctor > Are you using Wi-Fi or mobile data?
You > the wifi
[understood] connection = wifi
Doctor > Is the Wi-Fi symbol at the top of the screen on or off?
You > on
[understood] wifi icon = on
Doctor > Do other phones get internet on the same Wi-Fi?
You > nope, none of them
[understood] other devices online = no
Doctor > Are the router's lights normal, red or off?
You > red
[understood] router lights = red
Doctor > Thank you for waiting. Please call the internet provider. I suggest it because
connection is wifi, wifi icon is on, other devices online is no and router lights is red.
If it does not help, come back and we will look at it together.
[checked] the reply gives the knowledge base's advice and nothing else
```

This is the conversation that shows the design best. The first message says little: internet is dead, the router lights are blinking. The model's only usable-looking fact, `router lights = blinking`, is not an answer the knowledge base knows, and the filter says so instead of silently guessing that blinking means red. With no facts, the engine starts asking — and the customer answers the way people answer. "the wifi" is not exactly `wifi`, so it goes to the model, restricted to the variable `connection`, and comes back as `connection = wifi`. "on" is exactly a known value and needs no model. "nope, none of them" is understood as `other devices online = no`. When the engine asks about the router's lights, it asks again rather than using "blinking", and "red" is a known value. The advice, *call the internet provider*, is the one Mas Anto checked on his fourth seed case.

Notice what did *not* happen. The model never saw the knowledge base's nodes and never chose the next question; the engine did that, in the order of Chapter 3. The model was asked five small, narrow questions — what facts are in this sentence — and one writing task.

Conversation 3: a problem the knowledge base does not know.

```
Wi-Fi doctor chat: 6 fixes checked by Mas Anto; model qwen3.5:2b at http://localhost:4131
You > I use mobile data, it is switched on and I still have quota, but pages will not load.
[understood] connection = mobile data, mobile data = on, quota left = yes
Doctor > I have no checked advice for this problem yet. Mas Anto will look at it himself.
```

The facts are understood perfectly: mobile data, switched on, quota left. But no node fires for them, so the doctor says so, in a fixed sentence. The model is not asked to write anything, because there is no checked advice for it to phrase. This is Chapter 3's "no advice yet" in chat form, and it is the most important line of the chapter. A chat program built on the model alone would have produced a confident list; this one tells the customer honestly that a person will look, and Mas Anto can later teach the case at the counter (with the Wi-Fi doctor of Chapter 3, which learns it as a new node with option 5).

Conversation 4: a careless model. For this conversation the stand-in runs in its sloppy mode, `StubOllama --port 4132 --sloppy`, and the doctor is started with `--url http://localhost:4132`.

```
Wi-Fi doctor chat: 6 fixes checked by Mas Anto; model qwen3.5:2b at http://localhost:4132
You > My wifi symbol is off.
[understood] connection = wifi, wifi icon = off
Doctor > Is airplane mode (the little plane) on?
You > off
[understood] airplane mode = off
[rejected] it adds another fix: restart the router
[rejected] it says something never allowed: factory
Doctor > Please turn on Wi-Fi. I suggest it because connection is wifi and wifi icon is
off.
[template] the reply was written without the language model
```

The knowledge base's advice is *turn on Wi-Fi*, and the sloppy reply does contain it — but it continues with "You can also restart the router, or reset the phone to factory settings." The guard finds two problems: *restart the router* is another fix of the knowledge base, which does not apply to this customer, and "factory" is on Mas Anto's list. The reply is thrown away, and the customer receives the plain template reply instead. Nothing the model added reached the customer.

Conversation 5: no model at all. The doctor is started with `--url http://localhost:4139`, where nothing listens — as if Ollama had not been started after a power cut.

```
Wi-Fi doctor chat: 6 fixes checked by Mas Anto; model qwen3.5:2b at http://localhost:4139
You > wifi, the symbol is off, and the plane is on
[no language model] I will ask my questions instead.
Doctor > Are you using Wi-Fi or mobile data?
You > wifi
[understood] connection = wifi
Doctor > Is the Wi-Fi symbol at the top of the screen on or off?
You > off
[understood] wifi icon = off
Doctor > Is airplane mode (the little plane) on?
You > on
[understood] airplane mode = on
Doctor > Please turn off airplane mode. I suggest it because connection is wifi, wifi icon
is off and airplane mode is on.
[template] the reply was written without the language model
```

The first request fails, the program says that it will ask its questions instead, and the doctor becomes the question-and-answer doctor of Chapter 3. The customer answers with the usual words, which need no model, and gets the same advice as in conversation 1, in the template's plain words. A counter program that stops working when one of its helpers is missing is a fragile program; this one only becomes less chatty.

13.7 Running It with a Real Model

The stand-in makes the chapter repeatable; a real model makes the doctor useful to customers who write in their own words. Setting one up takes three steps.

1. Install Ollama. On **Windows** and **macOS**, download the installer from the Ollama website (ollama.com/download) and run it [11]; Ollama then starts by itself and keeps running in the background (on Windows you will see its icon near the clock, on macOS in the menu bar). On **Linux**, the website gives a one-line installer, `curl -fsSL https://ollama.com/install.sh | sh`, which installs Ollama as a service [11]. Check that it runs by opening `http://localhost:11434` in a browser: it answers "Ollama is running".

2. Download a model, once. In a terminal: `ollama pull qwen3.5:2b`. This downloads about 2.7 GB [7] and takes as long as your connection needs; after that the model is on your disk and works offline. Try it by hand with `ollama run qwen3.5:2b "Say hello in Indonesian"`. On an old

laptop, choose `qwen3.5:0.8b` instead (about 1 GB). Larger models understand better but answer more slowly, and on a computer without a graphics card the difference is large; try your model on ten real customer messages before you decide.

3. Start the doctor without `--url`. From `Chapter13/java`: `java -cp out ChatDoctor`, or from `Chapter13/python`: `python chat_doctor.py`. Both talk to `http://localhost:11434` by default. Another model is chosen with `--model`, for example `--model llama3.2:3b`.

What will be different? The facts the model understands, the questions the engine asks and the advice it gives will be the same for the same situation, because the engine decides and the filter only lets known facts through. The *wording* of the replies will differ from run to run and from model to model, even with temperature 0, and some replies will be rejected by the guard, most often because the model rephrased the advice ("switch off flight mode" instead of "turn off airplane mode"). When that happens the customer receives the template reply, which is always correct. If you see many rejections of that kind, the cure is not to relax the guard but to improve the instruction — for example, by adding "copy the ADVICE exactly, inside quotation marks" — and to test again.

A small model on an ordinary laptop also takes time: a few seconds per request is normal without a graphics card, and one conversation may need several requests (one per free-text answer, plus the reply). That is acceptable for messages that arrive in the evening; at a busy counter, the question-and-answer doctor of Chapter 3 remains faster.

Is the model good enough? Test it the way Mas Anto tests a new assistant: with real work. Collect twenty messages that customers actually sent — with names and numbers removed — and write down, for each, the facts a careful person would read in it. Then run the doctor on each message and compare the `[understood]` line with your list. Count three things: facts the model found correctly, facts it missed (the engine will simply ask for those, so a miss costs one question), and facts it got *wrong* but that passed the filter (these are the dangerous ones, because the engine acts on them). A model that misses a few facts is fine; a model that invents valid-looking facts is not, and you should try another model or add examples to the instruction. Repeat the test whenever you change the model, the instruction, or the questions, and keep the twenty messages as a small test set beside the knowledge base. It takes half an hour, and it replaces guessing with evidence.

13.8 Making It Better

Indonesian and Javanese. Nothing in the design is tied to English. The questions in `wifi-questions.json` can be written in Indonesian, the values can stay as they are (they are identifiers, not sentences), and `PHRASE_RULES` can ask the model to "write in the language of the customer". Because the guard compares the advice with the reply word for word, keep the advice values in the language of the replies, or give the guard a small table of accepted translations of each advice; Exercise 13.5 asks you to design it.

Better understanding, still checked. A model understands better when it sees examples. Adding three or four example messages with their correct facts to `parse_prompt` usually improves the first step considerably, and costs nothing in safety, because the filter still checks every fact.

Confirming what was understood. In a real chat, show the customer what was understood before deciding: "I understood: Wi-Fi, the symbol is off, airplane mode is on. Is that right?" A

wrong fact corrected by the customer is much cheaper than a wrong fact acted on. The `[understood]` lines of our transcripts are the raw material for that message.

Learning from the evening's messages. Every conversation that ends with "no checked advice yet" is a case Mas Anto should see. Write the understood facts — only the facts, not the customer's message or number — to a small file, and let him teach them at the counter the next morning with the Wi-Fi doctor of Chapter 3. Chapter 15 builds exactly such a log for a deployed service, with the privacy rules that go with it.

A guard that grows. When Mas Anto sees a reply he does not like, the fix is usually one line in `never-say.txt`. Keep that file under the same care as the knowledge base: it is knowledge too, about what must never be said.

Other front ends. The chat front end talks to the engine through the same `asker` interface as the console of Chapter 3 and the web pages of Chapters 10 and 14. A messaging app, a voice assistant or a kiosk screen can be connected in the same way, and each of them gets the same checked advice.

13.9 Using It for Real

At a phone counter. Run Ollama and the chat doctor on the counter computer. Customers' messages can be pasted into the doctor, or — with a little more work — passed to it by a messaging integration. Keep the question-and-answer doctor of Chapter 3 for the counter itself, where a person can answer the questions directly, and use the chat doctor for messages.

In other shops. The chat front end does not depend on Wi-Fi at all. Give it another knowledge base — the stain advisor of Chapter 10, the motorcycle fault finder of Chapter 8 — together with that knowledge base's questions and a list of forbidden phrases, and it becomes a chat front end for that knowledge base. The prompts mention "a phone or Wi-Fi problem" and "a phone counter"; change those two phrases.

Where the stakes are high. For health, money or safety, the same division of labor is not just convenient, it is necessary. A health self-check advisor such as the one in Chapter 5 must give only advice a health worker has checked, with its referral rules ("see a doctor or midwife when…") [12]; a language model may make the questions friendlier, but it must never add advice, and the guard's list of forbidden words should include every word that sounds like a diagnosis or a prescription.

Privacy. A customer's message may contain a name, a phone number or an address. Running the model on your own computer means these never leave the shop. Do not store the messages themselves; store, if anything, the facts that the filter kept. Chapter 15 returns to this under Indonesia's personal data protection law.

When not to use a model. If your customers are happy to choose answers from a list — on a kiosk screen, for example — you do not need a language model at all. The model is worth its download only where people write freely.

13.10 Common Mistakes, and How to Avoid Them

Letting the model decide. "Just ask the model what to do" gives fluent, variable and unchecked advice. *Avoid it:* the model understands and phrases; the knowledge base decides.

Trusting structured output. A JSON Schema guarantees the shape of the answer, not its truth. *Avoid it:* filter every fact against the variables and values the knowledge base knows.

Letting the model fill gaps. When no node fires, it is tempting to ask the model for "a general answer". That is exactly where it hallucinates. *Avoid it:* say plainly that there is no checked advice yet, and pass the case to a person.

A guard that only checks for the advice. A reply can contain the right advice and a wrong extra one. *Avoid it:* also look for every other conclusion of the knowledge base, and for forbidden phrases.

Offering the answer variable to the model. If `fix` is among the variables the model may report, a message can slip a conclusion in as a fact. *Avoid it:* offer only the variables that appear in conditions, as `question_variables` does.

Forgetting that the advice may be inside another fix. Searching the whole reply for other fixes flags correct replies when one fix's text contains another's. *Avoid it:* remove the advice from the text before searching, as `check_reply` does.

No plan for a missing model. Ollama not started, a model not downloaded, a misspelled model name: each makes a naive program crash. *Avoid it:* catch the error in one place, say what happened, and fall back to questions and template replies.

Sending local traffic through a proxy. On computers with a proxy, a request to `localhost` may be sent to the proxy and fail. *Avoid it:* build the HTTP client without a proxy, as both clients here do.

Testing only with the real model. Replies that change every run make tests that pass today and fail tomorrow. *Avoid it:* test the program's logic with the stand-in (or with a fake model object, as the chapter's tests do), and test the model separately, with real messages.

Storing customers' messages "for later". They contain personal data you do not need. *Avoid it:* store only the understood facts, and only when you will use them.

13.11 Chapter Summary

- A language model continues text fluently, but it has no checked knowledge, can **hallucinate**, and may answer differently each time; a VCIRS knowledge base decides from checked cases, the same way every time, with an explanation [1], [2].
- In the chat front end (**Figure 13.1**) the model has two narrow jobs — **understand** a message as facts and **phrase** the advice as sentences — and the knowledge base has the decision. A **filter** keeps only known variables and values, and a **guard** lets a reply through only if it gives the advice word for word, adds no other fix, and uses no forbidden phrase.
- **Ollama** runs open models on your own computer and answers `POST /api/chat` on port 11434; September 2026's release is v0.34.3 [6]. Small models such as `qwen3.5:2b` (2.7 GB) fit a counter computer (**Table 13.1**). The `format` field asks for **structured output** that matches a JSON Schema [5], [10].
- The client (**Listing 13.1**, **Listing 13.2**) and the first question (**Listing 13.3**, **Listing 13.4**) use only the standard libraries. A **stand-in** server (**Listing 13.5**, **Listing 13.6**) answers in Ollama's format from a phrase book, so the chapter runs offline and repeatably (**Figure 13.2**).

- The chat doctor (**Listing 13.7**, **Listing 13.8**) understood a full message, asked for missing facts in any words, refused an unknown value, said honestly when it had no advice, rejected a careless reply, and kept working with no model at all.

13.12 Exercises

13.1 In your own words, list the jobs of the language model and the jobs of the knowledge base in **Figure 13.1**. For each of the model's jobs, name the piece of code that checks its result.

13.2 In conversation 2 the model understood "the router lights are blinking" as `router lights = blinking`, and the filter ignored it. Mas Anto says that in his experience a blinking light means the router is still starting. What should he do so that the doctor handles blinking lights — change the filter, the prompt, or the knowledge base? Explain, and describe the case he would teach.

13.3 Which of these replies does the guard accept for the advice *turn off airplane mode*, and why? (a) "Please turn off airplane mode; your plane symbol is on." (b) "Switch off flight mode, then try again." (c) "Please turn off airplane mode. If that fails, turn on Wi-Fi." (d) "Please turn off airplane mode. The battery is fine."

13.4 Change `check_reply` (in either language) so that it also rejects a reply that contains any number with three or more digits, such as a phone number or a price the model made up. Show the new lines and say which conversation of Section 13.6 would be affected.

13.5 Design an extension that lets the doctor reply in Indonesian while the guard still checks the advice. Where do the Indonesian words for each advice come from, how does the guard use them, and what does the template reply look like?

13.6 In conversation 5 the customer had to type exact values. Suppose the model is missing but the customer answers "yes it is on" to the airplane question. What does the program print, and what does the engine do with the answer? Would you change this behavior?

13.7 The stand-in's phrase book maps "the others too" to `other devices online = no`. Explain why a phrase book can never replace a language model, using two example messages that the phrase book would understand wrongly.

13.8 Estimate how many requests to the model conversation 2 needed, and how many conversation 1 needed. If each request takes four seconds on the counter's laptop, how long does each conversation take? What does that suggest about when to use the chat doctor and when to use the question-and-answer doctor?

13.13 Solutions to the Exercises

Solution 13.1. The model has two jobs. (1) *Understand*: turn a message or an answer into facts; its result is checked by `understand`'s filter, which keeps only variables in the list and values the knowledge base knows. (2) *Phrase*: turn the advice and its reasons into sentences; its result is checked by `check_reply`, the guard. The knowledge base has three jobs: decide which node fires (`kb.infer`), ask for the facts it still needs (through the asker), and explain the advice (the conditions of the nodes that fired, which become the reasons). It also has a fourth, silent job: to say that it has no advice, which the program turns into a fixed sentence.

Solution 13.2. Not the filter: it is right to ignore a value the knowledge base does not know, because nobody has checked what it means. Not only the prompt either: telling the model "blinking means off" would put unchecked knowledge in a place nobody reviews. The right place

is the knowledge base. First, declare `blinking` as one of the usual answers of `router lights` in `wifi-questions.json`, so that the filter accepts it and the question offers it. Then, the next time a customer has a blinking light, teach the case at the counter with the doctor of Chapter 3: facts `connection = wifi`, `wifi icon = on`, `other devices online = no`, `router lights = blinking`, fix *wait five minutes for the router to start*. The engine hangs it as an exception of *restart the router*, next to the red and off exceptions, and the chat doctor uses it at once.

Solution 13.3. (a) Accepted: it contains the advice word for word, no other fix, no forbidden phrase, and is short. (b) Rejected: it does not contain "turn off airplane mode" word for word, even though it means the same; the customer receives the template reply. (c) Rejected: it adds another fix, *turn on Wi-Fi*. (d) Rejected: "battery" is on the list of forbidden phrases. The lesson of (b) is that the guard trades some good replies for certainty.

Solution 13.4. In Python, add before the length check:

```
if re.search(r"\d{3,}", reply):
    problems.append("it contains a long number")
```

with `import re` at the top; in Java, `if (reply.matches("(?s).*\\d{3,}.*")) { problems.add("it contains a long number"); }`. None of the five conversations is affected, because none of their replies contains a number of three or more digits; the rule protects against a model that invents a phone number or a price. (A reply mentioning "20 minutes" has two digits and still passes.)

Solution 13.5. Keep a small table, for example `advice-id.json`, that maps each advice value to its Indonesian wording, written and checked by Mas Anto: `"turn off airplane mode": "matikan mode pesawat"`. The program sends the Indonesian wording to the model as the `ADVICE` line, and the guard checks the reply for that wording, and for the Indonesian wordings of all other fixes. The template reply becomes "Silakan matikan mode pesawat. Saya menyarankannya karena …", with the reasons written as "wifi icon adalah off" or, better, with a second small table of Indonesian wordings for the facts. The knowledge base itself does not change; only the words around it do.

Solution 13.6. With no model, `ask` accepts only an answer that is exactly one of the known values. "yes it is on" is not, so the program prints `[not understood] please use one of: on / off` and returns `None`. The engine treats the fact as unknown, the exception *turn off airplane mode* does not fire, and the customer receives *turn on Wi-Fi* — the right advice for what the engine knows, but not for this customer. A better behavior is to ask again, up to two times, when the answer is not understood; that is a small loop around the reading in `ask`.

Solution 13.7. A phrase book matches text, not meaning. "The others too have no internet" contains "the others too" and is correctly understood as *no*; but "The others too are fine" also contains it and would be misunderstood as *no*. "My phone is not in airplane mode, the plane is off" contains "plane is off" and is right, but "the plane is on the table" contains "plane is on" and is wrong. A language model uses the whole sentence to decide what a phrase means; that is exactly the part we let it do.

Solution 13.8. Conversation 1 needed two requests: one to understand the message and one to phrase the reply. Conversation 2 needed four: one for the first message, one for "the wifi", one for "nope, none of them", and one for the reply; "on" and "red" were exact known values and needed no request at all. At four seconds each, conversation 1 makes the customer wait about 8 seconds and conversation 2 about 16, plus the time the customer needs to type. For messages that arrive in the evening this is fine. At a busy counter, where a person can pick answers from a list in a

second, the question-and-answer doctor of Chapter 3 is faster; the chat doctor earns its time where people write freely and do not know which facts matter.

References

[1] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, and P. Fung, "Survey of hallucination in natural language generation," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–38, 2023.

[2] M. M. I. Subakti, "A Variable-Centered Intelligent Rule System," M.Sc.Eng. thesis, Dept. of Computer Science and Information Engineering, Taipei, Taiwan, Jun. 2005.

[3] P. Compton and B. H. Kang, *Ripple-Down Rules: The Alternative to Machine Learning*. Boca Raton, FL, USA: CRC Press, 2021.

[4] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems 33 (NeurIPS 2020)*, 2020, pp. 9459–9474.

[5] Ollama, "Generate a chat message (POST /api/chat)," *Ollama API documentation*, 2026. [Online]. Available: https://docs.ollama.com/api/chat

[6] Ollama, "Releases: v0.34.3," GitHub, Sep. 19, 2026. [Online]. Available: https://github.com/ollama/ollama/releases

[7] Ollama, "qwen3.5," *Ollama model library*, 2026. [Online]. Available: https://ollama.com/library/qwen3.5

[8] Ollama, "llama3.2," *Ollama model library*, 2026. [Online]. Available: https://ollama.com/library/llama3.2

[9] Ollama, "gemma4," *Ollama model library*, 2026. [Online]. Available: https://ollama.com/library/gemma4

[10] Ollama, "Structured outputs," *Ollama blog*, Dec. 6, 2024. [Online]. Available: https://ollama.com/blog/structured-outputs

[11] Ollama, "Download Ollama," 2026. [Online]. Available: https://ollama.com/download

[12] I. Subakti and R. Hidayatullah, "Aplikasi sistem pakar untuk diagnosis awal gangguan kesehatan secara mandiri menggunakan Variable-Centered Intelligent Rule System," vol. 6, no. 1, pp. 11–16, Jan. 2007.

Chapter 14

Capstone: a VCIRS Expert-System Builder Anyone Can Use

By the end of this chapter you will be able to design a small web application with which a person who has never programmed builds an expert system of her own: she writes the questions, teaches real cases one by one, consults the knowledge, reads it as a tree and as rules, sees the usage statistics that VCIRS computes — Variable, Node and Rule Usage Rates — and what they mean for her, accepts or ignores the new nodes and rules that VCIRS proposes, and exports or imports the whole knowledge base as one JSON file. You will build the application twice, in Java with the web server that comes with the JDK and in Python with Flask, in such a way that the two servers show the same pages byte for byte and write the same knowledge-base file; you will separate what the application *does* from how it *looks* so that each part can be tested alone; and you will see the finished builder in English and in Indonesian, through screenshots of the running application.

14.1 Bu Wati's Notebook

Bu Wati makes cakes. Her shop is on the ground floor of her house: a glass case with *bolu*, *brownies* and layered cakes, two ovens at the back, and, on a shelf above the flour, a school exercise book with a soft blue cover. In that book, for eleven years, she has written down every cake that went wrong and what she found to be the reason. *Sank in the middle — Tini opened the oven to look. Dense like a brick — eggs straight from the fridge. Dry — forgot the timer, 15 minutes too long.* When a new helper joins, Bu Wati reads the book aloud to her over a few afternoons.

Two helpers work with her now, Rini and a younger girl who started this month. When a cake fails on a busy Saturday, they cannot stop to read eleven years of notes, and Bu Wati is not always there. She would like what the whole book knows to be available to them in a minute, on the old laptop by the ovens or on her phone. She would also like to keep writing in it — that is, to keep adding knowledge — without anyone's help, because she does not program, and she is not going to learn.

The earlier chapters of this book gave her most of what she needs. Chapter 3 showed how knowledge grows case by case, safely, as exceptions. Chapter 4 showed how to think in variables, Chapter 6 how VCIRS proposes new rules from its usage statistics, Chapter 7 what those statistics are, and Chapter 10 how to put an expert system on a web page. But every one of those applications was built by a programmer for one purpose. This chapter builds the tool that makes the programmer unnecessary: an **expert-system builder**, a small web application in which anyone can build, grow and share a VCIRS knowledge base about anything — cakes, phones, motorcycles, rice fields — using only a browser.

14.2 The Idea in Plain Words

14.2.1 What the builder must let a person do

Start from the person, not from the program. What does Bu Wati need to do, in her own words? She needs to write down the *questions* she asks about a failed cake, with the usual answers. She needs to *teach* a case: "this cake sank, the oven door was opened early, and the fix is to keep the door closed". She needs to *ask* — to consult the knowledge for a new cake. She needs to *see* what her knowledge base has become, and to *stop* a piece of advice she no longer believes. She needs to know *which questions matter most*, and to hear the machine's *suggestions* for simpler rules. And she needs to *save* the whole thing, to keep a copy or to give it to her sister, who runs a cake shop in another town.

Each of these becomes one page of the builder, and each page rests on one idea of VCIRS that an earlier chapter explained. **Table 14.1** puts them side by side.

Table 14.1 The pages of the builder, what a person does on each, and the VCIRS idea behind it.

Page	What the person does	The VCIRS idea behind it
Questions	writes the questions and their usual answers	variables and their values (Chapter 4)
Teach	describes a real case and gives the right answer	learning case by case, with cornerstone cases and the list of difference (Chapter 3)
Consult	picks the answers she knows and gets advice	RDR inference with an explanation (Chapters 3 and 8)
Knowledge	reads the nodes as a tree, stops a node	the Node and Rule Structures; stopping instead of deleting
Statistics	sees which questions and nodes are used most	Variable, Node and Rule Usage Rates (Chapter 7)
Proposals	keeps the new nodes and rules that make sense	rule and node generation (Chapter 6)
Home	exports, imports, starts a new project	the JSON knowledge-base file shared by Java and Python

Two principles guided every page. First, **the person speaks in cases, not in rules**. Nowhere in the builder does Bu Wati type "IF … THEN …". She describes a cake that failed and says what fixed it; the engine decides where that knowledge belongs, exactly as in Chapter 3, and tells her in one sentence what it did. Second, **nothing the person does can break what she did before**. Teaching adds exceptions that leave old cases alone; stopping a node keeps it and its history; proposals are only proposals; and every change is saved at once, so closing the browser loses nothing. These are the same promises that made Ripple Down Rules usable by pathologists who were not programmers [1], [2], now kept by a web page.

A good builder for laypeople also follows the general rules of usable software that designers have written down over the years: show the person where she is, speak her language rather than the system's, prevent errors rather than scold her for them, and make every action's result visible at once [3]. We will meet each of these in the pages, and we will also meet the places where the builder, honestly, still speaks the engine's language.

14.2.2 Usage statistics in plain words

A knowledge base built from cases grows in the order the cases arrived, not in the order of importance. VCIRS measures importance by counting how the knowledge is *used* by its own

structure — how many nodes share a variable, and how many later cases agreed with a condition [4], [5]. The builder shows three of these measures on its Statistics page. They look technical, but each answers a question Bu Wati would ask.

How many nodes use this question? That count is the variable's **node sharing**, written NS. In Bu Wati's knowledge base, `problem` appears in six nodes and `oven heat` in three: `problem` is the question that matters most, and the builder asks it first.

How much does a condition count inside its node? Every condition of a node stands at a position, its **variable order** (VO), among the node's TV conditions. The later a condition stands, the closer it is to the conclusion, and VCIRS gives it more weight through the **closeness degree** CD. Every condition also carries a **Credit**: it starts at 1 and grows by 1 each time a later case, taught as an exception of this node, agreed with the condition (Chapter 7 tells the full story). The **Variable Usage Rate** of a condition combines the three [4]:

$$\mathrm{VUR} = \mathrm{Credit} \times \mathrm{NS} \times \mathrm{CD}, \qquad \mathrm{CD} = \frac{\mathrm{VO}}{\mathrm{TV}} \tag{14.1}$$

How used is a node, and a rule? The **Node Usage Rate** of a node is the average VUR of its conditions, and the **Rule Usage Rate** of a rule — a path of nodes from the top level down — is the average NUR of its nodes [4]:

$$\mathrm{NUR}(n) = \frac{1}{\mathrm{TV}_n} \sum_{i=1}^{\mathrm{TV}_n} \mathrm{VUR}_i\,, \qquad \mathrm{RUR}(r) = \frac{1}{k} \sum_{j=1}^{k} \mathrm{NUR}\,(n_j) \tag{14.2}$$

Let us work one example by hand with Bu Wati's knowledge base, as it stands after the session of Section 14.4. The node *bake longer#1* has one condition, `problem = sank in the middle`, so TV is 1 and VO is 1, and CD is 1/1 = 1. The variable `problem` is used by six nodes, so NS is 6. The Credit of the condition started at 1 and was raised twice, once when *keep the door closed* was taught under this node and once when *heat the oven fully first* was, because both of those cases also had `problem = sank in the middle`; so Credit is 3. By Equation (14.1), VUR = 3 × 6 × 1 = 18, and since the node has only this one condition, its NUR is 18 as well. Its exception *keep the door closed#1* has one condition, `oven door opened early = yes`: Credit 1, NS 2 (the variable is used by two nodes), CD 1, so VUR = NUR = 2. The rule that ends at *keep the door closed#1* passes through both nodes, so by Equation (14.2) its RUR is (18 + 2) / 2 = 10.

What does Bu Wati learn from such numbers? A **large** rate marks knowledge that many other pieces of knowledge build on: the node *bake longer* is the root of a whole family of exceptions, and if it were wrong, all of them would be reached in the wrong way. Those are the nodes to review most carefully. A question with a large rate should be asked **first**, because it separates the cases best; the builder's Consult page and the question order use exactly this. A **small** rate marks a young, narrow piece of knowledge — often perfectly right, but resting on one case, and worth watching.

> **KEY INSIGHT.** The usage rates do not say whether a piece of advice is *true*. They say how much of the knowledge base *depends* on it. Truth comes from the cases a person checked; the rates tell her where to look first.

14.2.3 Proposals: the machine suggests, the person decides

From the same statistics VCIRS can **generate** new knowledge, as Chapter 6 showed [4], [6]. It finds the most important nodes (those shared by the most rules) and proposes a new rule that visits the nodes in a good order; and it finds the most important variables (those shared by the most nodes) and proposes, for each node that uses one, a new node whose conditions are taken from all nodes in a good order. The proposals are deliberately generous: a proposed node lists *many* conditions, and the person ticks the ones that belong together. VCIRS never adds a generated item by itself — the thesis says plainly that the user keeps only the semantically correct items [4].

This is where a builder for laypeople must be most careful with words. The Proposals page calls them *proposals* and says, at the top, "keep what makes sense and ignore the rest". A proposed node appears as a list of checkboxes, all unticked. In Section 14.7 we will see Bu Wati look at a proposal of eight conditions and keep exactly one — `problem = dry` — to make a shorter, more general node: *a dry cake means take it out earlier*, whatever else happened. That is the kind of generalization an expert makes naturally once she sees it written down, and the builder's job is to put it in front of her.

14.2.4 One application, two servers, the same pages

Figure 14.1 shows how the builder is put together.

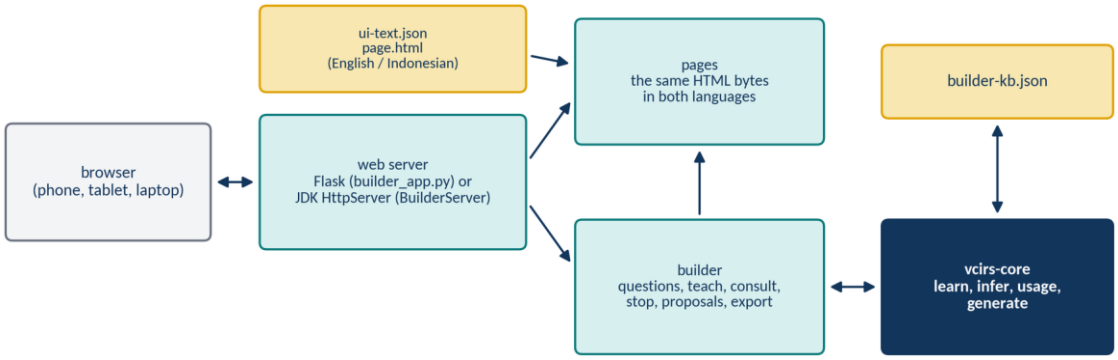

Figure 14.1 The builder. The browser talks to a small web server (Flask in Python, the JDK's HttpServer in Java). The server only connects the web to two parts that know nothing about the web: the pages, which build HTML from the knowledge base and from the words in `ui-text.json`, and the builder, which changes the knowledge base through vcirs-core and saves it after every change.

The most important decision in the picture is the separation into three parts. The **builder** (`Builder.java`, `builder.py`) is everything a person can do with a knowledge base, written as ordinary methods: `declare`, `teach`, `consult`, `stop`, `accept_node`, `export_json`, `import_json`. It knows nothing about browsers or HTML. The **pages** (`Pages.java`, `pages.py`) turn a knowledge base into HTML; they change nothing. The **server** (`BuilderServer.java`, `builder_app.py`) reads a web request, calls the builder, asks the pages for HTML, and sends it back. Because the builder knows nothing about the web, we can test it and demonstrate it with a plain script (Section 14.4) before a single page exists. Because the pages change nothing, we can compare the two languages' pages byte for byte.

The words on the pages are not in the code. They come from `ui-text.json`, which holds every label and hint in English and in Indonesian; a link at the top of every page switches between the two. The frame of every page — the header, the navigation and the style — comes from

`page.html`. Both files are shared by the Java and the Python servers, which is one reason why the two produce the same bytes.

14.3 Step 1: The Builder, Without the Web

Listing 14.1 is the builder in Python.

Listing 14.1 `builder.py`: everything a person can do with a knowledge base, without the web.

