HUMAN-COMPUTER INTERACTION

DESIGNING USABLE, ACCESSIBLE AND AI-READY INTERFACES, WITH JAVA AND PYTHON



Irfan Subakti

Profitina
profitina.com

Human-Computer Interaction

Designing Usable, Accessible and AI-Ready Interfaces, with Java and Python

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 ..... **xxi**
Who this book is for xxi
One idea, two languages xxi
What is in it xxii
How to read it xxiii
About the author xxiii
Acknowledgments xxiii

CHAPTER 1 Introduction to HCI: Why Interfaces Matter **1**
1.1 Bu Ratna's Warung Goes Online 1
1.2 What Human-Computer Interaction Is 2
1.2.1 A definition, and three words 2
1.2.2 The two gulfs 3
1.2.3 What HCI is not 3
1.3 From Punched Cards to Agents: a Short History and the Present 4
1.4 Why Interfaces Matter in Indonesia in 2026 5
1.4.1 Who is online, and on what 5
1.4.2 The law now asks for good interfaces 5
1.4.3 How this book works 6
1.5 Your Workbench: What We Install and Why 6
1.6 Installing on Windows 10 and 11 7
1.6.1 The JDK 7
1.6.2 Python 8
1.6.3 VS Code 8
1.7 Installing on macOS 8
1.7.1 The JDK 8
1.7.2 Python 9
1.7.3 VS Code 9
1.8 Installing on Linux 9
1.9 VS Code: Two Extensions and a Tour 10
1.10 Your First Programs 11
1.10.1 Hello, in Java 11
1.10.2 Hello, in Python 12
1.10.3 Checking the whole workbench 12
1.11 A First HCI Program: Counting the Cost of Clicks 15
1.11.1 The Keystroke-Level Model 15
1.11.2 The two pages, by hand 15
1.11.3 The program 16
1.11.4 Running it 20
1.11.5 Testing it 21
1.12 HCI in Practice: Profitina 22
1.13 Common Mistakes, and How to Avoid Them 23
1.14 Chapter Summary 23
1.15 Exercises 24
1.16 Solutions to the Exercises 24
References 25

CHAPTER 2 Human Factors: Perception, Cognition, Memory, Fitts's and Hick's Laws **29**
2.1 Bu Siti at the Puskesmas Kiosk 29
2.2 The Idea in Plain Words: a Person as an Information Processor 30

2.3 Perception: What Reaches the Mind 31
2.3.1 Vision 31
2.3.2 Grouping: the Gestalt principles 31
2.3.3 What stands out: preattentive features 32
2.3.4 Hearing and touch 32
2.4 Attention: One Thing at a Time 32
2.5 Memory: What Stays, and for How Long 33
2.5.1 Working memory 33
2.5.2 Recognition rather than recall 33
2.5.3 Mental models 33
2.6 Fitts's Law: How Long It Takes to Reach a Target 33
2.6.1 The law 33
2.6.2 A worked example 34
2.6.3 What the law means for design 35
2.7 Hick's Law: How Long It Takes to Decide 35
2.7.1 The law 35
2.7.2 When the law applies, and when it does not 36
2.8 Building the Calculator Step by Step 36
2.8.1 The data files 37
2.8.2 Fitting a line with least squares 38
2.8.3 The expected time of a layout 38
2.8.4 The program 39
2.9 Running It 44
2.10 Testing It 45
2.11 Using It for Real 47
2.12 HCI in Practice: Profitina 47
2.13 Common Mistakes, and How to Avoid Them 48
2.14 Chapter Summary 48
2.15 Exercises 48
2.16 Solutions to the Exercises 49
References 50

CHAPTER 3 Design Principles and Heuristics 52
3.1 Mbak Dewi's Laundry Leaves WhatsApp 52
3.2 The Idea in Plain Words: Principles, Guidelines and Heuristics 53
3.3 Norman's Principles 54
3.3.1 Affordances and signifiers 54
3.3.2 Mapping 54
3.3.3 Constraints and forcing functions 55
3.3.4 Feedback and the conceptual model 55
3.4 Three Families of Rules 56
3.5 Principles Meet the Design Languages of 2025–2026 57
3.6 Heuristic Evaluation Step by Step 57
3.6.1 The procedure 57
3.6.2 How many evaluators? 58
3.6.3 Heuristic evaluation and its relatives 59
3.7 Building the Report Generator 59
3.7.1 The data 60
3.7.2 The program 60
3.8 Running It 65
3.9 Testing It 66
3.10 Using It for Real 68
3.11 HCI in Practice: Profitina 68
3.12 Common Mistakes, and How to Avoid Them 68

3.13 Chapter Summary 69
3.14 Exercises 69
3.15 Solutions to the Exercises 70
References 71

CHAPTER 4 User-Centered and Design-Thinking Processes 73
4.1 An Ojek Base Wants an App 73
4.2 The Idea in Plain Words 74
4.3 The Human-Centred Design Process of ISO 9241-210 74
4.4 The Double Diamond and Design Thinking 75
4.4.1 The Double Diamond 75
4.4.2 Design thinking 76
4.4.3 Lean UX and agile teams 76
4.5 The Working Tools, with Templates 77
4.5.1 The affinity diagram 77
4.5.2 Personas 77
4.5.3 The journey map 78
4.5.4 Empathy maps and jobs to be done 78
4.5.5 Problem statements and "how might we" questions 78
4.5.6 Ideation and the impact-effort matrix 79
4.5.7 Measurable user requirements 79
4.6 AI in the Process, 2026 79
4.7 Building the Synthesis Tool 80
4.7.1 The data 80
4.7.2 The program 81
4.8 Running It 87
4.9 Testing It 89
4.10 Using It for Real 90
4.11 HCI in Practice: Profitina 91
4.12 Common Mistakes, and How to Avoid Them 91
4.13 Chapter Summary 91
4.14 Exercises 92
4.15 Solutions to the Exercises 92
References 94

CHAPTER 5 User Research Methods 95
5.1 Bu Rina's Missing Customers 95
5.2 The Idea in Plain Words 96
5.2.1 Finding out before building 96
5.2.2 What people say and what people do 96
5.2.3 Four questions that choose the method 97
5.3 Planning a Study 98
5.3.1 The research plan on one page 98
5.3.2 Who to talk to, and how many 98
5.3.3 Consent and personal data 99
5.4 Interviews and Contextual Inquiry 100
5.4.1 The semi-structured interview 100
5.4.2 Asking without leading 100
5.4.3 Contextual inquiry: the master and the apprentice 101
5.4.4 Field notes and diary studies 101
5.5 Surveys and Questionnaires 101
5.5.1 When a survey is the right tool 101
5.5.2 Writing good items 101
5.5.3 How precise is a survey? 102
5.6 Building the Survey Summarizer 103

5.6.1 The data 103
5.6.2 The program in Python 103
5.6.3 The program in Java 105
5.6.4 Running it 106
5.6.5 Reading the results 107
5.7 Making Sense of Qualitative Data 107
5.7.1 From notes to codes 107
5.7.2 The affinity diagram 108
5.7.3 When have we interviewed enough? 108
5.7.4 The saturation program 108
5.8 Testing the Programs 114
5.9 From Findings to Insights 115
5.9.1 Triangulating the evidence 115
5.9.2 Writing insight statements 115
5.9.3 What Bu Rina did 116
5.10 AI in User Research, as of 2026 116
5.11 Using It for Real 117
5.12 HCI in Practice: Profitina 118
5.13 Common Mistakes, and How to Avoid Them 118
5.14 Chapter Summary 119
5.15 Exercises 119
5.16 Solutions to the Exercises 120
References 121

CHAPTER 6 Prototyping: From Paper to Interactive (and AI-Assisted) Prototypes 123
6.1 The Queue at Dawn 123
6.2 The Idea in Plain Words 124
6.2.1 A prototype is a question 124
6.2.2 Sketches and prototypes; the right thing and the thing right 124
6.2.3 Horizontal and vertical prototypes 125
6.3 Paper Prototyping 125
6.3.1 Materials and roles 125
6.3.2 A session at the puskesmas 126
6.3.3 Wizard of Oz 126
6.4 Checking a Flow Before Building It 126
6.4.1 A flow as a text file 126
6.4.2 The checker 127
6.4.3 Running it on both versions 131
6.5 Wireframes and Clickable Mockups 133
6.5.1 Wireframes 133
6.5.2 Clickable mockups and the tools of 2026 133
6.6 Building a Coded Prototype 133
6.6.1 What it must do 133
6.6.2 The screens 134
6.6.3 The server in Python 135
6.6.4 The same prototype in Java 136
6.6.5 Running it and taking screenshots 141
6.7 Testing the Prototype Code 143
6.8 AI-Assisted Prototyping in 2026 144
6.8.1 What the tools do 144
6.8.2 Using AI without skipping the questions 145
6.9 Prototyping the Laundry Messages 145
6.10 Using It for Real 146
6.11 HCI in Practice: Profitina 146
6.12 Common Mistakes, and How to Avoid Them 147

6.13 Chapter Summary 147
6.14 Exercises 148
6.15 Solutions to the Exercises 148
References 149

CHAPTER 7 Usability Evaluation and Testing 151
7.1 Pre-Orders at Warung Bu Tini 151
7.2 The Idea in Plain Words 151
7.2.1 What “usable” means 151
7.2.2 Two families of evaluation 152
7.3 Heuristic Evaluation 152
7.3.1 The method 152
7.3.2 Severity ratings 153
7.4 Usability Testing with Real Users 153
7.4.1 Planning the test 153
7.4.2 Thinking aloud 154
7.4.3 What happened 154
7.5 Measuring Performance 154
7.5.1 Task success and its confidence interval 154
7.5.2 Time on task and the geometric mean 155
7.5.3 Errors 155
7.5.4 The results 155
7.6 Measuring Satisfaction: SUS and UMUX-Lite 156
7.6.1 The System Usability Scale 156
7.6.2 UMUX-Lite 156
7.6.3 The results 157
7.7 Building the Metrics Program 158
7.8 How Many Participants? 164
7.8.1 The problem-discovery model 164
7.8.2 Estimating p from the study 164
7.9 Testing the Programs 165
7.10 A/B Testing 167
7.10.1 From eight people to hundreds 167
7.10.2 The two-proportion z-test 168
7.10.3 How many visitors are needed? 168
7.10.4 The program 168
7.11 Evaluation in 2026: Remote, Unmoderated and AI-Assisted 172
7.12 Using It for Real 172
7.13 HCI in Practice: Profitina 173
7.14 Common Mistakes, and How to Avoid Them 173
7.15 Chapter Summary 174
7.16 Exercises 174
7.17 Solutions to the Exercises 175
References 176

CHAPTER 8 Visual Design, Design Systems and Information Architecture 178
8.1 A Rural Bank’s App 178
8.2 The Idea in Plain Words 178
8.3 Seeing Groups: Gestalt and Visual Hierarchy 179
8.3.1 The Gestalt principles 179
8.3.2 Visual hierarchy 180
8.4 Typography and Layout 180
8.4.1 Type that can be read 180
8.4.2 A modular type scale 180
8.4.3 Space and grids 180

8.5 Color and Contrast ... 181
8.5.1 Why contrast is measured, not judged ... 181
8.5.2 Relative luminance and the contrast ratio ... 181
8.6 Building the Contrast Checker ... 182
8.6.1 Colors as design tokens ... 182
8.6.2 The program in Python ... 182
8.6.3 The program in Java ... 184
8.6.4 Running it ... 186
8.7 Design Systems and Tokens ... 188
8.7.1 From decisions to a system ... 188
8.7.2 The design languages of 2026 ... 188
8.7.3 What a small team should do ... 189
8.8 Information Architecture ... 189
8.8.1 Organizing, labeling, navigating, searching ... 189
8.8.2 Card sorting and tree testing ... 189
8.9 Analyzing a Card Sort ... 190
8.9.1 Similarity between cards ... 190
8.9.2 Average-linkage clustering ... 190
8.9.3 The program ... 191
8.9.4 Reading the result ... 196
8.10 Testing the Programs ... 197
8.11 Using It for Real ... 199
8.12 HCI in Practice: Profitina ... 200
8.13 Common Mistakes, and How to Avoid Them ... 200
8.14 Chapter Summary ... 201
8.15 Exercises ... 201
8.16 Solutions to the Exercises ... 202
References ... 203

CHAPTER 9 Interaction Styles: GUI, Touch, Voice, Gesture ... 204
9.1 Bu Tini's Counter ... 204
9.2 The Idea in Plain Words ... 205
9.2.1 What an interaction style is ... 205
9.2.2 A map of the styles ... 205
9.2.3 A worked example: one glass of tea, five ways ... 206
9.3 The Theory with Evidence ... 206
9.3.1 Two gulfs, and why direct manipulation narrows them ... 206
9.3.2 Events, state and the speed of feedback ... 206
9.3.3 Touch: a finger is not a mouse pointer ... 207
9.3.4 Gestures: powerful, and invisible ... 208
9.3.5 Voice: natural, and ambiguous ... 208
9.3.6 Today's design languages, and interaction in space ... 209
9.4 A Hands-On Method: Choosing and Checking a Style ... 210
9.5 Building It Step by Step ... 210
9.5.1 The menu and the order core ... 211
9.5.2 The command line and the typed voice line ... 218
9.5.3 The touch kiosk ... 221
9.5.4 Measuring every target in a real browser ... 227
9.6 Running It ... 228
9.7 Testing It ... 229
9.8 Making It Better ... 231
9.8.1 A target that everyone can hit ... 231
9.8.2 Asking at the right moment ... 232
9.9 Using It for Real ... 232
9.10 HCI in Practice: Profitina ... 232

9.11 Common Mistakes, and How to Avoid Them ... 233
9.12 Chapter Summary ... 233
9.13 Exercises ... 234
9.14 Solutions to the Exercises ... 234
References ... 235

CHAPTER 10 Mobile and Responsive UX ... 237
10.1 Mas Yoga's Phone ... 237
10.2 The Idea in Plain Words ... 237
10.3 The Theory with Evidence ... 238
10.3.1 Indonesia's mobile reality, read carefully ... 238
10.3.2 Pixels, the viewport, and why a desktop page shrinks ... 238
10.3.3 Fluid layouts and content-chosen breakpoints ... 239
10.3.4 Hands and thumbs ... 239
10.3.5 Forms on phones ... 240
10.3.6 Speed: vitals, round trips and a budget ... 240
10.3.7 The WCAG criteria that bite hardest on phones ... 241
10.4 A Hands-On Method: the Mobile-First Checklist ... 241
10.5 Building It Step by Step ... 241
10.5.1 Two templates, one set of words ... 242
10.5.2 Filling and serving the pages ... 244
10.5.3 A browser that pretends to be a phone ... 247
10.5.4 The load-time estimate ... 248
10.6 Running It ... 250
10.7 Testing It ... 252
10.8 Making It Better ... 253
10.9 Using It for Real ... 253
10.10 HCI in Practice: Profitina ... 254
10.11 Common Mistakes, and How to Avoid Them ... 254
10.12 Chapter Summary ... 255
10.13 Exercises ... 255
10.14 Solutions to the Exercises ... 256
References ... 256

CHAPTER 11 Accessibility and Inclusive Design (WCAG 2.2) ... 258
11.1 A Morning at Puskesmas Sukamaju ... 258
11.2 The Idea in Plain Words ... 259
11.3 The Theory with Evidence ... 259
11.3.1 Who is affected ... 259
11.3.2 What the law requires ... 259
11.3.3 WCAG 2.2 in one page ... 260
11.3.4 Contrast, computed ... 261
11.3.5 How assistive technologies read a page ... 261
11.3.6 What the web looks like in 2026 ... 262
11.4 A Hands-On Method: Four Layers of Testing ... 262
11.5 Building It Step by Step ... 263
11.5.1 The first kiosk page ... 263
11.5.2 The contrast checker ... 264
11.5.3 The audit ... 267
11.5.4 The repaired page ... 272
11.6 Running It ... 273
11.7 Testing It ... 275
11.8 Making It Better ... 277
11.9 Using It for Real ... 277
11.10 HCI in Practice: Profitina ... 277