```
"""The expert-system builder: everything a person can do with a knowledge
base, without the web. The web app (builder_app.py) only shows forms and
calls these methods; builder_demo.py calls them from a script.
"""

import json
import os
import pathlib
import sys

sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import (Case, Conclusion, Condition, KnowledgeBase,  # noqa: E402
                   VcirsError)

DEFAULT_GOAL = "advice"


class Builder:
    def __init__(self, path):
        self.path = path
        if os.path.exists(path):
            self.kb = KnowledgeBase.load(path)
        else:
            self.kb = KnowledgeBase("My expert system", "")

    def save(self):
        self.kb.save(self.path)

    # ----- what the forms need --------------------------------------
    def goal(self):
        """The variable the answers are about: that of the first node."""
        for node in self.kb.nodes.values():
            return node.conclusions[0].variable
        return DEFAULT_GOAL
```

Listing 14.1 (Continued) *builder.py: everything a person can do with a knowledge base, without the web.*

```
    def question_variables(self):
        """Variables to ask about: the most used first, then the rest in
        the order they were made. Answer-only variables are left out."""
        asked = set(c.variable for n in self.kb.nodes.values()
                    for c in n.conditions)
        answers = set(c.variable for n in self.kb.nodes.values()
                      for c in n.conclusions)
        order = self.kb.usage().question_order()
        rest = [v for v in self.kb.variables if v not in order]
        return [v for v in order + rest
                if v in asked or v not in answers and v != self.goal()]

    # ----- changing the knowledge base -----------------------------------
    def new_project(self, name, description):
        name = name.strip()
        if not name:
            raise VcirsError("please give the project a name")
        self.kb = KnowledgeBase(name, description.strip())
        self.save()

    def declare(self, variable, question, choices_text):
        variable = variable.strip()
        if not variable or "=" in variable or "<" in variable \
                or ">" in variable:
            raise VcirsError("a question name must not be empty or contain "
                             "= < >")
        choices = [c.strip() for c in choices_text.split(",") if c.strip()]
        self.kb.declare(variable, question.strip(), choices)
        self.save()

    def make_case(self, pairs, answer=""):
        """pairs: (variable, value) with empty values left out; answer is a
        value of the goal, or 'variable = value'."""
        facts = [Condition(v, "=", x.strip()) for v, x in pairs if x.strip()]
        conclusions = []
        if answer.strip():
            if "=" in answer:
                conclusions.append(Conclusion.parse(answer.strip()))
            else:
                conclusions.append(Conclusion(self.goal(), answer.strip()))
        return Case(facts, conclusions)
```

Listing 14.1 (Continued) *builder.py: everything a person can do with a knowledge base, without the web.*

```
    def teach(self, pairs, answer, teacher):
        """Returns (what the knowledge base said before, LearnResult)."""
        case = self.make_case(pairs, answer)
        if not case.facts:
            raise VcirsError("please give at least one fact")
        if not case.conclusions:
            raise VcirsError("please give the right answer")
        before = self.kb.infer(case.known()).what()[0]
        result = self.kb.learn(case, teacher.strip())
        self.save()
        return before, result

    def consult(self, pairs):
        return self.kb.infer(self.make_case(pairs).known())

    def stop(self, node_id, teacher):
        self.kb.stop(node_id, teacher.strip())
        self.save()

    def accept_rule(self, rule_id, teacher):
        for g in self.kb.generate_rules():
            if g.id == rule_id:
                rule = self.kb.accept_rule(g, g.nodes, teacher.strip())
                self.save()
                return rule
        raise VcirsError("that proposal is no longer there")

    def accept_node(self, node_id, keep, teacher):
        for g in self.kb.generate_nodes():
            if g.id == node_id:
                node = self.kb.accept_node(g, keep, teacher.strip())
                self.save()
                return node
        raise VcirsError("that proposal is no longer there")

    # ----- export and import ---------------------------------------------
    def export_json(self):
        return self.kb.to_json()
```

Listing 14.1 (Continued) *builder.py: everything a person can do with a knowledge base, without the web.*

```
    def import_json(self, text):
        """A knowledge base file (vcirs-kb) or a case file with "cases"."""
        try:
            data = json.loads(text)
            if isinstance(data, dict) and "cases" in data:
                kb = cases_to_kb(data)
            else:
                kb = KnowledgeBase.from_json(text)
        except (ValueError, KeyError, TypeError, AttributeError, VcirsError):
            raise VcirsError("this is not a knowledge base or case file I "
                             "can read") from None
        self.kb = kb
        self.save()


def cases_to_kb(data):
    """Build a knowledge base from a case file, as if taught case by case."""
    kb = KnowledgeBase(data.get("name", "Imported cases"),
                       data.get("description", ""))
    for v in data.get("variables", []):
        kb.declare(v["variable"], v.get("question", ""), v.get("choices", []))
    for c in data["cases"]:
        case = Case.of(c["facts"], c["conclusions"], c.get("cf", 100))
        kb.learn(case, c.get("author", ""))
    return kb
```

Read it as a list of promises to the person.

It opens what was there. The constructor loads the knowledge-base file if it exists and otherwise starts an empty project called “My expert system”. `save` writes the file; every method that changes something calls it at the end, so the file on disk always holds the latest knowledge.

It knows what the answers are about. `goal` is the variable of the answers: the conclusion variable of the first node, or `advice` in an empty project. On the Teach page the person types only the value — “keep the door closed” — and `make_case` turns it into `advice = keep the door closed`. A person who needs a second kind of answer may type `variable = value` instead.

It asks the most useful questions first. `question_variables` lists the variables to show as questions: first in the order `usage().question_order()` gives — the variables ranked by their best VUR, Equation (14.1) — then the declared variables that no node uses yet, in the order they were made. It leaves out the variables that appear only as answers, so the goal `advice` never shows up as a question.

It protects the person from simple mistakes. `new_project` refuses an empty name. `declare` refuses a question name that contains `=`, `<` or `>`, because those characters separate a variable from its value in conditions; it splits the usual answers at commas and drops empty ones. `teach` refuses a case with no facts or no answer. Each refusal is a `VcirsError` with a sentence the person can understand; the server shows it at the top of the page.

It teaches, and says what happened. `teach` builds the case, asks the knowledge base what it would have answered *before* learning (the first line of `what()`), then calls `learn` — the one-call Ripple-Down-Rules loop of Chapter 3, which accepts the case, adds an exception under the node that answered wrongly, or adds a new top-level node — and returns both. The person sees “before, the knowledge base said…” and “what happened”, which is the most honest way to show that her teaching changed something, and exactly what.

It consults. `consult` turns the chosen answers into facts and runs `infer`. There is no asker here: the web form is itself the list of questions, and a question left empty is simply unknown.

It stops, and accepts proposals. `stop` stops a node (it is kept, with its history, but gives no advice any more). `accept_rule` and `accept_node` look for the proposal with the given identifier in a fresh list of proposals — the list may have changed since the page was shown — and accept it with the conditions the person kept.

It exports and imports. `export_json` is the knowledge base's own JSON. `import_json` accepts two kinds of file. A file with a `cases` list is a **case file**, like `cake-cases.json`: `cases_to_kb` declares its questions and teaches its cases one by one with `learn`, exactly as if a person had typed them into the Teach page. Anything else must be a VCIRS knowledge-base file. If the text is not JSON, or is JSON of the wrong shape, the person is told, in one sentence, that it is not a file the builder can read — and the knowledge base she had is left untouched, because `self.kb` is replaced only after the new one was built successfully.

Listing 14.2 is the same builder in Java.

Listing 14.2 `Builder.java`: the same builder in Java.

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

import vcirs.Case;
import vcirs.Conclusion;
import vcirs.Condition;
import vcirs.GeneratedNode;
import vcirs.GeneratedRule;
import vcirs.InferenceResult;
import vcirs.Json;
import vcirs.KnowledgeBase;
import vcirs.LearnResult;
import vcirs.Node;
import vcirs.Rule;
import vcirs.VcirsException;

/**
 * The expert-system builder: everything a person can do with a knowledge
 * base, without the web. The web app (BuilderServer) only shows forms and
 * calls these methods; BuilderDemo calls them from a script.
 */
public class Builder {
    public static final String DEFAULT_GOAL = "advice";

    /** What teach() tells the person: the old answer and what changed. */
    public record Taught(String before, LearnResult result) {
    }

    public final String path;
    public KnowledgeBase kb;

    public Builder(String path) {
        this.path = path;
        kb = Files.exists(Path.of(path)) ? KnowledgeBase.load(path)
                : new KnowledgeBase("My expert system", "");
    }

    public void save() {
        kb.save(path);
    }
```

Listing 14.2 (Continued) *Builder.java: the same builder in Java.*

```
    // ----- what the forms need ------------------------------------------
    /** The variable the answers are about: that of the first node. */
    public String goal() {
        for (Node node : kb.nodes.values()) {
            return node.conclusions.get(0).variable;
        }
        return DEFAULT_GOAL;
    }

    /**
     * Variables to ask about: the most used first, then the rest in the
     * order they were made. Answer-only variables are left out.
     */
    public List<String> questionVariables() {
        Set<String> asked = new HashSet<>();
        Set<String> answers = new HashSet<>();
        for (Node n : kb.nodes.values()) {
            n.conditions.forEach(c -> asked.add(c.variable));
            n.conclusions.forEach(c -> answers.add(c.variable));
        }
        List<String> all = new ArrayList<>(kb.usage().questionOrder());
        for (String v : kb.variables.keySet()) {
            if (!all.contains(v)) {
                all.add(v);
            }
        }
        List<String> out = new ArrayList<>();
        for (String v : all) {
            if (asked.contains(v) || !answers.contains(v) && !v.equals(goal())) {
                out.add(v);
            }
        }
        return out;
    }

    // ----- changing the knowledge base ----------------------------------
    public void newProject(String name, String description) {
        if (name.strip().isEmpty()) {
            throw new VcirsException("please give the project a name");
        }
        kb = new KnowledgeBase(name.strip(), description.strip());
        save();
    }
```

Listing 14.2 (Continued) *Builder.java: the same builder in Java.*

```
    public void declare(String variable, String question, String choicesText) {
        String v = variable.strip();
        if (v.isEmpty() || v.contains("=") || v.contains("<") || v.contains(">")) {
            throw new VcirsException("a question name must not be empty or "
                    + "contain = < >");
        }
        List<String> choices = new ArrayList<>();
        for (String c : choicesText.split(",")) {
            if (!c.strip().isEmpty()) {
                choices.add(c.strip());
            }
        }
        kb.declare(v, question.strip(), choices);
        save();
    }

    /**
     * pairs: {variable, value} with empty values left out; answer is a
     * value of the goal, or "variable = value".
     */
    public Case makeCase(List<String[]> pairs, String answer) {
        List<Condition> facts = new ArrayList<>();
        for (String[] p : pairs) {
            if (!p[1].strip().isEmpty()) {
                facts.add(new Condition(p[0], "=", p[1].strip()));
            }
        }
        List<Conclusion> conclusions = new ArrayList<>();
        if (!answer.strip().isEmpty()) {
            conclusions.add(answer.contains("=")
                    ? Conclusion.parse(answer.strip())
                    : new Conclusion(goal(), answer.strip()));
        }
        return new Case(facts, conclusions, 100);
    }
```

Listing 14.2 (Continued) *Builder.java: the same builder in Java.*

```
    public Taught teach(List<String[]> pairs, String answer, String teacher) {
        Case c = makeCase(pairs, answer);
        if (c.facts.isEmpty()) {
            throw new VcirsException("please give at least one fact");
        }
        if (c.conclusions.isEmpty()) {
            throw new VcirsException("please give the right answer");
        }
        String before = kb.infer(c.known()).what().get(0);
        LearnResult result = kb.learn(c, teacher.strip());
        save();
        return new Taught(before, result);
    }

    public InferenceResult consult(List<String[]> pairs) {
        return kb.infer(makeCase(pairs, "").known());
    }

    public void stop(String nodeId, String teacher) {
        kb.stop(nodeId, teacher.strip());
        save();
    }

    public Rule acceptRule(String ruleId, String teacher) {
        for (GeneratedRule g : kb.generateRules()) {
            if (g.id.equals(ruleId)) {
                Rule rule = kb.acceptRule(g, g.nodes, teacher.strip());
                save();
                return rule;
            }
        }
        throw new VcirsException("that proposal is no longer there");
    }

    public Node acceptNode(String nodeId, List<String> keep, String teacher) {
        for (GeneratedNode g : kb.generateNodes()) {
            if (g.id.equals(nodeId)) {
                Node node = kb.acceptNode(g, keep, teacher.strip());
                save();
                return node;
            }
        }
        throw new VcirsException("that proposal is no longer there");
    }
```

Listing 14.2 (Continued) *Builder.java: the same builder in Java.*

```
    // ----- export and import ---------------------------------------------
    public String exportJson() {
        return kb.toJson();
    }

    /** A knowledge base file (vcirs-kb) or a case file with "cases". */
    public void importJson(String text) {
        KnowledgeBase loaded;
        try {
            Object data = Json.parse(text);
            if (data instanceof Map && ((Map<?, ?>) data).containsKey("cases")) {
                loaded = casesToKb((Map<?, ?>) data);
            } else {
                loaded = KnowledgeBase.fromJson(text);
            }
        } catch (RuntimeException e) {
            throw new VcirsException("this is not a knowledge base or case file "
                    + "I can read");
        }
        kb = loaded;
        save();
    }

    /** Build a knowledge base from a case file, as if taught case by case. */
    static KnowledgeBase casesToKb(Map<?, ?> data) {
        Object name = data.get("name");
        Object description = data.get("description");
        KnowledgeBase k = new KnowledgeBase(
                name == null ? "Imported cases" : (String) name,
                description == null ? "" : (String) description);
        Object vars = data.get("variables");
        for (Object o : vars == null ? List.of() : (List<?>) vars) {
            Map<?, ?> v = (Map<?, ?>) o;
            Object q = v.get("question");
            Object ch = v.get("choices");
            k.declare((String) v.get("variable"), q == null ? "" : (String) q,
                    strings(ch == null ? List.of() : (List<?>) ch));
        }
        for (Object o : (List<?>) data.get("cases")) {
            Map<?, ?> c = (Map<?, ?>) o;
            Object cf = c.get("cf");
            Case aCase = Case.of(strings((List<?>) c.get("facts")),
                    strings((List<?>) c.get("conclusions")),
                    cf == null ? 100 : ((Number) cf).intValue(), 100);
            Object who = c.get("author");
            k.learn(aCase, who == null ? "" : (String) who);
```

Listing 14.2 (Continued) *Builder.java: the same builder in Java.*

```
        }
        return k;
    }

    static List<String> strings(List<?> list) {
        List<String> out = new ArrayList<>();
        for (Object o : list) {
            out.add((String) o);
        }
        return out;
    }
}
```

The Java version is longer mainly because Java has no dictionary literals: a pair of facts is a `String[]` of two elements, and the case file is read with `Json.parse` into maps and lists that must

be cast. `record Taught(String before, LearnResult result)` is a compact way to return two values from `teach`. `importJson` catches `RuntimeException`, the common parent of the many ways a wrong file can fail (a `ClassCastException` when a list is where a string should be, a `NullPointerException` when a field is missing, a `VcirsException` from the engine), and turns all of them into the same sentence.

14.4 Step 2: Bu Wati's Session, as a Script

Before building a single web page, we use the builder from a script that plays Bu Wati's first session: start a project, write six questions, teach the nine cases of her notebook's first pages, consult, teach a new case, look at the knowledge, the statistics and the proposals, accept one proposal, and export and import. The cases come from the chapter's example file, `cake-cases.json`, whose first lines are:

```
{
  "name": "Cake doctor",
  "description": "Why did the cake fail? What Bu Wati teaches the helpers of her cake
shop.",
  "variables": [
    {"variable": "problem", "question": "What went wrong with the cake?", "choices":
["sank in the middle", "dense and heavy", "cracked top", "raw inside", "dry", "burnt
bottom"]},
```

Listing 14.3 is the script in Python.

Listing 14.3 `builder_demo.py`: Bu Wati's session, played through the builder.

```
"""Bu Wati builds her cake doctor with the builder, step by step, the way
she would with the web app. Prints what each step does.

    python builder_demo.py [knowledge-base.json]
"""

import json
import os
import sys

from builder import Builder
from vcirs import VcirsError, num, report

DATA = os.path.join(os.path.dirname(os.path.abspath(__file__)), "..", "data")


def pairs(text):
    """'problem = dry; baking time = long' -> [('problem', 'dry'), ...]"""
    out = []
    for part in text.split(";"):
        var, value = part.split("=")
        out.append((var.strip(), value.strip()))
    return out


def teach(b, facts, answer, teacher):
    try:
        before, result = b.teach(pairs(facts), answer, teacher)
    except VcirsError as e:
        print("Teach " + facts + " -> " + answer + ": refused: " + str(e))
        return
    print("Teach " + facts + " -> " + answer + " (" + teacher + ")")
    print("  before: " + before.strip())
    print("  now:    " + str(result))


def consult(b, facts):
    result = b.consult(pairs(facts))
    print("Consult " + facts)
    for line in result.what() + result.how():
        print("  " + line)
```

Listing 14.3 (Continued) *builder_demo.py: Bu Wati's session, played through the builder.*

```
def main(args):
    path = args[0] if args else os.path.join(DATA, "cake-kb.json")
    b = Builder(path)
    with open(os.path.join(DATA, "cake-cases.json"), encoding="utf-8") as f:
        example = json.load(f)
    print("== 1. A new project")
    b.new_project(example["name"], example["description"])
    print(b.kb.summary())
    print("== 2. The questions")
    for v in example["variables"]:
        b.declare(v["variable"], v["question"], ", ".join(v["choices"]))
        print("Question " + v["variable"] + ": " + v["question"] + " ("
              + " / ".join(v["choices"]) + ")")
    print("== 3. Teaching, case by case")
    for c in example["cases"]:
        teach(b, "; ".join(c["facts"]), c["conclusions"][0].split("= ")[1],
              c["author"])
    print("== 4. Consulting")
    consult(b, "problem = sank in the middle; oven door opened early = no; "
               "oven heat = low")
    consult(b, "problem = burnt bottom")
    teach(b, "problem = burnt bottom", "bake on a higher rack", "Wati")
    teach(b, "problem = burnt bottom", "use a lighter pan", "Rini")
    teach(b, "problem = burnt bottom; pan = dark metal", "use a lighter pan",
          "Rini")
    print("== 5. The knowledge")
    for line in report.tree(b.kb):
        print(line)
    print("== 6. Usage statistics")
    u = b.kb.usage()
    print("Ask first: " + ", ".join(b.question_variables()))
    print("Most important question: " + ", ".join(u.important_variables()))
    print("Most important nodes: " + ", ".join(u.important_nodes()))
    for row in u.nur_rows():
        print("  NUR " + num.fixed(row[2], 3) + "  " + row[0])
    print("== 7. Proposals")
    for g in b.kb.generate_rules():
        print("Rule " + g.id + ": " + " -> ".join(g.nodes))
    for g in b.kb.generate_nodes():
        print("Node " + g.id + ": " + ", ".join(str(c) for c in g.conditions))
    node = b.accept_node("take it out earlier#1G", ["problem = dry"], "Wati")
    print("Accepted: " + str(node))
    consult(b, "problem = dry")
    print("== 8. Export and import")
    text = b.export_json()
    print("Exported " + str(len(text.splitlines())) + " lines of JSON")
```

Listing 14.3 (Continued) *builder_demo.py: Bu Wati's session, played through the builder.*

```
    b.import_json(text)
    print("Imported again: " + b.kb.summary())
    print("Same file: " + str(b.export_json() == text).lower())


if __name__ == "__main__":
    main(sys.argv[1:])
```

`pairs` turns the text `problem = dry; baking time = long` into the list of pairs the builder expects. `teach` prints what the knowledge base said before and what happened, or why it refused. `consult` prints the conclusion and the explanation. **Listing 14.4** is the same script in Java.

Listing 14.4 `BuilderDemo.java`: the same session in Java.

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;

import vcirs.Condition;
import vcirs.GeneratedNode;
import vcirs.GeneratedRule;
import vcirs.InferenceResult;
import vcirs.Json;
import vcirs.Node;
import vcirs.Num;
import vcirs.Report;
import vcirs.Usage;
import vcirs.VcirsException;

/**
 * Bu Wati builds her cake doctor with the builder, step by step, the way
 * she would with the web app. Prints what each step does.
 * java BuilderDemo [knowledge-base.json]
 */
public class BuilderDemo {
    static final String DATA = "../data/";

    /** "problem = dry; baking time = long" -> {problem, dry}, ... */
    static List<String[]> pairs(String text) {
        List<String[]> out = new ArrayList<>();
        for (String part : text.split(";")) {
            String[] kv = part.split("=");
            out.add(new String[] {kv[0].strip(), kv[1].strip()});
        }
        return out;
    }
```

Listing 14.4 (Continued) *BuilderDemo.java: the same session in Java.*

```
    static void teach(Builder b, String facts, String answer, String teacher) {
        Builder.Taught t;
        try {
            t = b.teach(pairs(facts), answer, teacher);
        } catch (VcirsException e) {
            System.out.println("Teach " + facts + " -> " + answer + ": refused: "
                    + e.getMessage());
            return;
        }
        System.out.println("Teach " + facts + " -> " + answer + " (" + teacher
                + ")");
        System.out.println("  before: " + t.before().strip());
        System.out.println("  now:    " + t.result());
    }

    static void consult(Builder b, String facts) {
        InferenceResult result = b.consult(pairs(facts));
        System.out.println("Consult " + facts);
        List<String> lines = new ArrayList<>(result.what());
        lines.addAll(result.how());
        for (String line : lines) {
            System.out.println("  " + line);
        }
    }

    @SuppressWarnings("unchecked")
    public static void main(String[] args) throws IOException {
        String path = args.length > 0 ? args[0] : DATA + "cake-kb.json";
        Builder b = new Builder(path);
        Map<String, Object> example = (Map<String, Object>) Json.parse(
                Files.readString(Path.of(DATA + "cake-cases.json"),
                        StandardCharsets.UTF_8));
        System.out.println("== 1. A new project");
        b.newProject((String) example.get("name"),
                (String) example.get("description"));
        System.out.println(b.kb.summary());
        System.out.println("== 2. The questions");
        for (Object o : (List<Object>) example.get("variables")) {
            Map<String, Object> v = (Map<String, Object>) o;
            List<String> choices = Builder.strings((List<?>) v.get("choices"));
            b.declare((String) v.get("variable"), (String) v.get("question"),
                    String.join(", ", choices));
            System.out.println("Question " + v.get("variable") + ": "
                    + v.get("question") + " (" + String.join(" / ", choices)
                    + ")");
        }
```

Listing 14.4 (Continued) *BuilderDemo.java: the same session in Java.*

```
        System.out.println("== 3. Teaching, case by case");
        for (Object o : (List<Object>) example.get("cases")) {
            Map<String, Object> c = (Map<String, Object>) o;
            String answer = Builder.strings((List<?>) c.get("conclusions"))
                    .get(0).split("= ")[1];
            teach(b, String.join("; ", Builder.strings((List<?>) c.get("facts"))),
                    answer, (String) c.get("author"));
        }
        System.out.println("== 4. Consulting");
        consult(b, "problem = sank in the middle; oven door opened early = no; "
                + "oven heat = low");
        consult(b, "problem = burnt bottom");
        teach(b, "problem = burnt bottom", "bake on a higher rack", "Wati");
        teach(b, "problem = burnt bottom", "use a lighter pan", "Rini");
        teach(b, "problem = burnt bottom; pan = dark metal", "use a lighter pan",
                "Rini");
        System.out.println("== 5. The knowledge");
        for (String line : Report.tree(b.kb)) {
            System.out.println(line);
        }
        System.out.println("== 6. Usage statistics");
        Usage u = b.kb.usage();
        System.out.println("Ask first: "
                + String.join(", ", b.questionVariables()));
        System.out.println("Most important question: "
                + String.join(", ", u.importantVariables()));
        System.out.println("Most important nodes: "
                + String.join(", ", u.importantNodes()));
        for (Usage.RateRow row : u.nurRows()) {
            System.out.println("  NUR " + Num.fixed(row.rate, 3) + "  " + row.id);
        }
        System.out.println("== 7. Proposals");
        for (GeneratedRule g : b.kb.generateRules()) {
            System.out.println("Rule " + g.id + ": "
                    + String.join(" -> ", g.nodes));
        }
        for (GeneratedNode g : b.kb.generateNodes()) {
            List<String> conds = new ArrayList<>();
            for (Condition c : g.conditions) {
                conds.add(c.toString());
            }
            System.out.println("Node " + g.id + ": " + String.join(", ", conds));
        }
        Node node = b.acceptNode("take it out earlier#1G", List.of("problem = dry"),
                "Wati");
        System.out.println("Accepted: " + node);
```

Listing 14.4 (Continued) *BuilderDemo.java: the same session in Java.*

```
        consult(b, "problem = dry");
        System.out.println("== 8. Export and import");
        String text = b.exportJson();
        System.out.println("Exported " + text.split("\n").length
                + " lines of JSON");
        b.importJson(text);
        System.out.println("Imported again: " + b.kb.summary());
        System.out.println("Same file: " + b.exportJson().equals(text));
    }
}
```

Run it from `Chapter14/java` with `javac -d out -sourcepath ../../vcirs-core/java/src *.java` and `java -cp out BuilderDemo`, or from `Chapter14/python` with `python builder_demo.py`. The two print the same text and write the same `cake-kb.json`, byte for byte. We read the output in five parts. The first part builds the knowledge:

```
== 1. A new project
Cake doctor: 0 variables, 0 nodes, 0 rules
== 2. The questions
Question problem: What went wrong with the cake? (sank in the middle / dense and heavy /
cracked top / raw inside / dry / burnt bottom)
Question oven door opened early: Was the oven door opened in the first 20 minutes? (yes /
no)
Question oven heat: How hot was the oven? (low / normal / high)
Question eggs: Were the eggs cold or at room temperature? (cold / room temperature)
Question flour mixing: How long was the batter mixed after the flour went in? (short /
long)
Question baking time: Was the baking time short, normal or long? (short / normal / long)
== 3. Teaching, case by case
Teach problem = sank in the middle -> bake longer (Wati)
  before: No node matched the case.
  now:    added: bake longer#1 added at the top level
Teach problem = sank in the middle; oven door opened early = yes -> keep the door closed
(Wati)
  before: Conclusion: advice = bake longer  (rule bake longer#1, CF 100)
  now:    refined: keep the door closed#1 added as an exception of bake longer#1
Teach problem = dense and heavy -> beat the eggs longer (Wati)
  before: No node matched the case.
  now:    added: beat the eggs longer#1 added at the top level
Teach problem = dense and heavy; eggs = cold -> use room-temperature eggs (Wati)
  before: Conclusion: advice = beat the eggs longer  (rule beat the eggs longer#1, CF 100)
  now:    refined: use room-temperature eggs#1 added as an exception of beat the eggs
longer#1
Teach problem = dense and heavy; eggs = room temperature; flour mixing = long -> fold the
flour in gently (Rini)
  before: Conclusion: advice = beat the eggs longer  (rule beat the eggs longer#1, CF 100)
  now:    refined: fold the flour in gently#1 added as an exception of beat the eggs
longer#1
Teach problem = cracked top; oven heat = high -> lower the heat (Wati)
  before: No node matched the case.
  now:    added: lower the heat#1 added at the top level
Teach problem = raw inside; oven heat = high -> lower the heat and bake longer (Wati)
  before: No node matched the case.
  now:    added: lower the heat and bake longer#1 added at the top level
Teach problem = dry; baking time = long -> take it out earlier (Rini)
  before: No node matched the case.
  now:    added: take it out earlier#1 added at the top level
Teach problem = sank in the middle; oven door opened early = no; oven heat = low -> heat
the oven fully first (Wati)
  before: Conclusion: advice = bake longer  (rule bake longer#1, CF 100)
  now:    refined: heat the oven fully first#1 added as an exception of bake longer#1
```

Six questions, then nine cases, and for each case one line of “before” and one of “now”. Follow the first two. The first case meets an empty knowledge base — “No node matched the case” — and becomes the top-level node *bake longer#1*. The second case has the same problem plus `oven door opened early = yes`; before, the knowledge base would have said *bake longer*, which is wrong for this cake, so `learn` adds *keep the door closed#1* as an exception of *bake longer#1*. The same pattern repeats for the dense cakes: a general node, *beat the eggs longer*, and two exceptions under it, one for cold eggs and one, taught by Rini, for batter mixed too long after the flour. Nobody chose where each piece of knowledge went; the cases did.

The second part consults, and then teaches something new:

```
== 4. Consulting
Consult problem = sank in the middle; oven door opened early = no; oven heat = low
  Conclusion: advice = heat the oven fully first  (rule heat the oven fully first#1, CF
100)
    on the way: advice = bake longer
  heat the oven fully first#1 fired because:
    problem = sank in the middle  (given, CF 100)
    oven door opened early = no  (given, CF 100)
    oven heat = low  (given, CF 100)
    so advice = heat the oven fully first  [rule CF 100, conclusion CF 100]
Consult problem = burnt bottom
  No node matched the case.
  No rule fired.
Teach problem = burnt bottom -> bake on a higher rack (Wati)
  before: No node matched the case.
  now:    added: bake on a higher rack#1 added at the top level
Teach problem = burnt bottom -> use a lighter pan: refused: no condition tells the new case
apart; add a fact that differs from the cornerstone case
Teach problem = burnt bottom; pan = dark metal -> use a lighter pan (Rini)
  before: Conclusion: advice = bake on a higher rack  (rule bake on a higher rack#1, CF
100)
  now:    refined: use a lighter pan#1 added as an exception of bake on a higher rack#1
```

The first consultation walks down to the exception *heat the oven fully first* and explains it by the three facts that made it fire; `on the way` records the general advice it passed. The second consultation, a burnt bottom, finds nothing. Bu Wati teaches *bake on a higher rack*, and it becomes a new top-level node. Then Rini tries to teach a different answer, *use a lighter pan*, for exactly the same facts — and the builder refuses: the new case cannot be told apart from the case that created *bake on a higher rack*. That refusal is the list of difference of Chapter 3 at work: two different answers for identical facts would make the knowledge base contradict itself. Rini adds the fact that really was different — her cake was baked in a dark metal pan — and the case is accepted as an exception. The variable `pan` did not exist before; the engine created it on the spot.

The third part prints the knowledge as a tree:

```
== 5. The knowledge
(root)
+- bake longer#1: problem = sank in the middle -> advice = bake longer
|  +- keep the door closed#1: oven door opened early = yes -> advice = keep the door closed
|  +- heat the oven fully first#1: oven door opened early = no, oven heat = low -> advice =
heat the oven fully first
+- beat the eggs longer#1: problem = dense and heavy -> advice = beat the eggs longer
|  +- use room-temperature eggs#1: eggs = cold -> advice = use room-temperature eggs
|  +- fold the flour in gently#1: eggs = room temperature, flour mixing = long -> advice =
fold the flour in gently
+- lower the heat#1: problem = cracked top, oven heat = high -> advice = lower the heat
+- lower the heat and bake longer#1: problem = raw inside, oven heat = high -> advice =
lower the heat and bake longer
+- take it out earlier#1: problem = dry, baking time = long -> advice = take it out earlier
+- bake on a higher rack#1: problem = burnt bottom -> advice = bake on a higher rack
   +- use a lighter pan#1: pan = dark metal -> advice = use a lighter pan
```

The fourth part shows the statistics:

```
== 6. Usage statistics
Ask first: problem, oven heat, oven door opened early, eggs, flour mixing, baking time, pan
Most important question: problem
Most important nodes: bake longer#1, beat the eggs longer#1
  NUR 1.000  fold the flour in gently#1
  NUR 1.000  use a lighter pan#1
  NUR 2.000  keep the door closed#1
  NUR 2.000  use room-temperature eggs#1
  NUR 2.000  take it out earlier#1
  NUR 2.000  heat the oven fully first#1
  NUR 3.000  lower the heat#1
  NUR 3.000  lower the heat and bake longer#1
  NUR 12.000  bake on a higher rack#1
  NUR 18.000  bake longer#1
  NUR 18.000  beat the eggs longer#1
```

"Ask first" is the question order of Equation (14.1): `problem` first, because its best VUR is the largest, then `oven heat`, and so on, ending with the new `pan`. `problem` is also the most important question in the sense of node sharing, and the two most important nodes are the two general nodes that have exceptions: *bake longer#1* and *beat the eggs longer#1*, each shared by three rules. The NUR list, smallest first, ends with the values we computed by hand in Section 14.2.2: 18 for *bake longer#1*, 2 for *keep the door closed#1*.

The last part shows the proposals, accepts one, and exports:

```
== 7. Proposals
Rule bake longer#1G: take it out earlier#1 -> lower the heat and bake longer#1 -> lower the
heat#1 -> bake on a higher rack#1 -> use a lighter pan#1 -> beat the eggs longer#1 -> fold
the flour in gently#1 -> use room-temperature eggs#1 -> bake longer#1
Rule beat the eggs longer#1G: take it out earlier#1 -> lower the heat and bake longer#1 ->
lower the heat#1 -> bake on a higher rack#1 -> use a lighter pan#1 -> beat the eggs
longer#1
Node bake longer#1G: pan = dark metal, eggs = room temperature, flour mixing = long, eggs =
cold, oven door opened early = yes, oven door opened early = no, oven heat = low, problem =
dry, baking time = long, problem = raw inside, problem = cracked top, oven heat = high,
problem = burnt bottom, problem = dense and heavy, problem = sank in the middle
Node beat the eggs longer#1G: pan = dark metal, eggs = room temperature, flour mixing =
long, eggs = cold, oven door opened early = yes, oven door opened early = no, oven heat =
low, problem = dry, baking time = long, problem = raw inside, problem = cracked top, oven
heat = high, problem = burnt bottom, problem = dense and heavy
Node lower the heat#1G: pan = dark metal, eggs = room temperature, flour mixing = long,
eggs = cold, oven door opened early = yes, oven door opened early = no, oven heat = low,
problem = dry, baking time = long, problem = raw inside, problem = cracked top
Node lower the heat and bake longer#1G: pan = dark metal, eggs = room temperature, flour
mixing = long, eggs = cold, oven door opened early = yes, oven door opened early = no, oven
heat = low, problem = dry, baking time = long, problem = raw inside
Node take it out earlier#1G: pan = dark metal, eggs = room temperature, flour mixing =
long, eggs = cold, oven door opened early = yes, oven door opened early = no, oven heat =
low, problem = dry
Node bake on a higher rack#1G: pan = dark metal, eggs = room temperature, flour mixing =
long, eggs = cold, oven door opened early = yes, oven door opened early = no, oven heat =
low, problem = dry, baking time = long, problem = raw inside, problem = cracked top, oven
heat = high, problem = burnt bottom
Accepted: take it out earlier#1G: problem = dry -> advice = take it out earlier
Consult problem = dry
  Conclusion: advice = take it out earlier  (rule take it out earlier#1G, CF 100)
  take it out earlier#1G fired because:
    problem = dry  (given, CF 100)
    so advice = take it out earlier  [rule CF 100, conclusion CF 100]
== 8. Export and import
Exported 848 lines of JSON
Imported again: Cake doctor: 8 variables, 12 nodes, 12 rules
Same file: true
```

The two proposed rules and six proposed nodes are long; that is how the generation of VCIRS works, and why the person must choose [4]. The proposed node *take it out earlier#1G* lists eight conditions collected across the knowledge base, ending in `problem = dry`. Keeping only that one condition gives a new, general top-level node: a dry cake means *take it out earlier*, even when nobody knows how long it baked. The consultation that follows uses it at once. Finally, the export is 848 lines of JSON; importing it again gives the same knowledge base, and exporting that gives the same file, byte for byte.

14.5 Step 3: The Pages

A page of the builder is plain HTML: headings, a few tables, and forms with drop-down lists, text boxes and buttons. There is no JavaScript framework and no stylesheet file to download; the only script on any page is one line on the Home page that copies a chosen file into the import box. Keeping the pages this simple has three advantages for this chapter. They load instantly even on an old phone. They work in every browser. And they can be written twice, in two languages, with the same bytes.

Every page is placed into the frame of `page.html`, whose top looks like this:

```
<!DOCTYPE html>
<html lang="{{lang}}">
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>{{title}}</title>
```

and whose body is `<header><b>{{app}}</b><nav>{{nav}}</nav></header>` followed by `<main>{{notice}}{{body}}</main>`, with a short style sheet in between that uses the colors of this book. The pages fill the `{{...}}` places with plain text replacement. The `viewport` line tells a phone to show the page at its own width instead of shrinking a desktop page, so the builder is usable on Bu Wati's phone.

The words come from `ui-text.json`, which has one section for English and one for Indonesian with the same keys:

```
    "t_title": "Teach a case", "t_hint": "Describe a real case: choose the answers that
were true, leave the rest empty, and give the right answer. The builder puts the
knowledge where it belongs.",
```

and, in the Indonesian section, `"t_title": "Ajarkan sebuah kasus"` with its own hint. A page never contains a word typed in the code, except the names of the knowledge base's own variables and values, which belong to the person.

Listing 14.5 is the pages module in Python.

Listing 14.5 `pages.py`: the HTML pages, built from the knowledge base and the words of `ui-text.json`.

```
"""The HTML pages of the builder. Every page is plain HTML built from the
knowledge base; the words come from ui-text.json (English or Indonesian).
The Java version (Pages.java) builds exactly the same bytes.
"""

import json
import os

from vcirs import num, report

DATA = os.path.join(os.path.dirname(os.path.abspath(__file__)), "..", "data")
NAV = [("/", "nav_home"), ("/variables", "nav_variables"),
       ("/teach", "nav_teach"), ("/consult", "nav_consult"),
       ("/knowledge", "nav_knowledge"), ("/stats", "nav_stats"),
       ("/proposals", "nav_proposals")]


def esc(text):
    """Make text safe inside HTML."""
    return (str(text).replace("&", "&amp;").replace("<", "&lt;")
            .replace(">", "&gt;").replace('"', "&quot;")
            .replace("'", "&#39;"))


def url_encode(text):
    """Letters, digits and - _ . stay; a space becomes +; the rest %XX."""
    out = []
    for byte in text.encode("utf-8"):
        ch = chr(byte)
        if ch.isascii() and (ch.isalnum() or ch in "-_."):
            out.append(ch)
        elif ch == " ":
            out.append("+")
        else:
            out.append("%%%02X" % byte)
    return "".join(out)
```

Listing 14.5 (Continued) *pages.py: the HTML pages, built from the knowledge base and the words of ui-text.json.*

```
def table(headers, rows):
    lines = ["<table>", "<tr>" + "".join("<th>" + esc(h) + "</th>"
                                         for h in headers) + "</tr>"]
    for row in rows:
        lines.append("<tr>" + "".join("<td>" + esc(c) + "</td>"
                                      for c in row) + "</tr>")
    lines.append("</table>")
    return "\n".join(lines)


class Pages:
    def __init__(self, lang="en"):
        with open(os.path.join(DATA, "ui-text.json"), encoding="utf-8") as f:
            self.texts = json.load(f)
        with open(os.path.join(DATA, "page.html"), encoding="utf-8") as f:
            self.frame = f.read()
        self.lang = lang

    def t(self, key):
        return self.texts[self.lang][key]

    def page(self, here, body, notice="", error=False):
        nav = []
        for href, key in NAV:
            mark = ' class="here"' if href == here else ""
            nav.append('<a href="' + href + '"' + mark + ">"
                       + esc(self.t(key)) + "</a>")
        other = "id" if self.lang == "en" else "en"
        nav.append('<a href="/lang?to=' + other + '">'
                   + esc(self.t("other_lang")) + "</a>")
        box = ""
        if notice:
            box = ('<p class="error">' + esc(self.t("error_prefix")) + " "
                   if error else '<p class="notice">') + esc(notice) + "</p>\n"
        title = self.t("app") + " - " + self.t(dict(NAV)[here])
        return (self.frame.replace("{{lang}}", self.lang)
                .replace("{{title}}", esc(title))
                .replace("{{app}}", esc(self.t("app")))
                .replace("{{nav}}", "\n".join(nav))
                .replace("{{notice}}", box).replace("{{body}}", body))

    def field(self, label, name, value=""):
        return ("<label>" + esc(label) + '</label>\n<input type="text" name="'
                + name + '" value="' + esc(value) + '">')
```

Listing 14.5 (Continued) *pages.py: the HTML pages, built from the knowledge base and the words of ui-text.json.*

```
    def fact_selects(self, b, chosen):
        """One drop-down list per question; chosen maps name -> value."""
        out = []
        for var in b.question_variables():
            question = b.kb.variables[var].question or var
            out.append("<label>" + esc(question) + "</label>")
            out.append('<select name="f:' + esc(var) + '">')
            out.append('<option value=""></option>')
            for value in b.kb.known_values(var):
                mark = " selected" if chosen.get(var) == value else ""
                out.append('<option value="' + esc(value) + '"' + mark + ">"
                           + esc(value) + "</option>")
            out.append("</select>")
        return "\n".join(out)

    # ----- the pages ------------------------------------------------
    def home(self, b):
        kb = b.kb
        return "\n".join([
            "<h1>" + esc(kb.name) + "</h1>",
            "<p>" + esc(kb.description) + "</p>",
            '<p class="answer">' + str(len(b.question_variables())) + " "
            + esc(self.t("questions")) + ", " + str(len(kb.nodes)) + " "
            + esc(self.t("nodes")) + ", " + str(len(kb.rules)) + " "
            + esc(self.t("rules")) + "</p>",
            "<h2>" + esc(self.t("export")) + "</h2>",
            '<p class="hint">' + esc(self.t("export_hint")) + "</p>",
            '<p><a href="/export">' + esc(self.t("download")) + "</a></p>",
            "<h2>" + esc(self.t("import")) + "</h2>",
            '<p class="hint">' + esc(self.t("import_hint")) + "</p>",
            '<form method="post" action="/import">',
            '<input type="file" accept=".json" onchange="var r=new '
            'FileReader();r.onload=function(){document.getElementById('
            "'json').value=r.result};r.readAsText(this.files[0])\">",
            '<textarea id="json" name="json" rows="5"></textarea>',
            "<br><button>" + esc(self.t("load")) + "</button>",
            "</form>",
            "<h2>" + esc(self.t("new_project")) + "</h2>",
            '<p class="hint">' + esc(self.t("new_project_hint")) + "</p>",
            '<form method="post" action="/project">',
            self.field(self.t("name"), "name"),
            self.field(self.t("description"), "description"),
            "<button>" + esc(self.t("create")) + "</button>",
            "</form>"])
```

Listing 14.5 (Continued) *pages.py: the HTML pages, built from the knowledge base and the words of ui-text.json.*

```
    def variables(self, b):
        u = b.kb.usage()
        rows = [[v.id, v.question, ", ".join(b.kb.known_values(v.id)),
                 u.ns(v.id)] for v in b.kb.variables.values()]
        return "\n".join([
            "<h1>" + esc(self.t("q_title")) + "</h1>",
            '<p class="hint">' + esc(self.t("q_hint")) + "</p>",
            table([self.t("q_name"), self.t("q_text"), self.t("q_choices"),
                   self.t("q_used")], rows),
            '<form method="post" action="/variables">',
            self.field(self.t("q_name"), "variable"),
            self.field(self.t("q_text"), "question"),
            self.field(self.t("q_choices"), "choices"),
            "<button>" + esc(self.t("save")) + "</button>",
            "</form>"])

    def teach(self, b, chosen):
        return "\n".join([
            "<h1>" + esc(self.t("t_title")) + "</h1>",
            '<p class="hint">' + esc(self.t("t_hint")) + "</p>",
            '<form method="post" action="/teach">',
            self.fact_selects(b, chosen),
            "<label>" + esc(self.t("t_extra")) + "</label>",
            '<textarea name="extra" rows="2"></textarea>',
            self.field(self.t("t_answer") + " (" + b.goal() + ")", "answer"),
            self.field(self.t("t_teacher"), "teacher"),
            "<button>" + esc(self.t("t_button")) + "</button>",
            "</form>"])

    def taught(self, before, result):
        lines = ["<h1>" + esc(self.t("t_title")) + "</h1>",
                 "<p>" + esc(self.t("t_before")) + ":</p>",
                 "<pre>" + esc(before.strip()) + "</pre>",
                 "<p>" + esc(self.t("t_now")) + ":</p>",
                 '<p class="answer">' + esc(str(result)) + "</p>"]
        if result.differences is not None:
            lines.append("<p>" + esc(self.t("t_diff")) + ":</p>")
            lines.append("<pre>" + esc("\n".join(result.differences.lines()))
                         + "</pre>")
        lines.append('<p><a href="/teach">' + esc(self.t("again")) + "</a></p>")
        return "\n".join(lines)
```

Listing 14.5 (Continued) *pages.py: the HTML pages, built from the knowledge base and the words of ui-text.json.*

```
    def consult(self, b, chosen, result=None):
        lines = ["<h1>" + esc(self.t("c_title")) + "</h1>"]
        if result is not None:
            if result.conclusions():
                lines.append("<h2>" + esc(self.t("c_answer")) + "</h2>")
                lines += ['<p class="answer">' + esc(c.value) + "</p>"
                          for c in result.conclusions()]
                lines.append("<p>" + esc(self.t("c_how")) + ":</p>")
                lines.append("<pre>" + esc("\n".join(result.how())) + "</pre>")
            else:
                lines.append('<p class="answer">' + esc(self.t("c_none"))
                             + "</p>")
            query = "&".join(url_encode("f:" + v) + "=" + url_encode(x)
                             for v, x in chosen.items())
            lines.append('<p><a href="/teach?' + esc(query) + '">'
                         + esc(self.t("c_wrong")) + "</a></p>")
        lines += ['<p class="hint">' + esc(self.t("c_hint")) + "</p>",
                  '<form method="post" action="/consult">',
                  self.fact_selects(b, chosen),
                  "<button>" + esc(self.t("c_button")) + "</button>",
                  "</form>"]
        return "\n".join(lines)

    def knowledge(self, b):
        lines = ["<h1>" + esc(self.t("k_title")) + "</h1>",
                 "<h2>" + esc(self.t("k_tree")) + "</h2>",
                 "<pre>" + esc("\n".join(report.tree(b.kb))) + "</pre>",
                 "<table>", "<tr><th>" + esc(self.t("k_node")) + "</th><th>"
                 + esc(self.t("k_when")) + "</th><th>" + esc(self.t("k_then"))
                 + "</th><th>" + esc(self.t("k_by")) + "</th><th></th></tr>"]
        for n in b.kb.nodes.values():
            if n.stopped:
                action = esc(self.t("k_stopped"))
            else:
                action = ('<form method="post" action="/stop">'
                          '<input type="hidden" name="node" value="'
                          + esc(n.id) + '"><button>' + esc(self.t("k_stop"))
                          + "</button></form>")
            lines.append("<tr><td>" + esc(n.id) + "</td><td>"
                         + esc(", ".join(str(c) for c in n.conditions))
                         + "</td><td>"
                         + esc(", ".join(str(c) for c in n.conclusions))
                         + "</td><td>" + esc(n.author) + "</td><td>" + action
                         + "</td></tr>")
        lines.append("</table>")
        lines.append("<h2>" + esc(self.t("k_history")) + "</h2>")
```

Listing 14.5 (Continued) *pages.py: the HTML pages, built from the knowledge base and the words of ui-text.json.*

```
        lines.append(table(["#", self.t("k_action"), self.t("k_node"),
                           self.t("k_option"), self.t("k_by")],
                          [[h["seq"], h["action"], h["node"], h["option"],
                            h["author"]] for h in b.kb.history]))
        return "\n".join(lines)

    def stats(self, b):
        u = b.kb.usage()
        none = self.t("s_none")
        return "\n".join([
            "<h1>" + esc(self.t("s_title")) + "</h1>",
            '<p class="hint">' + esc(self.t("s_hint")) + "</p>",
            "<h2>" + esc(self.t("s_order")) + "</h2>",
            table([self.t("s_order_col"), self.t("q_name"), self.t("s_ns")],
                  [[i + 1, v, u.ns(v)]
                   for i, v in enumerate(b.question_variables())]),
            "<p>" + esc(self.t("s_important")) + ": "
            + esc(", ".join(u.important_variables()) or none) + "</p>",
            "<p>" + esc(self.t("s_important_nodes")) + ": "
            + esc(", ".join(u.important_nodes()) or none) + "</p>",
            "<h2>" + esc(self.t("s_nur")) + "</h2>",
            table([self.t("k_node"), self.t("s_count"), self.t("s_rate")],
                  [[n, k, num.fixed(x, 3)]
                   for n, k, x in reversed(u.nur_rows())]),
            "<h2>" + esc(self.t("s_rur")) + "</h2>",
            table([self.t("s_rule"), self.t("s_rate")],
                  [[r, num.fixed(x, 3)] for r, x in reversed(u.rur_rows())]),
            "<h2>" + esc(self.t("s_vur")) + "</h2>",
            table([self.t("q_name"), self.t("k_node"), self.t("s_order_col"),
                   self.t("s_rate")],
                  [[v, n, o, num.fixed(x, 3)]
                   for v, n, o, x in reversed(u.vur_rows())])])

    def proposals(self, b):
        rules, nodes = b.kb.generate_rules(), b.kb.generate_nodes()
        lines = ["<h1>" + esc(self.t("p_title")) + "</h1>",
                 '<p class="hint">' + esc(self.t("p_hint")) + "</p>"]
        if not rules and not nodes:
            lines.append("<p>" + esc(self.t("p_none")) + "</p>")
        if rules:
            lines.append("<h2>" + esc(self.t("p_rules")) + "</h2>")
        for g in rules:
            lines += ['<form method="post" action="/accept-rule">',
                      '<input type="hidden" name="id" value="' + esc(g.id)
                      + '">', "<p><b>" + esc(g.id) + "</b>: "
                      + esc(" -> ".join(g.nodes)) + "</p>",
```

Listing 14.5 (Continued) *pages.py: the HTML pages, built from the knowledge base and the words of ui-text.json.*

```
                "<button>" + esc(self.t("p_accept")) + "</button>",
                "</form>"]
        if nodes:
            lines.append("<h2>" + esc(self.t("p_nodes")) + "</h2>")
        for g in nodes:
            lines += ['<form method="post" action="/accept-node">',
                      '<input type="hidden" name="id" value="' + esc(g.id)
                      + '">', "<p><b>" + esc(g.id) + "</b> &rarr; "
                      + esc(", ".join(str(c) for c in g.conclusions))
                      + "<br>" + esc(self.t("p_keep")) + ":</p>"]
            for c in g.conditions:
                lines.append('<label><input type="checkbox" name="keep" '
                             'value="' + esc(c.key()) + '"> ' + esc(c.key())
                             + "</label>")
            lines += ["<button>" + esc(self.t("p_accept")) + "</button>",
                      "</form>"]
        return "\n".join(lines)
```

Three small functions come first, and each exists for a reason.

`esc` makes any text safe to put inside HTML by replacing the five characters that have a meaning in HTML — `&`, `<`, `>`, `"` and `'` — with their escaped forms. Every piece of text that comes from the person or from the knowledge base passes through it. Without it, a question named `<b>eggs</b>` would appear in bold, and a malicious value could inject a script into the page of everyone who opens the builder: the attack called *cross-site scripting*, against which escaping every output is the standard defense [7].

`url_encode` writes the chosen answers into the link “Not right? Teach the right answer for this case” on the Consult page, so that the Teach page opens with the same answers already chosen. It encodes every byte except letters, digits and `- _ .`, and a space as `+`. Java and Python have their own URL encoders, and they differ in small ways (for example in whether `~` or `*` is encoded); writing our own few lines makes the two languages produce the same link.

`table` turns a header and rows into an HTML table, escaping every cell.

The class `Pages` then holds the language and the loaded texts. `page` puts a body into the frame: it builds the navigation, marking the current page, adds the link to the other language, and shows the message left by the last change — green for “Saved.”, red with “Sorry:” for an error. `fact_selects` builds one drop-down list per question, in the order of `question_variables`, with an empty first choice that means “I do not know”; the chosen answers stay selected when the page is shown again.

The page methods follow **Table 14.1**. `home` shows the project’s name, description and size, the export link, the import form (a file chooser whose one line of script copies the file’s text into the text box, and a **Load** button), and the form for a new project. `variables` lists the questions with their usual answers and how many nodes use each, and offers a form for a new question. `teach` is the case form: the questions, a box for other facts typed as `name = value`, the right answer, and the teacher’s name. `taught` shows what happened, and the list of difference when the case became an exception. `consult` shows the advice and its explanation above the form, and the link that sends the case to the Teach page. `knowledge` shows the tree, a table of nodes with a **Stop** button for each node still active, and the history. `stats` shows the question order with node sharing, the most important questions and nodes, and the NUR, RUR and VUR tables, largest first. `proposals` shows each proposed rule with an **Accept** button and each proposed node with a checkbox per condition.

Listing 14.6 is the same module in Java.

Listing 14.6 `Pages.java`: the same pages in Java, byte for byte.

```java
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Map;

import vcirs.Condition;
import vcirs.Conclusion;
import vcirs.GeneratedNode;
import vcirs.GeneratedRule;
import vcirs.InferenceResult;
import vcirs.Json;
import vcirs.LearnResult;
import vcirs.Node;
import vcirs.Num;
import vcirs.Report;
import vcirs.Usage;
import vcirs.Variable;

/**
 * The HTML pages of the builder. Every page is plain HTML built from the
 * knowledge base; the words come from ui-text.json (English or Indonesian).
 * The Python version (pages.py) builds exactly the same bytes.
 */
public class Pages {
    static final String DATA = "../data/";
    static final String[][] NAV = {{"/", "nav_home"},
        {"/variables", "nav_variables"}, {"/teach", "nav_teach"},
        {"/consult", "nav_consult"}, {"/knowledge", "nav_knowledge"},
        {"/stats", "nav_stats"}, {"/proposals", "nav_proposals"}};

    /** Make text safe inside HTML. */
    static String esc(Object text) {
        return String.valueOf(text).replace("&", "&amp;").replace("<", "&lt;")
                .replace(">", "&gt;").replace("\"", "&quot;")
                .replace("'", "&#39;");
    }
```

Listing 14.6 (Continued) *Pages.java: the same pages in Java, byte for byte.*

```
    /** Letters, digits and - _ . stay; a space becomes +; the rest %XX. */
    static String urlEncode(String text) {
        StringBuilder out = new StringBuilder();
        for (byte b : text.getBytes(StandardCharsets.UTF_8)) {
            char ch = (char) (b & 0xff);
            boolean plain = Character.isLetterOrDigit(ch)
                    || "-_.".indexOf(ch) >= 0;
            if (ch < 128 && plain) {
                out.append(ch);
            } else if (ch == ' ') {
                out.append('+');
            } else {
                out.append(String.format("%%%02X", b & 0xff));
            }
        }
        return out.toString();
    }

    static String table(List<String> headers, List<List<Object>> rows) {
        List<String> lines = new ArrayList<>();
        lines.add("<table>");
        StringBuilder head = new StringBuilder("<tr>");
        headers.forEach(h -> head.append("<th>").append(esc(h)).append("</th>"));
        lines.add(head + "</tr>");
        for (List<Object> row : rows) {
            StringBuilder r = new StringBuilder("<tr>");
            row.forEach(c -> r.append("<td>").append(esc(c)).append("</td>"));
            lines.add(r + "</tr>");
        }
        lines.add("</table>");
        return String.join("\n", lines);
    }

    final Map<String, Map<String, String>> texts;
    final String frame;
    String lang;
```

Listing 14.6 (Continued) *Pages.java: the same pages in Java, byte for byte.*

```
    @SuppressWarnings("unchecked")
    Pages(String lang) throws IOException {
        texts = (Map<String, Map<String, String>>) (Object) Json.parse(
                Files.readString(Path.of(DATA + "ui-text.json"),
                        StandardCharsets.UTF_8));
        frame = Files.readString(Path.of(DATA + "page.html"),
                StandardCharsets.UTF_8);
        this.lang = lang;
    }

    String t(String key) {
        return texts.get(lang).get(key);
    }

    String page(String here, String body, String notice, boolean error) {
        List<String> nav = new ArrayList<>();
        String title = "";
        for (String[] item : NAV) {
            String mark = item[0].equals(here) ? " class=\"here\"" : "";
            nav.add("<a href=\"" + item[0] + "\"" + mark + ">" + esc(t(item[1]))
                    + "</a>");
            if (item[0].equals(here)) {
                title = t("app") + " - " + t(item[1]);
            }
        }
        String other = lang.equals("en") ? "id" : "en";
        nav.add("<a href=\"/lang?to=" + other + "\">" + esc(t("other_lang"))
                + "</a>");
        String box = "";
        if (!notice.isEmpty()) {
            box = (error ? "<p class=\"error\">" + esc(t("error_prefix")) + " "
                    : "<p class=\"notice\">") + esc(notice) + "</p>\n";
        }
        return frame.replace("{{lang}}", lang).replace("{{title}}", esc(title))
                .replace("{{app}}", esc(t("app")))
                .replace("{{nav}}", String.join("\n", nav))
                .replace("{{notice}}", box).replace("{{body}}", body);
    }

    String field(String label, String name, String value) {
        return "<label>" + esc(label) + "</label>\n<input type=\"text\" name=\""
                + name + "\" value=\"" + esc(value) + "\">";
    }
```

Listing 14.6 (Continued) *Pages.java: the same pages in Java, byte for byte.*

```
    /** One drop-down list per question; chosen maps name -> value. */
    String factSelects(Builder b, Map<String, String> chosen) {
        List<String> out = new ArrayList<>();
        for (String var : b.questionVariables()) {
            String q = b.kb.variables.get(var).question;
            out.add("<label>" + esc(q.isEmpty() ? var : q) + "</label>");
            out.add("<select name=\"f:" + esc(var) + "\">");
            out.add("<option value=\"\"></option>");
            for (String value : b.kb.knownValues(var)) {
                String mark = value.equals(chosen.get(var)) ? " selected" : "";
                out.add("<option value=\"" + esc(value) + "\"" + mark + ">"
                        + esc(value) + "</option>");
            }
            out.add("</select>");
        }
        return String.join("\n", out);
    }

    static String joined(List<?> items, String sep) {
        List<String> parts = new ArrayList<>();
        items.forEach(x -> parts.add(String.valueOf(x)));
        return String.join(sep, parts);
    }

    // ----- the pages -------------------------------------------------------
    String home(Builder b) {
        return String.join("\n", List.of(
            "<h1>" + esc(b.kb.name) + "</h1>",
            "<p>" + esc(b.kb.description) + "</p>",
            "<p class=\"answer\">" + b.questionVariables().size() + " "
            + esc(t("questions")) + ", " + b.kb.nodes.size() + " "
            + esc(t("nodes")) + ", " + b.kb.rules.size() + " "
            + esc(t("rules")) + "</p>",
            "<h2>" + esc(t("export")) + "</h2>",
            "<p class=\"hint\">" + esc(t("export_hint")) + "</p>",
            "<p><a href=\"/export\">" + esc(t("download")) + "</a></p>",
            "<h2>" + esc(t("import")) + "</h2>",
            "<p class=\"hint\">" + esc(t("import_hint")) + "</p>",
            "<form method=\"post\" action=\"/import\">",
            "<input type=\"file\" accept=\".json\" onchange=\"var r=new "
            + "FileReader();r.onload=function(){document.getElementById("
            + "'json').value=r.result};r.readAsText(this.files[0])\">",
            "<textarea id=\"json\" name=\"json\" rows=\"5\"></textarea>",
            "<br><button>" + esc(t("load")) + "</button>",
            "</form>",
            "<h2>" + esc(t("new_project")) + "</h2>",
```

Listing 14.6 (Continued) *Pages.java: the same pages in Java, byte for byte.*

```
            "<p class=\"hint\">" + esc(t("new_project_hint")) + "</p>",
            "<form method=\"post\" action=\"/project\">",
            field(t("name"), "name", ""),
            field(t("description"), "description", ""),
            "<button>" + esc(t("create")) + "</button>",
            "</form>"));
    }

    String variables(Builder b) {
        Usage u = b.kb.usage();
        List<List<Object>> rows = new ArrayList<>();
        for (Variable v : b.kb.variables.values()) {
            rows.add(List.of(v.id, v.question,
                    String.join(", ", b.kb.knownValues(v.id)), u.ns(v.id)));
        }
        return String.join("\n", List.of(
            "<h1>" + esc(t("q_title")) + "</h1>",
            "<p class=\"hint\">" + esc(t("q_hint")) + "</p>",
            table(List.of(t("q_name"), t("q_text"), t("q_choices"), t("q_used")),
                    rows),
            "<form method=\"post\" action=\"/variables\">",
            field(t("q_name"), "variable", ""),
            field(t("q_text"), "question", ""),
            field(t("q_choices"), "choices", ""),
            "<button>" + esc(t("save")) + "</button>",
            "</form>"));
    }

    String teach(Builder b, Map<String, String> chosen) {
        return String.join("\n", List.of(
            "<h1>" + esc(t("t_title")) + "</h1>",
            "<p class=\"hint\">" + esc(t("t_hint")) + "</p>",
            "<form method=\"post\" action=\"/teach\">",
            factSelects(b, chosen),
            "<label>" + esc(t("t_extra")) + "</label>",
            "<textarea name=\"extra\" rows=\"2\"></textarea>",
            field(t("t_answer") + " (" + b.goal() + ")", "answer", ""),
            field(t("t_teacher"), "teacher", ""),
            "<button>" + esc(t("t_button")) + "</button>",
            "</form>"));
    }
```

Listing 14.6 (Continued) *Pages.java: the same pages in Java, byte for byte.*

```
    String taught(String before, LearnResult result) {
        List<String> lines = new ArrayList<>(List.of(
            "<h1>" + esc(t("t_title")) + "</h1>",
            "<p>" + esc(t("t_before")) + ":</p>",
            "<pre>" + esc(before.strip()) + "</pre>",
            "<p>" + esc(t("t_now")) + ":</p>",
            "<p class=\"answer\">" + esc(result) + "</p>"));
        if (result.differences != null) {
            lines.add("<p>" + esc(t("t_diff")) + ":</p>");
            lines.add("<pre>" + esc(String.join("\n", result.differences.lines()))
                    + "</pre>");
        }
        lines.add("<p><a href=\"/teach\">" + esc(t("again")) + "</a></p>");
        return String.join("\n", lines);
    }