11.11 Common Mistakes, and How to Avoid Them 278
11.12 Chapter Summary 278
11.13 Exercises 279
11.14 Solutions to the Exercises 279
References 280

CHAPTER 12 Emotional Design, Trust and Ethical UX 282
12.1 Bu Sri's Membership 282
12.2 The Idea in Plain Words 283
12.3 The Theory with Evidence 283
12.3.1 Three levels of emotional design 283
12.3.2 How trust is built and lost 284
12.3.3 A catalog of dark patterns 284
12.3.4 What the law says 285
12.3.5 Measuring feelings 286
12.3.6 Persuasion that respects the user 286
12.4 A Hands-On Method: the Fair-Flow Review 287
12.5 Building It Step by Step 287
12.5.1 Describing a flow 287
12.5.2 The linter 288
12.5.3 The second flow 290
12.5.4 The consent ledger 291
12.6 Running It 293
12.7 Testing It 294
12.8 Making It Better 295
12.9 Using It for Real 296
12.10 HCI in Practice: Profitina 296
12.11 Common Mistakes, and How to Avoid Them 297
12.12 Chapter Summary 297
12.13 Exercises 298
12.14 Solutions to the Exercises 298
References 299

CHAPTER 13 Designing AI Interfaces: Chat, Copilots and Agents 301
13.1 Bu Wati's Midnight Messages 301
13.2 The Idea in Plain Words 302
13.2.1 Chat, copilot and agent 302
13.2.2 Four questions every AI interface must answer 302
13.2.3 A worked example by hand 303
13.3 The Theory, with Evidence 303
13.3.1 Guidelines for human-AI interaction 303
13.3.2 Levels of automation and the autonomy ladder 304
13.3.3 Trust, reliance and calibration 305
13.3.4 Agents in 2025 and 2026 306
13.3.5 The rules in 2026: disclosure and consent 306
13.4 A Method: The AI Interaction Canvas 307
13.5 Building the Order Assistant 307
13.5.1 The program in Python 308
13.5.2 The program in Java 310
13.5.3 Running it 314
13.6 Measuring Reliance and Calibration 315
13.7 Testing It 319
13.8 Making It Better 321
13.9 Using It for Real 321
13.10 HCI in Practice: Profitina 322

13.11 Common Mistakes, and How to Avoid Them 322
13.12 Chapter Summary 323
13.13 Exercises 323
13.14 Solutions to the Exercises 324
References 325

CHAPTER 14 Data-Informed UX: Analytics, Experiments and Metrics 326
14.1 Bu Tini's Quiet Checkout 326
14.2 The Idea in Plain Words 326
14.2.1 What people say and what people do 326
14.2.2 Events, funnels and metrics 327
14.2.3 A worked example by hand 327
14.3 The Theory, with Evidence 327
14.3.1 Choosing metrics: the HEART framework 327
14.3.2 Where the users are: Indonesia in 2026 328
14.3.3 Controlled experiments 328
14.3.4 Trustworthy experiments: the traps 329
14.3.5 Data with consent 330
14.3.6 Metrics for AI features 330
14.4 A Method: The Experiment Plan 331
14.5 Building the Funnel Analysis 331
14.5.1 The event log 331
14.5.2 The funnel in Python and Java 332
14.5.3 Running it 334
14.6 Building the A/B Check 335
14.6.1 The program in Python 335
14.6.2 The program in Java 337
14.6.3 Running it on the healthy log 339
14.6.4 Running it on the broken log 339
14.7 Testing It 340
14.8 Making It Better 341
14.9 Using It for Real 342
14.10 HCI in Practice: Profitina 342
14.11 Common Mistakes, and How to Avoid Them 343
14.12 Chapter Summary 343
14.13 Exercises 344
14.14 Solutions to the Exercises 344
References 345

CHAPTER 15 Emerging Interfaces: Spatial, Wearable and Ambient Computing 346
15.1 Mbah Sumi and the Queue Screen 346
15.2 The Idea in Plain Words 347
15.2.1 Beyond the rectangle 347
15.2.2 Size is an angle 347
15.2.3 A worked example by hand 348
15.3 The Theory, with Evidence 348
15.3.1 The devices of 2025 and 2026 348
15.3.2 Input in space: looking is not choosing 349
15.3.3 Wearables: glance, tap, go 349
15.3.4 Ambient and calm technology 350
15.3.5 Comfort, safety, privacy and accessibility 350
15.3.6 Spatial ideas on flat screens 351
15.4 A Method: The Context-of-Use Checklist 351
15.5 Building the Target-Size Calculator 352
15.6 Choosing a Dwell Time 354

15.6.1 The simulated gaze log 354
15.6.2 The analysis in Python and Java 354
15.6.3 Reading the result 356
15.7 Testing It 357
15.8 Making It Better 359
15.9 Using It for Real 359
15.10 HCI in Practice: Profitina 360
15.11 Common Mistakes, and How to Avoid Them 360
15.12 Chapter Summary 360
15.13 Exercises 361
15.14 Solutions to the Exercises 361
References 362

CHAPTER 16 Capstone: Designing and Evaluating a Complete Interactive System 364

16.1 A Kiosk for Puskesmas Sehat Selalu 364
16.2 The Idea in Plain Words 364
16.2.1 A complete interactive system 364
16.2.2 Every chapter as a decision 365
16.3 Discover: Research at the Registration Desk 366
16.4 Define: Requirements and Constraints 367
16.4.1 Requirements 367
16.4.2 Human-factors budget 367
16.4.3 Accessibility, law and ethics as constraints 368
16.5 Develop: From Paper to a Working Kiosk 368
16.5.1 The paper prototype 368
16.5.2 The design decisions 369
16.6 Building the Kiosk 369
16.6.1 The kiosk in Python 369
16.6.2 The kiosk in Java 372
16.6.3 Running it 376
16.7 Evaluating the Kiosk 378
16.7.1 The pilot test 378
16.7.2 The measures 378
16.7.3 The evaluation program 379
16.7.4 The results 384
16.8 Iterate: From Findings to Changes 385
16.9 Testing the Code 386
16.10 Presenting Design Decisions 387
16.11 Using It for Real 387
16.12 HCI in Practice: Profitina 388
16.13 Common Mistakes, and How to Avoid Them 388
16.14 Chapter Summary 388
16.15 Exercises 389
16.16 Solutions to the Exercises 389
References 390

List of Figures

Figure 1.1 Norman's two gulfs: execution (how do I do it?) and evaluation (did it work?), and the design qualities that bridge each one. 3
Figure 1.2 How a Java program and a Python program run. Java is compiled once by `javac` into bytecode, which the Java Virtual Machine runs; Python is read and run by the interpreter every time. 7
Figure 1.3 The parts of Visual Studio Code used in this book: (1) Activity Bar, (2) Explorer, (3) editor, (4) Run button, (5) integrated terminal, (6) Status Bar. 10
Figure 1.4 The predicted time of one order on the two pages, split by operator: pointing (P) dominates design A. 16
Figure 2.1 The Model Human Processor: perceptual, cognitive and motor processors with typical cycle times, the brief sensory stores and the two memories. 30
Figure 2.2 Four Gestalt principles: proximity (two groups by distance), similarity (columns by color), common region (pairs by enclosure) and continuity (two smooth lines rather than a zigzag). 31
Figure 2.3 The old and the new first screen of the queue kiosk, to scale, with each service's share of patients and the resting point of the hand. 37
Figure 2.4 The practice logs with the fitted lines: Fitts's law (left) and Hick's law (right). 45
Figure 3.1 The order page of Profitina Laundry before and after a heuristic evaluation, rendered from our own teaching pages; the numbered markers point at problems P01, P02, P03, P07 and P14 of **Listing 3.1**. 53
Figure 3.2 Mapping on a stove: knobs in a row give an arbitrary mapping, knobs in the same square as the burners give a natural one. 55
Figure 3.3 The share of problems found by i evaluators, from Equation (3.1), for $\lambda = 0.464$ (this evaluation) and $\lambda = 0.31$ (Nielsen and Landauer's average). 59
Figure 4.1 The human-centred design activities of ISO 9241-210: plan, then cycle through context, requirements, design and evaluation until the solution meets the user requirements. 75
Figure 4.2 The Double Diamond: discover and define in the problem space, develop and deliver in the solution space; each diamond first widens, then narrows. 76
Figure 4.3 The emotional line of the journey map: mean pain per stage from the 37 notes (the axis is reversed, so the curve dips where it hurts; the shaded band marks severe pain). 88
Figure 4.4 The ten ideas in the impact-effort matrix, numbered in the order of **Listing 4.3**; the size of each circle shows its votes. 88
Figure 5.1 The landscape of user research methods: behavioral methods (top) watch what people do, attitudinal methods (bottom) record what people say; qualitative methods (left) explain why, quantitative methods (right) count how many. 97
Figure 5.2 The six survey items as a diverging stacked bar chart (60 respondents); numbers are percentages of respondents, and item Q2 leans furthest to the disagreeing side. 107
Figure 5.3 New codes per interview (bars) and distinct codes found so far (line) in the laundry study; the rule of Equation (5.4) declares saturation at interview 7. 113
Figure 6.1 The fidelity spectrum: from sketches that ask whether this is the right thing to coded prototypes that ask whether the thing is right; AI-assisted tools reach high fidelity quickly, which makes it easy to skip the left-hand questions. 125
Figure 6.2 The kiosk flow before and after the paper test: version 1 has an unreachable screen and a dead end and needs four or five taps to a ticket; version 2 needs three, and every screen has a way back. 132
Figure 6.3 Three screens of the running kiosk prototype, captured by `shoot.py`: the welcome screen, the service choice and the ticket with the number of people ahead and the estimated wait. 142
Figure 7.1 SUS scores of the eight participants against the curved grading bands; the mean (70.3) is just above the average product (68), but the two older participants score in the F band. 157

Figure 7.2 The nine problems of the warung test and the participants who met each; larger colored dots mark a sighting, colored by severity, and P3 and P6 meet the most problems. 165
Figure 7.3 Expected share of problems found as participants are added, from $1 - (1 - p)n$: with p around 0.3, five participants find about 85%; with rarer problems ($p = 0.15$), five find only about 56%. 165
Figure 8.1 Four Gestalt principles used in interface design: proximity, similarity, common region and closure. 179
Figure 8.2 The ten color pairs of the bank app as swatches, with their use, contrast ratio and WCAG 2.2 AA result. 187
Figure 8.3 Design tokens in a design system: primitive tokens hold values, semantic tokens name purposes, components use semantic tokens, and build tools turn one token file into code for each platform. 188
Figure 8.4 Dendrogram of the open card sort (20 participants, 15 cards, average linkage); cutting at a similarity of 0.5 gives four groups, and the loan installment joins the bills. 196
Figure 9.1 One order core with three front ends: the command style, the typed voice style with its confirmation rule, and the direct-manipulation kiosk all use the same menu, cart and total. 210
Figure 9.2 The final kiosk at a width of 360 CSS pixels after tapping “Iced tea” twice and “Fried rice” once: two columns of 64-pixel dish buttons, a 48-pixel minus button on each order line, and a language link with a 48-pixel touch area. 231
Figure 10.1 The two booking pages on a 360-pixel phone: (a) the desktop page, laid out at 980 pixels and shrunk to fit, and (b) the mobile-first page with readable text, labeled 48-pixel fields and the Call and WhatsApp bar within thumb reach. 251
Figure 10.2 Estimated arrival time of the first (2,492 kB) and the slim (76 kB) booking page on three networks, from Equation (10.4) with four round trips; the dashed line is the 2.5-second budget for a good LCP. The time axis is logarithmic. 252
Figure 11.1 The kiosk’s seven color pairs with their contrast ratios from Equation (11.3): three fail the WCAG 2.2 AA thresholds (4.5:1 for text, 3:1 for large text and for the parts of controls). 274
Figure 11.2 The repaired kiosk page at 360 CSS pixels after three presses of Tab: real buttons with visible text, a labeled card field with a hint and a 3:1 border, and a 4-pixel amber focus ring on the focused button. 275
Figure 12.1 Norman’s three levels of emotional design applied to Laundry Wangi: the visceral first impression, the behavioral experience of ordering and tracking, and the reflective judgment of honesty that decides whether a customer recommends the laundry. 283
Figure 12.2 The two consent screens on a phone: (a) the first version, with a red banner, a restarting countdown, one pre-ticked box for four purposes, a large “agree” button and a small shaming link; (b) the second version, with one choice per purpose, optional purposes off, plain explanations and two equal buttons. 294
Figure 13.1 The autonomy ladder: the risk and reversibility of each action decide how much the AI may do without asking, with an example from the laundry for every rung. 305
Figure 13.2 Calibration of the simulated care-label model: points below the diagonal mean the model claims more confidence than it earns; the red lines are the gaps that the ECE averages. 319
Figure 14.1 The checkout funnel of the simulated experiment: share of the users who opened the menu that reached each step, for the old checkout (A) and the new one (B). 335
Figure 15.1 The visual angle θ of an object of size s at distance d: the same angle can be produced by a small object nearby or a large object far away. 347
Figure 15.2 The dwell-time trade-off on the simulated gaze log: false activations fall and missed selections rise as the dwell grows; the lowest cost of Equation (15.4) is at 700 ms. 357
Figure 16.1 The capstone path: four phases, the chapters whose methods each phase applies, and the loop back that turns one design into an iterative process. 365
Figure 16.2 The running kiosk, captured with Playwright: the home screen and the number screen. Markers: 1 language and text size, 2 urgent-help note, 3 the four services, 4 the number and the people ahead in a status region. 369
Figure 16.3 The larger-text mode after one press of the Tab key: the skip link appears with a navy focus ring, and the text reflows. 378
Figure 16.4 SUS scores in the pilot test, colored by age group, with the mean (72.5) and the commonly used average of 68; all three participants aged 60 or over scored below the average. 385

List of Tables

Table 1.1 The eras of human-computer interaction and what each asked of its users. 4
Table 1.2 The tools of the workbench and the versions used throughout the book. 6
Table 1.3 The Keystroke-Level Model operators and their standard unit times. 15
Table 2.1 Minimum sizes for touch targets in current guidelines. 35
Table 2.2 A human-factors checklist for reviewing a screen. 47
Table 3.1 Nielsen's ten usability heuristics, in plain words, with examples from the first order page of Profitina Laundry. 56
Table 3.2 Nielsen's severity scale for usability problems. 58
Table 3.3 A recording template for heuristic evaluation. 68
Table 4.1 The six principles of human-centred design in ISO 9241-210:2019, with what they mean at the ojek base. 74
Table 4.2 The same cycle in three vocabularies, with typical methods for each step. 77
Table 4.3 A persona built from research notes, with the design implications it suggests. 78
Table 4.4 A one-day synthesis workshop plan with its outputs. 90
Table 5.1 Choosing a first method from the research question, with typical sample sizes and durations for a small team. 98
Table 5.2 A one-page research plan for the Laundry Bersih study. 98
Table 5.3 Interview traps and better questions for the laundry study. 100
Table 7.1 The severity scale for usability problems, with examples from the ordering page. 153
Table 9.1 The five primary interaction styles, what each asks of the user, and where each appears at Bu Tini's counter. 205
Table 9.2 Minimum sizes for touch and pointer targets as of September 2026. 208
Table 9.3 A worksheet for choosing and checking an interaction style, filled in for the warung kiosk. 210
Table 10.1 A mobile-first checklist, with the results for the first version of the booking page. 241
Table 11.1 The nine success criteria added in WCAG 2.2. 260
Table 11.2 The six most common failures of the WebAIM Million 2026, the WCAG 2.2 criteria behind them, and the checks of this chapter's programs. 263
Table 12.1 The dark patterns of the first Laundry Wangi flow, why each is unfair, and the fair alternative. 285
Table 12.2 The fair-flow review: questions to answer "yes" to before releasing a sign-up, subscription or consent flow. 287
Table 13.1 The four questions a person has about any AI system, and how the order assistant answers each one. 302
Table 13.2 A selection of the 18 guidelines for human-AI interaction and how each shapes the order assistant. 304
Table 13.3 The AI interaction canvas, filled in for the laundry's order assistant. 307
Table 14.1 The HEART Goals–Signals–Metrics process applied to the warung's ordering website; engagement is left out on purpose, because more time on an ordering page is not a goal. 328
Table 14.2 The experiment plan, filled in for the warung's checkout; every row is decided before the first visitor is assigned. 331
Table 15.1 Devices that shape spatial and wearable interface design in 2026, with the inputs and outputs a designer must plan for; prices are at launch in the United States. 348
Table 15.2 The context-of-use checklist for emerging interfaces, answered for the puskesmas waiting room. 351
Table 16.1 The methods of Chapters 2–15 and the decision each one contributed to the queue kiosk. 366
Table 16.2 The WCAG 2.2 success criteria checked for the queue kiosk and how each is met. 368
Table 16.3 The three pilot-test tasks and their success criteria. 378