    String consult(Builder b, Map<String, String> chosen, InferenceResult result) {
        List<String> lines = new ArrayList<>();
        lines.add("<h1>" + esc(t("c_title")) + "</h1>");
        if (result != null) {
            if (!result.conclusions().isEmpty()) {
                lines.add("<h2>" + esc(t("c_answer")) + "</h2>");
                for (Conclusion c : result.conclusions()) {
                    lines.add("<p class=\"answer\">" + esc(c.value) + "</p>");
                }
                lines.add("<p>" + esc(t("c_how")) + ":</p>");
                lines.add("<pre>" + esc(String.join("\n", result.how()))
                        + "</pre>");
            } else {
                lines.add("<p class=\"answer\">" + esc(t("c_none")) + "</p>");
            }
            List<String> query = new ArrayList<>();
            chosen.forEach((v, x) -> query.add(urlEncode("f:" + v) + "="
                    + urlEncode(x)));
            lines.add("<p><a href=\"/teach?" + esc(String.join("&", query))
                    + "\">" + esc(t("c_wrong")) + "</a></p>");
        }
        lines.add("<p class=\"hint\">" + esc(t("c_hint")) + "</p>");
        lines.add("<form method=\"post\" action=\"/consult\">");
        lines.add(factSelects(b, chosen));
        lines.add("<button>" + esc(t("c_button")) + "</button>");
        lines.add("</form>");
        return String.join("\n", lines);
    }
```

Listing 14.6 (Continued) *Pages.java: the same pages in Java, byte for byte.*

```
    String knowledge(Builder b) {
        List<String> lines = new ArrayList<>(List.of(
            "<h1>" + esc(t("k_title")) + "</h1>",
            "<h2>" + esc(t("k_tree")) + "</h2>",
            "<pre>" + esc(String.join("\n", Report.tree(b.kb))) + "</pre>",
            "<table>", "<tr><th>" + esc(t("k_node")) + "</th><th>"
            + esc(t("k_when")) + "</th><th>" + esc(t("k_then"))
            + "</th><th>" + esc(t("k_by")) + "</th><th></th></tr>"));
        for (Node n : b.kb.nodes.values()) {
            String action = n.stopped ? esc(t("k_stopped"))
                    : "<form method=\"post\" action=\"/stop\">"
                    + "<input type=\"hidden\" name=\"node\" value=\""
                    + esc(n.id) + "\"><button>" + esc(t("k_stop"))
                    + "</button></form>";
            lines.add("<tr><td>" + esc(n.id) + "</td><td>"
                    + esc(joined(n.conditions, ", ")) + "</td><td>"
                    + esc(joined(n.conclusions, ", ")) + "</td><td>"
                    + esc(n.author) + "</td><td>" + action + "</td></tr>");
        }
        lines.add("</table>");
        lines.add("<h2>" + esc(t("k_history")) + "</h2>");
        List<List<Object>> rows = new ArrayList<>();
        for (Map<String, Object> h : b.kb.history) {
            rows.add(List.of(h.get("seq"), h.get("action"), h.get("node"),
                    h.get("option"), h.get("author")));
        }
        lines.add(table(List.of("#", t("k_action"), t("k_node"), t("k_option"),
                t("k_by")), rows));
        return String.join("\n", lines);
    }

    String stats(Builder b) {
        Usage u = b.kb.usage();
        String none = t("s_none");
        List<List<Object>> order = new ArrayList<>();
        List<String> vars = b.questionVariables();
        for (int i = 0; i < vars.size(); i++) {
            order.add(List.of(i + 1, vars.get(i), u.ns(vars.get(i))));
        }
        List<List<Object>> nur = new ArrayList<>();
        for (Usage.RateRow r : u.nurRows()) {
            nur.add(List.of(r.id, r.size, Num.fixed(r.rate, 3)));
        }
        List<List<Object>> rur = new ArrayList<>();
        for (Usage.RateRow r : u.rurRows()) {
            rur.add(List.of(r.id, Num.fixed(r.rate, 3)));
```

Listing 14.6 (Continued) *Pages.java: the same pages in Java, byte for byte.*

```
        }
        List<List<Object>> vur = new ArrayList<>();
        for (Usage.VurRow r : u.vurRows()) {
            vur.add(List.of(r.variable, r.node, r.order, Num.fixed(r.vur, 3)));
        }
        Collections.reverse(nur);
        Collections.reverse(rur);
        Collections.reverse(vur);
        String important = String.join(", ", u.importantVariables());
        String importantNodes = String.join(", ", u.importantNodes());
        return String.join("\n", List.of(
            "<h1>" + esc(t("s_title")) + "</h1>",
            "<p class=\"hint\">" + esc(t("s_hint")) + "</p>",
            "<h2>" + esc(t("s_order")) + "</h2>",
            table(List.of(t("s_order_col"), t("q_name"), t("s_ns")), order),
            "<p>" + esc(t("s_important")) + ": "
            + esc(important.isEmpty() ? none : important) + "</p>",
            "<p>" + esc(t("s_important_nodes")) + ": "
            + esc(importantNodes.isEmpty() ? none : importantNodes) + "</p>",
            "<h2>" + esc(t("s_nur")) + "</h2>",
            table(List.of(t("k_node"), t("s_count"), t("s_rate")), nur),
            "<h2>" + esc(t("s_rur")) + "</h2>",
            table(List.of(t("s_rule"), t("s_rate")), rur),
            "<h2>" + esc(t("s_vur")) + "</h2>",
            table(List.of(t("q_name"), t("k_node"), t("s_order_col"),
                    t("s_rate")), vur)));
    }

    String proposals(Builder b) {
        List<GeneratedRule> rules = b.kb.generateRules();
        List<GeneratedNode> nodes = b.kb.generateNodes();
        List<String> lines = new ArrayList<>();
        lines.add("<h1>" + esc(t("p_title")) + "</h1>");
        lines.add("<p class=\"hint\">" + esc(t("p_hint")) + "</p>");
        if (rules.isEmpty() && nodes.isEmpty()) {
            lines.add("<p>" + esc(t("p_none")) + "</p>");
        }
        if (!rules.isEmpty()) {
            lines.add("<h2>" + esc(t("p_rules")) + "</h2>");
        }
        for (GeneratedRule g : rules) {
            lines.add("<form method=\"post\" action=\"/accept-rule\">");
            lines.add("<input type=\"hidden\" name=\"id\" value=\"" + esc(g.id)
                    + "\">");
            lines.add("<p><b>" + esc(g.id) + "</b>: "
                    + esc(String.join(" -> ", g.nodes)) + "</p>");
```

Listing 14.6 (Continued) *Pages.java: the same pages in Java, byte for byte.*

```
            lines.add("<button>" + esc(t("p_accept")) + "</button>");
            lines.add("</form>");
        }
        if (!nodes.isEmpty()) {
            lines.add("<h2>" + esc(t("p_nodes")) + "</h2>");
        }
        for (GeneratedNode g : nodes) {
            lines.add("<form method=\"post\" action=\"/accept-node\">");
            lines.add("<input type=\"hidden\" name=\"id\" value=\"" + esc(g.id)
                    + "\">");
            lines.add("<p><b>" + esc(g.id) + "</b> &rarr; "
                    + esc(joined(g.conclusions, ", ")) + "<br>"
                    + esc(t("p_keep")) + ":</p>");
            for (Condition c : g.conditions) {
                lines.add("<label><input type=\"checkbox\" name=\"keep\" value=\""
                        + esc(c.key()) + "\"> " + esc(c.key()) + "</label>");
            }
            lines.add("<button>" + esc(t("p_accept")) + "</button>");
            lines.add("</form>");
        }
        return String.join("\n", lines);
    }
}
```

Writing two programs that produce the same bytes asks for care in small places, and the pages show most of them. Both languages build each page as a list of lines joined with `\n`. Both escape with the same five replacements in the same order (the `&` first, so that the `&` of `<` is not escaped again). Numbers are written with `num.fixed` / `Num.fixed`, the engine's shared formatter, never with the language's own. Tables are reversed with `reversed(...)` in Python and `Collections.reverse` in Java, from lists that the engine sorts the same way in both. And maps keep the order in which keys were added: a Python `dict` does so by definition, and in Java we use `LinkedHashMap` everywhere.

14.6 Step 4: The Web Servers

The servers are the thinnest part. **Listing 14.7** is the Python server with **Flask**, a small and widely used web framework for Python; its current version, 3.1.3, was released in February 2026 [8].

Listing 14.7 `builder_app.py`: the builder's web server in Flask.

```
"""The VCIRS expert-system builder as a small web app (Flask).

    python builder_app.py [knowledge-base.json] [--port 4141]

Then open http://localhost:4141 in a browser. Everything the pages show
comes from builder.py and pages.py; this file only connects them to the web.
"""

import os
import sys

from flask import Flask, Response, redirect, request

from builder import Builder
from pages import DATA, Pages
from vcirs import VcirsError

app = Flask(__name__)
state = {"builder": None, "pages": Pages("en"), "notice": "", "error": False}


def show(here, body):
    """A whole page, with the message left by the last change (once)."""
    html = state["pages"].page(here, body, state["notice"], state["error"])
    state["notice"], state["error"] = "", False
    return Response(html, content_type="text/html; charset=utf-8")


def done(where, error=None):
    """After a change: remember what to tell the person, and go on."""
    state["notice"] = str(error) if error else state["pages"].t("saved")
    state["error"] = error is not None
    return redirect(where, code=303)


def chosen_facts(values):
    """The f:name fields that were filled in, as a dict name -> value."""
    return {k[2:]: v.strip() for k, v in values.items()
            if k.startswith("f:") and v.strip()}
```

Listing 14.7 (Continued) *builder_app.py: the builder's web server in Flask.*

```
def extra_facts(text):
    """'name = value' lines typed by hand."""
    out = []
    for line in text.splitlines():
        if "=" in line:
            var, value = line.split("=", 1)
            if var.strip() and value.strip():
                out.append((var.strip(), value.strip()))
    return out


@app.get("/")
def home():
    return show("/", state["pages"].home(state["builder"]))


@app.get("/variables")
def variables():
    return show("/variables", state["pages"].variables(state["builder"]))


@app.get("/teach")
def teach_form():
    b = state["builder"]
    return show("/teach", state["pages"].teach(b, chosen_facts(request.args)))


@app.get("/consult")
def consult_form():
    return show("/consult", state["pages"].consult(state["builder"], {}))


@app.get("/knowledge")
def knowledge():
    return show("/knowledge", state["pages"].knowledge(state["builder"]))


@app.get("/stats")
def stats():
    return show("/stats", state["pages"].stats(state["builder"]))


@app.get("/proposals")
def proposals():
    return show("/proposals", state["pages"].proposals(state["builder"]))
```

Listing 14.7 (Continued) *builder_app.py: the builder's web server in Flask.*

```
@app.get("/export")
def export():
    return Response(state["builder"].export_json(),
                    content_type="application/json; charset=utf-8",
                    headers={"Content-Disposition":
                             'attachment; filename="knowledge-base.json"'})


@app.get("/lang")
def language():
    if request.args.get("to") in ("en", "id"):
        state["pages"].lang = request.args["to"]
    return redirect("/", code=303)


@app.post("/project")
def project():
    try:
        state["builder"].new_project(request.form.get("name", ""),
                                     request.form.get("description", ""))
    except VcirsError as e:
        return done("/", e)
    return done("/")


@app.post("/import")
def import_json():
    try:
        state["builder"].import_json(request.form.get("json", ""))
    except VcirsError as e:
        return done("/", e)
    return done("/")


@app.post("/variables")
def declare():
    f = request.form
    try:
        state["builder"].declare(f.get("variable", ""), f.get("question", ""),
                                 f.get("choices", ""))
    except VcirsError as e:
        return done("/variables", e)
    return done("/variables")
```

Listing 14.7 (Continued) *builder_app.py: the builder's web server in Flask.*

```
@app.post("/teach")
def teach():
    b, f = state["builder"], request.form
    pairs = list(chosen_facts(f).items()) + extra_facts(f.get("extra", ""))
    try:
        before, result = b.teach(pairs, f.get("answer", ""),
                                 f.get("teacher", ""))
    except VcirsError as e:
        state["notice"], state["error"] = str(e), True
        return show("/teach", state["pages"].teach(b, chosen_facts(f)))
    return show("/teach", state["pages"].taught(before, result))


@app.post("/consult")
def consult():
    b = state["builder"]
    chosen = chosen_facts(request.form)
    result = b.consult(list(chosen.items()))
    return show("/consult", state["pages"].consult(b, chosen, result))


@app.post("/stop")
def stop():
    try:
        state["builder"].stop(request.form.get("node", ""), "")
    except VcirsError as e:
        return done("/knowledge", e)
    return done("/knowledge")


@app.post("/accept-rule")
def accept_rule():
    try:
        state["builder"].accept_rule(request.form.get("id", ""), "")
    except VcirsError as e:
        return done("/proposals", e)
    return done("/proposals")
```

Listing 14.7 (Continued) *builder_app.py: the builder's web server in Flask.*

```
@app.post("/accept-node")
def accept_node():
    f = request.form
    try:
        state["builder"].accept_node(f.get("id", ""), f.getlist("keep"), "")
    except VcirsError as e:
        return done("/proposals", e)
    return done("/proposals")


def main(args):
    port = int(args[args.index("--port") + 1]) if "--port" in args else 4141
    files = [a for a in args if a.endswith(".json")]
    state["builder"] = Builder(files[0] if files
                               else os.path.join(DATA, "builder-kb.json"))
    print("VCIRS Builder: open http://localhost:" + str(port)
          + " in your browser; Ctrl+C stops it.", flush=True)
    app.run(host="127.0.0.1", port=port, threaded=False)  # one at a time


if __name__ == "__main__":
    main(sys.argv[1:])
```

A Flask application is a set of functions, each marked with the address and method it answers: `@app.get("/stats")` answers the browser's request for the Statistics page, `@app.post("/teach")` answers the Teach form. `request.form` holds the fields of a submitted form, `request.args` those of a link's query string; `getlist("keep")` returns every ticked checkbox named `keep`.

The server keeps its whole state in one dictionary: the builder, the pages (with the chosen language), and a *notice* — the message to show once on the next page. Two helpers do all the talking. `show` wraps a body in the frame, with the notice, and clears the notice so that it appears only once. `done` is called after a change: it sets the notice to "Saved." or to the error's sentence and **redirects** the browser, with status 303, to the page it should see next. This pattern — *post, redirect, get* — means that after saving a question, the browser shows the Questions page through a fresh `GET`; if Bu Wati presses the browser's reload button, the page is shown again, but the question is not saved twice. The Teach and Consult forms are different: their result *is* a page (what happened, or the advice), so they show it directly.

`chosen_facts` picks the fields whose names start with `f:` — the drop-down lists — and keeps the non-empty ones; `extra_facts` reads the box of other facts, one `name = value` per line. `main` reads the file name and port from the command line, opens the builder, and starts Flask on `127.0.0.1`, which means *this computer only*.

Listing 14.8 is the Java server with `com.sun.net.httpserver`, the small web server that has been part of the JDK for many years and is documented with the JDK 25 API [9].

Listing 14.8 `BuilderServer.java`: the same server with the JDK's own HttpServer.

```
import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.net.URLDecoder;
import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;

import vcirs.InferenceResult;
import vcirs.VcirsException;

/**
 * The VCIRS expert-system builder as a small web app (the JDK's own HTTP
 * server). java BuilderServer [knowledge-base.json] [--port 4141]
 * Then open http://localhost:4141 in a browser. Everything the pages show
 * comes from Builder and Pages; this class only connects them to the web.
 */
public class BuilderServer {
    static Builder builder;
    static Pages pages;
    static String notice = "";
    static boolean error = false;

    /** "a=1&b=x+y" -> {a: [1], b: [x y]}, in the order given. */
    static Map<String, List<String>> form(String text) {
        Map<String, List<String>> out = new LinkedHashMap<>();
        if (text == null || text.isEmpty()) {
            return out;
        }
        for (String part : text.split("&")) {
            int eq = part.indexOf('=');
            String k = eq < 0 ? part : part.substring(0, eq);
            String v = eq < 0 ? "" : part.substring(eq + 1);
            out.computeIfAbsent(URLDecoder.decode(k, StandardCharsets.UTF_8),
                    x -> new ArrayList<>())
                    .add(URLDecoder.decode(v, StandardCharsets.UTF_8));
        }
        return out;
    }
```

Listing 14.8 (Continued) *BuilderServer.java: the same server with the JDK's own HttpServer.*

```
    static String get(Map<String, List<String>> f, String key) {
        return f.containsKey(key) ? f.get(key).get(0) : "";
    }

    /** The f:name fields that were filled in, as name -> value. */
    static Map<String, String> chosenFacts(Map<String, List<String>> f) {
        Map<String, String> out = new LinkedHashMap<>();
        f.forEach((k, v) -> {
            if (k.startsWith("f:") && !v.get(0).strip().isEmpty()) {
                out.put(k.substring(2), v.get(0).strip());
            }
        });
        return out;
    }

    /** "name = value" lines typed by hand. */
    static List<String[]> extraFacts(String text) {
        List<String[]> out = new ArrayList<>();
        for (String line : text.split("\\R")) {
            int eq = line.indexOf('=');
            if (eq > 0 && !line.substring(0, eq).strip().isEmpty()
                    && !line.substring(eq + 1).strip().isEmpty()) {
                out.add(new String[] {line.substring(0, eq).strip(),
                    line.substring(eq + 1).strip()});
            }
        }
        return out;
    }

    static void send(HttpExchange ex, int status, String type, String body,
                     String extraHeader, String extraValue) throws IOException {
        byte[] bytes = body.getBytes(StandardCharsets.UTF_8);
        ex.getResponseHeaders().set("Content-Type", type);
        if (extraHeader != null) {
            ex.getResponseHeaders().set(extraHeader, extraValue);
        }
        ex.sendResponseHeaders(status, bytes.length == 0 ? -1 : bytes.length);
        try (OutputStream out = ex.getResponseBody()) {
            out.write(bytes);
        }
    }
```

Listing 14.8 (Continued) *BuilderServer.java: the same server with the JDK's own HttpServer.*

```
    /** A whole page, with the message left by the last change (once). */
    static void show(HttpExchange ex, String here, String body)
            throws IOException {
        String html = pages.page(here, body, notice, error);
        notice = "";
        error = false;
        send(ex, 200, "text/html; charset=utf-8", html, null, null);
    }

    /** After a change: remember what to tell the person, and go on. */
    static void done(HttpExchange ex, String where, VcirsException e)
            throws IOException {
        notice = e == null ? pages.t("saved") : e.getMessage();
        error = e != null;
        send(ex, 303, "text/html; charset=utf-8", "", "Location", where);
    }

    static synchronized void handle(HttpExchange ex) throws IOException {
        String path = ex.getRequestURI().getPath();
        boolean post = ex.getRequestMethod().equals("POST");
        Map<String, List<String>> f = form(post
                ? new String(ex.getRequestBody().readAllBytes(),
                        StandardCharsets.UTF_8)
                : ex.getRequestURI().getRawQuery());
        Builder b = builder;
        try {
            switch ((post ? "POST " : "GET ") + path) {
                case "GET /" -> show(ex, "/", pages.home(b));
                case "GET /variables" -> show(ex, path, pages.variables(b));
                case "GET /teach" -> show(ex, path, pages.teach(b, chosenFacts(f)));
                case "GET /consult" -> show(ex, path, pages.consult(b,
                        new LinkedHashMap<>(), null));
                case "GET /knowledge" -> show(ex, path, pages.knowledge(b));
                case "GET /stats" -> show(ex, path, pages.stats(b));
                case "GET /proposals" -> show(ex, path, pages.proposals(b));
                case "GET /export" -> send(ex, 200,
                        "application/json; charset=utf-8", b.exportJson(),
                        "Content-Disposition",
                        "attachment; filename=\"knowledge-base.json\"");
                case "GET /lang" -> {
                    String to = get(f, "to");
                    if (to.equals("en") || to.equals("id")) {
                        pages.lang = to;
                    }
                    send(ex, 303, "text/html", "", "Location", "/");
                }
```

Listing 14.8 (Continued) *BuilderServer.java: the same server with the JDK's own HttpServer.*

```
                case "POST /project" -> {
                    b.newProject(get(f, "name"), get(f, "description"));
                    done(ex, "/", null);
                }
                case "POST /import" -> {
                    b.importJson(get(f, "json"));
                    done(ex, "/", null);
                }
                case "POST /variables" -> {
                    b.declare(get(f, "variable"), get(f, "question"),
                            get(f, "choices"));
                    done(ex, "/variables", null);
                }
                case "POST /teach" -> teach(ex, b, f);
                case "POST /consult" -> {
                    Map<String, String> chosen = chosenFacts(f);
                    List<String[]> pairs = new ArrayList<>();
                    chosen.forEach((k, v) -> pairs.add(new String[] {k, v}));
                    InferenceResult result = b.consult(pairs);
                    show(ex, "/consult", pages.consult(b, chosen, result));
                }
                case "POST /stop" -> {
                    b.stop(get(f, "node"), "");
                    done(ex, "/knowledge", null);
                }
                case "POST /accept-rule" -> {
                    b.acceptRule(get(f, "id"), "");
                    done(ex, "/proposals", null);
                }
                case "POST /accept-node" -> {
                    b.acceptNode(get(f, "id"),
                            f.getOrDefault("keep", List.of()), "");
                    done(ex, "/proposals", null);
                }
                default -> send(ex, 404, "text/plain; charset=utf-8",
                        "Not found", null, null);
            }
        } catch (VcirsException e) {
            String back = path.equals("/project") || path.equals("/import")
                    ? "/" : path.startsWith("/accept") ? "/proposals"
                    : path.equals("/stop") ? "/knowledge" : path;
            done(ex, back, e);
        }
    }
```

Listing 14.8 (Continued) *BuilderServer.java: the same server with the JDK's own HttpServer.*

```
    static void teach(HttpExchange ex, Builder b, Map<String, List<String>> f)
            throws IOException {
        List<String[]> pairs = new ArrayList<>();
        chosenFacts(f).forEach((k, v) -> pairs.add(new String[] {k, v}));
        pairs.addAll(extraFacts(get(f, "extra")));
        Builder.Taught t;
        try {
            t = b.teach(pairs, get(f, "answer"), get(f, "teacher"));
        } catch (VcirsException e) {
            notice = e.getMessage();
            error = true;
            show(ex, "/teach", pages.teach(b, chosenFacts(f)));
            return;
        }
        show(ex, "/teach", pages.taught(t.before(), t.result()));
    }

    public static void main(String[] args) throws IOException {
        int port = 4141;
        String path = "../data/builder-kb.json";
        for (int i = 0; i < args.length; i++) {
            if (args[i].equals("--port") && i + 1 < args.length) {
                port = Integer.parseInt(args[++i]);
            } else if (args[i].endsWith(".json")) {
                path = args[i];
            }
        }
        builder = new Builder(path);
        pages = new Pages("en");
        HttpServer server = HttpServer.create(
                new InetSocketAddress("127.0.0.1", port), 0);
        server.createContext("/", BuilderServer::handle);
        server.start();
        System.out.println("VCIRS Builder: open http://localhost:" + port
                + " in your browser; Ctrl+C stops it.");
    }
}
```

The JDK's server gives less help than Flask, so the Java server does three things by hand. `form` parses `application/x-www-form-urlencoded` text — the format in which browsers send forms, `a=1&b=x+y` — into a map from names to lists of values, with `URLDecoder`. `send` writes a response with its content type and length. And `handle` does the routing itself, with one `switch` over the method and path, such as `"POST /teach"`. `handle` is `synchronized` so that two requests arriving at the same time — Bu Wati on the laptop and Rini on her phone — never change the knowledge base at the same moment. The Python server gets the same guarantee from `threaded=False` in `app.run`, which makes Flask answer one request at a time; for a shop with two or three people, one request at a time is more than fast enough.

To run the builder, start either server:

```
cd Chapter14/java
javac -d out -sourcepath ../../vcirs-core/java/src *.java
java -cp out BuilderServer
```

or, after `python -m pip install -r requirements.txt` in `Chapter14/python`, `python builder_app.py`. Both print a line like `VCIRS Builder: open http://localhost:4141 in your browser; Ctrl+C stops it.` Open that address. Both servers keep their knowledge base in `data/builder-kb.json`, so you can stop one and start the other and continue where you were.

Do the two servers really agree? To check, a small test program (kept outside the chapter's folder) sent the same 33 requests to each server — importing the cake cases, opening every page

in both languages, consulting, teaching three cases including a refused one, declaring a question, accepting a proposal and a wrong proposal, stopping a node, importing a broken file and making a project without a name — and saved every page. The 33 pages of the Java server and the 33 pages of the Python server were identical, byte for byte, and so were the two knowledge-base files at the end.

14.7 Using the Builder

Here is Bu Wati's first evening with the builder, in the screenshots of the running application. On the Home page she chooses `cake-cases.json` under *Import* — the file into which her nephew typed the first pages of her notebook — and presses **Load**.

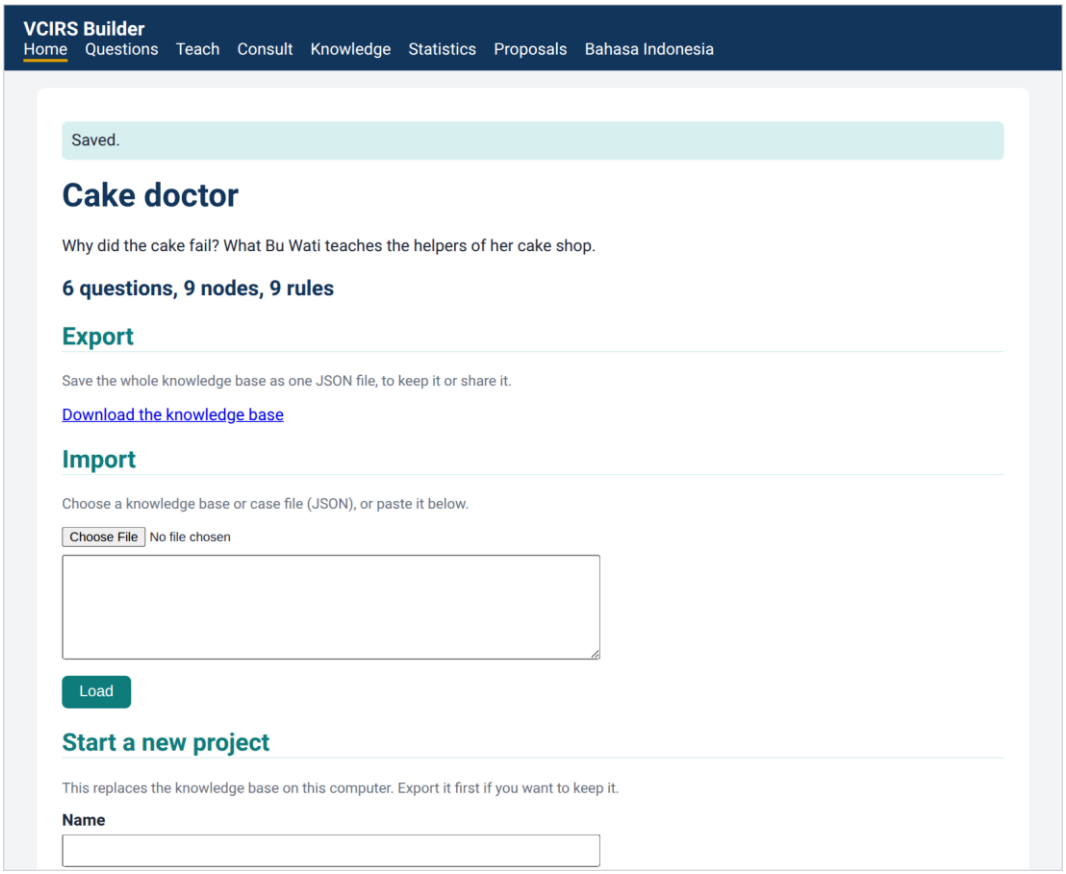

Figure 14.2 The Home page after importing the cake cases: the project's name and description, its size, and the export, import and new-project sections. "Saved." confirms the last change.

Figure 14.2 shows the result: the project is now called *Cake doctor*, with 6 questions, 9 nodes and 9 rules. Nothing on the page says "node" in a way she has to understand yet; the numbers are there for when she wants them.

Rini has a cake that sank. She opens **Consult**, chooses *sank in the middle*, *low* for the oven, *no* for the door, and presses **Give advice**.

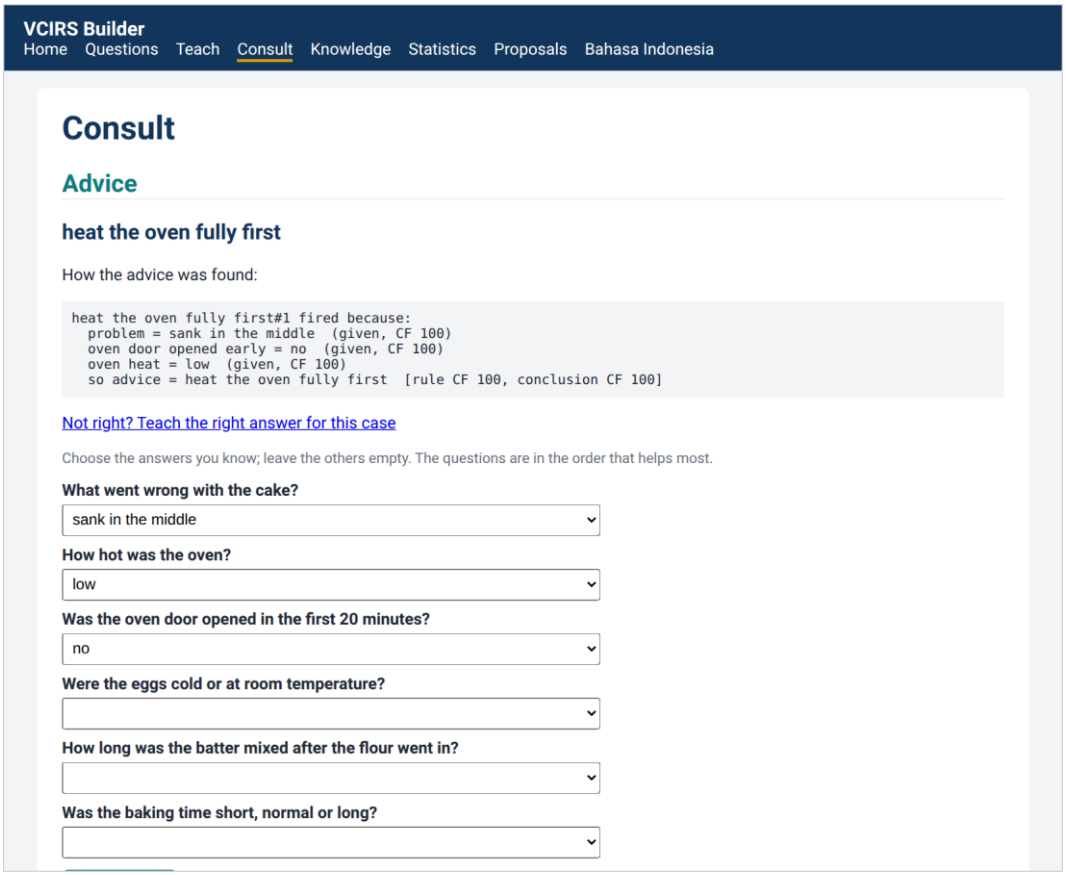

Figure 14.3 The Consult page: the advice in large letters, how it was found, a link to teach the right answer if it is wrong, and the questions in the order that helps most.

Figure 14.3 shows the advice, *heat the oven fully first*, with the explanation below it: the node that fired and the three facts that made it fire. The questions are in the order of the Statistics page, the most useful first, and the answers she chose stay selected. If the advice had been wrong, the link *Not right? Teach the right answer for this case* would have opened the Teach page with the same answers already chosen.

That is what happens next, with a burnt cake. The knowledge base says *bake on a higher rack*, but this time the cake was in a dark metal pan and the higher rack did not help; a lighter pan did. Rini follows the link, types `pan = dark metal` in the box for other facts, *use a lighter pan* as the right answer, and her name.

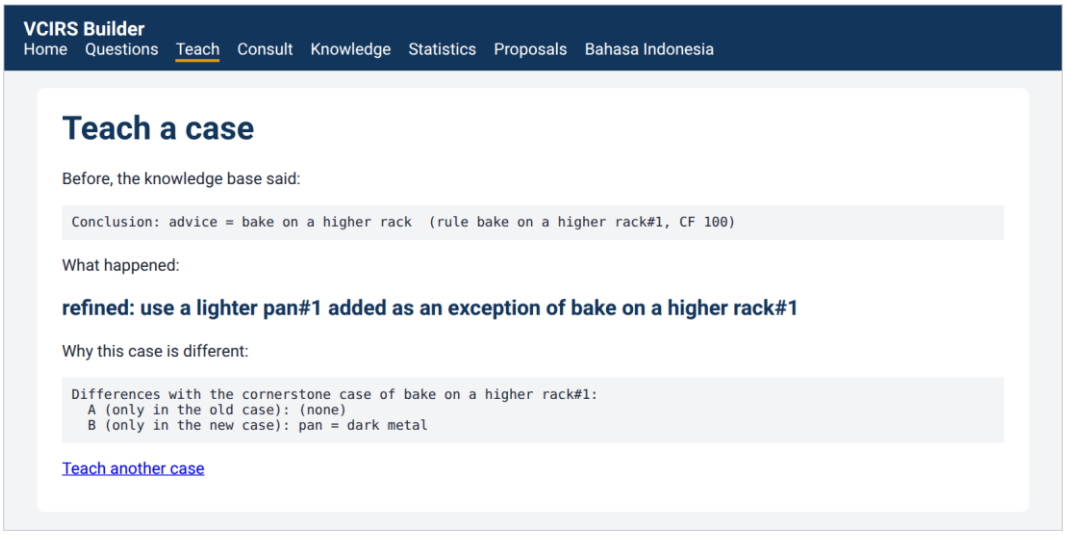

Figure 14.4 The result of teaching: what the knowledge base said before, what happened, and why the new case is different from the case that created the old node.

Figure 14.4 is the builder's most important page. It says what the knowledge base answered before, what happened — the new node became an exception of the old one — and the list of difference that justified it: nothing only in the old case, `pan = dark metal` only in the new case. Rini did not need to know what a node or an exception is to teach it; the page tells her, in the engine's words, and the next consultation with a dark pan will give her advice.

Later, Bu Wati opens **Statistics**.

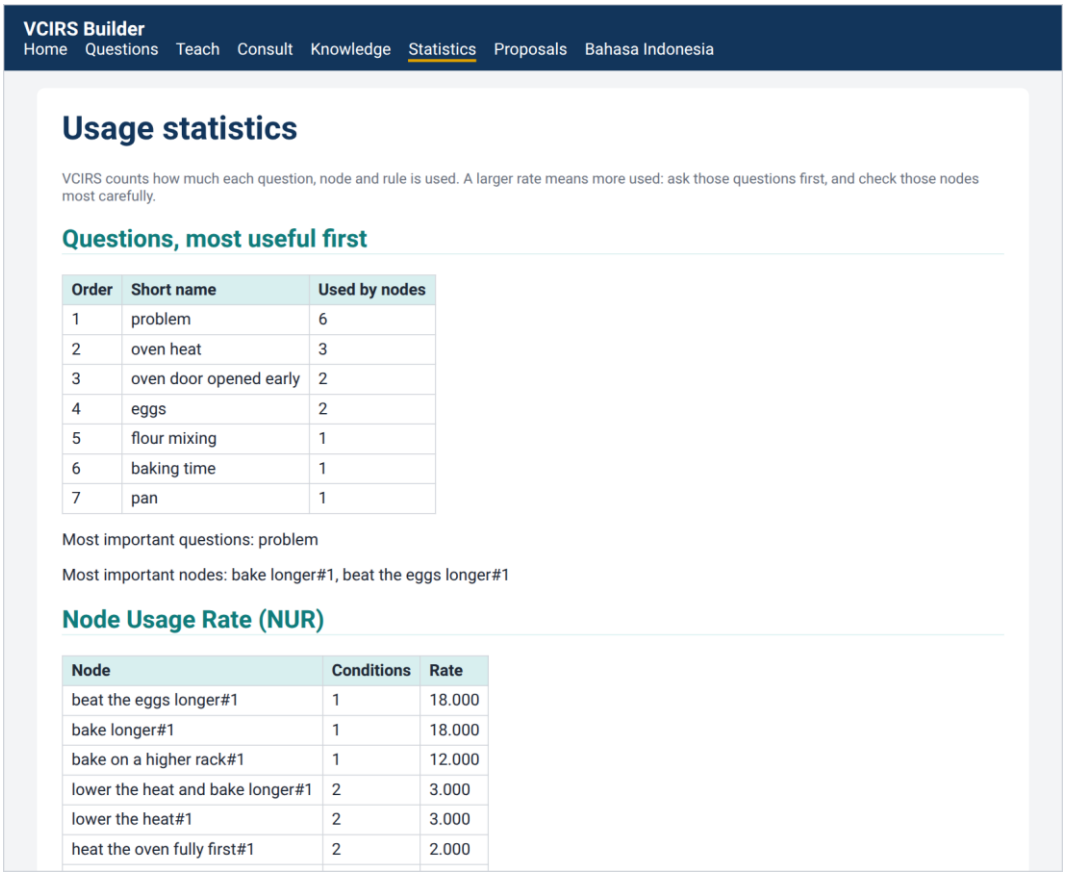

Figure 14.5 The Statistics page: the questions from most to least useful with the number of nodes that use each, the most important questions and nodes, and the Node Usage Rates, largest first.

Figure 14.5 answers two of her questions at a glance. Which questions matter most? `problem`, used by six nodes, then `oven heat`. Which pieces of knowledge carry the most weight? *beat the eggs longer* and *bake longer*, with NUR 18 each: every exception about eggs and oven doors hangs from them. The rest of the page, below the screenshot, lists the RUR of every rule and the VUR of every condition, for the day she wants the details.

On the **Proposals** page she finds the list of proposed rules and nodes. Most of it looks strange to her, and the page tells her that is fine: keep what makes sense. In the proposal for *take it out earlier* she ticks only `problem = dry`.

Figure 14.6 A proposed node on the Proposals page. Of the conditions VCIRS collected in a good order, Bu Wati keeps one, `problem = dry`, to make a shorter and more general node.

Figure 14.6 shows the proposal with her tick. After **Accept**, a new top-level node *take it out earlier#1G* answers every dry cake, even when nobody remembers how long it baked. Finally she opens **Knowledge**.

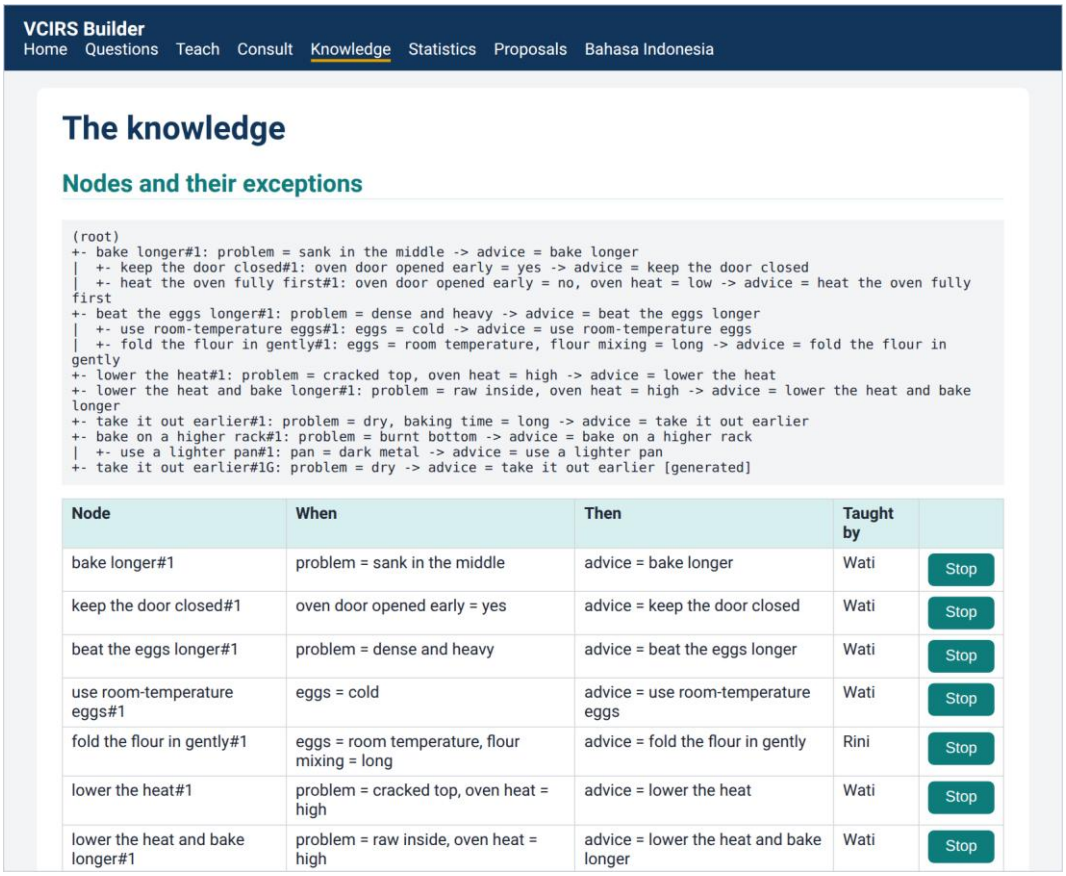

Figure 14.7 The Knowledge page: the tree of nodes and exceptions, marked when a node was generated, and a table with who taught each node and a Stop button.

Figure 14.7 shows her knowledge as a tree: each general node with its exceptions under it, the new exception for dark pans, and the generated node at the bottom, marked `[generated]`. The table below says who taught each node — Wati or Rini — and offers a **Stop** button for any advice she no longer wants given. Further down, the history lists every change with its option number, the same audit trail that Chapter 11 builds on for teams.

Everything she saw also exists in Indonesian. The link *Bahasa Indonesia* at the top switches every label, hint and button; the knowledge base's own questions and answers are in whatever language she wrote them. The Indonesian screenshots of this chapter's Indonesian edition show the same builder with *Dokter kue*, imported from `kue-cases-id.json`.

14.8 Making It Better

Several projects. The builder keeps one knowledge base at a time. A simple extension keeps a folder of knowledge bases and a drop-down list on the Home page to choose one; the builder class does not change, because it already takes the file as a parameter.

People and permissions. Anyone who can open the page can teach and stop nodes. In a family shop that is right. In a larger team, add a login and let only some people teach; Chapter 11 shows the team features — several experts, versions and an audit trail — that such a builder would use, and the name of the teacher that the builder already records with every node is the first step.

Engine messages in the person's language. The page labels switch to Indonesian, but the engine's own sentences — "refined: … added as an exception of …", "Differences with the cornerstone case of …" — stay in English, because they come from vcirs-core. A friendlier builder

would build its own sentences from `LearnResult` and `DifferenceList` (their parts are all available) in the chosen language. Exercise 14.5 asks for it.

Checking the affected cornerstones. Before accepting a taught case, the builder could show the cornerstone cases whose answers would change, with `affected_cornerstones`, and ask for confirmation, as Chapter 3 described. With a few dozen nodes this matters little; with hundreds it is the safety net that makes large knowledge bases maintainable [1].

A fuzzy question. A question such as *how long did it bake?* is really a number. The engine understands fuzzy values (Chapter 9); the Questions page could let a person define *short*, *normal* and *long* as ranges of minutes, and the Consult page could accept a number.

Undo. Because nothing is ever deleted, an "undo" is simply "stop the node I just added". A button that does this for the last change of the current person would make experimenting less frightening.

14.9 Using It for Real

At Bu Wati's shop. Start the server on the laptop by the ovens and open it in the laptop's browser. To use it from phones in the shop, start it with the host set to the laptop's address in the shop's Wi-Fi (Flask: `app.run(host="0.0.0.0")`; Java: `new InetSocketAddress("0.0.0.0", port)`) and open `http://<laptop-address>:4141` on the phone. Do this only inside the shop's own network: the builder has no passwords, and anyone who can reach it can change the knowledge. Chapter 15 shows how to put a finished knowledge base behind a small, read-only service instead.

Starting from a notebook. The fastest way to begin is the way this chapter began: type the notebook's cases into a case file — one line per case, with the facts, the answer and who taught it — and import it. The builder teaches them one by one, so the result is the same knowledge base as if they had been typed into the Teach page, and the history shows every case. After that, teach at the counter or the oven, when a real case happens.

Sharing. The exported JSON is the whole knowledge: questions, nodes, rules, cornerstone cases and history. Bu Wati can send it to her sister, who imports it and continues teaching with her own cases; the two shops' knowledge bases then grow apart, each with its own history. It is also the file that the Wi-Fi doctor of Chapter 3, the web app of Chapter 10 and the service of Chapter 15 read, because they all use vcirs-core.

Other domains. Nothing in the builder is about cakes. A motorcycle workshop, a rice farmer's group, a small clinic's front desk (with the safe referral rules of Chapter 5 [10]), a school's computer helper: each starts with an empty project, writes its questions, and teaches. The only thing the builder assumes is that someone knows the answers when she sees a case.

Training new helpers. The builder is also a teacher. A new helper can open the Knowledge page and read the tree the way Bu Wati once read her notebook aloud: each general piece of advice with its exceptions under it, and the name of the person who taught each one. Better still, she can use the Consult page as a practice game for herself — choose the answers for a cake she saw fail, guess the advice, then press **Give advice** and read the explanation. After a month she will know most of what the knowledge base knows, and she will have seen *why*, fact by fact. And when she finds a case the knowledge base gets wrong, she teaches it herself, which is the best sign that she has become one of the experts.

Keep copies. Export the knowledge base every week and keep the files. It is small, it is the shop's knowledge, and Chapter 15 shows how to make those copies automatically and check them.

14.10 Common Mistakes, and How to Avoid Them

Writing rules instead of teaching cases. A person who types every rule she can imagine into case files, without real cases, builds a flat rule base in disguise. *Avoid it:* teach from cakes that really failed; each case then has a cornerstone that was really checked.

Different words for the same answer. `dense`, `dense and heavy` and `heavy` are three different values to the engine. *Avoid it:* write the usual answers on the Questions page and choose them from the drop-down lists; type a new value only when it is really new.

Accepting proposals without reading them. A proposed node with all its conditions ticked is usually contradictory (`problem = dry` *and* `problem = raw inside`). *Avoid it:* tick only the conditions that belong together, and ignore proposals that make no sense.

Putting HTML together without escaping. One question name with `<` in it breaks the page; a malicious one attacks every visitor. *Avoid it:* pass every piece of text through `esc`, as every page here does [7].

Changing the knowledge base from two places at once. Two servers on the same file, or two tabs of an old copy, can overwrite each other's changes. *Avoid it:* run one server per knowledge-base file, and let everyone use it through the browser.

Showing the result of a POST without redirecting. A reload then repeats the change. *Avoid it:* redirect after every change (post, redirect, get), as `done` does; show a page directly only when the page is the result, as for Teach and Consult.

Letting the builder be reached from the internet. It has no passwords. *Avoid it:* listen on `127.0.0.1`, or on the shop's own network at most.

Keeping only one copy. A laptop that falls off a shelf takes eleven years of notes with it. *Avoid it:* export regularly and keep the copies elsewhere.

14.11 Chapter Summary

- A builder lets a person who does not program create, grow and share a VCIRS expert system; each page rests on one VCIRS idea (**Table 14.1**), and the person speaks in **cases**, never in rules.
- The **usage statistics** say how much of the knowledge base depends on each question, node and rule: VUR = Credit × NS × CD (Equation (14.1)), NUR and RUR are averages (Equation (14.2)); large rates mark knowledge to review first and questions to ask first.
- **Proposals** from rule and node generation are only proposals: the person keeps the conditions that belong together (**Figure 14.6**).
- The application separates the **builder** (**Listing 14.1**, **Listing 14.2**), the **pages** (**Listing 14.5**, **Listing 14.6**) and the **servers** (**Listing 14.7**, **Listing 14.8**) (**Figure 14.1**); a script plays a whole session without the web (**Listing 14.3**, **Listing 14.4**).
- The Java server (JDK HttpServer) and the Python server (Flask) produce the same pages and the same knowledge-base file byte for byte, in English and Indonesian, and the screenshots (**Figure 14.2**, **Figure 14.3**, **Figure 14.4**, **Figure 14.5**, **Figure 14.7**) show a whole evening of use.

14.12 Exercises

14.1 Using Equation (14.1) and Equation (14.2), compute by hand the VUR of both conditions of *fold the flour in gently#1* (`eggs = room temperature`, then `flour mixing = long`), its NUR, and the RUR of the rule that ends at it. Compare with the output of Section 14.4.

14.2 Rini teaches `problem = burnt bottom` → *use a lighter pan* and the builder refuses. Explain why, in terms of the cornerstone case and the list of difference. How can she make the builder accept her knowledge, and why would stopping the old node first not help?

14.3 Bu Wati thinks *lower the heat#1* (`problem = cracked top`, `oven heat = high`) is wrong: in her experience a cracked top comes from too much baking powder. Which page does she use, and what does she do, so that the old advice is no longer given but the history still shows it? What does she teach next?

14.4 Explain why the builder saves the knowledge base after every change instead of offering a Save button. What would a Save button cost Bu Wati on a busy Saturday?

14.5 The engine's sentences on the Teach result page are in English even in the Indonesian builder. Write, in words or code, how `pages.taught` could build an Indonesian sentence from `result.action`, `result.node.id` and `result.fired`, using two new keys in `ui-text.json`.

14.6 A helper types the question name `oven <heat>` on the Questions page. What does the builder do? And if she types the answer `<b>hot</b>` as a usual answer of `oven heat`, what does the Consult page show, and why is that safe?

14.7 Two proposed rules in Section 14.4 contain long sequences of top-level nodes. Why would accepting them add nothing useful to Bu Wati's knowledge base? When do proposed rules become useful?

14.8 Describe the requests a browser sends when Bu Wati adds a question on the Questions page and then presses the browser's reload button. How many times is the question saved, and why?

14.13 Solutions to the Exercises

Solution 14.1. The node has TV = 2 conditions. For `eggs = room temperature`: VO = 1, CD = 1/2; NS(`eggs`) = 2 (this node and *use room-temperature eggs#1*); Credit = 1 (no case was ever taught under this node). VUR = 1 × 2 × 0.5 = 1. For `flour mixing = long`: VO = 2, CD = 2/2 = 1; NS = 1; Credit = 1; VUR = 1 × 1 × 1 = 1. NUR = (1 + 1)/2 = 1, which is the first line of the NUR list in Section 14.4 (`NUR 1.000  fold the flour in gently#1`). The rule ending at it passes through *beat the eggs longer#1* (NUR 18) and this node (NUR 1), so RUR = (18 + 1)/2 = 9.5.

Solution 14.2. The node *bake on a higher rack#1* was created by the case with the single fact `problem = burnt bottom`; that is its cornerstone case. Rini's case has exactly the same facts, so its list of difference is empty in both sections: there is no condition that is true for her case and false for the cornerstone case, and an exception built from nothing would fire for every burnt cake, changing the old case's answer. The builder therefore refuses. The way out is to add a fact that really was different in her case — `pan = dark metal`, or any other fact the cornerstone case does not have — which gives section B one condition and lets the case become an exception, as in the chapter. Stopping *bake on a higher rack#1* first would not help: a stopped node still fires, it only gives no advice, so the new case would again be compared with the same cornerstone case and refused for the same reason. A difference must be named; that is what keeps the knowledge base from contradicting itself.

Solution 14.3. She opens **Knowledge** and presses **Stop** beside *lower the heat#1*. The node stays in the knowledge base with its cornerstone case, its exceptions (none here) and its history entry, but it no longer gives advice; the history gains a `stop` line. Then, the next time a cake cracks, she teaches that case on the Teach page with the facts she knows and the right answer, *use less baking powder*, adding the new fact `baking powder = too much`. What `learn` does depends on the facts. If the case also has `oven heat = high`, the stopped node still fires — a stopped node only stops giving advice — so the new case is hung under it as an exception built from the difference `baking powder = too much`. If the oven heat is not given, the stopped node does not fire, nothing answers, and the case becomes a new top-level node. Either way the old advice is never given again, and the history keeps both steps.

Solution 14.4. A person who does not program does not think of "the file" at all; she thinks she told the machine something. If saving were a separate step, a Saturday with three failed cakes, a crying child and a customer at the door would end with a closed browser and nothing saved. Saving after every change costs a few milliseconds and makes the page's promise true: what you taught is kept. It also makes the file on disk always consistent with what the pages show, which Chapter 15's backups rely on.

Solution 14.5. Add two keys to `ui-text.json`, for example `"r_refined": "{new} ditambahkan sebagai pengecualian dari {old}"` and `"r_added": "{new} ditambahkan di tingkat teratas"` (and the English versions), plus `"r_accepted"`. In `taught`, choose the key from `result.action` and replace `{new}` with `result.node.id` and `{old}` with `result.fired` before escaping; for `accepted` there is no new node. The same idea works for the difference list: build "Hanya pada kasus lama: …" and "Hanya pada kasus baru: …" from `result.differences.a` and `.b`. In Java the replacements are `String.replace`, in Python `str.replace`, so both languages still produce the same bytes.

Solution 14.6. `oven <heat>` contains `<`, so `declare` refuses it with "a question name must not be empty or contain = < >", shown in red at the top of the Questions page, and nothing is saved. The answer `<b>hot</b>` is accepted as a value (only question names are restricted), and on the Consult page it appears in the drop-down list as the literal text `<b>hot</b>`, because `esc` turns it into `<b>hot</b>` before it is placed in the HTML. The browser shows the characters instead of interpreting them as a tag, so no value can change the page or run a script.

Solution 14.7. A rule of VCIRS is a path of nodes, and the RDR engine already follows the paths that exist: every node with its ancestors. The proposed rules string together top-level nodes about different problems (dry, raw, cracked, burnt, dense, sank), which never belong to one case, so a rule through them would never fire for a real cake. Proposed rules become useful in knowledge bases whose nodes share many variables and whose rules are long paths, as in the Zookeeper example of the thesis, where a proposed rule visits the shared nodes in a good order and can make inference faster [4]. In a young knowledge base organized around one question, `problem`, the proposed *nodes* are the useful part, because a person can pick a generalization from them.

Solution 14.8. The browser sends `POST /variables` with the form fields. The server declares the question, saves the file, sets the notice "Saved." and answers `303 See Other` with `Location: /variables`. The browser then sends `GET /variables` and shows the page with the notice. When Bu Wati presses reload, the browser repeats the *last* request, which is the `GET`; the page is shown again (without the notice, which was shown once) and the question is saved exactly once. Without the redirect, reload would repeat the `POST` — most browsers would first ask "resend the form?" — and declare the question again.

References

[1] P. Compton and B. H. Kang, *Ripple-Down Rules: The Alternative to Machine Learning*. Boca Raton, FL, USA: CRC Press, 2021.

[2] G. Edwards, P. Compton, R. Malor, A. Srinivasan, and L. Lazarus, "PEIRS: a pathologist-maintained expert system for the interpretation of chemical pathology reports," *Pathology*, vol. 25, no. 1, pp. 27–34, 1993.

[3] J. Nielsen, "10 usability heuristics for user interface design," Nielsen Norman Group, updated Jan. 30, 2024. [Online]. Available: https://www.nngroup.com/articles/ten-usability-heuristics/

[4] M. M. I. Subakti, "A Variable-Centered Intelligent Rule System," M.Sc.Eng. thesis, Dept. of Computer Science and Information Engineering, Taipei, Taiwan, Jun. 2005.

[5] I. Subakti, "A variable-centered intelligent rule system," in *Proc. 1st Annual Int. Conf. Information and Communication Technology Seminar (ICTS 2005)*, vol. 1, no. 1, Surabaya, Indonesia, Aug. 2005, pp. 167–174.

[6] I. Subakti, "Some revisions in VCIRS and cases reconstructing perspectives," in *Proc. Information and Communication Technology Seminar (ICTS 2006)*, Surabaya, Indonesia, 2006.

[7] OWASP Foundation, "Cross site scripting prevention cheat sheet," *OWASP Cheat Sheet Series*, 2026. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

[8] Pallets, "Flask 3.1.3," Python Package Index, Feb. 19, 2026. [Online]. Available: https://pypi.org/project/Flask/

[9] Oracle, "Package com.sun.net.httpserver," *Java SE 25 & JDK 25 API Specification*, 2025. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/jdk.httpserver/com/sun/net/httpserver/package-summary.html

[10] I. Subakti and R. Hidayatullah, "Aplikasi sistem pakar untuk diagnosis awal gangguan kesehatan secara mandiri menggunakan Variable-Centered Intelligent Rule System," vol. 6, no. 1, pp. 11–16, Jan. 2007.

Chapter 15

Deploying, Maintaining and Growing Your Expert Systems

By the end of this chapter you will be able to turn a finished knowledge base into a small **advice service** that runs every day on an ordinary computer and answers any phone, tablet or program in the shop over HTTP; package the service and its tools as **one file** — a `.jar` for Java or a `.pyz` for Python — and install it on Windows, macOS or Linux so that it starts by itself; check every deployment with a short automatic test; make **backups** of the knowledge base that you can trust, check them, and restore one; keep a **usage log** that records which advice was given and which cases found no answer, without recording anything about the people who asked; read a monthly **usage report** that tells you what to teach next and which knowledge is never used; follow Indonesia's personal data protection law, whose implementing regulation was issued in July 2026; think clearly about the ethics of giving advice by machine; and see where to go next, with this book behind you.

15.1 The Morning the Laptop Would Not Start

For three months Bu Wati's cake doctor, built with the builder of Chapter 14, lived on the old laptop by the ovens. It worked well. Rini used it on busy Saturdays, the new helper learned from it, and Bu Wati taught it a new case almost every week. Then, on a Monday in September, the laptop would not start. The screen stayed black, a repairman said the disk had failed, and for one frightening afternoon Bu Wati believed that three months of teaching were gone.

They were not, but only by luck: two weeks earlier her nephew had exported the knowledge base to his phone to show a friend. Two weeks of cases were lost for good. That afternoon made Bu Wati ask three questions that every owner of an expert system eventually asks. *How do I make sure the knowledge survives when a computer dies? How do I let everyone in the shop use the advice without letting everyone change it?* And, after the scare had passed, a quieter one: *Is the cake doctor actually helping? Which of its advice is used, and what do people ask that it cannot answer?*

This chapter answers those questions for the cake doctor, and through it for every application in this book. It is the chapter about everything that happens *after* the program works: deploying it, keeping it alive, keeping its knowledge safe, watching how it is used, obeying the law about the people it serves, and growing it for years. None of it is glamorous, and all of it decides whether an expert system is still helping somebody three years from now.

15.2 The Idea in Plain Words

15.2.1 From a program you run to a service that runs

Until now, every application in this book was a program somebody started: the Wi-Fi doctor asked its questions in a terminal and ended; the builder ran while its window was open. To **deploy** an application means to put it where it will be used, in a form that keeps running without

its programmer: it starts when the computer starts, restarts itself if it fails, keeps its data in a known place, and can be checked, backed up and updated by someone who is not a programmer.

For the cake doctor the right form is a small **service**: a program that runs in the background on one computer in the shop and answers questions over HTTP — the same kind of HTTP requests that Chapters 10, 13 and 14 used. The laptop by the ovens, a tablet on the counter, or a phone can ask it for advice; a later program, such as the chat front end of Chapter 13, can ask it too. The service has three doors. `GET /health` answers "I am alive, and this is the knowledge base I use". `GET /questions` lists the questions, most useful first, with their usual answers, so that a screen can show them. `POST /advise` receives facts as JSON and returns the advice and its explanation.

The service only *reads* the knowledge base; it never changes it. Teaching still happens in the builder, by the people Bu Wati trusts, on her own laptop. When a new version of the knowledge base is ready, it is copied to the service's computer, and the service notices the new file and loads it by itself. This separation — one place where knowledge is changed, one where it is used — is the simplest way to let everyone use the advice without letting everyone change it.

15.2.2 One file to copy

A deployment is easiest when there is one thing to copy. Java has had such a thing since its first years: the **JAR file**, a compressed archive of compiled classes with a small manifest that names the class whose `main` runs first; `java -jar cake-doctor.jar` then starts the program on any computer with a Java runtime [1]. Python has the same idea in the **zipapp**: a ZIP archive of Python files with a `__main__.py` at the top, which Python runs directly with `python cake-doctor.pyz`; the standard library's `zipapp` module builds it [2]. Both files contain the application *and* the vcirs-core engine, so the shop computer needs nothing but Java or Python and the knowledge-base file.

We put the service and its three tools — the deployment check, the usage report and the backup tool — into the same file, as four **commands**: `serve`, `check`, `report` and `backup`. One file, one command name, four jobs.

15.2.3 A knowledge base is data: backups you can trust

Bu Wati's lesson is the oldest lesson in computing: data that exists in one place does not really exist. The usual advice is the **3-2-1 rule**: keep three copies of important data, on two different kinds of storage, with one copy in another place [3]. For a knowledge base of a few hundred kilobytes this costs nothing. The copy on the service computer is the first; a nightly backup on a USB stick or a second disk is the second; a copy sent every week to the owner's phone or to a cloud folder is the third.

But a backup is only useful if it can be restored, and a surprising number of backups cannot: the copy was taken while the file was half written, or the USB stick corrupted it quietly. Our backup tool takes three precautions. It copies a knowledge base only after loading it successfully, so a broken file is never backed up over good ones. It writes a **fingerprint** of every copy — a SHA-256 hash, a 64-character number computed from the file's bytes that changes completely if even one byte changes — into an index file. And it has a `check` command that reads every copy again, recomputes its fingerprint and loads it, so that a damaged backup is discovered on the day it is damaged, not on the day it is needed. Finally, `restore` refuses a damaged copy, and keeps the file it replaces, just in case.

15.2.4 Watching how the knowledge is used

The usage rates of Chapter 14 tell Bu Wati how the knowledge is built: which nodes others depend on. They do not tell her how it is *used*: which advice the helpers actually receive on Saturdays, and which questions they ask that the knowledge base cannot answer. For that, the service keeps a **usage log**: one line per consultation, with the time, the facts that were given, the node that answered (or none), and the advice.

The log is where maintenance begins. Once a month a small tool reads it and writes a **usage report** with four findings: how many consultations there were each month; how many got an answer; which nodes gave advice, most used first, and which nodes never did; and which unanswered cases came back most often. The last list is the most valuable: it is literally a list of the cases to teach next, sorted by how much they would help. The list of nodes that never gave advice is the second most valuable: such a node may be fine — some problems are rare — or it may be knowledge that no longer matches the shop, and it deserves a look.

15.2.5 Personal data and the law

A usage log records what people asked. That makes it the part of an expert system where the law about personal data applies most directly, and where a careless design can do the most harm.

Indonesia's **Law No. 27 of 2022 on Personal Data Protection** (*Undang-Undang Pelindungan Data Pribadi*, UU PDP) was enacted on October 17, 2022, and gave organizations two years to adjust, until October 2024 [4], [5]. It applies to anyone who processes personal data — data about an identified or identifiable person — including small businesses, and it asks, among other things, that personal data be collected in a limited and specific way for a lawful purpose, processed accurately and securely, kept no longer than needed, and that a data breach be reported to the people affected within 3 × 24 hours [4], [6]. Its implementing regulation, **Government Regulation No. 33 of 2026** (PP 33/2026), was enacted on July 16, 2026, began to circulate publicly in late August, and takes effect on January 16, 2027, after a six-month transition; as of the end of August 2026, the Personal Data Protection Agency that will enforce the law had not yet been established [5]. The regulation gives the operational detail the law itself left open, for example on consent, the rights of data subjects and the data protection officer [5], [6].

What does this mean for the cake doctor? The simplest way to comply with a law about personal data is **not to collect personal data at all**, and the usage log is designed that way. It records the time of a consultation, the facts about a *cake* (`problem = dry`), the node that answered and the advice. It records no name, no phone number, no IP address — the service's web server is even told not to write the usual access log with addresses — and no free text that could contain any of those. The log is still kept only as long as it is useful: the report tool can delete lines older than a number of days, and running it every month with, say, 180 days keeps six months of history and nothing more.

For an expert system about *people* — a health self-check, a loan advisor, a clinic triage — the facts themselves may be personal data, and more of the law applies: a clear purpose, consent where needed, access limits, retention limits, and a plan for breaches. Section 15.9 returns to this.

> **KEY INSIGHT.** Design the log so that it answers your maintenance questions — which advice is used, what cannot be answered — and nothing else. Data you never collect cannot leak, does not need consent, and costs nothing to protect.

15.3 Step 1: The Advice Service

Listing 15.1 is the service in Python, using only the standard library so that it can be packed into a `.pyz` file with no dependencies.

Listing 15.1 `advice_service.py`: the cake doctor as a small advice service with a privacy-minded usage log.