Table 16.4 Pilot-test findings with severity ratings and the changes planned for the second iteration. 385

List of Listings

Listing 1.1 `Hello.java`: the first Java program, which also reports the Java version and the operating system. 11
Listing 1.2 `hello.py`: the first Python program. 12
Listing 1.3 `check_setup.py`: checks the Python version and looks for Java. 12
Listing 1.4 `CheckSetup.java`: checks the Java version and looks for Python. 13
Listing 1.5 `data/orders.txt`: the two designs written as Keystroke-Level Model operators. 16
Listing 1.6 `klm.py`: reads designs as operator lists and predicts their times. 16
Listing 1.7 `Klm.java`: the same Keystroke-Level Model calculator in Java. 18
Listing 1.8 `test_klm.py`: tests of the Keystroke-Level Model calculator. 21
Listing 1.9 `KlmTest.java`: the same checks in Java. 21
Listing 2.1 `data/kiosk_old.csv`: the old layout, twelve buttons on one screen. 37
Listing 2.2 `data/kiosk_new.csv`: the new layout, six large buttons and *Other* on screen 1, six more on screen 2. 37
Listing 2.3 `humanfactors.py`: fits Fitts's and Hick's laws to the logs and computes the expected time of each kiosk layout. 39
Listing 2.4 `HumanFactors.java`: the same fitting and layout comparison in Java. 41
Listing 2.5 `test_humanfactors.py`: tests of the two laws and of the least-squares fit. 45
Listing 2.6 `HumanFactorsTest.java`: the same checks in Java. 46
Listing 3.1 `data/problems.csv`: the fourteen distinct problems of the first order page. 60
Listing 3.2 `data/findings.csv`: 26 findings of four evaluators, each with a severity from 0 to 4. 60
Listing 3.3 `heuristics.py`: merges the findings, ranks the problems and estimates how many evaluators are needed. 60
Listing 3.4 `Heuristics.java`: the same report in Java. 62
Listing 3.5 `test_heuristics.py`: tests of the model and of the ranking. 66
Listing 3.6 `HeuristicsTest.java`: the same checks in Java, plus CSV splitting. 67
Listing 4.1 `data/themes.csv`: the themes of the affinity diagram and their "how might we" phrases. 80
Listing 4.2 `data/notes.csv`: 37 notes from six passengers and four drivers, each with a stage, a theme and a pain score. 80
Listing 4.3 `data/ideas.csv`: ten ideas with impact, effort and votes. 81
Listing 4.4 `synthesis.py`: themes with their reach, pain per journey stage, "how might we" questions and prioritized ideas. 81
Listing 4.5 `Synthesis.java`: the same synthesis in Java. 83
Listing 4.6 `test_synthesis.py`: tests of the quadrants, the ranking of themes and the journey mean. 89
Listing 4.7 `SynthesisTest.java`: the same checks in Java. 89
Listing 5.1 `survey_stats.py`: summarizes a Likert survey with means, 95% confidence intervals and top-2-box shares. 103
Listing 5.2 `SurveyStats.java`: the same survey summary in Java. 105
Listing 5.3 `saturation.py`: code frequencies, new codes per interview and the saturation rule of Guest, Namey and Chen. 109
Listing 5.4 `Saturation.java`: the same analysis in Java. 110
Listing 5.5 `test_research.py`: tests of the survey statistics and the saturation rule. 114
Listing 5.6 `ResearchTest.java`: the same checks in Java. 115
Listing 6.1 `flow_v1.txt`: the kiosk flow of the paper test, one line per button. 126
Listing 6.2 `flow_check.py`: finds unreachable screens, dead ends and the fewest taps to every goal. 127

Listing 6.3 `FlowCheck.java`: the same checker in Java. 129
Listing 6.4 `flow_v2.txt`: the flow after the paper test. 132
Listing 6.5 `kiosk_pages.py`: the texts, services, style and the six screens of the kiosk prototype. 134
Listing 6.6 `kiosk_proto.py`: the kiosk prototype served by Flask on port 4060. 135
Listing 6.7 `KioskPages.java`: the same screens in Java. 136
Listing 6.8 `KioskProto.java`: the kiosk prototype served by the JDK's built-in HTTP server on port 4061. 139
Listing 6.9 `shoot.py`: walks through the running prototype with Playwright and saves screenshots. 141
Listing 6.10 `test_proto.py`: tests of the flow checker and the prototype. 143
Listing 6.11 `ProtoTest.java`: the same checks in Java. 144
Listing 7.1 `usability_metrics.py`: task success with adjusted-Wald intervals, geometric mean times, SUS with grade, UMUX-Lite and problem discovery. 158
Listing 7.2 `UsabilityMetrics.java`: the same metrics in Java. 160
Listing 7.3 `test_eval.py`: tests of the usability metrics and the A/B test. 166
Listing 7.4 `EvalTest.java`: the same checks in Java 166
Listing 7.5 `ab_test.py`: the two-proportion z-test, the confidence interval of the difference and the sample size. 169
Listing 7.6 `AbTest.java`: the same A/B analysis in Java 170
Listing 8.1 `contrast.py`: checks token color pairs against WCAG 2.2 contrast minimums and suggests a compliant color. 182
Listing 8.2 `Contrast.java`: the same checker in Java. 184
Listing 8.3 `cardsort.py`: co-occurrence similarities and average-linkage clustering of an open card sort. 191
Listing 8.4 `CardSort.java`: the same analysis in Java. 193
Listing 8.5 `test_design.py`: tests of the contrast checker and the card-sort analysis. 197
Listing 8.6 `DesignTest.java`: the same checks in Java. 198
Listing 9.1 `menu.csv`: the five dishes, their prices and the words customers use for them. 211
Listing 9.2 `order_core.py`: the menu, the cart and the two parsers for the command and voice styles. 211
Listing 9.3 `OrderCore.java`: the same menu, cart and parsers in Java. 214
Listing 9.4 `cli.py`: the command style and the typed voice style, with explicit confirmation below the confidence threshold. 218
Listing 9.5 `Cli.java`: the same command and voice front end in Java. 219
Listing 9.6 `session.txt`: a scripted session with commands, utterances and three deliberate troubles. 221
Listing 9.7 `kiosk_v1.html`: the first version of the kiosk page template, with the placeholders that the servers fill in. 221
Listing 9.8 `kiosk.py`: the direct-manipulation kiosk served by Flask on port 4090. 222
Listing 9.9 `Kiosk.java`: the same kiosk with the JDK's built-in HTTP server on port 4091. 224
Listing 9.10 `audit_targets.py`: measures every tappable element of the running kiosk at phone width and compares it with the three guidelines. 227
Listing 9.11 `test_order_core.py`: tests of the order core for pytest. 229
Listing 9.12 `OrderCoreTest.java`: the same checks in plain Java 229
Listing 10.1 `naive.html`: the first, desktop-only version of the booking page. 242
Listing 10.2 `responsive.html`: the mobile-first rewrite with a viewport line, fluid layout, labeled fields and a thumb-reach action bar. 242
Listing 10.3 `strings.csv`: the page's words in English and Indonesian. 243
Listing 10.4 `pages.py`: fills either template in either language 244
Listing 10.5 `app.py`: serves both pages with Flask on port 4100. 244
Listing 10.6 `Pages.java`: the same template filling in Java. 245
Listing 10.7 `Server.java`: serves both pages with the JDK's HTTP server on port 4101. 246

Listing 10.8 `check_responsive.py`: measures layout width, overflow, effective text and target sizes and the phone field at four widths. 247
Listing 10.9 `resources_v1.csv`: the resources of the first, finished design (the slim design is in `resources_v2.csv`). 249
Listing 10.10 `networks.csv`: the three network profiles used for the estimate. 249
Listing 10.11 `budget.py`: estimates the arrival time of a page on each network and compares it with the LCP budget. 249
Listing 10.12 `Budget.java`: the same estimate in Java. 249
Listing 10.13 `test_ch10.py`: tests of the estimate and of the page rendering. 252
Listing 10.14 `Ch10Test.java`: the same checks in plain Java. 252
Listing 11.1 `queue_v1.html`: the first kiosk page, with the most common accessibility failures. 263
Listing 11.2 `colors.csv`: the kiosk's color pairs and the kind of content each colors. 264
Listing 11.3 `contrast.py`: the WCAG contrast ratio and a darker color that passes. 264
Listing 11.4 `Contrast.java`: the same contrast checker in Java. 265
Listing 11.5 `a11y_audit.py`: finds missing language, alternative texts, labels and names, keyboard traps of clickable elements, heading jumps and duplicate ids. 267
Listing 11.6 `A11yAudit.java`: the same audit in Java with `java.util.regex`. 269
Listing 11.7 `queue_v2.html`: the repaired kiosk page, using native HTML elements, labels, text alternatives, a live region and a visible focus ring. 272
Listing 11.8 `test_ch11.py`: tests of the contrast checker and the audit. 275
Listing 11.9 `Ch11Test.java`: the same checks in plain Java. 276
Listing 12.1 `flow_v1.txt`: the first sign-up and membership flow of Laundry Wangi, described line by line. 287
Listing 12.2 `consent_lint.py`: checks a described flow for pre-ticked and bundled consent, interface interference, confirmshaming, false urgency and hard exits. 288
Listing 12.3 `ConsentLint.java`: the same linter in Java. 289
Listing 12.4 `flow_v2.txt`: the repaired flow, with purposes off by default, equal buttons, no countdown and an exit as easy as the entrance. 291
Listing 12.5 `consent_log.csv`: consent events with time, user, purpose, action and the version of the consent text shown. 291
Listing 12.6 `consent_ledger.py`: replays consent events into the current state and answers whether a purpose is allowed. 291
Listing 12.7 `ConsentLedger.java`: the same ledger in Java. 292
Listing 12.8 `test_ch12.py`: tests of the linter and the ledger. 294
Listing 12.9 `Ch12Test.java`: the same checks in plain Java. 295
Listing 13.1 `laundry_assistant.py`: the order assistant, with confidence words, questions for missing slots, confirmation, undo and a way to a person. 308
Listing 13.2 `LaundryAssistant.java`: the same assistant in Java. 310
Listing 13.3 `make_log.py`: writes a seeded, simulated log of 200 reviewed cases for each version of the screen. 315
Listing 13.4 `reliance.py`: over-reliance, under-reliance, right final decisions and the expected calibration error. 315
Listing 13.5 `Reliance.java`: the same analysis in Java. 316
Listing 13.6 `test_assistant.py`: tests of parsing, price and the undo window. 319
Listing 13.7 `AssistantTest.java`: the same checks in Java. 320
Listing 14.1 `make_events.py`: writes two seeded, simulated event logs, one healthy and one with a logging bug in variant B. 332
Listing 14.2 `funnel.py`: users per checkout step and variant, with step and overall conversion. 332
Listing 14.3 `Funnel.java`: the same funnel in Java. 333
Listing 14.4 `abtest.py`: sample ratio check, difference with 95% confidence interval, *z*-test and sample size. 335
Listing 14.5 `AbTest.java`: the same check in Java. 337

Listing 14.6 `test_abtest.py`: tests of the error function, the ratio check, the *z*-test and the sample size. 340
Listing 14.7 `AbTestTest.java`: the same checks in Java. 340
Listing 15.1 `visual_angle.py`: physical size for a visual angle at four distances, and the angle of a 1 cm button. 352
Listing 15.2 `VisualAngle.java`: the same calculator in Java. 352
Listing 15.3 `make_gaze.py`: writes a seeded, simulated log of 1,000 glances, marking which ones were intended selections. 354
Listing 15.4 `dwell.py`: false activations and missed selections for each dwell time, and the dwell with the lowest cost. 355
Listing 15.5 `Dwell.java`: the same analysis in Java. 355
Listing 15.6 `test_spatial.py`: tests of the geometry and of the direction of the dwell trade-off. 357
Listing 15.7 `SpatialTest.java`: the same checks in Java. 358
Listing 16.1 `kiosk_app.py`: the queue kiosk in Python with Flask, in two languages and with a larger-text mode. 369
Listing 16.2 `KioskServer.java`: the same kiosk with the JDK's HTTP server. 372
Listing 16.3 `shoot.py`: Playwright screenshots of our own running kiosk, in both languages. 377
Listing 16.4 `evaluate.py`: SUS, UMUX-Lite, task completion with adjusted Wald intervals, and WCAG color contrast. 379
Listing 16.5 `Evaluate.java`: the same evaluation in Java. 381
Listing 16.6 `test_evaluate.py`: tests of SUS scoring, contrast and the adjusted Wald interval. 386
Listing 16.7 `EvaluateTest.java`: the same checks in Java. 386

List of Equations

(1.1) 15
(2.1) 34
(2.2) 34
(2.3) 34
(2.4) 35
(2.5) 36
(2.6) 38
(2.7) 38
(3.1) 58
(3.2) 58
(3.3) 58
(4.1) 77
(4.2) 78
(4.3) 79
(5.1) 102
(5.2) 102
(5.3) 102
(5.4) 108
(7.1) 154
(7.2) 155
(7.3) 156
(7.4) 157
(7.5) 164
(7.6) 168

(7.7) 168
(8.1) 180
(8.2) 181
(8.3) 181
(8.4) 182
(8.5) 190
(8.6) 190
(9.1) 207
(9.2) 207
(9.3) 209
(10.1) 239
(10.2) 239
(10.3) 239
(10.4) 240
(11.1) 261
(11.2) 261
(11.3) 261
(12.1) 285
(12.2) 285
(13.1) 303
(13.2) 305
(13.3) 305
(13.4) 305
(13.5) 306
(14.1) 327
(14.2) 327
(14.3) 329
(14.4) 329
(14.5) 329
(14.6) 330
(15.1) 347
(15.2) 347
(15.3) 347
(15.4) 355
(16.1) 367
(16.2) 367
(16.3) 379
(16.4) 379
(16.5) 379
(16.6) 379

Preface

Bu Ratna's warung takes the same order — two iced teas, one fried rice, one geprek chicken — on two different screens, and on one of them customers give up halfway. An elderly patient stands in front of a puskesmas queue kiosk and cannot find the button she needs. A laundry's customers stop asking by WhatsApp and stop coming. A chat assistant quotes a price at midnight, and nobody can tell whether it is right. None of these problems is a bug in the code; every one of them is a problem in the meeting between a person and a machine. Human-computer interaction is the discipline that studies that meeting, and it has methods, measurements and evidence for making it go well.

This book grew out of the course Human-Computer Interaction at ITS, where the subject has been taught to students of informatics for many years, and it has been written from the ground up for 2026. Its foundations are the lasting ones — human perception and memory, Fitts's and Hick's laws, Norman's principles and Nielsen's heuristics, user-centred design, user research, prototyping and usability testing — because they do not change when the software does. Everything built on them has been brought up to date: WCAG 2.2 and the accessibility laws now in force, from the European Accessibility Act to Indonesia's own rules on disability and personal data; the design languages renewed in 2025, Apple's Liquid Glass, Material 3 Expressive and Fluent 2; passkeys, dark-pattern regulation and consent that is honest; and the new interfaces of generative AI — chat, copilots and agents — together with the spatial, wearable and ambient devices that are leaving the rectangle of the screen behind.