```
"""The cake doctor as a small, deployable advice service (standard library).

    python advice_service.py [--kb FILE] [--port 4151] [--host 127.0.0.1]
                             [--logs DIR] [--replay]

GET  /health     is the service alive, and which knowledge base it uses
GET  /questions  the questions, most useful first, with their usual answers
POST /advise     {"facts": {"problem": "dry"}} -> the advice and why
Every consultation is written to DIR/usage-YYYY-MM.jsonl: the time, the
facts and the answer, nothing about the person. When the knowledge-base
file changes, the service loads it again by itself. --replay lets a client
give the time of each consultation in an X-Time header (for old records);
a request with an X-Test header is answered but not written to the log.
"""

import datetime
import json
import os
import pathlib
import sys
import threading
from http.server import BaseHTTPRequestHandler, ThreadingHTTPServer

sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import KnowledgeBase, VcirsError                    # noqa: E402
from vcirs.jsonio import dumps                                 # noqa: E402

ENGINE = "vcirs-core 1.0.0"


class Service:
    """Everything except HTTP: the knowledge base, advice, and the log."""

    def __init__(self, kb_path, log_dir):
        self.kb_path = kb_path
        self.log_dir = log_dir
        self.lock = threading.Lock()
        self.kb = KnowledgeBase.load(kb_path)
        self.stamp = os.path.getmtime(kb_path)
        os.makedirs(log_dir, exist_ok=True)
```

Listing 15.1 (Continued) *advice_service.py: the cake doctor as a small advice service with a privacy-minded usage log.*

```
    def fresh_kb(self):
        """Load the file again if it changed; keep the old one if the new
        one cannot be read (a half-copied file, a mistake)."""
        stamp = os.path.getmtime(self.kb_path)
        if stamp != self.stamp:
            try:
                self.kb = KnowledgeBase.load(self.kb_path)
                print("reloaded " + self.kb_path, file=sys.stderr)
            except (OSError, ValueError, KeyError, VcirsError):
                print("could not reload " + self.kb_path + "; keeping the "
                      "old knowledge base", file=sys.stderr)
            self.stamp = stamp
        return self.kb

    def health(self):
        kb = self.fresh_kb()
        return {"status": "ok", "name": kb.name, "nodes": len(kb.nodes),
                "engine": ENGINE}

    def questions(self):
        kb = self.fresh_kb()
        used = [c.variable for n in kb.nodes.values() for c in n.conditions]
        order = kb.usage().question_order()
        order += [v for v in kb.variables if v in used and v not in order]
        return {"questions": [{"variable": v,
                               "question": kb.variables[v].question,
                               "choices": kb.known_values(v)}
                              for v in order]}

    def advise(self, facts, time, record=True):
        kb = self.fresh_kb()
        facts = {str(k).strip(): str(v).strip() for k, v in facts.items()
                 if str(k).strip() and str(v).strip()}
        result = kb.infer(facts)
        advice = [c.value for c in result.conclusions()]
        rule = result.paths[0].rule_id() if result.paths else None
        answer = {"advice": advice, "rule": rule,
                  "how": result.how() if advice else []}
        if record:                          # a check is not a real consultation
            self.log({"time": time, "rule": rule or "", "advice": advice,
                      "facts": facts})
        return answer
```

Listing 15.1 (Continued) *advice_service.py: the cake doctor as a small advice service with a privacy-minded usage log.*

```
    def log(self, record):
        """One line of JSON per consultation, one file per month."""
        name = "usage-" + record["time"][:7] + ".jsonl"
        with self.lock, open(os.path.join(self.log_dir, name), "a",
                             encoding="utf-8", newline="\n") as f:
            f.write(json.dumps(record, ensure_ascii=False) + "\n")


def now():
    return datetime.datetime.now(datetime.timezone.utc).strftime(
        "%Y-%m-%dT%H:%M:%SZ")


def make_handler(service, replay):
    class Handler(BaseHTTPRequestHandler):
        protocol_version = "HTTP/1.1"        # keep the connection open

        def reply(self, status, value):
            body = (dumps(value) + "\n").encode("utf-8")
            self.send_response(status)
            self.send_header("Content-Type", "application/json; charset=utf-8")
            self.send_header("Content-Length", str(len(body)))
            self.end_headers()
            self.wfile.write(body)

        def do_GET(self):
            if self.path == "/health":
                self.reply(200, service.health())
            elif self.path == "/questions":
                self.reply(200, service.questions())
            else:
                self.reply(404, {"error": "no such page"})

        def do_POST(self):
            size = int(self.headers.get("Content-Length", "0"))
            text = self.rfile.read(size).decode("utf-8", "replace")
            if self.path != "/advise":
                self.reply(404, {"error": "no such page"})
                return
            try:
                data = json.loads(text)
                facts = data["facts"]
                if not isinstance(facts, dict):
                    raise ValueError
            except (ValueError, KeyError, TypeError):
                self.reply(400, {"error": 'send {"facts": {"name": "value"}}'})
```

Listing 15.1 (Continued) *advice_service.py: the cake doctor as a small advice service with a privacy-minded usage log.*

```
                return
            time = self.headers.get("X-Time") if replay else None
            test = self.headers.get("X-Test") is not None
            self.reply(200, service.advise(facts, time or now(), not test))

        def log_message(self, fmt, *args):   # no addresses in any log
            pass

    return Handler


def main(args):
    def option(name, default):
        return args[args.index(name) + 1] if name in args else default

    here = os.path.dirname(os.path.abspath(sys.argv[0]))
    kb_path = option("--kb", os.path.join(here, "..", "data", "cake-kb.json"))
    service = Service(kb_path, option("--logs", "logs"))
    host, port = option("--host", "127.0.0.1"), int(option("--port", "4151"))
    server = ThreadingHTTPServer((host, port),
                                 make_handler(service, "--replay" in args))
    print("Advice service for '" + service.kb.name + "' on http://" + host
          + ":" + str(port) + "; Ctrl+C stops it.", flush=True)
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        pass


if __name__ == "__main__":
    main(sys.argv[1:])
```

The class `Service` holds everything except HTTP, which keeps it testable without a network.

It loads the knowledge base, and loads it again when it changes. The constructor loads the file and remembers its modification time. `fresh_kb`, called at the start of every request, compares the file's modification time with the remembered one; if it changed, it loads the file again. If the new file cannot be read — perhaps it is still being copied, or someone copied the wrong file — the service keeps the old knowledge base and writes a warning to the error output. The shop never finds itself with a service that stopped answering because of a bad copy.

It answers. `health` returns the status, the knowledge base's name, its number of nodes and the engine's version, the four things you want to know when checking a deployment. `questions` returns the question order of Chapter 14 — the variables used in conditions, most useful first — with each question's text and known answers. `advise` cleans the given facts (names and values are trimmed, empty ones are dropped), runs `infer`, and returns the advice, the rule that gave it, and the explanation.

It logs, carefully. `log` appends one line of JSON to a file named after the month of the consultation, `usage-2026-09.jsonl`; one file per month makes deleting old months easy. A lock makes sure that two requests arriving at the same time never write half-lines into each other. A consultation whose request carries the header `X-Test` is answered but not logged: checks after a deployment are not real use, and should not appear in the statistics.

The HTTP part is `make_handler`. It uses `ThreadingHTTPServer` from the standard library with `protocol_version = "HTTP/1.1"`, reads the request body first (so that an unknown path does not leave unread data on a kept-open connection), answers 400 with a short explanation when

the body is not `{"facts": {...}}`, and replaces `log_message`, the method that normally prints every request with the caller's IP address, with one that prints nothing. `--replay` is for one special job: entering consultations that happened earlier, such as the three months of paper notes in Section 15.7, with their real dates in an `X-Time` header. Without `--replay`, the service ignores `X-Time` and uses its own clock, so nobody can falsify the dates in the log.

It listens only on this computer unless told otherwise: `--host 127.0.0.1` is the default. To serve the tablets in the shop, start it with `--host 0.0.0.0`, which listens on every network the computer is connected to — do that only on the shop's own Wi-Fi, never on a computer connected directly to the internet, because the service has no passwords.

Listing 15.2 is the same service in Java, with the JDK's `HttpServer`.

Listing 15.2 `AdviceService.java`: the same service in Java.

```
import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardOpenOption;
import java.nio.file.attribute.FileTime;
import java.time.Instant;
import java.time.ZoneOffset;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;

import vcirs.Conclusion;
import vcirs.Facts;
import vcirs.InferenceResult;
import vcirs.Json;
import vcirs.KnowledgeBase;
import vcirs.Node;
import vcirs.VcirsException;

/**
 * The cake doctor as a small, deployable advice service (JDK only).
 * java AdviceService [--kb FILE] [--port 4151] [--host 127.0.0.1]
 *                    [--logs DIR] [--replay]
 * GET /health, GET /questions, POST /advise {"facts": {"problem": "dry"}}.
 * Every consultation is written to DIR/usage-YYYY-MM.jsonl: the time, the
 * facts and the answer, nothing about the person. When the knowledge-base
 * file changes, the service loads it again by itself. --replay lets a client
 * give the time of each consultation in an X-Time header (for old records);
 * a request with an X-Test header is answered but not written to the log.
 */
public class AdviceService {
    static final String ENGINE = "vcirs-core 1.0.0";
    static final DateTimeFormatter TIME = DateTimeFormatter
            .ofPattern("yyyy-MM-dd'T'HH:mm:ss'Z'").withZone(ZoneOffset.UTC);
```

Listing 15.2 (Continued) *AdviceService.java: the same service in Java.*

```
    final Path kbPath;
    final Path logDir;
    KnowledgeBase kb;
    FileTime stamp;

    AdviceService(Path kbPath, Path logDir) throws IOException {
        this.kbPath = kbPath;
        this.logDir = logDir;
        kb = KnowledgeBase.load(kbPath.toString());
        stamp = Files.getLastModifiedTime(kbPath);
        Files.createDirectories(logDir);
    }

    /**
     * Load the file again if it changed; keep the old one if the new one
     * cannot be read (a half-copied file, a mistake).
     */
    synchronized KnowledgeBase freshKb() throws IOException {
        FileTime now = Files.getLastModifiedTime(kbPath);
        if (!now.equals(stamp)) {
            try {
                kb = KnowledgeBase.load(kbPath.toString());
                System.err.println("reloaded " + kbPath);
            } catch (RuntimeException e) {
                System.err.println("could not reload " + kbPath
                        + "; keeping the old knowledge base");
            }
            stamp = now;
        }
        return kb;
    }

    Map<String, Object> health() throws IOException {
        KnowledgeBase k = freshKb();
        Map<String, Object> out = new LinkedHashMap<>();
        out.put("status", "ok");
        out.put("name", k.name);
        out.put("nodes", k.nodes.size());
        out.put("engine", ENGINE);
        return out;
    }
```

Listing 15.2 (Continued) *AdviceService.java: the same service in Java.*

```
    Map<String, Object> questions() throws IOException {
        KnowledgeBase k = freshKb();
        List<String> used = new ArrayList<>();
        for (Node n : k.nodes.values()) {
            n.conditions.forEach(c -> used.add(c.variable));
        }
        List<String> order = new ArrayList<>(k.usage().questionOrder());
        for (String v : k.variables.keySet()) {
            if (used.contains(v) && !order.contains(v)) {
                order.add(v);
            }
        }
        List<Object> list = new ArrayList<>();
        for (String v : order) {
            Map<String, Object> q = new LinkedHashMap<>();
            q.put("variable", v);
            q.put("question", k.variables.get(v).question);
            q.put("choices", new ArrayList<Object>(k.knownValues(v)));
            list.add(q);
        }
        return Map.of("questions", list);
    }

    Map<String, Object> advise(Map<?, ?> given, String time, boolean record)
            throws IOException {
        KnowledgeBase k = freshKb();
        Map<String, String> facts = new LinkedHashMap<>();
        given.forEach((name, value) -> {
            String n = String.valueOf(name).strip();
            String v = String.valueOf(value).strip();
            if (!n.isEmpty() && !v.isEmpty()) {
                facts.put(n, v);
            }
        });
        InferenceResult result = k.infer(Facts.map(facts));
        List<Object> advice = new ArrayList<>();
        for (Conclusion c : result.conclusions()) {
            advice.add(c.value);
        }
        String rule = result.paths.isEmpty() ? null
                : result.paths.get(0).ruleId();
        Map<String, Object> answer = new LinkedHashMap<>();
        answer.put("advice", advice);
        answer.put("rule", rule);
        answer.put("how", advice.isEmpty() ? List.of()
                : new ArrayList<Object>(result.how()));
```

Listing 15.2 (Continued) *AdviceService.java: the same service in Java.*

```
        if (record) {                    // a check is not a real consultation
            Map<String, Object> line = new LinkedHashMap<>();
            line.put("time", time);
            line.put("rule", rule == null ? "" : rule);
            line.put("advice", advice);
            line.put("facts", facts);
            log(line);
        }
        return answer;
    }

    /** One line of JSON per consultation, one file per month. */
    synchronized void log(Map<String, Object> line) throws IOException {
        String time = (String) line.get("time");
        Path file = logDir.resolve("usage-" + time.substring(0, 7) + ".jsonl");
        Files.writeString(file, compact(line) + "\n", StandardCharsets.UTF_8,
                StandardOpenOption.CREATE, StandardOpenOption.APPEND);
    }

    /** JSON on one line, written exactly as Python's json.dumps writes it. */
    static String compact(Object value) {
        if (value instanceof Map<?, ?> map) {
            List<String> parts = new ArrayList<>();
            map.forEach((k, v) -> parts.add(Json.quote(k.toString()) + ": "
                    + compact(v)));
            return "{" + String.join(", ", parts) + "}";
        }
        if (value instanceof List<?> list) {
            List<String> parts = new ArrayList<>();
            list.forEach(v -> parts.add(compact(v)));
            return "[" + String.join(", ", parts) + "]";
        }
        return value == null ? "null" : Json.quote(value.toString());
    }

    void reply(HttpExchange ex, int status, Object value) throws IOException {
        byte[] body = (Json.write(value) + "\n").getBytes(StandardCharsets.UTF_8);
        ex.getResponseHeaders().set("Content-Type",
                "application/json; charset=utf-8");
        ex.sendResponseHeaders(status, body.length);
        try (OutputStream out = ex.getResponseBody()) {
            out.write(body);
        }
    }
```

Listing 15.2 (Continued) *AdviceService.java: the same service in Java.*

```
    void handle(HttpExchange ex, boolean replay) throws IOException {
        String what = ex.getRequestMethod() + " " + ex.getRequestURI().getPath();
        switch (what) {
            case "GET /health" -> reply(ex, 200, health());
            case "GET /questions" -> reply(ex, 200, questions());
            case "POST /advise" -> {
                Object facts;
                try {
                    Object data = Json.parse(new String(
                            ex.getRequestBody().readAllBytes(),
                            StandardCharsets.UTF_8));
                    facts = data instanceof Map<?, ?> m ? m.get("facts") : null;
                } catch (VcirsException e) {
                    facts = null;
                }
                if (!(facts instanceof Map<?, ?> given)) {
                    reply(ex, 400, Map.of("error",
                            "send {\"facts\": {\"name\": \"value\"}}"));
                    return;
                }
                String time = replay
                        ? ex.getRequestHeaders().getFirst("X-Time") : null;
                boolean test = ex.getRequestHeaders().containsKey("X-Test");
                reply(ex, 200, advise(given, time == null
                        ? TIME.format(Instant.now()) : time, !test));
            }
            default -> reply(ex, 404, Map.of("error", "no such page"));
        }
    }

    public static void main(String[] args) throws IOException {
        Map<String, String> opt = new LinkedHashMap<>();
        for (int i = 0; i < args.length; i++) {
            if (args[i].startsWith("--") && i + 1 < args.length
                    && !args[i + 1].startsWith("--")) {
                opt.put(args[i], args[++i]);
            } else {
                opt.put(args[i], "");
            }
        }
        AdviceService service = new AdviceService(
                Path.of(opt.getOrDefault("--kb", "../data/cake-kb.json")),
                Path.of(opt.getOrDefault("--logs", "logs")));
        String host = opt.getOrDefault("--host", "127.0.0.1");
        int port = Integer.parseInt(opt.getOrDefault("--port", "4151"));
        boolean replay = opt.containsKey("--replay");
```

Listing 15.2 (Continued) *AdviceService.java: the same service in Java.*

```
        HttpServer server = HttpServer.create(new InetSocketAddress(host, port), 0);
        server.createContext("/", ex -> service.handle(ex, replay));
        server.start();
        System.out.println("Advice service for '" + service.kb.name + "' on http://"
                + host + ":" + port + "; Ctrl+C stops it.");
    }
}
```

Two details make the Java log lines identical to the Python ones. The body of every answer is written with `Json.write`, the same canonical writer as Python's `dumps` from vcirs-core. The log lines, which must fit on one line, are written by `compact`, which reproduces exactly what Python's `json.dumps` writes: `{"time": "...", "rule": "...", ...}`, with a space after every colon and comma. The time is formatted as `2026-09-25T08:00:00Z` in both languages, always in UTC, so that

logs from a shop in Makassar and a shop in Medan can be compared without thinking about time zones.

15.4 Step 2: Packaging and Deploying

One program, four commands. **Listing 15.3** is the entry point that the `.pyz` file runs.

Listing 15.3 `cake_doctor.py`: one program with four commands, the entry point of the `.pyz` file.

```
"""One program with four commands, so that one file can be deployed.

    python cake-doctor.pyz serve  [options of advice_service]
    python cake-doctor.pyz check  [options of service_check]
    python cake-doctor.pyz report [options of usage_report]
    python cake-doctor.pyz backup [options of backup]
"""

import sys

import advice_service
import backup
import service_check
import usage_report

COMMANDS = {"serve": advice_service.main, "check": service_check.main,
            "report": usage_report.main, "backup": backup.main}


def main(args):
    if not args or args[0] not in COMMANDS:
        print("usage: cake-doctor serve|check|report|backup [options]")
        sys.exit(1)
    COMMANDS[args[0]](args[1:])


if __name__ == "__main__":
    main(sys.argv[1:])
```

Listing 15.4 is the Java entry point that the `.jar` file names in its manifest.

Listing 15.4 `CakeDoctor.java`: the same four commands in Java.

```
import java.util.Arrays;

/**
 * One program with four commands, so that one file can be deployed.
 * java -jar cake-doctor.jar serve|check|report|backup [options]
 */
public class CakeDoctor {
    public static void main(String[] args) throws Exception {
        String[] rest = args.length > 0
                ? Arrays.copyOfRange(args, 1, args.length) : args;
        switch (args.length > 0 ? args[0] : "") {
            case "serve" -> AdviceService.main(rest);
            case "check" -> ServiceCheck.main(rest);
            case "report" -> UsageReport.main(rest);
            case "backup" -> Backup.main(rest);
            default -> {
                System.out.println("usage: cake-doctor serve|check|report|backup "
                        + "[options]");
                System.exit(1);
            }
        }
    }
}
```

Building the `.pyz`. **Listing 15.5** copies the four Python files and the vcirs-core package into a temporary folder, puts `cake_doctor.py` there as `__main__.py`, and asks `zipapp` to pack the folder into `dist/cake-doctor.pyz`, compressed, with a first line that lets Linux and macOS run it directly [2].

Listing 15.5 `build_pyz.py`: packs the service, its tools and vcirs-core into one `.pyz` file.

```
"""Build ../dist/cake-doctor.pyz: the service, its tools and vcirs-core in
one file that any Python 3.10 or later can run (python cake-doctor.pyz ...).

    python build_pyz.py
"""

import os
import pathlib
import shutil
import tempfile
import zipapp

HERE = pathlib.Path(__file__).resolve().parent
ENGINE = HERE.parents[1] / "vcirs-core" / "python" / "vcirs"
DIST = HERE.parent / "dist"
FILES = ["advice_service.py", "backup.py", "service_check.py",
         "usage_report.py"]


def main():
    with tempfile.TemporaryDirectory() as tmp:
        root = pathlib.Path(tmp)
        for name in FILES:
            shutil.copy(HERE / name, root / name)
        shutil.copy(HERE / "cake_doctor.py", root / "__main__.py")
        shutil.copytree(ENGINE, root / "vcirs",
                        ignore=shutil.ignore_patterns("__pycache__"))
        DIST.mkdir(exist_ok=True)
        target = DIST / "cake-doctor.pyz"
        zipapp.create_archive(root, target, interpreter="/usr/bin/env python3",
                              compressed=True)
    print("built " + os.path.relpath(target, HERE.parent) + " ("
          + str(target.stat().st_size // 1024) + " KB)")


if __name__ == "__main__":
    main()
```

Running `python build_pyz.py` in `Chapter15/python` prints:

```
built dist/cake-doctor.pyz (34 KB)
```

Thirty-four kilobytes: the whole cake doctor, engine included, is smaller than one photo of a cake.

Building the `.jar`. Java needs no script; the JDK's `jar` tool does it [1]. From `Chapter15/java`:

```
javac -d build -sourcepath ../../vcirs-core/java/src CakeDoctor.java \
      AdviceService.java ServiceCheck.java UsageReport.java Backup.java
jar --create --file ../dist/cake-doctor.jar --main-class CakeDoctor -C build .
```

`javac` compiles the five classes and, through `-sourcepath`, exactly the engine classes they use; `jar --create` packs the compiled folder and writes `Main-Class: CakeDoctor` into the manifest. (On Windows, write the first command on one line without the `\`.) The result is about 94 kilobytes.

Figure 15.1 shows what the finished deployment looks like in Bu Wati's shop.

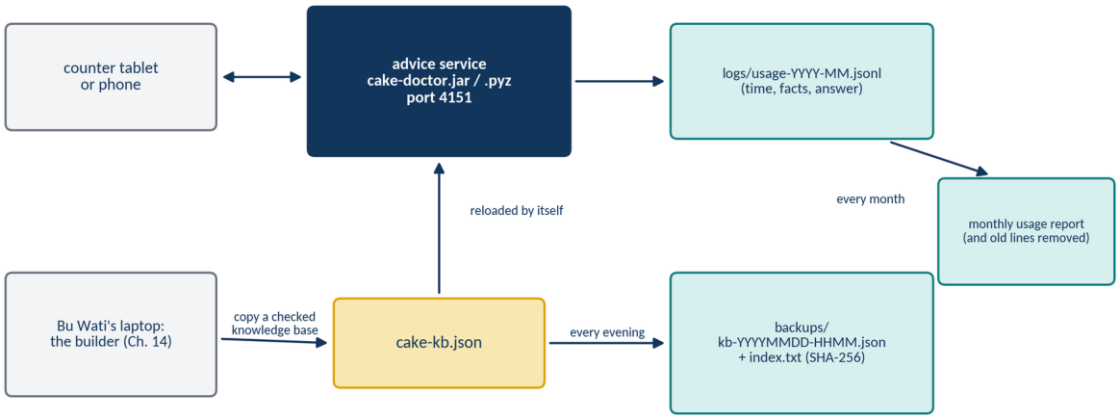

Figure 15.1 The cake doctor deployed. The service runs from one file on the shop computer and answers the counter's tablet or phone; it reloads the knowledge base when a checked copy arrives from Bu Wati's laptop, writes a usage log, and is backed up every evening and reported on every month.

Installing it. On the shop computer, make a folder — `C:\cake-doctor` on Windows, `/opt/cake-doctor` on Linux, a folder in your home folder on macOS — and put in it the `.jar` (or the `.pyz`), a `data` folder with `cake-kb.json`, and empty `logs` and `backups` folders. The computer needs Java 17 or later (Chapter 1 installed JDK 25) or Python 3.10 or later (Chapter 1 installed Python 3.14). Test it by hand first: `java -jar cake-doctor.jar serve --kb data/cake-kb.json --logs logs`, and open `http://localhost:4151/health` in a browser.

Then make it start by itself. The chapter's `deploy` folder holds a ready-made file for each system.

On **Linux**, the service manager is **systemd**, and a service is described by a small *unit file* [7]:

```
# systemd unit for Linux: copy to /etc/systemd/system/cake-doctor.service,
# then: sudo systemctl daemon-reload && sudo systemctl enable --now cake-doctor
[Unit]
Description=Cake doctor advice service (VCIRS)
After=network.target

[Service]
User=cakedoctor
WorkingDirectory=/opt/cake-doctor
ExecStart=/usr/bin/java -jar /opt/cake-doctor/cake-doctor.jar serve \
    --kb /opt/cake-doctor/data/cake-kb.json \
    --logs /opt/cake-doctor/logs --port 4151
# Python instead of Java: the same line with
# /usr/bin/python3 /opt/cake-doctor/cake-doctor.pyz instead of java -jar ...
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

Copy it to `/etc/systemd/system/cake-doctor.service`, create a user `cakedoctor` that owns the folder (`sudo useradd --system cakedoctor`, `sudo chown -R cakedoctor /opt/cake-doctor`), and run `sudo systemctl daemon-reload` and `sudo systemctl enable --now cake-doctor`. `Restart=on-failure` restarts the service if it ever crashes; `sudo systemctl status cake-doctor` shows whether it runs, and `journalctl -u cake-doctor` shows its messages, including any "could not reload" warning. Running it as its own user, not as `root`, means that even a mistake in the service cannot touch the rest of the computer.

On **Windows**, `start-windows.cmd` starts the service from `C:\cake-doctor`:

```
@echo off
rem Start the cake doctor on Windows. Put this file, cake-doctor.jar (or
rem cake-doctor.pyz) and the data folder in C:\cake-doctor. To start it with
rem Windows, add it in Task Scheduler: Create Task > Triggers: At startup.
cd /d C:\cake-doctor
java -jar cake-doctor.jar serve --kb data\cake-kb.json --logs logs --port 4151
rem Python instead of Java:
rem python cake-doctor.pyz serve --kb data\cake-kb.json --logs logs --port 4151
```

Open **Task Scheduler**, choose *Create Task*, give it a name, add a trigger *At startup*, and an action *Start a program* with `C:\cake-doctor\start-windows.cmd`; under *General*, choose *Run whether user is logged on or not*. The first time Java opens a network port, Windows Firewall may ask for permission: allow it on *private* networks only.

On **macOS**, the equivalent of a unit file is a *launch agent*: a small `.plist` file in `~/Library/LaunchAgents` that names the program and its arguments and sets `RunAtLoad` and `KeepAlive` to true; `launchctl load` activates it. Exercise 15.6 asks you to write one.

Backups and reports on a schedule. On Linux and macOS, two tiny scripts do the two regular jobs. `backup.sh` makes the evening backup:

```
#!/bin/sh
# The evening backup (Linux, macOS): keep the newest 30 copies.
cd /opt/cake-doctor || exit 1
java -jar cake-doctor.jar backup save data/cake-kb.json backups --keep 30
```

and `report.sh` writes the monthly report into a file named after the month, removing log lines older than 180 days on the way:

```
#!/bin/sh
# The monthly report; it also removes log lines older than 180 days.
cd /opt/cake-doctor || exit 1
mkdir -p reports
java -jar cake-doctor.jar report logs --kb data/cake-kb.json \
    --keep-days 180 > "reports/$(date +%Y-%m).txt"
```

Copy both scripts into the service's folder, and two lines added with `crontab -e` run them at the right times:

```
# crontab -e (Linux, macOS): the backup every evening at 21:30, and the
# report at 7:00 on the first day of every month.
30 21 * * * /opt/cake-doctor/backup.sh
0 7 1 * *   /opt/cake-doctor/report.sh
```

The five fields before each command are minute, hour, day of the month, month and day of the week, and `*` means "any": the backup runs every day at 21:30, the report at 7:00 on the first day of every month. On Windows, `backup-windows.cmd` in the same folder does the backup and is scheduled in Task Scheduler with a *Daily* trigger. Remember the other two copies of the 3-2-1 rule: the backups folder should be on a USB stick or second disk, and once a week one backup goes to another place.

15.5 Step 3: Checking a Deployment

Every time the service is installed, updated or moved, check it. A check that takes ten seconds and is always the same catches the mistakes that are easy to make and hard to notice: the wrong knowledge base copied, an old version of the file, a firewall that blocks the port, a Java that is too old. **Listing 15.6** is the check.

Listing 15.6 `service_check.py`: the check after every deployment, and the replay of recorded consultations.

```
"""Check a running advice service, the way you would after every deployment.

    python service_check.py [URL] [--replay visits.txt]

Without --replay it asks /health, /questions and two pieces of advice, and
says whether everything answered as expected. With --replay it sends every
line "time | name = value; name = value" of the file to /advise.
"""

import json
import sys
import urllib.request

OPENER = urllib.request.build_opener(urllib.request.ProxyHandler({}))


def call(url, body=None, time=None):
    data = None if body is None else json.dumps(body).encode("utf-8")
    request = urllib.request.Request(url, data=data)
    request.add_header("Content-Type", "application/json")
    if time:
        request.add_header("X-Time", time)
    else:
        request.add_header("X-Test", "yes")        # do not count checks
    with OPENER.open(request, timeout=10) as response:
        return json.loads(response.read().decode("utf-8"))


def facts_of(text):
    """'problem = dry; baking time = long' -> {'problem': 'dry', ...}"""
    out = {}
    for part in text.split(";"):
        name, value = part.split("=")
        out[name.strip()] = value.strip()
    return out


def advise(url, text):
    answer = call(url + "/advise", {"facts": facts_of(text)})
    if not answer["advice"]:
        print("advise " + text + " -> (no answer yet)")
    else:
        print("advise " + text + " -> " + ", ".join(answer["advice"])
              + " (rule " + answer["rule"] + ")")
    return answer
```

Listing 15.6 (Continued) *service_check.py: the check after every deployment, and the replay of recorded consultations.*

```
def main(args):
    urls = [a for a in args if a.startswith("http")]
    url = urls[0] if urls else "http://localhost:4151"
    if "--replay" in args:
        path = args[args.index("--replay") + 1]
        count = 0
        with open(path, encoding="utf-8") as f:
            for line in f:
                if "|" in line:
                    time, text = line.split("|", 1)
                    call(url + "/advise", {"facts": facts_of(text)},
                         time.strip())
                    count += 1
        print("replayed " + str(count) + " consultations to " + url)
        return
    print("Checking " + url)
    try:
        health = call(url + "/health")
    except OSError:
        print("FAIL: the service does not answer")
        sys.exit(1)
    print("health: " + health["status"] + ", " + health["name"] + ", "
          + str(health["nodes"]) + " nodes, " + health["engine"])
    names = [q["variable"] for q in call(url + "/questions")["questions"]]
    print("questions: " + ", ".join(names))
    good = advise(url, "problem = dry; baking time = long")
    none = advise(url, "problem = sticky top")
    ok = good["advice"] == ["take it out earlier"] and not none["advice"]
    print("all checks passed" if ok else "FAIL: unexpected advice")
    if not ok:
        sys.exit(1)


if __name__ == "__main__":
    main(sys.argv[1:])
```

It asks `/health` and `/questions`, then two pieces of advice whose answers are known: a dry cake baked too long must get *take it out earlier*, and a sticky top, which the knowledge base has never been taught, must get no answer. The second test is as important as the first. A deployment that suddenly gives advice for a sticky top is running the wrong knowledge base. Every request of the check carries `X-Test`, so none of them appears in the usage log. If anything is wrong, the program says `FAIL` and ends with exit code 1, which a script or a scheduled task can detect.

Listing 15.7 is the same check in Java.

Listing 15.7 `ServiceCheck.java`: the same check in Java.

```
import java.io.IOException;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.time.Duration;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

import vcirs.Json;

/**
 * Check a running advice service, the way you would after every deployment.
 * java ServiceCheck [URL] [--replay visits.txt]
 * Without --replay it asks /health, /questions and two pieces of advice, and
 * says whether everything answered as expected. With --replay it sends every
 * line "time | name = value; name = value" of the file to /advise.
 */
public class ServiceCheck {
    static final HttpClient HTTP = HttpClient.newBuilder()
            .proxy(HttpClient.Builder.NO_PROXY).build();

    @SuppressWarnings("unchecked")
    static Map<String, Object> call(String url, Object body, String time)
            throws IOException, InterruptedException {
        HttpRequest.Builder r = HttpRequest.newBuilder(URI.create(url))
                .timeout(Duration.ofSeconds(10))
                .header("Content-Type", "application/json");
        if (time != null) {
            r.header("X-Time", time);
        } else {
            r.header("X-Test", "yes");               // do not count checks
        }
        r = body == null ? r.GET() : r.POST(HttpRequest.BodyPublishers
                .ofString(Json.write(body), StandardCharsets.UTF_8));
        HttpResponse<String> response = HTTP.send(r.build(),
                HttpResponse.BodyHandlers.ofString(StandardCharsets.UTF_8));
        return (Map<String, Object>) Json.parse(response.body());
    }
```

Listing 15.7 (Continued) *ServiceCheck.java: the same check in Java.*

```
    /** "problem = dry; baking time = long" -> {problem=dry, ...} */
    static Map<String, Object> factsOf(String text) {
        Map<String, Object> out = new LinkedHashMap<>();
        for (String part : text.split(";")) {
            String[] kv = part.split("=");
            out.put(kv[0].strip(), kv[1].strip());
        }
        return out;
    }

    @SuppressWarnings("unchecked")
    static List<Object> advise(String url, String text)
            throws IOException, InterruptedException {
        Map<String, Object> answer = call(url + "/advise",
                Map.of("facts", factsOf(text)), null);
        List<Object> advice = (List<Object>) answer.get("advice");
        if (advice.isEmpty()) {
            System.out.println("advise " + text + " -> (no answer yet)");
        } else {
            System.out.println("advise " + text + " -> "
                    + String.join(", ", advice.stream().map(String::valueOf)
                            .toList()) + " (rule " + answer.get("rule") + ")");
        }
        return advice;
    }

    @SuppressWarnings("unchecked")
    public static void main(String[] args) throws Exception {
        String url = "http://localhost:4151";
        String replay = null;
        for (int i = 0; i < args.length; i++) {
            if (args[i].equals("--replay") && i + 1 < args.length) {
                replay = args[++i];
            } else if (args[i].startsWith("http")) {
                url = args[i];
            }
        }
        if (replay != null) {
            int count = 0;
            for (String line : Files.readAllLines(Path.of(replay),
                    StandardCharsets.UTF_8)) {
                int bar = line.indexOf('|');
                if (bar >= 0) {
                    call(url + "/advise", Map.of("facts",
                            factsOf(line.substring(bar + 1))),
                            line.substring(0, bar).strip());
```

Listing 15.7 (Continued) *ServiceCheck.java: the same check in Java.*

```
                    count++;
                }
            }
            System.out.println("replayed " + count + " consultations to " + url);
            return;
        }
        System.out.println("Checking " + url);
        Map<String, Object> health;
        try {
            health = call(url + "/health", null, null);
        } catch (IOException e) {
            System.out.println("FAIL: the service does not answer");
            System.exit(1);
            return;
        }
        System.out.println("health: " + health.get("status") + ", "
                + health.get("name") + ", " + health.get("nodes") + " nodes, "
                + health.get("engine"));
        List<String> names = new ArrayList<>();
        for (Object q : (List<Object>) call(url + "/questions", null, null)
                .get("questions")) {
            names.add((String) ((Map<String, Object>) q).get("variable"));
        }
        System.out.println("questions: " + String.join(", ", names));
        List<Object> good = advise(url, "problem = dry; baking time = long");
        List<Object> none = advise(url, "problem = sticky top");
        boolean ok = good.equals(List.of("take it out earlier")) && none.isEmpty();
        System.out.println(ok ? "all checks passed" : "FAIL: unexpected advice");
        if (!ok) {
            System.exit(1);
        }
    }
}
```

Here is the deployment in Bu Wati's shop folder, started from the `.jar` with `java -jar cake-doctor.jar serve --kb data/cake-kb.json --logs logs --replay`, which prints:

```
Advice service for 'Cake doctor' on http://127.0.0.1:4151; Ctrl+C stops it.
```

and checked from another terminal with the `.pyz`, `python cake-doctor.pyz check`:

```
Checking http://localhost:4151
health: ok, Cake doctor, 12 nodes, vcirs-core 1.0.0
questions: problem, oven heat, oven door opened early, eggs, flour mixing, baking time, pan
advise problem = dry; baking time = long -> take it out earlier (rule take it out
earlier#1)
advise problem = sticky top -> (no answer yet)
all checks passed
```

The Java service and the Python check work together without any adjustment, because both speak the same HTTP and the same JSON; the Python service with the Java check prints the same lines. The service can also be asked by hand. In a browser, `http://localhost:4151/health` shows:

```
{
  "status": "ok",
  "name": "Cake doctor",
  "nodes": 12,
  "engine": "vcirs-core 1.0.0"
}
```

and a consultation can be sent with `curl`, the command-line HTTP tool included with Windows 10 and 11, macOS and most Linux systems: `curl -X POST -H "X-Test: yes" -d "{\"facts\":`

```
{\"problem\": \"dry\", \"baking time\": \"long\"}" http://localhost:4151/advise
```

answers

```
{
  "advice": [
    "take it out earlier"
  ],
  "rule": "take it out earlier#1",
  "how": [
    "take it out earlier#1 fired because:",
    "  problem = dry  (given, CF 100)",
    "  baking time = long  (given, CF 100)",
    "  so advice = take it out earlier  [rule CF 100, conclusion CF 100]"
  ]
}
```

with the advice, the rule and the full explanation. Any program in any language that can send HTTP — a phone app, a spreadsheet macro, the chat front end of Chapter 13 — can use the cake doctor in exactly this way.

15.6 Step 4: Backups

Listing 15.8 is the backup tool.

Listing 15.8 `backup.py`: save, check and restore backups of a knowledge base, with SHA-256 fingerprints.

```
"""Back up a knowledge base, check the backups, and restore one.

    python backup.py save    KB BACKUPDIR [--keep 30] [--stamp YYYYMMDD-HHMM]
    python backup.py check   BACKUPDIR
    python backup.py restore BACKUPDIR NAME KB

save copies the file only if it loads as a knowledge base, names the copy
after the time, writes its SHA-256 fingerprint to BACKUPDIR/index.txt and
keeps only the newest --keep copies. check reads every copy again and
compares it with its fingerprint. restore puts a copy back (after checking
it), keeping the file it replaces as KB.before-restore.
"""

import datetime
import hashlib
import os
import pathlib
import shutil
import sys

sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import KnowledgeBase, VcirsError                    # noqa: E402


def fingerprint(path):
    with open(path, "rb") as f:
        return hashlib.sha256(f.read()).hexdigest()


def loads(path):
    """The knowledge base in the file, or None if it cannot be read."""
    try:
        return KnowledgeBase.load(path)
    except (OSError, ValueError, KeyError, TypeError, VcirsError):
        return None


def read_index(folder):
    path = os.path.join(folder, "index.txt")
    if not os.path.exists(path):
        return []
    with open(path, encoding="utf-8") as f:
        return [line.split() for line in f if line.strip()]
```

Listing 15.8 (Continued) *backup.py: save, check and restore backups of a knowledge base, with SHA-256 fingerprints.*

```
def write_index(folder, rows):
    with open(os.path.join(folder, "index.txt"), "w", encoding="utf-8",
              newline="\n") as f:
        f.writelines(" ".join(r) + "\n" for r in rows)


def save(kb_path, folder, keep, stamp):
    kb = loads(kb_path)
    if kb is None:
        print("NOT saved: " + kb_path + " is not a readable knowledge base")
        return False
    os.makedirs(folder, exist_ok=True)
    name = "kb-" + stamp + ".json"
    shutil.copyfile(kb_path, os.path.join(folder, name))
    rows = [r for r in read_index(folder) if r[0] != name]
    rows.append([name, fingerprint(kb_path), str(len(kb.nodes))])
    for old in rows[:-keep] if len(rows) > keep else []:
        os.remove(os.path.join(folder, old[0]))
        print("removed old backup " + old[0])
    write_index(folder, rows[-keep:])
    print("saved " + name + " (" + str(len(kb.nodes)) + " nodes, sha256 "
          + rows[-1][1][:12] + "...)")
    return True


def check(folder):
    ok = True
    for name, digest, nodes in read_index(folder):
        path = os.path.join(folder, name)
        if not os.path.exists(path):
            state = "MISSING"
        elif fingerprint(path) != digest:
            state = "DAMAGED (fingerprint differs)"
        elif loads(path) is None:
            state = "DAMAGED (cannot be read)"
        else:
            state = "ok, " + nodes + " nodes"
        ok = ok and state.startswith("ok")
        print(name + ": " + state)
    return ok
```

Listing 15.8 (Continued) *backup.py: save, check and restore backups of a knowledge base, with SHA-256 fingerprints.*

```
def restore(folder, name, kb_path):
    rows = [r for r in read_index(folder) if r[0] == name]
    path = os.path.join(folder, name)
    if not rows or not os.path.exists(path) \
            or fingerprint(path) != rows[0][1] or loads(path) is None:
        print("NOT restored: " + name + " is missing or damaged")
        return False
    if os.path.exists(kb_path):
        shutil.copyfile(kb_path, kb_path + ".before-restore")
    shutil.copyfile(path, kb_path)
    print("restored " + name + " to " + kb_path)
    return True


def main(args):
    def option(name, default):
        return args[args.index(name) + 1] if name in args else default

    words = [a for i, a in enumerate(args) if not a.startswith("--")
             and (i == 0 or not args[i - 1].startswith("--"))]
    stamp = option("--stamp", datetime.datetime.now().strftime("%Y%m%d-%H%M"))
    if len(words) >= 3 and words[0] == "save":
        good = save(words[1], words[2], int(option("--keep", "30")), stamp)
    elif len(words) >= 2 and words[0] == "check":
        good = check(words[1])
    elif len(words) >= 4 and words[0] == "restore":
        good = restore(words[1], words[2], words[3])
    else:
        print("usage: backup save KB DIR [--keep N] [--stamp S] | check DIR"
              " | restore DIR NAME KB")
        good = False
    sys.exit(0 if good else 1)


if __name__ == "__main__":
    main(sys.argv[1:])
```

`save` refuses to back up a file that does not load as a knowledge base; copies it under a name made from the date and time, `kb-20260925-2130.json`, so that the names sort in time order; appends the name, the fingerprint and the number of nodes to `index.txt`; and, when there are more copies than `--keep`, removes the oldest. `check` goes through the index: a missing file, a file whose fingerprint differs, and a file that does not load are all reported, and the exit code is 1 if anything is wrong. `restore` checks the chosen copy the same way before using it, and saves the current file as `…before-restore` first. **Listing 15.9** is the Java version; `MessageDigest` computes SHA-256 and `HexFormat` writes it as hexadecimal digits.

Listing 15.9 `Backup.java`: the same backup tool in Java.

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardCopyOption;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.HexFormat;
import java.util.List;

import vcirs.KnowledgeBase;

/**
 * Back up a knowledge base, check the backups, and restore one.
 * java Backup save    KB BACKUPDIR [--keep 30] [--stamp YYYYMMDD-HHMM]
 * java Backup check   BACKUPDIR
 * java Backup restore BACKUPDIR NAME KB
 * save copies the file only if it loads as a knowledge base, names the copy
 * after the time, writes its SHA-256 fingerprint to BACKUPDIR/index.txt and
 * keeps only the newest --keep copies. check reads every copy again and
 * compares it with its fingerprint. restore puts a copy back (after checking
 * it), keeping the file it replaces as KB.before-restore.
 */
public class Backup {
    static String fingerprint(Path path) throws IOException {
        try {
            MessageDigest sha = MessageDigest.getInstance("SHA-256");
            return HexFormat.of().formatHex(sha.digest(Files.readAllBytes(path)));
        } catch (NoSuchAlgorithmException e) {
            throw new IllegalStateException(e);
        }
    }

    /** The knowledge base in the file, or null if it cannot be read. */
    static KnowledgeBase loads(Path path) {
        try {
            return KnowledgeBase.load(path.toString());
        } catch (RuntimeException e) {
            return null;
        }
    }
```

Listing 15.9 (Continued) *Backup.java: the same backup tool in Java.*

```
    static List<String[]> readIndex(Path folder) throws IOException {
        Path path = folder.resolve("index.txt");
        List<String[]> rows = new ArrayList<>();
        if (Files.exists(path)) {
            for (String line : Files.readAllLines(path, StandardCharsets.UTF_8)) {
                if (!line.isBlank()) {
                    rows.add(line.strip().split("\\s+"));
                }
            }
        }
        return rows;
    }

    static void writeIndex(Path folder, List<String[]> rows) throws IOException {
        StringBuilder b = new StringBuilder();
        rows.forEach(r -> b.append(String.join(" ", r)).append("\n"));
        Files.writeString(folder.resolve("index.txt"), b, StandardCharsets.UTF_8);
    }

    static boolean save(Path kbPath, Path folder, int keep, String stamp)
            throws IOException {
        KnowledgeBase kb = loads(kbPath);
        if (kb == null) {
            System.out.println("NOT saved: " + kbPath
                    + " is not a readable knowledge base");
            return false;
        }
        Files.createDirectories(folder);
        String name = "kb-" + stamp + ".json";
        Files.copy(kbPath, folder.resolve(name),
                StandardCopyOption.REPLACE_EXISTING);
        List<String[]> rows = new ArrayList<>();
        for (String[] r : readIndex(folder)) {
            if (!r[0].equals(name)) {
                rows.add(r);
            }
        }
        rows.add(new String[] {name, fingerprint(kbPath),
            String.valueOf(kb.nodes.size())});
        while (rows.size() > keep) {
            Files.delete(folder.resolve(rows.get(0)[0]));
            System.out.println("removed old backup " + rows.get(0)[0]);
            rows.remove(0);
        }
        writeIndex(folder, rows);
        String[] last = rows.get(rows.size() - 1);
```

Listing 15.9 (Continued) *Backup.java: the same backup tool in Java.*

```
        System.out.println("saved " + name + " (" + last[2] + " nodes, sha256 "
                + last[1].substring(0, 12) + "...)");
        return true;
    }

    static boolean check(Path folder) throws IOException {
        boolean ok = true;
        for (String[] r : readIndex(folder)) {
            Path path = folder.resolve(r[0]);
            String state;
            if (!Files.exists(path)) {
                state = "MISSING";
            } else if (!fingerprint(path).equals(r[1])) {
                state = "DAMAGED (fingerprint differs)";
            } else if (loads(path) == null) {
                state = "DAMAGED (cannot be read)";
            } else {
                state = "ok, " + r[2] + " nodes";
            }
            ok = ok && state.startsWith("ok");
            System.out.println(r[0] + ": " + state);
        }
        return ok;
    }

    static boolean restore(Path folder, String name, Path kbPath)
            throws IOException {
        String[] row = null;
        for (String[] r : readIndex(folder)) {
            if (r[0].equals(name)) {
                row = r;
            }
        }
        Path path = folder.resolve(name);
        if (row == null || !Files.exists(path)
                || !fingerprint(path).equals(row[1]) || loads(path) == null) {
            System.out.println("NOT restored: " + name + " is missing or damaged");
            return false;
        }
        if (Files.exists(kbPath)) {
            Files.copy(kbPath, Path.of(kbPath + ".before-restore"),
                    StandardCopyOption.REPLACE_EXISTING);
        }
        Files.copy(path, kbPath, StandardCopyOption.REPLACE_EXISTING);
        System.out.println("restored " + name + " to " + kbPath);
        return true;
```

Listing 15.9 (Continued) *Backup.java: the same backup tool in Java.*

```
    }

    public static void main(String[] args) throws IOException {
        List<String> words = new ArrayList<>();
        int keep = 30;
        String stamp = LocalDateTime.now().format(
                DateTimeFormatter.ofPattern("yyyyMMdd-HHmm"));
        for (int i = 0; i < args.length; i++) {
            if (args[i].equals("--keep") && i + 1 < args.length) {
                keep = Integer.parseInt(args[++i]);
            } else if (args[i].equals("--stamp") && i + 1 < args.length) {
                stamp = args[++i];
            } else {
                words.add(args[i]);
            }
        }
        boolean good;
        String what = words.isEmpty() ? "" : words.get(0);
        if (what.equals("save") && words.size() >= 3) {
            good = save(Path.of(words.get(1)), Path.of(words.get(2)), keep, stamp);
        } else if (what.equals("check") && words.size() >= 2) {
            good = check(Path.of(words.get(1)));
        } else if (what.equals("restore") && words.size() >= 4) {
            good = restore(Path.of(words.get(1)), words.get(2),
                    Path.of(words.get(3)));
        } else {
            System.out.println("usage: backup save KB DIR [--keep N] [--stamp S]"
                    + " | check DIR | restore DIR NAME KB");
            good = false;
        }
        System.exit(good ? 0 : 1);
    }
}
```

Here is a week of backups compressed into a few minutes. `--stamp` gives each backup the name of the evening it stands for, and `--keep 2` keeps only two, so that we can see the oldest one removed:

```
$ java -jar cake-doctor.jar backup save data/cake-kb.json backups --keep 2 --stamp
20260925-2130
saved kb-20260925-2130.json (12 nodes, sha256 b54cda04bbc9...)
$ java -jar cake-doctor.jar backup save data/cake-kb.json backups --keep 2 --stamp
20260926-2130
saved kb-20260926-2130.json (12 nodes, sha256 b54cda04bbc9...)
$ java -jar cake-doctor.jar backup save data/cake-kb.json backups --keep 2 --stamp
20260927-2130
removed old backup kb-20260925-2130.json
saved kb-20260927-2130.json (12 nodes, sha256 b54cda04bbc9...)
$ java -jar cake-doctor.jar backup check backups
kb-20260926-2130.json: ok, 12 nodes
kb-20260927-2130.json: ok, 12 nodes
```

Three saves, the oldest removed at the third, and a check that finds both remaining copies intact. The fingerprints are the same because the knowledge base did not change in between; in a real week they would differ. Now we damage one backup on purpose by appending a line to it, as a failing USB stick might, and check again:

```
$ java -jar cake-doctor.jar backup check backups
kb-20260926-2130.json: DAMAGED (fingerprint differs)
kb-20260927-2130.json: ok, 12 nodes
$ java -jar cake-doctor.jar backup restore backups kb-20260926-2130.json data/cake-kb.json
NOT restored: kb-20260926-2130.json is missing or damaged
$ java -jar cake-doctor.jar backup restore backups kb-20260927-2130.json data/cake-kb.json
restored kb-20260927-2130.json to data/cake-kb.json
```

The check finds the damage at once. The restore of the damaged copy is refused; the restore of the good copy succeeds, and the previous `cake-kb.json` is kept as `cake-kb.json.before-restore`. On the Monday of Section 15.1, this is the command Bu Wati's nephew would have typed on the new laptop, and the shop would have lost at most one day of teaching.

15.7 Step 5: Monitoring Rule Usage

From July to September 2026, Bu Wati's helpers wrote each consultation on a card by the ovens — the date, the facts, what the doctor said — while the service was being prepared. The chapter's `data/visits.txt` holds those 120 consultations, one per line, such as

```
2026-07-01T04:16:00Z | problem = sank in the middle; oven door opened early = no; oven
heat = low
```

and `python cake-doctor.pyz check --replay visits.txt`, sent to the service started with `--replay`, enters them into the log with their real dates:

```
replayed 120 consultations to http://localhost:4151
```

Each consultation became one line of the log file of its month; the first line of `logs/usage-2026-07.jsonl` is

```
{"time": "2026-07-01T04:16:00Z", "rule": "heat the oven fully first#1", "advice": ["heat
the oven fully first"], "facts": {"problem": "sank in the middle", "oven door opened
early": "no", "oven heat": "low"}}
```

— a time, the node, the advice and three facts about a cake. Nothing in it identifies a person. **Listing 15.10** is the tool that reads these lines.

Listing 15.10 `usage_report.py`: the monthly usage report, and the removal of old log lines.

```
"""Read the usage logs of the advice service and say what they show.

    python usage_report.py [LOGDIR] [--kb FILE] [--keep-days N --today DATE]

It counts consultations per month, how many got an answer, which nodes gave
the advice (and which never did), and which unanswered cases came back most
often: those are the cases to teach next. With --keep-days it also deletes
the log lines older than N days before DATE (default: today), so that the
logs are not kept longer than they are needed.
"""

import datetime
import json
import os
import pathlib
import sys

sys.path.insert(0, str(pathlib.Path(__file__).resolve().parents[2]
                       / "vcirs-core" / "python"))
from vcirs import KnowledgeBase, num                          # noqa: E402

MONTHS = ["January", "February", "March", "April", "May", "June", "July",
          "August", "September", "October", "November", "December"]


def read_logs(log_dir):
    """Every record of every usage-*.jsonl file, oldest file first."""
    records = []
    for name in sorted(os.listdir(log_dir)):
        if name.startswith("usage-") and name.endswith(".jsonl"):
            with open(os.path.join(log_dir, name), encoding="utf-8") as f:
                records += [json.loads(line) for line in f if line.strip()]
    return records


def prune(log_dir, keep_days, today):
    """Delete records older than keep_days before today; return how many."""
    limit = (today - datetime.timedelta(days=keep_days)).isoformat()
    removed = 0
    for name in sorted(os.listdir(log_dir)):
        if not (name.startswith("usage-") and name.endswith(".jsonl")):
            continue
        path = os.path.join(log_dir, name)
        with open(path, encoding="utf-8") as f:
            lines = [line for line in f if line.strip()]
        keep = [line for line in lines if json.loads(line)["time"] >= limit]
```

Listing 15.10 (Continued) *usage_report.py: the monthly usage report, and the removal of old log lines.*

```
        removed += len(lines) - len(keep)
        if not keep:
            os.remove(path)
        elif len(keep) < len(lines):
            with open(path, "w", encoding="utf-8", newline="\n") as f:
                f.writelines(keep)
    return removed


def percent(part, whole):
    return num.fixed(100.0 * part / whole, 1) + "%" if whole else "-"


def report(records, kb):
    if not records:
        return ["No consultations in the logs."]
    out = ["Consultations: " + str(len(records)) + ", from "
           + records[0]["time"][:10] + " to " + records[-1]["time"][:10]]
    months = {}
    for r in records:
        months[r["time"][:7]] = months.get(r["time"][:7], 0) + 1
    out.append("  " + ", ".join(MONTHS[int(m[5:]) - 1] + " " + m[:4] + ": "
                                + str(n) for m, n in months.items()))
    answered = [r for r in records if r["rule"]]
    out.append("Answered: " + str(len(answered)) + " ("
               + percent(len(answered), len(records)) + "), no answer: "
               + str(len(records) - len(answered)) + " ("
               + percent(len(records) - len(answered), len(records)) + ")")
    counts = {}
    for r in answered:
        counts[r["rule"]] = counts.get(r["rule"], 0) + 1
    out.append("Advice given, most used first:")
    for rule, n in sorted(counts.items(), key=lambda x: (-x[1], x[0])):
        out.append("  %4d  %-6s %s" % (n, percent(n, len(records)), rule))
    if kb is not None:
        idle = [n.id for n in kb.nodes.values()
                if n.id not in counts and not n.stopped]
        out.append("Nodes that never gave the advice: "
                   + (", ".join(idle) if idle else "none"))
    missing = {}
    for r in records:
        if not r["rule"]:
            key = "; ".join(k + " = " + v for k, v in r["facts"].items())
            missing[key] = missing.get(key, 0) + 1
    out.append("Cases to teach next (no answer), most frequent first:")
    for key, n in sorted(missing.items(), key=lambda x: (-x[1], x[0])):
```

Listing 15.10 (Continued) *usage_report.py: the monthly usage report, and the removal of old log lines.*

```
        out.append("  %4d  %s" % (n, key))
    return out


def main(args):
    def option(name, default):
        return args[args.index(name) + 1] if name in args else default

    values = [args[i] for i in range(len(args)) if not args[i].startswith("--")
              and (i == 0 or not args[i - 1].startswith("--"))]
    log_dir = values[0] if values else "logs"
    kb_path = option("--kb", None)
    if "--keep-days" in args:
        today = datetime.date.fromisoformat(
            option("--today", datetime.date.today().isoformat()))
        removed = prune(log_dir, int(option("--keep-days", "0")), today)
        print("Removed " + str(removed) + " log lines older than "
              + option("--keep-days", "0") + " days before "
              + today.isoformat() + ".")
    kb = KnowledgeBase.load(kb_path) if kb_path else None
    print("Usage report for " + log_dir
          + (" and " + kb.name if kb else ""))
    for line in report(read_logs(log_dir), kb):
        print(line)


if __name__ == "__main__":
    main(sys.argv[1:])
```

`read_logs` reads every `usage-*.jsonl` file in name order, which is time order. `report` counts consultations per month; answered and unanswered ones, as numbers and percentages; how often each node gave advice, most used first; which nodes of the knowledge base never did (stopped nodes are left out, since they give no advice by design); and the unanswered cases, grouped by their facts and sorted by how often they occurred. `prune` removes the lines older than `--keep-days` days before `--today` (by default, the real today), rewriting a month's file or deleting it when nothing in it is recent enough. Percentages are formatted with the engine's `num.fixed`, so that Java and Python round them the same way. **Listing 15.11** is the Java version.

Listing 15.11 `UsageReport.java`: the same report in Java.

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Stream;

import vcirs.Json;
import vcirs.KnowledgeBase;
import vcirs.Node;
import vcirs.Num;

/**
 * Read the usage logs of the advice service and say what they show.
 * java UsageReport [LOGDIR] [--kb FILE] [--keep-days N --today DATE]
 * It counts consultations per month, how many got an answer, which nodes
 * gave the advice (and which never did), and which unanswered cases came
 * back most often: those are the cases to teach next. With --keep-days it
 * also deletes the log lines older than N days before DATE (default:
 * today), so that the logs are not kept longer than they are needed.
 */
public class UsageReport {
    static final String[] MONTHS = {"January", "February", "March", "April",
        "May", "June", "July", "August", "September", "October", "November",
        "December"};

    static List<Path> logFiles(Path dir) throws IOException {
        try (Stream<Path> s = Files.list(dir)) {
            return s.filter(p -> p.getFileName().toString().startsWith("usage-")
                    && p.getFileName().toString().endsWith(".jsonl"))
                    .sorted().toList();
        }
    }
```

Listing 15.11 (Continued) *UsageReport.java: the same report in Java.*

```
    /** Every record of every usage-*.jsonl file, oldest file first. */
    @SuppressWarnings("unchecked")
    static List<Map<String, Object>> readLogs(Path dir) throws IOException {
        List<Map<String, Object>> records = new ArrayList<>();
        for (Path p : logFiles(dir)) {
            for (String line : Files.readAllLines(p, StandardCharsets.UTF_8)) {
                if (!line.isBlank()) {
                    records.add((Map<String, Object>) Json.parse(line));
                }
            }
        }
        return records;
    }

    /** Delete records older than keepDays before today; return how many. */
    @SuppressWarnings("unchecked")
    static int prune(Path dir, int keepDays, LocalDate today) throws IOException {
        String limit = today.minusDays(keepDays).toString();
        int removed = 0;
        for (Path p : logFiles(dir)) {
            List<String> lines = new ArrayList<>();
            for (String line : Files.readAllLines(p, StandardCharsets.UTF_8)) {
                if (!line.isBlank()) {
                    lines.add(line);
                }
            }
            List<String> keep = new ArrayList<>();
            for (String line : lines) {
                String time = (String) ((Map<String, Object>) Json.parse(line))
                        .get("time");
                if (time.compareTo(limit) >= 0) {
                    keep.add(line);
                }
            }
            removed += lines.size() - keep.size();
            if (keep.isEmpty()) {
                Files.delete(p);
            } else if (keep.size() < lines.size()) {
                Files.writeString(p, String.join("\n", keep) + "\n",
                        StandardCharsets.UTF_8);
            }
        }
        return removed;
    }
```

Listing 15.11 (Continued) *UsageReport.java: the same report in Java.*

```
    static String percent(int part, int whole) {
        return whole == 0 ? "-" : Num.fixed(100.0 * part / whole, 1) + "%";
    }

    static void count(Map<String, Integer> counts, String key) {
        counts.merge(key, 1, Integer::sum);
    }

    /** The keys, the largest count first, equal counts in name order. */
    static List<String> byCount(Map<String, Integer> counts) {
        List<String> keys = new ArrayList<>(counts.keySet());
        keys.sort((a, b) -> counts.get(a).equals(counts.get(b))
                ? a.compareTo(b) : counts.get(b) - counts.get(a));
        return keys;
    }

    @SuppressWarnings("unchecked")
    static List<String> report(List<Map<String, Object>> records,
                               KnowledgeBase kb) {
        List<String> out = new ArrayList<>();
        if (records.isEmpty()) {
            out.add("No consultations in the logs.");
            return out;
        }
        int total = records.size();
        String first = (String) records.get(0).get("time");
        String last = (String) records.get(total - 1).get("time");
        out.add("Consultations: " + total + ", from " + first.substring(0, 10)
                + " to " + last.substring(0, 10));
        Map<String, Integer> months = new LinkedHashMap<>();
        Map<String, Integer> counts = new LinkedHashMap<>();
        Map<String, Integer> missing = new LinkedHashMap<>();
        int answered = 0;
        for (Map<String, Object> r : records) {
            count(months, ((String) r.get("time")).substring(0, 7));
            String rule = (String) r.get("rule");
            if (!rule.isEmpty()) {
                answered++;
                count(counts, rule);
            } else {
                List<String> parts = new ArrayList<>();
                ((Map<String, Object>) r.get("facts")).forEach((k, v) ->
                        parts.add(k + " = " + v));
                count(missing, String.join("; ", parts));
            }
        }
```

Listing 15.11 (Continued) *UsageReport.java: the same report in Java.*

```
        List<String> perMonth = new ArrayList<>();
        months.forEach((m, n) -> perMonth.add(MONTHS[Integer.parseInt(
                m.substring(5)) - 1] + " " + m.substring(0, 4) + ": " + n));
        out.add("  " + String.join(", ", perMonth));
        out.add("Answered: " + answered + " (" + percent(answered, total)
                + "), no answer: " + (total - answered) + " ("
                + percent(total - answered, total) + ")");
        out.add("Advice given, most used first:");
        for (String rule : byCount(counts)) {
            out.add(String.format("  %4d  %-6s %s", counts.get(rule),
                    percent(counts.get(rule), total), rule));
        }
        if (kb != null) {
            List<String> idle = new ArrayList<>();
            for (Node n : kb.nodes.values()) {
                if (!counts.containsKey(n.id) && !n.stopped) {
                    idle.add(n.id);
                }
            }
            out.add("Nodes that never gave the advice: "
                    + (idle.isEmpty() ? "none" : String.join(", ", idle)));
        }
        out.add("Cases to teach next (no answer), most frequent first:");
        for (String key : byCount(missing)) {
            out.add(String.format("  %4d  %s", missing.get(key), key));
        }
        return out;
    }

    public static void main(String[] args) throws IOException {
        Map<String, String> opt = new LinkedHashMap<>();
        String logDir = "logs";
        for (int i = 0; i < args.length; i++) {
            if (args[i].startsWith("--") && i + 1 < args.length) {
                opt.put(args[i], args[++i]);
            } else {
                logDir = args[i];
            }
        }
        if (opt.containsKey("--keep-days")) {
            LocalDate today = LocalDate.parse(opt.getOrDefault("--today",
                    LocalDate.now().toString()));
            int removed = prune(Path.of(logDir),
                    Integer.parseInt(opt.get("--keep-days")), today);
            System.out.println("Removed " + removed + " log lines older than "
                    + opt.get("--keep-days") + " days before " + today + ".");
```

Listing 15.11 (Continued) *UsageReport.java: the same report in Java.*

```
        }
        KnowledgeBase kb = opt.containsKey("--kb")
                ? KnowledgeBase.load(opt.get("--kb")) : null;
        System.out.println("Usage report for " + logDir
                + (kb != null ? " and " + kb.name : ""));
        for (String line : report(readLogs(Path.of(logDir)), kb)) {
            System.out.println(line);
        }
    }
}
```

`java -jar cake-doctor.jar report logs --kb data/cake-kb.json` prints the report for the three months:

```
Usage report for logs and Cake doctor
Consultations: 120, from 2026-07-01 to 2026-09-30
  July 2026: 41, August 2026: 40, September 2026: 39
Answered: 106 (88.3%), no answer: 14 (11.7%)
Advice given, most used first:
    22  18.3%  keep the door closed#1
    14  11.7%  use room-temperature eggs#1
    13  10.8%  bake longer#1
    12  10.0%  take it out earlier#1
     9  7.5%   bake on a higher rack#1
     9  7.5%   heat the oven fully first#1
     7  5.8%   fold the flour in gently#1
     6  5.0%   take it out earlier#1G
     6  5.0%   use a lighter pan#1
     5  4.2%   beat the eggs longer#1
     3  2.5%   lower the heat and bake longer#1
Nodes that never gave the advice: lower the heat#1
Cases to teach next (no answer), most frequent first:
     8  problem = sticky top
     4  problem = uneven rise
     2  problem = cracked top; oven heat = normal
```

Figure 15.2 draws the same numbers.