Who this book is for

The book is self-contained, and you are assumed to be a beginner. If you have never run a usability test, never written a line of code and never heard of a heuristic, you are exactly the reader this book was written for. Every term is introduced in plain words before it is used, every method is first shown on a small, real situation, the theory is given with the evidence behind it and the sources where it can be checked, and every program in the book was run before its output was printed here. Chapter 1 begins with one order on two designs and ends with a workbench that is ready for everything that follows.

Readers who already build interfaces, and students who need the rigor of a university course, will find the other half of the book: the research behind each principle, standardized questionnaires and metrics with their formulas and confidence intervals, experiments analysed properly, templates and checklists ready for real projects, and complete, tested programs to adapt to their own work. 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

Human-computer interaction is not only a matter of opinion and sketches; much of it can be measured, and measuring is what programs are for. So wherever a program helps, this book writes it twice, in **Java** and in **Python**, and the two versions read the same data and print identical output. Java is the language of large, long-lived systems, of Android phones and of many banks and government services; Python is the language of quick experiments and of data work. The

programs are deliberately small: calculators for Fitts's and Hick's laws, a heuristic-evaluation report generator, a survey summarizer, a SUS and UMUX-Lite scorer, a contrast checker, a consent-flow linter, an A/B test analysis, and small accessible web pages served by Java's built-in `com.sun.net.httpserver` and by **Flask**, checked in a real browser with **Playwright**. If you know neither language, start with Python and read the Java version after it.

All of it is 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. The design work itself needs nothing more than paper, a pen and the people who will use what you design.

What is in it

Chapter 1 explains why interfaces matter, following Bu Ratna's warung as it goes online: what HCI is, how it grew from punched cards to AI agents, why it matters in Indonesia in 2026, and it sets up the workbench.

Chapters 2 and 3 lay the foundations. Chapter 2 watches Bu Siti at a puskesmas kiosk to explain the human factors that no software update can change — perception, attention and memory — and turns Fitts's and Hick's laws into calculators. Chapter 3 follows Mbak Dewi's laundry as it leaves WhatsApp, presents Norman's principles and the families of design heuristics, meets the design languages of 2025–2026, and builds a heuristic-evaluation report generator.

Chapters 4 to 7 teach the process of designing the right thing. Chapter 4 helps an ojek base that wants an app with the human-centred design process of ISO 9241-210, the Double Diamond and design thinking, and looks at AI in the design process of 2026. Chapter 5 finds out why Bu Rina's laundry is losing customers with interviews, contextual inquiry, surveys and the coding of qualitative data. Chapter 6 prototypes a puskesmas queue kiosk, from paper to wireframes, clickable mockups and a coded prototype. Chapter 7 evaluates the pre-orders at Warung Bu Tini with heuristic evaluation, usability tests, performance measures and the SUS and UMUX-Lite questionnaires.

Chapters 8 to 10 are about the interface itself. Chapter 8 redesigns a rural bank's app with Gestalt, visual hierarchy, typography, color and contrast, design systems and information architecture. Chapter 9 compares the ways of ordering at Bu Tini's counter — GUI, touch, voice and gesture — with one order core and three front ends. Chapter 10 designs for Mas Yoga's budget phone, measuring a mobile-first booking page in a real browser at four widths and on three networks.

Chapters 11 and 12 are about people who are too often left out or misled. Chapter 11 makes a puskesmas website accessible with WCAG 2.2 and the laws that now require it, and tests it in four layers. Chapter 12 examines the membership flow of Bu Sri's laundry to show emotional design, trust, dark patterns and honest consent, and lints a consent flow automatically.

Chapters 13 to 15 look at the interfaces that are arriving now. Chapter 13 designs Bu Wati's midnight order assistant — chat, copilots and agents — around confirmation, undo and calibrated trust. Chapter 14 settles arguments about Bu Tini's quiet checkout with analytics, funnels, A/B experiments and metrics. Chapter 15 follows Mbah Sumi and a queue screen into spatial, wearable and ambient computing, with target sizes and dwell times computed rather than guessed.

Chapter 16, the capstone, designs and evaluates a complete interactive system: a self-service queue kiosk for Puskesmas Sehat Selalu, from research at the registration desk to a working bilingual kiosk and the evidence that it helps its patients.

Every chapter follows the same path: a situation from everyday Indonesian life, the idea in plain words, the theory with its evidence, a hands-on method with templates and checklists, a small companion program in Java and Python where it helps, running and testing it, HCI in practice at Profitina, common mistakes, a summary, exercises with complete solutions, and references.

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 `Chapter16`, 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.

How to read it

With a pen, some paper and VS Code open. Chapter 1 comes first for everyone, because it explains why interfaces matter and builds the workbench; after that the chapters climb from the person, through the process, to the interface and the new kinds of interfaces, and each one names the earlier chapters it builds on. Do the methods, not only read about them: sketch the paper prototype, interview two people, run a five-person usability test on something you use every day, and score it. When a program works, change it — a target size, a color, one answer in a questionnaire — and watch how the result changes. 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. He supervises and examines undergraduate theses, teaches MLOps for the IBM SkillsBuild program, and builds products: he founded Profitina, an AI-visibility platform, whose everyday design decisions close every chapter of this book. 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 worked through the earlier forms of this material in the Human-Computer Interaction course and asked the questions that showed where an explanation was still thin, and to the colleagues at ITS with whom the subject has been taught and refined over the years. To the warung owners, launderers, drivers, health workers and older customers whose everyday encounters with screens gave each chapter its story. The book owes its standard, too, to the published standards and guidelines of the W3C, ISO and the platform makers, to the official

documentation of Java, Python, Flask, Playwright and VS Code, and to the classic textbooks and research literature on human-computer interaction, which are cited wherever they are used.

Surabaya, September 2026



Irfan Subakti · yifana@gmail.com

Profitina

profitina.com

Chapter 1

Introduction to HCI: Why Interfaces Matter

Every day millions of people in Indonesia order food, pay for parking, queue at a clinic and send money through a screen they did not design and cannot change. When the screen is clear they hardly notice it; when it is not, they lose time, money and sometimes their dignity. This chapter explains what human-computer interaction (HCI) is and why it matters, sets up the workbench used throughout the book, and ends with a first small program that turns a design argument into numbers. By the end of the chapter you will be able to define usability, user experience and accessibility in the terms of the international standards; describe the two gulfs that separate a person's goal from a system's response; explain why interface quality is a public concern in Indonesia in 2026, with current figures and current law; install and check JDK 25, Python 3.14 and VS Code on Windows, macOS or Linux; and predict, with the Keystroke-Level Model, how many seconds and how many working hours a better screen saves.

1.1 Bu Ratna's Warung Goes Online

Bu Ratna runs a small warung near a campus in Surabaya: rice with fried chicken, geprek chicken, fried rice, iced tea and coffee. Since 2024 most of her orders arrive through WhatsApp, and her nephew Dimas, a student, built her a simple web page on the old laptop at the counter so that she can type each order in, see the queue and print a slip for the kitchen. Dimas is proud of it. It has a dropdown list of all forty-two menu items, a box for the quantity, a button *Add*, and at the end a button *Save*.

On the first Saturday, Bu Ratna stops using it at eleven in the morning. It is not that the page is broken; every order it receives is stored correctly. It is that each order takes her a long time. She must click the dropdown, scroll to find *Es Teh*, click it, click the quantity box, move her hand to the keyboard to type a 2, move her hand back to the mouse, click *Add*, and then do all of that again for the next item. When twenty students arrive at once after the midday prayer, she goes back to her notebook and a ballpoint pen.

Dimas's first reaction is the reaction of almost every programmer: *she will get used to it.* His second reaction, a week later, is more interesting. He watches her for an hour, counts the steps, and redraws the page as a grid of large tiles, one per menu item, with the most frequently ordered items at the top. A click on a tile adds one portion; a second click adds another. The same order that took more than twenty seconds now takes about ten. Bu Ratna uses the new page without being asked.

Nothing about the database, the network or the programming language changed between the two versions. What changed was the *interface*, the part of the system that a person sees, hears and touches, and through which the person tells the system what to do. The story repeats itself everywhere: at the queue kiosk of a puskesmas where an elderly patient cannot find the button for the dental clinic among twelve small labels; at a bank's ATM that keeps the card of anyone who walks away with the cash; in an ojek app whose pickup pin lands on the wrong side of a

river; in a laundry's WhatsApp ordering where no customer can remember which of four numbered options means "iron only". Section 1.11 of this chapter measures Bu Ratna's two pages with a model that predicts how long the steps take. The chapters that follow teach how to see such problems before the users do, how to design better, and how to prove that the result is better.

1.2 What Human-Computer Interaction Is

1.2.1 A definition, and three words

The classic definition was written by the curriculum committee of ACM SIGCHI, the association's special interest group on computer-human interaction: *human-computer interaction is a discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them* [1]. Four words in that sentence carry the whole book. *Design* means deciding what the system should be like before it is built. *Evaluation* means finding out, with real people and real tasks, whether it works for them. *Implementation* means building it, which is why this book contains programs. And *human use* means that the measure of success is not whether the code runs but whether a person can reach a goal with it.

HCI draws on computer science for the machine side, on psychology and cognitive science for the human side, and on design, sociology, ergonomics and linguistics for everything in between. It is practiced under many names: usability engineering, user experience (UX) design, interaction design, human factors, accessibility engineering. In Indonesian universities the course is called *Interaksi Manusia dan Komputer* (IMK).

Three words are used constantly, and the international standards give them precise meanings.

> **DEFINITION.** *Usability* is the extent to which a system, product or service can be used by specified users to achieve specified goals with *effectiveness* (accuracy and completeness), *efficiency* (the resources used, such as time and effort) and *satisfaction* (the user's physical, cognitive and emotional responses) in a specified context of use (ISO 9241-11:2018) [2]. *User experience* is a person's perceptions and responses that result from the use or anticipated use of a system, before, during and after use (ISO 9241-210:2019) [3]. *Accessibility* is the extent to which the system can be used by people with the widest range of needs, characteristics and capabilities, including people with disabilities [2].

Notice the word *specified* that appears three times in the definition of usability. There is no such thing as "usable" in general. Dimas's first page is perfectly usable by Dimas at his desk. It fails for a specified user (Bu Ratna, fifty-four, a fast cook and a slow mouse user), a specified goal (enter an order of three items), and a specified context of use (twenty hungry students at the counter). Change any of the three and the verdict can change. This is why every method in this book begins by asking *who*, *what for* and *where*.

The three qualities of usability can also be measured, and that is what makes the definition useful. Effectiveness is measured, for example, as the percentage of orders entered without a mistake; efficiency as the seconds per order; satisfaction with a short questionnaire such as the System Usability Scale of Chapter 7. User experience is wider: it includes Bu Ratna's worry on Friday night that tomorrow will be as slow as last Saturday, and her small pleasure, a week later, when the queue is empty before the next wave.

1.2.2 The two gulfs

Why is one interface hard and another easy? The most useful single answer comes from the cognitive scientist Don Norman, whose book *The Design of Everyday Things* is still the best first reading in the field [4]. Norman describes every action as a small cycle. A person has a goal (order two iced teas); plans an action, specifies the steps and performs them (the *execution* side); then perceives what the system shows, interprets it and compares it with the goal (the *evaluation* side). Between the person and the system lie two gulfs, drawn in **Figure 1.1**.

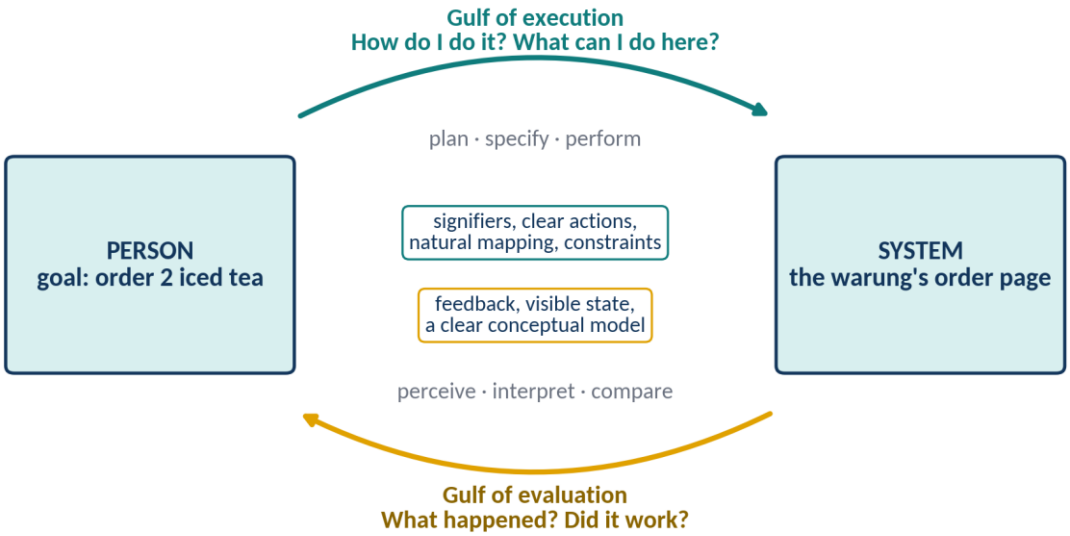


Figure 1.1 Norman's two gulfs: execution (how do I do it?) and evaluation (did it work?), and the design qualities that bridge each one.

The *gulf of execution* is the distance between what the person wants to do and what the system lets them do, as they understand it. On Dimas's first page the gulf is wide: the goal "two iced teas" must be translated into five separate controls. The *gulf of evaluation* is the distance between what the system shows and what the person needs to know to judge whether the goal was reached. When Bu Ratna clicked *Add* and nothing visible happened except a small line at the bottom of a long list, she clicked again, and the kitchen made four iced teas.

Good design bridges both gulfs. Signifiers that show what can be done, a natural mapping between controls and effects, and constraints that make wrong actions impossible narrow the gulf of execution. Immediate feedback, a visible state and a clear conceptual model narrow the gulf of evaluation. Chapter 3 treats each of these principles in detail; for now it is enough to notice that every problem in Section 1.1 is a wide gulf on one side or the other. The puskesmas kiosk has a wide gulf of execution (which of twelve labels is "the dentist"?); the ATM that keeps forgotten cards once had a wide gulf of evaluation (the person's goal, cash, was complete, so the card was forgotten), and banks bridged it by returning the card *before* the cash.

> **KEY INSIGHT:** When a person makes a mistake with an interface, the first question is not "what is wrong with the person?" but "which gulf was too wide?". Blaming users is easy and changes nothing; narrowing a gulf changes the behavior of every future user.

1.2.3 What HCI is not

Three misunderstandings are common enough to name. HCI is not *making things pretty*: visual design matters (Chapter 8), but a beautiful page that hides its main action has failed. HCI is not *asking users what they want*: people are poor predictors of their own behavior, and Chapter 5

shows why observation beats opinion. And HCI is not *common sense*: if it were, Dimas's first page would have been his last. What looks obvious to the person who built a system is precisely what that person can no longer see, because they already know how it works. The discipline exists to replace that blind spot with evidence.

1.3 From Punched Cards to Agents: a Short History and the Present

The history of HCI is a history of who the user is. In the 1950s and 1960s the user was a specialist who prepared punched cards and waited hours for a printout; the interface was a queue. Ivan Sutherland's Sketchpad (1963) let a person draw directly on a screen with a light pen, and Douglas Engelbart's demonstration of 1968 showed the mouse, windows and shared editing. At Xerox PARC in the 1970s the Alto combined them into the graphical user interface, which the Apple Macintosh (1984) and Microsoft Windows brought to offices and homes. The web of the 1990s put interfaces in front of people who had never been trained on any computer; the iPhone (2007) and Android made the touchscreen the most common interface on earth. Jonathan Grudin's history of the field traces how HCI moved in the same period from tools for experts to partners for everyone [5].

Table 1.1 summarizes the eras with what each one asked of the user.

Table 1.1 The eras of human-computer interaction and what each asked of its users.

Era	Typical interface	Who the user was	What the user had to learn
1950s–1970s	Batch jobs, punched cards, command lines	Trained specialists	Exact syntax of commands
1980s–1990s	Graphical user interface: windows, icons, menus, pointer	Office workers, then home users	Metaphors such as desktop, folder, trash
1990s–2000s	The web	Anyone with a browser	Links, forms, search
2007–2020s	Touchscreens, mobile apps	Almost everyone, including first-time users	Gestures, app conventions
2020s	Voice, conversational and generative AI, agents, spatial computing	Everyone, often without any screen at all	What to ask, and when to trust the answer

Where are we in September 2026? Four developments shape what a student of HCI meets today.

The design languages were renewed. On June 9, 2025 Apple introduced Liquid Glass, a translucent material that "reflects and refracts its surroundings" and adapts to the content beneath it, across iOS 26, iPadOS 26, macOS Tahoe 26, watchOS 26 and tvOS 26 [6]. Google's Material 3 Expressive, released the same spring, was backed by 46 research studies with more than 18,000 participants; Google reports that people found key interface elements up to four times faster in expressive layouts [7]. Microsoft's Fluent 2 is the shared design system of Windows and Microsoft 365 [8]. These are not fashions to copy; Chapter 3 shows how each one applies (and sometimes strains) the same principles.

Computing left the flat screen. Apple Vision Pro went on sale on February 2, 2024 and made "spatial computing" a mainstream term [9]; watches, earbuds and cars carry their own interfaces. Chapter 15 treats these.

Generative AI became an interface. In February 2026 ChatGPT alone reported 900 million weekly users [10]. A text box that answers in paragraphs, a copilot inside a spreadsheet and an *agent* that books a ticket on the user's behalf are new interaction styles with old problems: the gulf of execution becomes "what should I ask?" and the gulf of evaluation becomes "is this answer true,

and what did the agent just do?". Research on these questions began before the boom; the eighteen guidelines for human-AI interaction of Amershi et al. at CHI 2019 remain the standard starting point [11]. Chapter 13 is devoted to them.

The research community is larger than ever. The ACM CHI conference, the main venue of the field, met in Barcelona from April 13 to 17, 2026, accepted 1,702 full papers out of 6,730 submissions (25.3%) and welcomed 5,254 participants on site [12]. Design tools changed as fast: at Figma's Config 2026 (June 23–24, 2026) the company announced an AI agent inside the design canvas, code layers and motion design [13]. Chapter 6 uses such tools; this book never lets them replace watching real people.

1.4 Why Interfaces Matter in Indonesia in 2026

1.4.1 Who is online, and on what

The Indonesian Internet Service Providers Association (APJII) released its 2026 survey on May 22, 2026: 235.2 million Indonesians, 81.72% of a population of 287.9 million, use the internet, about six million more than a year before [14]. Penetration ranges from 85.95% in Java to 69.74% in Maluku and Papua. Among the people who are still offline, 34% say they have no device and 31.5% say they lack digital literacy [14]. The second number is an HCI number: a third of the people who are not yet online are held back not by cables or money but by interfaces that ask too much of a first-time user.

The devices are mostly Android phones. In August 2026 StatCounter measured 77.07% of mobile web page views in Indonesia on Android and 22.88% on iOS [15]. The same service counted 60.59% of all page views in Indonesia from desktop computers and 38.98% from phones in August 2026 [16]. That second figure surprises people, and it teaches a lesson worth learning early: StatCounter counts *page views on the websites that carry its tracking code*, not people, and heavy desktop users at work view many pages. Most of the everyday life of Indonesians online happens inside apps, which such counters do not see. Always ask what a statistic counts before designing for it.

The users are also getting older. According to Statistics Indonesia (BPS), people aged sixty and over were 11.97% of the population in 2025, which means that Indonesia has entered the phase of an *ageing population*; the share ranges from 17.83% in Yogyakarta to 6.71% in Central Papua [17]. Chapter 2 explains how vision, hearing, memory and movement change with age, and why a screen designed by twenty-year-olds for twenty-year-olds fails a growing part of the country.

1.4.2 The law now asks for good interfaces

Interface quality has become a legal matter. Indonesia's Law No. 8 of 2016 on Persons with Disabilities recognizes the right to accessibility, including accessible information and communication [18], and Government Regulation No. 42 of 2020 details accessibility of public services for persons with disabilities [19]. Law No. 27 of 2022 on Personal Data Protection (UU PDP) requires, among other lawful bases, explicit and valid consent for specific purposes; its implementing regulation, Government Regulation No. 33 of 2026, was signed on July 16, 2026 and takes effect on January 16, 2027 [20], [21]. A consent screen that a person cannot understand is therefore not only bad design; it is a weak legal basis. Chapter 12 treats consent design in depth.

Abroad, the European Accessibility Act (Directive (EU) 2019/882) has applied since June 28, 2025 to products and services such as e-commerce, banking, e-books and transport ticketing sold

in the European Union, including by companies outside it [22]. The technical yardstick behind most of these laws is the Web Content Accessibility Guidelines of the W3C. WCAG 2.2 is the current Recommendation [23]; its successor, WCAG 3.0, is still a working draft, updated for review most recently in September 2026, and the W3C says it is not expected to be a completed standard for a few more years [24]. Chapter 11 works through WCAG 2.2 criterion by criterion.

1.4.3 How this book works

Every chapter follows the same path. It starts with a situation from everyday Indonesian life, explains the idea in plain words, gives the theory with its evidence and sources, turns it into a hands-on method with templates and checklists, and, where a program helps, builds a small companion program in Java and in Python whose output is printed exactly as the program produced it. A section *HCI in Practice: Profitina* shows where the chapter's ideas appear in a real Indonesian product, followed by common mistakes, a summary, exercises and their complete solutions. The programs are deliberately small: the aim is not to build the warung's system but to make HCI measurable. Fitts's and Hick's laws become calculators in Chapter 2; heuristic evaluation becomes a report generator in Chapter 3; research notes become an affinity board in Chapter 4.

1.5 Your Workbench: What We Install and Why

A carpenter's first job in a new workshop is to set up the bench. Ours has three main tools, summarized in **Table 1.2**. The versions are the current ones as of September 2026; later patch releases (a JDK 25.0.5 or a Python 3.14.8) work exactly the same.

Table 1.2 The tools of the workbench and the versions used throughout the book.

Tool	What it is	Version used in this book
JDK (Java Development Kit)	The Java compiler `javac`, the Java runtime `java` and the standard library	JDK 25 LTS, update 25.0.4 (July 2026) [25], [26]
Python	The Python interpreter `python` and its standard library, with `pip` and `venv`	3.14.7 (August 5, 2026) [27]
Visual Studio Code	A free code editor for Windows, macOS and Linux	1.139 (September 23, 2026) [28]
Extension Pack for Java	Adds Java editing, running, debugging and testing to VS Code	from the Extensions view [29]
Python extension	Adds Python editing (Pylance), running and debugging to VS Code	from the Extensions view [30]

Why JDK 25? Java has a new version every six months, and every two years one of them is a *long-term support* (LTS) release that receives security updates for many years. JDK 25 was released on September 16, 2025 as the current LTS [26]; Oracle provides free updates for it until at least September 2028, and the July 2026 update is 25.0.4 [25]. JDK 27 appeared on September 15, 2026 [31], but it is a short-term release, and for a book whose programs should keep working for years the LTS is the right choice. Every program in this book uses only features that exist since Java 17, so JDK 17 and 21 also work; JDK 25 is simply the best one to install today.

There are several builds of the same JDK, all made from the open-source OpenJDK code: Oracle's own, Eclipse Temurin from the Adoptium project, Microsoft's, Amazon Corretto, and the packages of Linux distributions. They behave identically for our purposes. We use **Eclipse Temurin** in the instructions because it is free for any use, has a simple installer for every system, and appears in

the package managers of all three systems; the September 2026 security release is Temurin 25.0.4.1 [32].

Why Python 3.14? Python 3.14 was first released on October 7, 2025 and brought, among other things, template strings, deferred evaluation of annotations, an officially supported build without the global interpreter lock, and a friendlier interactive shell with colors [33]. The programs of this book do not need those features; they use plain, long-established Python and also run on Python 3.11 or newer. We install 3.14 because it is current and because later chapters use libraries tested against it.

Why VS Code? Because it is free, runs on all three systems, handles Java and Python equally well, and has a built-in terminal. Any other editor works too; IntelliJ IDEA and PyCharm are excellent. The instructions in this book use VS Code so that one set of screenshots and menu names fits everyone.

Before installing anything it helps to know what happens when you "run" a program in each language, because the difference explains most first-day error messages. **Figure 1.2** shows the two paths.

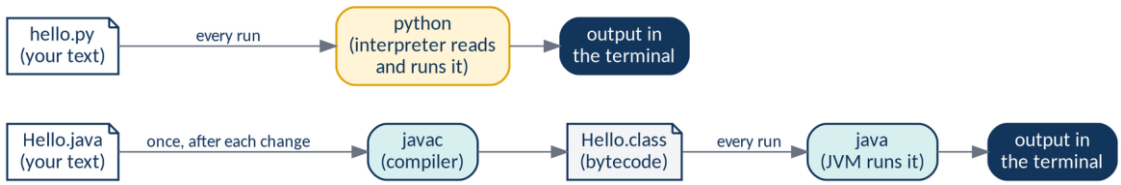


Figure 1.2 How a Java program and a Python program run. Java is compiled once by `javac` into bytecode, which the Java Virtual Machine runs; Python is read and run by the interpreter every time.

A Java program is text in a file such as `Hello.java`. The compiler `javac` checks it and translates it into *bytecode*, a compact set of instructions stored in `Hello.class`. The command `java` starts the Java Virtual Machine (JVM), which runs the bytecode. You compile once after each change and run as often as you like. A Python program is text in a file such as `hello.py`, and the command `python` reads and runs it directly each time; mistakes that Java finds while compiling, Python finds only when it reaches the faulty line. Since Java 11 you can also run a single Java file without compiling it first, with `java Hello.java`: the JVM compiles it in memory and runs it [34]. It is handy for small experiments; for programs made of several files we compile with `javac`.

1.6 Installing on Windows 10 and 11

1.6.1 The JDK

The simplest way is **winget**, the package manager built into Windows 10 and 11. Open **PowerShell** (press the Windows key, type `powershell`, press Enter) and type:

```
PS C:\Users\you> winget install EclipseAdoptium.Temurin.25.JDK
```

If you prefer clicking, open `https://adoptium.net`, choose **Temurin 25 (LTS)**, **Windows** and **x64** (or **aarch64** on an ARM laptop), download the `.msi` installer and run it. On the page **Custom Setup**, make sure that **Add to PATH** and **Set JAVA_HOME variable** are both set to "Will be installed on local hard drive"; the second is off by default, and some tools need it.

Close PowerShell, open a new one (a window opened before the installation does not see the new program), and check both commands:

```
PS C:\Users\you> java -version
openjdk version "25.0.4" 2026-07-21 LTS
PS C:\Users\you> javac -version
javac 25.0.4
```

The first line of `java -version` is what matters; two more lines follow that name the build, and the exact update number may be newer than 25.0.4. If Windows answers that `java` "is not recognized as the name of a cmdlet", the installer did not add Java to the `PATH`, the list of folders in which Windows looks for commands. Run the installer again and choose **Add to PATH**, or add `C:\Program Files\Eclipse Adoptium\jdk-25...\bin` yourself through **Settings › System › About › Advanced system settings › Environment Variables**.

1.6.2 Python

On Windows, Python is now installed with the **Python install manager**, a small program that downloads, updates and switches between Python versions [35]. Get it from the Microsoft Store (search for "Python Install Manager"), or from `https://www.python.org/downloads/windows/`, or with winget:

```
PS C:\Users\you> winget install 9NQ7512CXL7T -e --accept-package-agreements
```

Then open a new PowerShell window, install Python 3.14 and check it:

```
PS C:\Users\you> py install 3.14
PS C:\Users\you> python --version
Python 3.14.7
```

After this, `python`, `py` and `python3` all start Python. If `python --version` opens the Microsoft Store instead, close every terminal, open a new one and try again; the manager can also repair the settings with `py install --configure` [35].

1.6.3 VS Code

Install VS Code with `winget install Microsoft.VisualStudioCode`, or download the **User Installer** from `https://code.visualstudio.com`. In the installer, tick **Add to PATH** and **Add "Open with Code" action to Windows Explorer file context menu**: the first lets you type `code .` in a terminal to open the current folder, and the second lets you right-click a folder and open it in VS Code.

1.7 Installing on macOS

1.7.1 The JDK

The easiest way on a Mac is the Temurin installer package. Open `https://adoptium.net`, choose **Temurin 25 (LTS)**, **macOS** and **aarch64** for a Mac with an Apple chip (M1 and later) or **x64** for an older Intel Mac, download the `.pkg` file and double-click it. If you already use the Homebrew package manager, one command does the same:

```
you@mac ~ % brew install --cask temurin@25
```

Open **Terminal** (press Cmd+Space, type "Terminal", press Enter) and check:

```
you@mac ~ % java -version
openjdk version "25.0.4" 2026-07-21 LTS
you@mac ~ % javac -version
javac 25.0.4
```

If several JDKs are installed, macOS picks the newest. The command `/usr/libexec/java_home -V` lists them all.

1.7.2 Python

macOS may already contain a `python3` command, but it is an older version meant for Apple's developer tools; do not build on it. Download the macOS installer for Python 3.14.7 from `https://www.python.org/downloads/macos/` and double-click it; it is a universal package that runs natively on Apple and Intel chips. At the end the installer opens a folder **Applications › Python 3.14**: double-click **Install Certificates.command** there once, so that Python can make secure internet connections. With Homebrew, `brew install python@3.14` works as well. Then check:

```
you@mac ~ % python3 --version
Python 3.14.7
```

On macOS and Linux the command is `python3`, not `python`. The book writes `python3` in commands for macOS and Linux and `python` for Windows.

1.7.3 VS Code

Download VS Code for macOS from `https://code.visualstudio.com`, open the downloaded file and drag **Visual Studio Code** into **Applications**. To be able to type `code .` in the terminal, start VS Code, press Cmd+Shift+P to open the Command Palette, and run **Shell Command: Install 'code' command in PATH**.

1.8 Installing on Linux

Most Linux distributions carry recent versions of Java and Python in their own package repositories, and those are the best choice because the system keeps them updated. Ubuntu 26.04 LTS uses OpenJDK 25 as its default Java [36] and Python 3.14 as its default `python3`, but it installs neither `javac` nor Python's `venv` module unless you ask:

```
$ sudo apt update
$ sudo apt install openjdk-25-jdk python3 python3-venv python3-pip
$ java -version
openjdk version "25.0.4" 2026-07-21
$ javac -version
javac 25.0.4
$ python3 --version
Python 3.14.7
```

On Fedora the equivalent is `sudo dnf install java-25-openjdk-devel python3 python3-pip`. On an older distribution whose repository has no JDK 25, install Temurin from Adoptium's own repository following the instructions at `https://adoptium.net/installation`, or keep the JDK 21 of the distribution: every program in this book runs on JDK 17 or newer. The same holds for Python: Ubuntu 24.04 ships Python 3.12, which runs all the programs of this book, so you may keep it. Never remove or replace the system Python on Linux; parts of the operating system depend on it.

For VS Code, download the `.deb` package (Ubuntu, Debian, Mint) or the `.rpm` package (Fedora, openSUSE) from `https://code.visualstudio.com` and install it with `sudo apt install ./code_*.deb` or `sudo dnf install ./code-*.rpm`. The package also adds Microsoft's repository, so VS Code is then updated together with the rest of the system.

1.9 VS Code: Two Extensions and a Tour

Start VS Code. Click the **Extensions** icon in the Activity Bar on the left (or press Ctrl+Shift+X, on macOS Cmd+Shift+X) and install two extensions, both published by Microsoft:

- **Extension Pack for Java** (identifier `vscjava.vscode-java-pack`). It bundles the Java language support, the debugger, the test runner, Maven and Gradle support and a project manager [29]. It finds the JDK you installed by itself; if you have several, the command **Java: Configure Java Runtime** in the Command Palette lets you choose.
- **Python** (identifier `ms-python.python`). It brings **Pylance**, which completes names and marks mistakes while you type, and the **Python Debugger** [30].