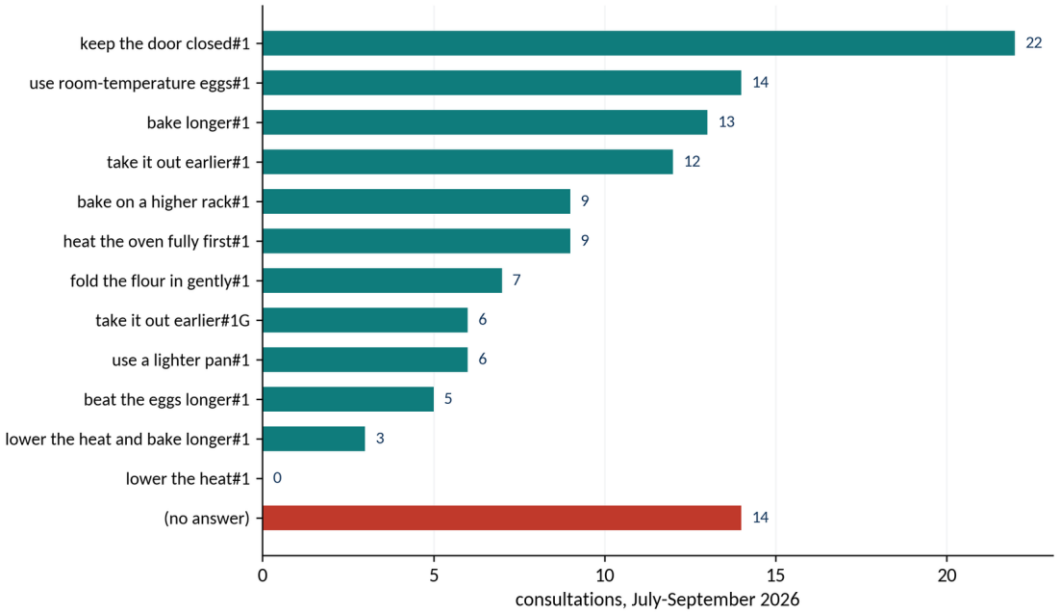

Figure 15.2 How often each node gave advice in 120 consultations, July to September 2026 (from the replayed log), with the node that never did and the consultations that got no answer.

Read the report the way Bu Wati read it, with a cup of tea on the first of October.

The doctor is used, and mostly answers. 120 consultations in three months, about ten a week, and 88.3% of them got advice. A share of answered consultations that grows over the months means the knowledge base is catching up with the shop; a share that suddenly falls means something new is happening — a new flour, a new oven — and is the moment to look closely.

Opening the oven door is the most common mistake. *keep the door closed* gave 22 of the 120 answers, 18.3%. That is not knowledge Bu Wati needs to change; it is knowledge she can *act* on: a small sign on the oven, *do not open in the first 20 minutes*, would prevent more failed cakes than any new rule.

The generated node earns its place. *take it out earlier#1G*, the general node Bu Wati accepted from a proposal in Chapter 14, answered 6 consultations: dry cakes for which nobody knew the baking time. Without it, those six would have got no answer.

One node was never used. *lower the heat#1* — a cracked top with a high oven — never gave advice. That alone proves nothing; but look at the next list.

The cases to teach next. A sticky top came up 8 times with no answer, an uneven rise 4 times, and a cracked top with a *normal* oven twice. These fourteen consultations are the most useful thing in the whole log. The sticky top is an easy decision: the next time it happens, Bu Wati finds the cause and teaches the case in the builder. The cracked tops are more interesting: they happened with a normal oven, so the existing node for a hot oven never fired, and the node's own case never happened again in three months. Together with Exercise 14.3, this is the evidence Bu Wati needed to teach the cause she suspected all along — too much baking powder.

The last step keeps the log only as long as it is needed. With `--keep-days 60`, as if the date were September 30, the report tool first removes the older lines:

```
Removed 41 log lines older than 60 days before 2026-09-30.
Usage report for logs
Consultations: 79, from 2026-08-01 to 2026-09-30
  August 2026: 40, September 2026: 39
Answered: 71 (89.9%), no answer: 8 (10.1%)
Advice given, most used first:
    14  17.7%  keep the door closed#1
    10  12.7%  take it out earlier#1
    10  12.7%  use room-temperature eggs#1
     9  11.4%  bake longer#1
     7  8.9%   fold the flour in gently#1
     5  6.3%   heat the oven fully first#1
     4  5.1%   bake on a higher rack#1
     4  5.1%   beat the eggs longer#1
     3  3.8%   lower the heat and bake longer#1
     3  3.8%   take it out earlier#1G
     2  2.5%   use a lighter pan#1
Cases to teach next (no answer), most frequent first:
     4  problem = sticky top
     3  problem = uneven rise
     1  problem = cracked top; oven heat = normal
```

July's 41 lines are gone, and so is the file `usage-2026-07.jsonl`. In the shop the scheduled report keeps 180 days; choose the number by asking how far back you really look.

15.8 Maintaining and Growing the Knowledge

A deployed expert system is never finished. **Figure 15.3** shows the cycle that keeps it alive.

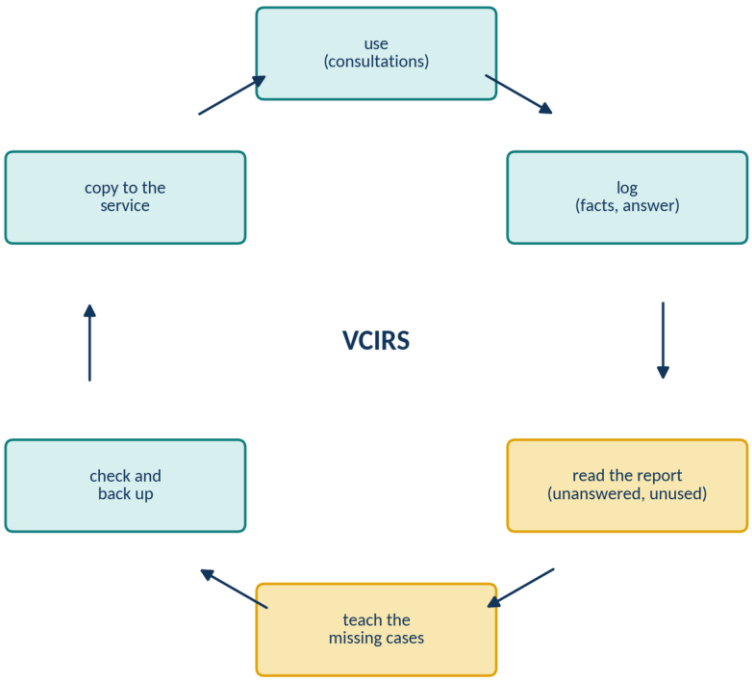

Figure 15.3 The maintenance cycle: the service is used, every consultation is logged, the monthly report shows what is unanswered and unused, the expert teaches the missing cases in the builder, the new knowledge base is checked and backed up, and the checked copy goes to the service.

A monthly routine. For a small shop, one hour a month is enough. Read the report. Teach the two or three most frequent unanswered cases the next time they happen (or at once, if you remember a real case well). Look at the nodes that never gave advice and decide whether to keep them, and look at the most used nodes and ask whether their advice could be acted on, like the oven sign. Then copy the new knowledge base to the service, run the check, and make a backup by hand as well as the automatic one.

One place to teach. Teach in one copy of the knowledge base — the builder on the owner's laptop — and copy it to the service. Never teach in two copies at once: two knowledge bases that grow apart cannot be merged automatically, because each new node was chosen against the cornerstone cases that existed *in that copy*. If two people must teach, let them teach in the same builder, one after the other; the history records who taught what.

Versions. Every export is a version. Name the copies you send to the service by date — the backup tool already does — and keep the history inside the file: it lists every change with its option and the name of the person who made it, which is what lets you answer "since when does the doctor say that?" Chapter 11 goes further, with versions and an audit trail for a team of experts.

Stop, do not delete. When advice becomes wrong — a new oven, a new recipe — stop the node in the builder rather than trying to remove it. Its cornerstone case stays, its exceptions keep working, the history records the decision, and the usage log shows the node's last use. VCIRS never deletes a node, and the reason is exactly this: knowledge that was once right carries the evidence of why the knowledge after it is right [8], [9].

Updating the program, not only the knowledge. Sometimes the program itself changes: a new version of vcirs-core, a fix in the service, a newer Java or Python on the shop computer. Treat such an update with more care than a new knowledge base, because it changes how *every* answer is computed. Keep the old `.jar` or `.pyz` beside the new one (`cake-doctor-2026-09.jar`, `cake-`

`doctor-2026-12.jar`), stop the service, start the new file, and run the check. Then consult ten cases whose answers you know — the ten most common lines of the last report are a good choice — and compare the advice with what the old version gave. If anything differs and you cannot explain why, stop the new version and start the old one again: that is a **rollback**, and it takes one minute because the old file is still there. The knowledge-base file does not change in an update, so both versions read the same knowledge.

Checking on a schedule, too. The check of Section 15.5 is useful after a deployment, and just as useful every morning. Scheduled at 7:00, before the shop opens, it answers the question "is the doctor alive today?" before anybody needs it. On Linux and macOS, a crontab line such as `0 7 * * * cd /opt/cake-doctor && java -jar cake-doctor.jar check || echo "cake doctor is down" | mail -s "cake doctor" wati@example.com` sends a message only when the check fails (the `||` runs the second command only after a failure); on Windows, a scheduled task can run the check and show a message box when its exit code is not zero. A service that fails silently at night and is discovered by a helper with a burnt cake at ten in the morning is a sure way for a deployed system to lose people's trust.

When the knowledge base grows large. A few hundred nodes are easy; a few thousand need more care. The usage rates of Chapter 14 point to the nodes on which much depends; review those first. The list of affected cornerstones (Chapter 3) becomes important when you teach: with many nodes, a new exception may change the answers of old cases that you do not remember, and the engine can list them. Long experience with Ripple Down Rules in pathology laboratories showed that the time to add a rule stays small even as knowledge bases grow to thousands of rules, because the expert only ever looks at one case and its difference list [8]; the work that grows is keeping the *domain* in view, which is what the monthly report is for.

When to start a second knowledge base. When the questions of two groups of cases have nothing in common — cakes and bread, phones and laptops — two small knowledge bases are easier to teach, check and understand than one large one. The builder and the service each handle one file; run two services on two ports.

15.9 Privacy, Ethics and Safety

15.9.1 Personal data, in practice

The cake doctor collects no personal data, and that was a design decision, not an accident. Many expert systems are not so lucky: a health self-check must know symptoms, a loan advisor must know income, a clinic triage must know a patient. For such systems, UU PDP and, from January 2027, PP 33/2026 ask for more [4], [5], [6]. In plain words:

- **Know your purpose, and collect only what serves it.** Every question of the knowledge base should be one the advice needs. If the advice never depends on a person's name, do not ask for it. If a variable is used by no node, remove the question.
- **Tell people, and ask when needed.** Before a consultation that involves personal data, say what is collected, why, and for how long; where consent is the lawful basis, ask for it clearly and record that it was given.
- **Separate identity from facts.** If you must link consultations to a person — a patient's file, for example — keep the identity in a separate place from the usage log, and use a random number to link them. The usage log itself should never need a name.
- **Keep data only as long as it is needed**, and delete it on schedule, as `--keep-days` does.

- **Protect what you keep.** Keep the service on the internal network, the log files readable only by the service's user, the backups where only the owner can reach them.
- **Be ready for a breach.** If personal data leaks — a stolen laptop with unencrypted logs — the law asks that the people affected be told within 3 × 24 hours [4], [6]. A written plan of who does what is part of deployment.
- **Respect the rights of the people in the data**: to see what is kept about them, to correct it, and to have it deleted.

Larger organizations have further duties, such as appointing a data protection officer and assessing the impact of high-risk processing [5], [6]. The regulation's details are new in September 2026, and the agency that will enforce them was not yet in place at the end of August [5]; if your expert system handles personal data at scale, read PP 33/2026 itself or ask someone who knows it well before January 16, 2027.

15.9.2 The ethics of advice by machine

An expert system gives advice, and people act on advice. Indonesia's Ministry of Communication and Informatics set out ethical values for artificial intelligence in a 2023 circular — among them inclusivity, humanity, security, accessibility, transparency, credibility and accountability, personal data protection, sustainability and respect for intellectual property [10] — and UNESCO's worldwide recommendation on the ethics of AI asks for human oversight, transparency and explainability, fairness, and proportionality [11]. A VCIRS system meets many of these by its nature, and the rest by the way it is used.

Explain every answer. Every application in this book shows *why* it gives its advice, fact by fact. Keep that visible to the people who receive the advice, not hidden in a log. An answer that can be explained can be questioned, and an answer that can be questioned can be corrected.

Keep a person responsible. The knowledge came from people who saw cases solved, and every node records who taught it. The system does not replace their judgment; it spreads it. Say so to the people who use it, and make it easy for them to reach a person when the advice does not fit.

Say "I do not know". The honest "no advice yet" of Chapter 3 and the fixed sentence of Chapter 13 are ethical features, not gaps. A system that always answers will sometimes answer wrongly with full confidence.

Know where the stakes are high. For health, money and safety, give only advice that a qualified person has checked, include clear referral rules — *see a doctor or midwife when…*, *ask a certified electrician before…* — and never present the output as a diagnosis or a financial recommendation [12]. The knowledge base can hold that referral knowledge as nodes like any other, and it should.

Watch for unfairness. A knowledge base learns from the cases its teachers saw. If they saw only one kind of customer, it may serve others badly. The usage report is a first check: many unanswered cases from one group of people is a sign that the knowledge does not yet cover them.

Do not use the log for what it was not meant for. A usage log collected to improve advice must not quietly become a tool to judge the helpers who used it.

15.10 Where to Go Next

This book began with a becak driver, a siomay seller and a parking attendant, and with a promise: that a person who knows nothing about programming can build an expert system that learns. Fifteen chapters later, you have built a dozen of them, from an electricity-bill saver with nine

rules to a builder that lets anyone make their own and a service that runs them every day. Here are some directions from here.

Go deeper into the ideas. The source of this book's engine is the Variable-Centered Intelligent Rule System, in its thesis, its conference papers and their revisions [9], [13], [14]. Its roots are in Ripple Down Rules, for which the best single book is Compton and Kang's [8], and a review of two decades of research [15] shows how widely the idea was applied. Reading them now, after building the applications, you will recognize every concept.

Extend the engine. vcirs-core is small enough to read in a weekend, and it is written twice, so every idea can be tried in the language you prefer. Some extensions the sources suggest and this book did not build: fuzzy numbers for uncertain facts (Chapter 9 built the fuzzy values; the fuzzy VCIRS paper goes further [16]), multiple conclusions in every application, and a knowledge base shared by several shops that merges knowledge by teaching each other's unanswered cases.

Combine it with learning from data. Chapter 12 turned a decision tree into a VCIRS knowledge base; Chapter 13 put a language model in front of one. Both show the same pattern: let statistical methods propose, and let checked knowledge decide. That pattern will stay useful as the statistical methods improve.

Share what you build, carefully. A knowledge base is small, readable and portable, which makes it easy to share: Bu Wati's sister imported the cake doctor and grew her own. When you share one, share its history too, so that the next person can see who taught what and when; remove from it anything personal that a cornerstone case might contain; and say plainly what it was built for and by whom. A knowledge base about cakes in Surabaya may be wrong about cakes at a higher altitude in Malang, and its new owner deserves to know where it came from before trusting it.

Build for your own community. The most important applications of this book are the ones you will build next — for your family's shop, your village's farmers' group, your mosque's or church's office, your school. Start with a few real cases, teach the rest as they come, keep backups, read the report once a month, and keep a person responsible. That is all an expert system needs to keep helping for years.

15.11 Common Mistakes, and How to Avoid Them

Running the service from a terminal window. Closing the window, or logging out, stops it. *Avoid it:* install it as a service (systemd, Task Scheduler, launchd) with a restart policy.

Teaching in the deployed copy. Two copies that both change cannot be merged. *Avoid it:* teach in one place, copy the checked result to the service.

Backing up without checking. A backup that cannot be restored is discovered only when it is needed. *Avoid it:* fingerprint every copy, check the copies regularly, and practise a restore.

One copy in one place. A fire, a theft or a failed disk takes the original and the backup together. *Avoid it:* follow the 3-2-1 rule [3].

Logging everything "in case". Names, phone numbers, IP addresses and free text turn a maintenance log into personal data with legal duties. *Avoid it:* log only what the maintenance questions need, and switch off the web server's default access log.

Keeping logs forever. Old logs are a liability, not an asset. *Avoid it:* delete on a schedule, as `--keep-days` does.

Counting tests as use. Checks after deployment inflate the statistics. *Avoid it:* mark test requests (`X-Test`) and leave them out of the log.

Exposing the service to the internet. It has no passwords and no encryption. *Avoid it:* listen on `127.0.0.1` or the shop network only; if it must be reachable from outside, put it behind a properly configured web server with HTTPS and authentication, and ask someone experienced.

Deleting advice that became wrong. Deleting loses the evidence and breaks exceptions. *Avoid it:* stop the node; the history records why.

Never reading the report. A log nobody reads is a cost with no benefit. *Avoid it:* put the monthly hour in the calendar.

15.12 Chapter Summary

- To **deploy** is to make an application run without its programmer: the cake doctor became an **advice service** with three doors, `/health`, `/questions` and `/advise` (**Listing 15.1**, **Listing 15.2**), that reads a knowledge base, reloads it when a new copy arrives, and never changes it.
- The service and its tools are packed into **one file**, a `.jar` or a `.pyz`, with four commands (**Listing 15.3**, **Listing 15.4**, **Listing 15.5**), and installed to start by itself with systemd, Task Scheduler or launchd (**Figure 15.1**).
- A short **check** after every deployment tests health, questions, one known answer and one known non-answer (**Listing 15.6**, **Listing 15.7**).
- **Backups** are taken only of readable files, fingerprinted with SHA-256, checked regularly, restored only when intact, and kept in three copies (**Listing 15.8**, **Listing 15.9**).
- The **usage log** records time, facts, node and advice — nothing about people — and the monthly **report** shows how much advice was given, which nodes are never used, and which unanswered cases to teach next (**Listing 15.10**, **Listing 15.11**, **Figure 15.2**); old lines are deleted on schedule.
- Maintenance is a **cycle** of use, logging, reading, teaching, checking and deploying (**Figure 15.3**); advice that became wrong is stopped, never deleted.
- **UU PDP** (Law 27/2022) applies to personal data; its implementing regulation **PP 33/2026**, enacted on July 16, 2026, takes effect on January 16, 2027 [4], [5]. The simplest compliance is not to collect personal data; where it is needed, collect little, keep it briefly, protect it, and be ready for a breach. Explanations, a responsible person, honest “I do not know” answers and referral rules make advice by machine ethical [10], [11].

15.13 Exercises

15.1 List the three doors of the advice service and, for each, one situation in the shop where it is used.

15.2 Bu Wati copies a new `cake-kb.json` to the service’s folder with the Windows Explorer, and the copy takes two seconds. A helper asks for advice during those two seconds. What happens, and why is the answer still correct?

15.3 From the report of Section 15.7, compute by hand the percentage of consultations answered by *bake longer#1* and by *take it out earlier#1G*, and check them against the report.

15.4 The helpers ask whether the log could also record which helper asked, "to see who needs more training". Discuss this request in the light of Section 15.9. If Bu Wati agrees, how would you do it with the least personal data?

15.5 Write the lines of a scheduled task that, every Sunday at 22:00, copies the newest file of `backups` to a USB stick mounted at `/media/usb` (Linux) and then runs `backup check` on the stick. What does the check protect against that the evening backup does not?

15.6 Write a macOS launch agent `.plist` that starts the service from `~/cake-doctor` at login and restarts it if it stops.

15.7 The report says *lower the heat#1* never gave advice. Give two different explanations, and say what Bu Wati should do in each case.

15.8 A clinic wants to deploy the health self-check of Chapter 5 as a service like this one. Name four changes to the usage log and the deployment that the clinic needs and the cake shop does not.

15.14 Solutions to the Exercises

Solution 15.1. `GET /health`: after installing or updating the service, or when a helper thinks it is not answering, the check (or a browser) asks it. `GET /questions`: the counter tablet's screen shows the questions in the most useful order, with their usual answers as buttons. `POST /advise`: the tablet sends the answers chosen for a failed cake and shows the advice and its explanation; the chat front end of Chapter 13 can use the same door.

Solution 15.2. During the copy the file's modification time changes, so `fresh_kb` tries to load it. If the file is still half written, loading fails — the JSON is incomplete — and the service keeps the old knowledge base and writes "could not reload … keeping the old knowledge base" to its error output. The helper gets the advice of the old knowledge base, which was correct until a moment ago. Because the service remembers the new modification time, it does not retry at once; the next time the file changes (when the copy finishes and its time is set again), it loads the complete file. A safer habit is to copy the file under another name, such as `cake-kb.new`, and then rename it to `cake-kb.json`: a rename replaces the file in one step.

Solution 15.3. *bake longer#1* gave advice 13 times out of 120: 13/120 = 0.10833…, that is 10.8%, as in the report. *take it out earlier#1G* gave advice 6 times: 6/120 = 0.05 = 5.0%, as in the report.

Solution 15.4. Recording who asked turns the usage log into personal data about employees and changes its purpose from improving advice to judging people, which Section 15.9.2 warns against. Under UU PDP the shop would need a clear purpose, would have to tell the helpers, and would have to protect and eventually delete the data [4]. A better answer to the real need — "who needs more training?" — is to look at the unanswered and the frequent cases together with the helpers, without names. If Bu Wati still wants it, record a short code per helper (not a name), tell the helpers what is recorded and why, keep the table linking codes to names on her own laptop only, and delete the codes from the log after the training.

Solution 15.5. The simplest way is to let the backup tool itself save on the stick, so that the stick has its own `index.txt` with fingerprints. Write a script `weekly-usb.sh` beside the other two:

```
#!/bin/sh
cd /opt/cake-doctor || exit 1
java -jar cake-doctor.jar backup save data/cake-kb.json \
    /media/usb/backups --keep 52 &&
java -jar cake-doctor.jar backup check /media/usb/backups
```

and add `0 22 * * 0 /opt/cake-doctor/weekly-usb.sh` with `crontab -e`: minute 0, hour 22, any day of the month, any month, day of the week 0 (Sunday). `--keep 52` keeps a year of weekly copies, and `&&` runs the check only if the save succeeded. The weekly copy on a separate device protects against the failure of the service computer's own disk, which would take the evening backups with it; the check on the stick protects against the stick itself silently damaging files, which is common with cheap USB sticks. Once a month, take the stick home: that is the "one copy in another place" of the 3-2-1 rule.

Solution 15.6. Save as `~/Library/LaunchAgents/com.example.cakedoctor.plist`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>com.example.cakedoctor</string>
  <key>WorkingDirectory</key>
  <string>/Users/wati/cake-doctor</string>
  <key>ProgramArguments</key>
  <array>
    <string>/usr/bin/java</string>
    <string>-jar</string>
    <string>cake-doctor.jar</string>
    <string>serve</string>
    <string>--kb</string>
    <string>data/cake-kb.json</string>
    <string>--logs</string>
    <string>logs</string>
  </array>
  <key>RunAtLoad</key>
  <true/>
  <key>KeepAlive</key>
  <true/>
</dict>
</plist>
```

and activate it with `launchctl load ~/Library/LaunchAgents/com.example.cakedoctor.plist`. Replace `/Users/wati` with your home folder, and `/usr/bin/java` with the output of `which java` if Java is elsewhere.

Solution 15.7. (1) The problem is rare: a cracked top with a *hot* oven simply did not happen in three months, because the ovens were set correctly. Then the node is fine; keep it and look again in a year. (2) The node's knowledge is incomplete or wrong: cracked tops do happen (twice, with a normal oven), but for another reason. Then Bu Wati should teach the real cause when the next cracked cake happens, and, if she comes to believe the high-oven advice is wrong too, stop the node. The report cannot tell the two apart; the person who knows cakes can.

Solution 15.8. (1) The facts are health data, which UU PDP treats as specific (sensitive) personal data [4]; the clinic needs a lawful basis — usually explicit consent — and must tell patients what is kept and why. (2) The log must not be linkable to a patient in the log itself; if a link is needed, keep it separately, protected, with a random identifier. (3) Access to the service and the logs must be limited to the clinic's staff, with the service on the internal network and the files readable only by the service's user; backups must be encrypted. (4) The retention period should be set by the clinic's policy and kept by the scheduled pruning, and a breach plan with the 3 × 24-hour

notification must be written down. In addition, the knowledge base must carry referral rules and never present its output as a diagnosis [12].

References

[1] Oracle, "The jar command," *Java Development Kit Version 25 Tool Specifications*, 2025. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/specs/man/jar.html

[2] Python Software Foundation, "zipapp — Manage executable Python zip archives," *The Python Standard Library*, 3.14. [Online]. Available: https://docs.python.org/3/library/zipapp.html

[3] Cybersecurity and Infrastructure Security Agency (US-CERT), "Data backup options," 2012. [Online]. Available: https://www.cisa.gov/sites/default/files/publications/data_backup_options.pdf

[4] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*, Lembaran Negara RI Tahun 2022 Nomor 196, Oct. 17, 2022. [Online]. Available: https://peraturan.bpk.go.id/Details/229798/uu-no-27-tahun-2022

[5] C. Aw, T. Darmawan, A. Rabbani, F. Seow, and C. O'Leary, "Indonesia's Personal Data Protection Law implementing regulation arrives quietly: a first look at GR 33/2026," Hogan Lovells, Aug. 31, 2026. [Online]. Available: https://www.hlc.com/en/publications/indonesias-personal-data-protection-law-implementing-regulation-arrives-quietly

[6] Robere & Associates, "PP 33 2026: panduan pelindungan data pribadi," 2026. [Online]. Available: https://robere.co.id/id/pp-33-2026-pelindungan-data-pribadi/

[7] systemd project, "systemd.service — Service unit configuration," *systemd manual pages*. [Online]. Available: https://www.freedesktop.org/software/systemd/man/latest/systemd.service.html

[8] P. Compton and B. H. Kang, *Ripple-Down Rules: The Alternative to Machine Learning*. Boca Raton, FL, USA: CRC Press, 2021.

[9] M. M. I. Subakti, "A Variable-Centered Intelligent Rule System," M.Sc.Eng. thesis, Dept. of Computer Science and Information Engineering, Taipei, Taiwan, Jun. 2005.

[10] Kementerian Komunikasi dan Informatika Republik Indonesia, *Surat Edaran Menteri Komunikasi dan Informatika Nomor 9 Tahun 2023 tentang Etika Kecerdasan Artifisial*, Dec. 19, 2023.

[11] UNESCO, *Recommendation on the Ethics of Artificial Intelligence*. Paris, France: UNESCO, 2021.

[12] I. Subakti and R. Hidayatullah, "Aplikasi sistem pakar untuk diagnosis awal gangguan kesehatan secara mandiri menggunakan Variable-Centered Intelligent Rule System," vol. 6, no. 1, pp. 11–16, Jan. 2007.

[13] I. Subakti, "A variable-centered intelligent rule system," in *Proc. 1st Annual Int. Conf. Information and Communication Technology Seminar (ICTS 2005)*, vol. 1, no. 1, Surabaya, Indonesia, Aug. 2005, pp. 167–174.

[14] I. Subakti, "Some revisions in VCIRS and cases reconstructing perspectives," in *Proc. Information and Communication Technology Seminar (ICTS 2006)*, Surabaya, Indonesia, 2006.

[15] D. Richards, "Two decades of ripple down rules research," *The Knowledge Engineering Review*, vol. 24, no. 2, pp. 159–184, 2009.

[16] I. Subakti and O. Wijayanto, "Penerapan konsep fuzzy dalam Variable-Centered Intelligent Rule System (studi kasus: pemilihan jurusan di Chinese University of Hongkong)."