Figure 1.3 shows the parts of the window that the book refers to. The Activity Bar (1) switches between Explorer, Search, Source Control, Run and Debug, and Extensions. The Explorer (2) shows the files of the folder you opened. The editor (3) is where you type. The Run button (4), a triangle at the top right of the editor, runs the current file. The integrated terminal (5), opened with **Terminal › New Terminal** or Ctrl+`, is an ordinary PowerShell, zsh or bash window that already sits in your project folder. The Status Bar (6) shows, among other things, which Python interpreter and which Java runtime VS Code is using; clicking the Python version lets you choose another.

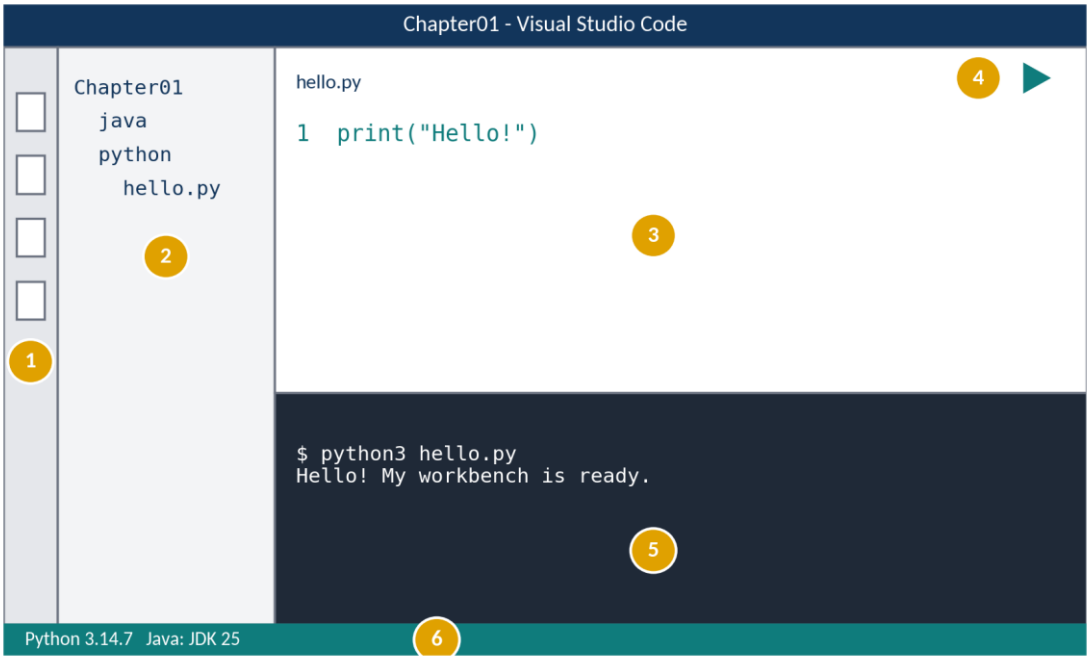


Figure 1.3 The parts of Visual Studio Code used in this book: (1) Activity Bar, (2) Explorer, (3) editor, (4) Run button, (5) integrated terminal, (6) Status Bar.

Always open a *folder* in VS Code (**File › Open Folder**), not a single file. The folder becomes your workspace: the Explorer shows it, the terminal starts in it, and the Java extension treats the `.java` files inside as one project. For this book, open the folder of the chapter you are reading, such as `Chapter01`.

A word about Python *virtual environments*. A virtual environment is a private folder, by convention `.venv`, with its own copy of `python` and its own place for installed packages, so that the libraries of one project do not collide with those of another [37]. The early chapters of this book use only the standard library and need no environment; the chapters that install libraries (such as Flask for the small web interfaces of later chapters) tell you to create one with `python -m`

`venv .venv` and to activate it with `.venv\Scripts\Activate.ps1` on Windows or `source .venv/bin/activate` on macOS and Linux. The check program of Section 1.10 reports whether an environment is active, but for Chapters 1 to 4 it does not matter.

1.10 Your First Programs

1.10.1 Hello, in Java

Create a folder `Chapter01` with the subfolders `java` and `python`, or open the `Chapter01` folder of the book's code. In VS Code choose **File › New File**, type the program in **Listing 1.1** and save it as `java/Hello.java`. In Java the file name must be the class name followed by `.java`, with the same capital letters.

Listing 1.1 `Hello.java`: the first Java program, which also reports the Java version and the operating system.

```
/** The first program: say hello and show which Java runs it. */
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello! My HCI workbench is ready.");
        System.out.println("Java version: " + System.getProperty("java.version"));
        System.out.println("Running on:   " + System.getProperty("os.name"));
    }
}
```

Line by line: `public class Hello` declares a *class*, the unit in which Java keeps code, named `Hello`. The comment above it, between `/**` and `*/`, is for people and is ignored by the compiler. `public static void main(String[] args)` is the *main method*, where every Java application starts; the words `public static void` say that it can be called from outside, needs no object, and returns nothing, and `String[] args` receives the words typed after the program's name on the command line. `System.out.println(...)` prints a line. `System.getProperty("java.version")` asks the running JVM for its version, so the program tells you which Java actually runs it, which is the first thing to check when something behaves strangely.

Open the terminal in VS Code (Ctrl+`), go into the `java` folder, compile and run:

```
$ cd java
$ javac -d out Hello.java
$ java -cp out Hello
Hello! My HCI workbench is ready.
Java version: 25.0.4
Running on:   Windows 11
```

The option `-d out` tells `javac` to put the compiled `Hello.class` in a folder called `out`, which keeps the source folder tidy, and `-cp out` (*class path*) tells `java` where to find it. On macOS the last line says `Mac OS X`, on Linux `Linux`. You can also press the Run button (4 in **Figure 1.3**); the Java extension compiles and runs the file and shows the output in the terminal. And, as Section 1.5 mentioned, `java Hello.java` compiles and runs a single file in one step.

JDK 25 also allows a much shorter first program. Since JEP 512 became final in JDK 25, a file may contain just a method `void main()` without any class around it, and a new helper class `IO` prints a line [38]:

```
void main() {
    IO.println("Hello!");
}
```

This is a welcome simplification for beginners. The programs in this book nevertheless use the classic form with a class and `public static void main`, because they are real applications made of several classes, and because the classic form also runs on JDK 17 and 21.

1.10.2 Hello, in Python

Listing 1.2 is the same program in Python. Save it as `python/hello.py`.

Listing 1.2 `hello.py`: the first Python program.

```
"""The first program: say hello and show which Python runs it."""
import platform

print("Hello! My HCI workbench is ready.")
print("Python version: " + platform.python_version())
print("Running on:     " + platform.system())
```

The text between the triple quotes at the top is a *docstring*, a description of the file. `import platform` loads a module from the standard library that knows about the computer and the interpreter; `platform.python_version()` returns the version as text and `platform.system()` the operating system. There is no class and no `main` method: Python simply runs the file from top to bottom. Run it from the terminal, in the `python` folder:

```
$ cd ../python
$ python3 hello.py
Hello! My HCI workbench is ready.
Python version: 3.14.7
Running on:     Linux
```

On Windows type `python hello.py`, and the last line says `Windows`; on macOS it says `Darwin`, the name of the core of macOS. In VS Code, open `hello.py` and press the Run button; if VS Code asks which interpreter to use, choose Python 3.14.

1.10.3 Checking the whole workbench

Rather than trusting that everything went well, we write a program that checks. **Listing 1.3** reports the Python version, whether a virtual environment is active, and whether a Java runtime can be found; **Listing 1.4** does the same from the Java side.

Listing 1.3 `check_setup.py`: checks the Python version and looks for Java.

```
"""Check the workbench: the Python version, the virtual environment and Java."""
import platform
import shutil
import subprocess
import sys

RECOMMENDED = (3, 14)      # the version this book is written for
MINIMUM = (3, 11)          # the oldest version the book's code runs on


def java_version():
    """Ask the 'java' command for its version; None when it is not found."""
    if shutil.which("java") is None:
        return None
    done = subprocess.run(["java", "-version"], capture_output=True, text=True)
    for line in (done.stderr + done.stdout).splitlines():  # java uses stderr
        if " version " in line and '"' in line:
            return line.split('"')[1]         # openjdk version "25.0.4" ...
    return "unknown"


def main():
    version = sys.version_info[:2]
    in_venv = sys.prefix != sys.base_prefix
    print(f"Python version : {platform.python_version()}")
    print(f"System         : {platform.system()} ({platform.machine()})")
    print(f"Virtual env.   : {'yes' if in_venv else 'no (fine for this book)'}")
    java = java_version()
    print(f"Java version   : {java or 'NOT FOUND - install JDK 25'}")
    if version < MINIMUM:
        print("Result         : Python is too old, please install Python 3.14")
    elif java is None:
        print("Result         : Python is fine; now install the JDK")
    elif version < RECOMMENDED:
        print("Result         : works; Python 3.14 is recommended")
    else:
        print("Result         : your workbench is ready")


if __name__ == "__main__":
    main()
```

The program uses four standard modules. `sys.version_info[:2]` is the version as a pair of numbers such as `(3, 14)`, and Python compares such pairs element by element, so `(3, 12) < (3, 14)` is true. `sys.prefix` and `sys.base_prefix` differ exactly when a virtual environment is active. `shutil.which("java")` looks for a `java` command on the `PATH` and returns `None` if there is none. `subprocess.run` starts `java -version` as a separate program and captures what it prints; Java prints its version to the *error* stream rather than the normal output, a historical quirk, which is why the function reads `done.stderr`. The function then picks the line that contains the word `version` and a quotation mark and returns the text between the first pair of quotation marks.

Listing 1.4 `CheckSetup.java`: checks the Java version and looks for Python.

```java
/** Check the workbench: the Java version, and whether Python can be found. */
public class CheckSetup {
    static final int RECOMMENDED = 25;       // the JDK this book is written for
    static final int MINIMUM = 17;           // the oldest JDK the code runs on

    /** Ask a command for its version; null when the command is not found. */
    static String versionOf(String... command) {
        try {
            Process p = new ProcessBuilder(command).redirectErrorStream(true)
                                                   .start();
            String out = new String(p.getInputStream().readAllBytes()).trim();
            p.waitFor();
            return out.isEmpty() ? null : out.lines().findFirst().get();
        } catch (Exception e) {
            return null;                     // not installed, or not on PATH
        }
    }

    public static void main(String[] args) {
        int feature = Runtime.version().feature();
        System.out.println("Java version    : " + Runtime.version());
        System.out.println("Java vendor     : " + System.getProperty("java.vendor"));
        System.out.println("System          : " + System.getProperty("os.name")
                           + " (" + System.getProperty("os.arch") + ")");
        String python = versionOf("python3", "--version");
        if (python == null) {
            python = versionOf("python", "--version");  // the name on Windows
        }
        System.out.println("Python          : "
                           + (python != null ? python : "NOT FOUND"));
        if (feature < MINIMUM) {
            System.out.println("Result          : Java is too old, install JDK 25");
        } else if (python == null) {
            System.out.println("Result          : Java is fine; now install Python");
        } else if (feature < RECOMMENDED) {
            System.out.println("Result          : works; JDK 25 is recommended");
        } else {
            System.out.println("Result          : your workbench is ready");
        }
    }
}
```

`Runtime.version().feature()` is the main version number of the running Java, 25 for JDK 25. `ProcessBuilder` starts another program; `redirectErrorStream(true)` merges its error stream into its normal output, so that the version is found whichever stream Python uses. If the command does not exist, starting it throws an exception, which the `catch` turns into `null`, meaning “not found”. Because Python on Windows is usually called `python` and on macOS and Linux `python3`, the program tries both.

On a finished workbench the two programs print:

```
$ python3 check_setup.py
Python version : 3.14.7
System         : Linux (x86_64)
Virtual env.   : no (fine for this book)
Java version   : 25.0.4
Result         : your workbench is ready
$ java -cp out CheckSetup
Java version   : 25.0.4+8-LTS
Java vendor    : Eclipse Adoptium
System         : Linux (amd64)
Python         : Python 3.14.7
Result         : your workbench is ready
```

The exact build numbers after the `+` differ from computer to computer. The last line is the one to look at. If it says anything other than "your workbench is ready", it also says what to install, and the programs of the book will still run as long as it says "works".

1.11 A First HCI Program: Counting the Cost of Clicks

1.11.1 The Keystroke-Level Model

Dimas's argument with himself in Section 1.1 ("she will get used to it" against "the page is slow") can be settled without guessing. In 1980 Stuart Card, Thomas Moran and Allen Newell published the *Keystroke-Level Model* (KLM), which predicts how long an experienced user needs to carry out a task on a given interface, before anyone builds it [39]. It became part of their larger GOMS family of models in *The Psychology of Human-Computer Interaction* [40] and is still taught and used because it is simple, cheap and surprisingly accurate for routine tasks.

The idea is to write the task as a sequence of elementary *operators* and to add up their times. **Table 1.3** lists the five operators we use, with the unit times recommended by Card, Moran and Newell and repeated in Kieras's practical guide [39], [41].

Table 1.3 The Keystroke-Level Model operators and their standard unit times.

Operator	Meaning	Time (s)
K	Press a key or a button on the keyboard (average skilled typist)	0.28
P	Point at a target on the screen with the mouse	1.10
B	Press or release a mouse button (a click is B B)	0.10
H	Move a hand from the keyboard to the mouse or back	0.40
M	Prepare mentally for the next step: decide, find, recall	1.35

If a task needs n_K keystrokes, n_P pointing movements, and so on, the predicted time is

$$T = n_K t_K + n_P t_P + n_B t_B + n_H t_H + n_M t_M \tag{1.1}$$

where $t_K = 0.28$ s, $t_P = 1.10$ s, $t_B = 0.10$ s, $t_H = 0.40$ s and $t_M = 1.35$ s are the unit times. Equation (1.1) is nothing more than "count each kind of step and multiply by how long it takes".

The hardest part is placing the M operators. Kieras's rule of thumb is to put one M before each *chunk* of the task that the user must think about, such as deciding which item comes next, but not inside a chunk that the hands carry out automatically, such as the two clicks of a double click [41]. We place one M before each ordered item and one before *Save*.

1.11.2 The two pages, by hand

Take the order that arrives most often at lunchtime: two iced teas, one fried rice, one geprek chicken. On Dimas's first page (design A), each item needs: M (decide the item), P B B (open the dropdown), P B B (choose the item), P B B (click the quantity box), H (hand to the keyboard), K (type the digit), H (hand back to the mouse), P B B (click *Add*). That is 1 M, 4 P, 8 B, 2 H and 1 K per item. Three items and a final M P B B for *Save* give

$$n_M = 4, \quad n_P = 13, \quad n_B = 26, \quad n_H = 6, \quad n_K = 3.$$

With Equation (1.1):

$$T_A = 4(1.35) + 13(1.10) + 26(0.10) + 6(0.40) + 3(0.28)$$
$$= 5.40 + 14.30 + 2.60 + 2.40 + 0.84 = 25.54 \text{ s}.$$

On the tile page (design B), the iced tea is M P B B B B (think, point at the tile, click it twice), the other two items are M P B B each, and *Save* is M P B B:

$$T_B = 4(1.35) + 4(1.10) + 10(0.10) = 5.40 + 4.40 + 1.00 = 10.80 \text{ s}.$$

The tiles save $25.54 - 10.80 = 14.74$ seconds, or 57.7% of the time, on every such order. **Figure 1.4** shows where the time goes: the four M operators are the same in both designs, because the thinking does not change; the saving comes almost entirely from pointing, the operator that design A needs thirteen times.

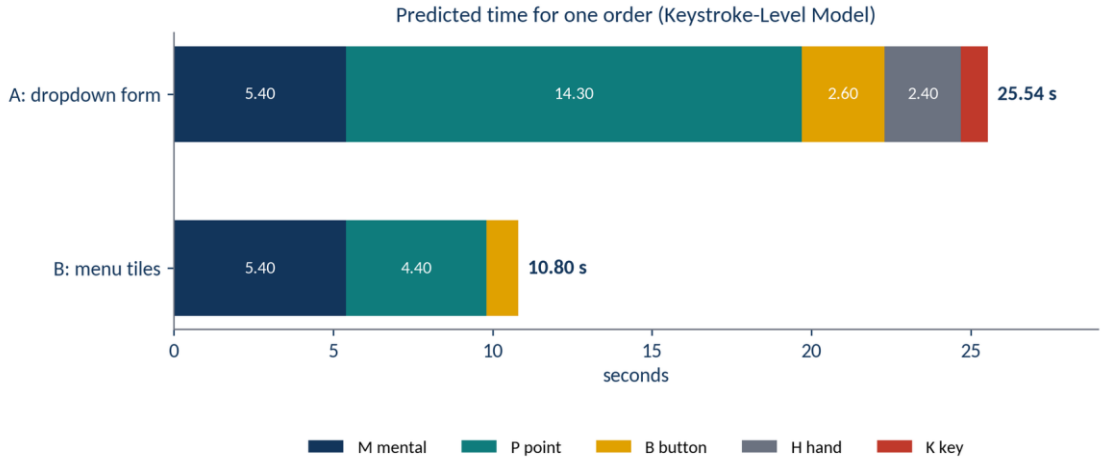


Figure 1.4 The predicted time of one order on the two pages, split by operator: pointing (P) dominates design A.

1.11.3 The program

Counting by hand is fine for one task and tedious for ten. The program in this section reads a small text file that describes each design as a list of operators and prints the counts, the predicted times and what the faster design saves over a working month. **Listing 1.5** is the file for Bu Ratna's order; "2B" is short for "B B", and everything after `#` is a comment for people.

Listing 1.5 `data/orders.txt`: the two designs written as Keystroke-Level Model operators.

```
# The order "2 iced tea, 1 fried rice, 1 geprek chicken" on two designs.
# K key, P point, B mouse button press or release, H hand moves between
# keyboard and mouse, M mental preparation. "2B" means "B B".
A: dropdown form
  M P 2B P 2B P 2B H K H P 2B     # iced tea: item list, quantity, Add
  M P 2B P 2B P 2B H K H P 2B     # fried rice
  M P 2B P 2B P 2B H K H P 2B     # geprek chicken
  M P 2B                          # Save
B: menu tiles
  M P 4B                          # iced tea tile, clicked twice
  M P 2B                          # fried rice tile
  M P 2B                          # geprek chicken tile
  M P 2B                          # Save
```

Listing 1.6 is the Python version.

Listing 1.6 `klm.py`: reads designs as operator lists and predicts their times.

```
"""Keystroke-Level Model: how long does an expert need for a task?"""
import sys

# Operator times in seconds, from Card, Moran and Newell (1980).
TIMES = {"K": 0.28,   # press a key (average typist)
         "P": 1.10,   # point at a target with the mouse
         "B": 0.10,   # press or release the mouse button
         "H": 0.40,   # move a hand between keyboard and mouse
         "M": 1.35}   # prepare mentally for the next step
ORDER = "KPBHM"


def count(ops):
    """Count the operators in a text such as 'M P 2B H K'."""
    counts = dict.fromkeys(ORDER, 0)
    for token in ops.split():
        times, op = int(token[:-1] or 1), token[-1]   # '2B' -> 2, 'B'
        if op not in TIMES:
            raise ValueError(f"unknown operator {token!r}")
        counts[op] += times
    return counts


def seconds(counts):
    """The predicted time: every operator count times its unit time."""
    return sum(counts[op] * TIMES[op] for op in ORDER)


def read_designs(path):
    """Read 'name' lines, each followed by indented lines of operators."""
    designs = {}
    name = None
    with open(path, encoding="utf-8") as f:
        for line in f:
            text = line.split("#")[0].rstrip()
            if not text:
                continue
            if not line[0].isspace():
                name = text
                designs[name] = ""
            else:
                designs[name] += " " + text
    return {n: count(ops) for n, ops in designs.items()}
```

Listing 1.6 (Continued) *klm.py: reads designs as operator lists and predicts their times.*

```
def main():
    path = sys.argv[1] if len(sys.argv) > 1 else "data/orders.txt"
    per_day = int(sys.argv[2]) if len(sys.argv) > 2 else 120
    designs = read_designs(path)
    print("Keystroke-Level Model, times in seconds (Card, Moran and Newell)")
    print(f"{'design':18}" + "".join(f"{op:>4}" for op in ORDER) + "   time")
    for name, counts in designs.items():
        cols = "".join(f"{counts[op]:>4}" for op in ORDER)
        print(f"{name:18}{cols}{seconds(counts):7.2f}")
    ranked = sorted(designs.items(), key=lambda d: seconds(d[1]))
    (fast, b), (_, a) = ranked[0], ranked[-1]        # fastest, slowest
    saved = seconds(a) - seconds(b)
    print(f"{fast[0]} saves {saved:.2f} s per order "
          f"({100 * saved / seconds(a):.1f}%).")
    minutes = saved * per_day / 60
    print(f"At {per_day} orders a day: {minutes:.1f} minutes a day, "
          f"{minutes * 26 / 60:.1f} hours a month (26 days).")


if __name__ == "__main__":
    main()
```

The dictionary `TIMES` holds the unit times of **Table 1.3**. The function `count` splits a text such as `M P 2B` into tokens; the last character of a token is the operator and whatever comes before it is a repeat count, so `token[:-1] or 1` turns `"2B"` into 2 and `"B"` into 1 (an empty string counts as false in Python, and `or` then supplies 1). An unknown letter raises an error rather than being ignored, because a typo that silently removes a step would make a design look faster than it is. The function `seconds` is Equation (1.1) written in one line. The function `read_designs` treats a line that starts at the left margin as the name of a new design and every indented line as more operators for it, which lets a long task be written over several commented lines. Finally `main` prints a table, finds the fastest and the slowest design by sorting on the predicted time, and scales the difference to a day and to a month of 26 working days. Two optional command-line arguments choose another file and another number of orders per day.

Listing 1.7 is the same program in Java.

Listing 1.7 `Klm.java`: the same Keystroke-Level Model calculator in Java.

```java
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.LinkedHashMap;
import java.util.Map;

/** Keystroke-Level Model: how long does an expert need for a task? */
public class Klm {
    static final String ORDER = "KPBHM";
    // Operator times in seconds, from Card, Moran and Newell (1980):
    // key, point, mouse button, hand to keyboard or mouse, mental step.
    static final double[] TIMES = {0.28, 1.10, 0.10, 0.40, 1.35};

    /** Count the operators in a text such as "M P 2B H K". */
    static int[] count(String ops) {
        int[] counts = new int[ORDER.length()];
        for (String token : ops.trim().split("\\s+")) {
            if (token.isEmpty()) continue;
            char op = token.charAt(token.length() - 1);
            String n = token.substring(0, token.length() - 1);
            int i = ORDER.indexOf(op);
            if (i < 0) throw new IllegalArgumentException("unknown " + token);
            counts[i] += n.isEmpty() ? 1 : Integer.parseInt(n);
        }
        return counts;
    }

    /** The predicted time: every operator count times its unit time. */
    static double seconds(int[] counts) {
        double total = 0;
        for (int i = 0; i < counts.length; i++) total += counts[i] * TIMES[i];
        return total;
    }

    /** Read "name" lines, each followed by indented lines of operators. */
    static Map<String, int[]> readDesigns(String path) throws IOException {
        Map<String, String> ops = new LinkedHashMap<>();
        String name = null;
        for (String line : Files.readAllLines(Path.of(path))) {
            String text = line.split("#")[0].stripTrailing();
            if (text.isEmpty()) continue;
            if (!Character.isWhitespace(line.charAt(0))) {
                name = text;
                ops.put(name, "");
            } else {
                ops.put(name, ops.get(name) + " " + text);
```

Listing 1.7 (Continued) *Klm.java: the same Keystroke-Level Model calculator in Java.*

```
            }
        }
        Map<String, int[]> designs = new LinkedHashMap<>();
        ops.forEach((n, o) -> designs.put(n, count(o)));
        return designs;
    }

    public static void main(String[] args) throws IOException {
        String path = args.length > 0 ? args[0] : "data/orders.txt";
        int perDay = args.length > 1 ? Integer.parseInt(args[1]) : 120;
        Map<String, int[]> designs = readDesigns(path);
        System.out.println("Keystroke-Level Model, times in seconds"
                           + " (Card, Moran and Newell)");
        var head = new StringBuilder(String.format("%-18s", "design"));
        for (char op : ORDER.toCharArray()) head.append(String.format("%4s", op));
        System.out.println(head + "   time");
        String slow = null, fast = null;
        for (var e : designs.entrySet()) {
            var row = new StringBuilder(String.format("%-18s", e.getKey()));
            for (int c : e.getValue()) row.append(String.format("%4d", c));
            System.out.println(row + String.format("%7.2f", seconds(e.getValue())));
            double t = seconds(e.getValue());
            if (slow == null || t > seconds(designs.get(slow))) slow = e.getKey();
            if (fast == null || t < seconds(designs.get(fast))) fast = e.getKey();
        }
        double a = seconds(designs.get(slow));
        double saved = a - seconds(designs.get(fast));
        System.out.printf("%s saves %.2f s per order (%.1f%%).%n",
                          fast.charAt(0), saved, 100 * saved / a);
        double minutes = saved * perDay / 60;
        System.out.printf("At %d orders a day: %.1f minutes a day, "
                          + "%.1f hours a month (26 days).%n",
                          perDay, minutes, minutes * 26 / 60);
    }
}
```

The Java version keeps the counts in an `int` array in the order K, P, B, H, M, so that `ORDER.indexOf(op)` gives the position of an operator and `TIMES[i]` its time. `Files.readAllLines` reads the whole file; a `LinkedHashMap` keeps the designs in the order of the file, which a plain `HashMap` would not. Instead of sorting, the loop that prints the rows remembers the slowest and the fastest design as it goes.

1.11.4 Running it

From the `python` folder, and from the `java` folder after compiling, the two programs print the same lines:

```
$ python3 klm.py
$ java -cp out Klm
```

```
Keystroke-Level Model, times in seconds (Card, Moran and Newell)
design               K   P   B   H   M   time
A: dropdown form     3  13  26   6   4  25.54
B: menu tiles        0   4  10   0   4  10.80
B saves 14.74 s per order (57.7%).
At 120 orders a day: 29.5 minutes a day, 12.8 hours a month (26 days).
```

The numbers agree with the hand calculation of Section 1.11.2. On a day with 120 orders of this size the tiles give Bu Ratna back half an hour, and more than twelve working hours a month. With a quieter day of 60 orders, `python3 klm.py data/orders.txt 60` reports 14.7 minutes a day and 6.4 hours a month.

It is worth being honest about what the model does and does not say. The KLM predicts the time of an *expert* who makes no errors; Bu Ratna in her first week is not an expert, and every error (the four iced teas of Section 1.2.2) costs more time than the model sees. It says nothing about learning, satisfaction or whether customers get the right food. It is a first estimate that is cheap enough to make before writing any code, and in this case it confirms what watching Bu Ratna showed: the design, not the user, was slow.

1.11.5 Testing it

A program whose purpose is to settle arguments must itself be trustworthy. **Listing 1.8** checks the counting, one single click, the rejection of an unknown operator and the two totals of Section 1.11.2.

Listing 1.8 `test_klm.py`: tests of the Keystroke-Level Model calculator.

```
"""Tests for the Keystroke-Level Model: python -m unittest test_klm.py"""
import unittest
from klm import count, seconds, read_designs


class KlmTests(unittest.TestCase):
    def test_counting_with_repeats(self):
        c = count("M P 2B H K 3K")
        self.assertEqual((c["M"], c["P"], c["B"], c["H"], c["K"]),
                         (1, 1, 2, 1, 4))

    def test_one_click(self):
        self.assertAlmostEqual(seconds(count("M P 2B")), 2.65)

    def test_unknown_operator(self):
        with self.assertRaises(ValueError):
            count("M X")

    def test_the_two_designs(self):
        d = read_designs("data/orders.txt")
        self.assertAlmostEqual(seconds(d["A: dropdown form"]), 25.54)
        self.assertAlmostEqual(seconds(d["B: menu tiles"]), 10.80)


if __name__ == "__main__":
    unittest.main()
```

Listing 1.9 does the same in Java without any test library, so that it runs on a bare JDK.

Listing 1.9 `KlmTest.java`: the same checks in Java.

```java
import java.util.Map;

/** Tests for the Keystroke-Level Model: java -cp out KlmTest */
public class KlmTest {
    static int passed = 0;

    static void expect(boolean ok, String what) {
        if (!ok) throw new AssertionError(what);
        passed++;
    }

    public static void main(String[] args) throws Exception {
        int[] c = Klm.count("M P 2B H K 3K");          // K P B H M
        expect(c[0] == 4 && c[1] == 1 && c[2] == 2 && c[3] == 1 && c[4] == 1,
               "counting with repeats");
        expect(Math.abs(Klm.seconds(Klm.count("M P 2B")) - 2.65) < 1e-9,
               "one click");
        try {
            Klm.count("M X");
            expect(false, "unknown operator");
        } catch (IllegalArgumentException e) {
            passed++;
        }
        Map<String, int[]> d = Klm.readDesigns("data/orders.txt");
        expect(Math.abs(Klm.seconds(d.get("A: dropdown form")) - 25.54) < 1e-9,
               "design A");
        expect(Math.abs(Klm.seconds(d.get("B: menu tiles")) - 10.80) < 1e-9,
               "design B");
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
$ python3 -m unittest test_klm.py
....
----------------------------------------------------------------------
Ran 4 tests in 0.001s

OK
$ java -cp out KlmTest
All 5 checks passed.
```

1.12 HCI in Practice: Profitina

Profitina is an Indonesian AI Visibility (GEO, generative engine optimization) platform, live at app.profitina.com: it measures how ready a brand's website is to be cited by AI answer engines, explains each signal it measured, and helps fix what it finds [42]. Its users are mostly brand owners and small-business marketers, not engineers, and that single fact makes it an HCI problem before it is a software problem.

Three places in the product show this chapter's ideas at work. The first is the *score*. A number such as "62" means nothing to a shop owner; the dashboard therefore shows each signal with a plain explanation and a weekly trend, so that the gulf of evaluation ("is my site getting better?") can be crossed at a glance. The dashboard is designed as a cockpit: one screen gathers notifications, followed channels, rank and activity, and every item is clickable straight to its section, which shortens the gulf of execution for the most common next step. The second is *consent*. Under UU PDP the service asks for explicit consent and offers data export, email and password change and full account deletion as self-service actions; each is a small interface that must be understood, not merely clicked, which is exactly the standard that Section 1.4.2 described. The third is *feedback timing* in the community feed: when a member posts, the

composer clears at once and the post appears immediately, and if sending fails the text is restored so that nothing typed is lost. That is Norman's feedback principle turned into a feature.

The product also follows a rule that every AI-powered interface in this book will follow: the app advises and the user decides. The Content Engine drafts text with a locally running model, and when no model is available the app says so instead of silently showing a template. Chapter 13 explains why such honesty is a design decision and not an engineering detail. The hands-on case study of this book is not the real product but a teaching example, *Profitina Laundry*, which later chapters use for exercises that need a complete small service.

1.13 Common Mistakes, and How to Avoid Them

1. **Designing for yourself.** The person who built a system knows how it works and cannot see what a newcomer sees. Watch at least one real user before believing that a design is clear (Chapter 5 shows how).
2. **Blaming the user.** "She will get used to it" hides a wide gulf. Ask which gulf is too wide and narrow it.
3. **Saying "usable" without saying for whom.** Usability is always for specified users, goals and contexts of use. Write them down.
4. **Trusting a statistic without asking what it counts.** StatCounter counts page views on tracked websites, not people; a survey counts respondents who answered. Read the method before designing on the number.
5. **Treating the KLM as the truth.** It predicts error-free expert time. Use it to compare designs early, then test with real people.
6. **Placing M operators everywhere, or nowhere.** Both distort the prediction. Put one before each chunk that needs a decision, following Kieras's rules.
7. **Installing from random websites.** Use the official sources and package managers of Section 1.6 to 1.8, and check the result with `check_setup.py` and `CheckSetup.java`.
8. **Opening a single file instead of a folder in VS Code.** The terminal, the Explorer and the Java project all work from the opened folder.

1.14 Chapter Summary

- Human-computer interaction is the discipline of designing, evaluating and implementing interactive systems for human use. Usability (ISO 9241-11) is effectiveness, efficiency and satisfaction for specified users, goals and contexts; user experience (ISO 9241-210) covers perceptions before, during and after use; accessibility extends usability to the widest range of people.
- Norman's gulf of execution (how do I do it?) and gulf of evaluation (did it work?) explain most interface problems; signifiers, mapping and constraints narrow the first, feedback and a clear conceptual model the second.
- The field moved from specialists at command lines to everyone on phones, and in 2025–2026 to Liquid Glass, Material 3 Expressive, spatial computing and AI agents. CHI 2026 accepted 1,702 papers.
- In Indonesia, 81.72% of the population is online (APJII 2026), Android carries 77.07% of mobile page views (StatCounter, August 2026), and 11.97% of the population is sixty or older (BPS, 2025). UU 8/2016, PP 42/2020, UU PDP with PP 33/2026, the European Accessibility Act and WCAG 2.2 make good interfaces a legal matter.

- The workbench is JDK 25 LTS, Python 3.14 and VS Code with the Java and Python extensions, on Windows, macOS or Linux.
- The Keystroke-Level Model predicts expert task time as a sum of operator times (Equation (1.1)); for Bu Ratna's order the tile page takes 10.80 s instead of 25.54 s, 12.8 working hours a month at 120 orders a day.

1.15 Exercises

1. Using Equation (1.1) and **Table 1.3**, compute by hand the KLM time on design A and on design B for an order of *one* item only (one iced tea). How large is the saving in seconds and in percent? Why is the percentage smaller than for the three-item order?
2. A customer orders five different items, one portion each. Write the operator sequence for both designs, compute the times, and check your answer by adding the designs to a copy of `data/orders.txt` and running `klm.py`.
3. Dimas proposes a third design C for Bu Ratna, who types well: every menu item has a two-letter code (ET for iced tea), and an order is typed as code, quantity and Enter per item, with an empty Enter to save. Write design C for the order of Section 1.11.2 in the file format of **Listing 1.5** (remember the hand must first move to the keyboard), run the program and report what it prints. Is C the best design for Bu Ratna? Give one argument that the KLM cannot see.
4. At the puskesmas kiosk of Section 1.1 a seventy-year-old patient must choose one of twelve small, text-only buttons, and after pressing one, the kiosk prints a number without saying which clinic it is for. Name the gulf of execution and the gulf of evaluation in this situation and propose one change for each.
5. Write three measurable usability goals for Bu Ratna's order page, one each for effectiveness, efficiency and satisfaction, in the sense of ISO 9241-11. Each goal must name the user, the task and the number to reach.
6. StatCounter reports that 60.59% of page views in Indonesia in August 2026 came from desktop computers. A classmate concludes: "So we should design Indonesian services for desktops first." Explain in three or four sentences why this conclusion does not follow.
7. Extend `klm.py` and `Klm.java` so that an optional third command-line argument gives the number of working days per month instead of the fixed 26. Show the output for 120 orders a day and 30 working days.

1.16 Solutions to the Exercises

1. Design A for one item: M P B B P B B P B B H K H P B B, then M P B B for *Save*: $n_M = 2$, $n_P = 5$, $n_B = 10$, $n_H = 2$, $n_K = 1$, so $T_A = 2.70 + 5.50 + 1.00 + 0.80 + 0.28 = 10.28$ s. Design B: M P B B for the tile and M P B B for *Save*: $T_B = 2.70 + 2.20 + 0.40 = 5.30$ s. The saving is 4.98 s, or 48.4%. The percentage is smaller because the closing *Save* costs the same M P B B in both designs, and with only one item that shared cost is a larger part of the whole.

2. Design A: five times the per-item chunk plus *Save*: $n_M = 6$, $n_P = 21$, $n_B = 42$, $n_H = 10$, $n_K = 5$; $T_A = 8.10 + 23.10 + 4.20 + 4.00 + 1.40 = 40.80$ s. Design B: five times M P B B plus *Save*: $n_M = 6$, $n_P = 6$, $n_B = 12$; $T_B = 8.10 + 6.60 + 1.20 = 15.90$ s. The saving grows to 24.90 s (61.0%). In the file, write `M P 2B P 2B P 2B H K H P 2B` five times under design A and `M P 2B` five times under design B, each followed by the *Save* line; the program prints the same totals.

3. Design C in the file format:

```
C: keyboard codes
  H                              # hand from mouse to keyboard
  M 2K K K                       # "ET", quantity 2, Enter
  M 2K K K                       # "NG", 1, Enter
  M 2K K K                       # "AG", 1, Enter
  M K                            # Enter on an empty line saves
```

Appended to the file, the program prints:

```
Keystroke-Level Model, times in seconds (Card, Moran and Newell)
design               K   P   B   H   M   time
A: dropdown form     3  13  26   6   4  25.54
B: menu tiles        0   4  10   0   4  10.80
C: keyboard codes   13   0   0   1   4   9.44
C saves 16.10 s per order (63.0%).
At 120 orders a day: 32.2 minutes a day, 14.0 hours a month (26 days).
```

C is the fastest *for an expert who knows all the codes*, 1.36 s faster than the tiles. The KLM cannot see the cost of learning and remembering forty-two codes. Tiles rely on *recognition* (the item is visible), codes on *recall* (the item must be remembered), and Chapter 2 explains why recall is slower and more error-prone for newcomers and under stress. A sensible product offers both: tiles for everyone, codes as a shortcut for the practiced user, which is Nielsen's heuristic "flexibility and efficiency of use" of Chapter 3.

4. Gulf of execution: the patient cannot map the goal "see the dentist" to one of twelve small text labels, perhaps with poor eyesight. Change: fewer, larger buttons with icons and plain words ("Gigi / Dentist"), grouped by frequency, and a staff member or a voice prompt for help. Gulf of evaluation: after pressing, the ticket shows only a number, so the patient cannot check that the right clinic was chosen. Change: print and show the clinic name in large text next to the number and let the patient cancel within a few seconds.

5. For example. Effectiveness: "Bu Ratna enters 95% of three-item orders without a correction, measured over one Saturday lunch." Efficiency: "Bu Ratna, after one week of use, enters a three-item order in at most 15 seconds on average." Satisfaction: "Bu Ratna and her helper score the page at least 70 on the System Usability Scale (Chapter 7) after two weeks." Each goal names the user, the task and the target.

6. The figure counts page views on websites that carry StatCounter's code, not people and not time. A few heavy desktop users at offices can generate many page views, while most Indonesians use the internet on phones inside apps such as WhatsApp and marketplaces, which web counters do not see. APJII's survey of people, and the fact that the people still offline mostly lack devices and digital literacy, point to phones as the primary device. The right conclusion is to design mobile-first and check the actual users of the specific service.

7. In Python, add after `per_day`: `days = int(sys.argv[3]) if len(sys.argv) > 3 else 26` and replace `26` in the last print by `days`, both in the calculation and in the text `({days} days)`. In Java, add `int days = args.length > 2 ? Integer.parseInt(args[2]) : 26;` and use `days` in the same two places. For `python3 klm.py data/orders.txt 120 30` the last line becomes `At 120 orders a day: 29.5 minutes a day, 14.7 hours a month (30 days).`, because $1768.8 \times 30 / 3600 = 14.74$ hours.

References

[1] T. T. Hewett, R. Baecker, S. Card, T. Carey, J. Gasen, M. Mantei, G. Perlman, G. Strong, and W. Verplank, *ACM SIGCHI Curricula for Human-Computer Interaction*. New York, NY, USA: ACM, 1992. [Online]. Available: https://dl.acm.org/doi/book/10.1145/2594128

[2] International Organization for Standardization, *ISO 9241-11:2018 Ergonomics of human-system interaction — Part 11: Usability: Definitions and concepts*. Geneva, Switzerland: ISO, 2018. [Online]. Available: https://www.iso.org/standard/63500.html

[3] International Organization for Standardization, *ISO 9241-210:2019 Ergonomics of human-system interaction — Part 210: Human-centred design for interactive systems*. Geneva, Switzerland: ISO, 2019. [Online]. Available: https://www.iso.org/standard/77520.html

[4] D. A. Norman, *The Design of Everyday Things*, revised and expanded ed. New York, NY, USA: Basic Books, 2013.

[5] J. Grudin, *From Tool to Partner: The Evolution of Human-Computer Interaction*. San Rafael, CA, USA: Morgan & Claypool, 2017.

[6] Apple Inc., "Apple introduces a delightful and elegant new software design," Apple Newsroom, Jun. 9, 2025. [Online]. Available: https://www.apple.com/newsroom/2025/06/apple-introduces-a-delightful-and-elegant-new-software-design/

[7] Google Design, "Expressive Design: Google's UX Research," accessed Sep. 2026. [Online]. Available: https://design.google/library/expressive-material-design-google-research

[8] Microsoft, "Fluent 2 Design System," accessed Sep. 2026. [Online]. Available: https://fluent2.microsoft.design/

[9] Apple Inc., "Apple Vision Pro available in the U.S. on February 2," Apple Newsroom, Jan. 8, 2024. [Online]. Available: https://www.apple.com/newsroom/2024/01/apple-vision-pro-available-in-the-us-on-february-2/

[10] TechCrunch, "ChatGPT reaches 900M weekly active users," Feb. 27, 2026. [Online]. Available: https://techcrunch.com/2026/02/27/chatgpt-reaches-900m-weekly-active-users

[11] S. Amershi, D. Weld, M. Vorvoreanu, A. Fourney, B. Nushi, P. Collisson, J. Suh, S. Iqbal, P. N. Bennett, K. Inkpen, J. Teevan, R. Kikin-Gil, and E. Horvitz, "Guidelines for Human-AI Interaction," in *Proc. CHI Conference on Human Factors in Computing Systems (CHI '19)*, Glasgow, UK, 2019, paper 3. doi: 10.1145/3290605.3300233

[12] Interactive Media Lab Dresden, "ACM CHI 2026 in Barcelona," Apr. 2026. [Online]. Available: https://imld.de/en/news/2026/04/acm-chi-2026-in-barcelona/

[13] Figma, "What's new from Config 2026," Figma Help Center, Jun. 2026. [Online]. Available: https://help.figma.com/hc/en-us/articles/39582753756695-What-s-new-from-Config-2026

[14] detikInet, "Pengguna Internet Indonesia Tembus 235 Juta, Penetrasi 81,72%," May 2026, reporting Asosiasi Penyelenggara Jasa Internet Indonesia (APJII), *Survei Penetrasi Internet Indonesia 2026*. [Online]. Available: https://inet.detik.com/cyberlife/d-8499661/pengguna-internet-indonesia-tembus-235-juta-penetrasi-81-72

[15] StatCounter Global Stats, "Mobile Operating System Market Share Indonesia," August 2026. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/indonesia

[16] StatCounter Global Stats, "Desktop vs Mobile vs Tablet Market Share Indonesia," August 2026. [Online]. Available: https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/indonesia

[17] Periskop.id, "Jumlah Lansia Indonesia 2025 11,97%, Indonesia Masuk Aging Population," May 5, 2026, reporting Badan Pusat Statistik, *Survei Penduduk Antar Sensus (SUPAS) 2025*. [Online]. Available: https://periskop.id/kependudukan/20260505/jumlah-lansia-indonesia-2025-1197-indonesia-masuk-aging-population

[18] Republik Indonesia, *Undang-Undang Nomor 8 Tahun 2016 tentang Penyandang Disabilitas*. JDIH BPK RI. [Online]. Available: https://peraturan.bpk.go.id/Details/37251/uu-no-8-tahun-2016

[19] Republik Indonesia, *Peraturan Pemerintah Nomor 42 Tahun 2020 tentang Aksesibilitas terhadap Permukiman, Pelayanan Publik, dan Pelindungan dari Bencana bagi Penyandang Disabilitas*. JDIH BPK RI. [Online]. Available: https://peraturan.bpk.go.id/Details/142301/pp-no-42-tahun-2020

[20] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. JDIH BPK RI. [Online]. Available: https://peraturan.bpk.go.id/Details/229798/uu-no-27-tahun-2022

[21] Veritask, “Pengaturan Teknis Pelindungan Data Pribadi dan Kewajiban Pengendali serta Prosesor Akhirnya Terbit Lewat PP 33/2026,” 2026, on *Peraturan Pemerintah Nomor 33 Tahun 2026 tentang Peraturan Pelaksanaan Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. [Online]. Available: https://veritask.ai/id/artikel/pengaturan-teknis-pelindungan-data-pribadi-dan-kewajiban-pengendali-serta-prosesor-akhirnya-terbit-lewat-pp-33-2026

[22] European Parliament and Council of the European Union, “Directive (EU) 2019/882 on the accessibility requirements for products and services (European Accessibility Act),” *Official Journal of the European Union*, L 151, Jun. 7, 2019, applicable from Jun. 28, 2025. [Online]. Available: https://eur-lex.europa.eu/eli/dir/2019/882/oj

[23] World Wide Web Consortium, *Web Content Accessibility Guidelines (WCAG) 2.2*, W3C Recommendation. [Online]. Available: https://www.w3.org/TR/WCAG22/

[24] W3C Web Accessibility Initiative, “For Review: WCAG 3 Working Draft – September 2026,” Sep. 10, 2026, and “WCAG 3 Introduction.” [Online]. Available: https://www.w3.org/WAI/news/2026-09-10/wcag3/

[25] Oracle, “Java Downloads: JDK 25 (LTS) and JDK 27,” accessed Sep. 2026. [Online]. Available: https://www.oracle.com/java/technologies/downloads/

[26] OpenJDK, “JDK 25,” General Availability Sep. 16, 2025. [Online]. Available: https://openjdk.org/projects/jdk/25/

[27] Python Software Foundation, “Python 3.14.7,” Aug. 5, 2026. [Online]. Available: https://www.python.org/downloads/release/python-3147/

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

[29] Microsoft, “Java in Visual Studio Code,” *VS Code Documentation*. [Online]. Available: https://code.visualstudio.com/docs/languages/java

[30] Microsoft, “Python in Visual Studio Code,” *VS Code Documentation*. [Online]. Available: https://code.visualstudio.com/docs/languages/python

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

[32] 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

[33] Python Software Foundation, “What’s New in Python 3.14,” *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3.14/whatsnew/3.14.html

[34] OpenJDK, “JEP 330: Launch Single-File Source-Code Programs,” JDK 11. [Online]. Available: https://openjdk.org/jeps/330

[35] Python Software Foundation, “Using Python on Windows,” *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3.14/using/windows.html

[36] Phoronix, “Ubuntu 26.04 LTS Moving To OpenJDK 25 By Default,” 2026. [Online]. Available: https://www.phoronix.com/news/Ubuntu-26.04-OpenJDK-25

[37] Python Software Foundation, “venv — Creation of virtual environments,” *The Python Standard Library*. [Online]. Available: https://docs.python.org/3/library/venv.html

[38] OpenJDK, “JEP 512: Compact Source Files and Instance Main Methods,” JDK 25. [Online]. Available: https://openjdk.org/jeps/512

[39] S. K. Card, T. P. Moran, and A. Newell, “The Keystroke-Level Model for User Performance Time with Interactive Systems,” *Communications of the ACM*, vol. 23, no. 7, pp. 396–410, 1980. doi: 10.1145/358886.358895

[40] S. K. Card, T. P. Moran, and A. Newell, *The Psychology of Human-Computer Interaction*. Hillsdale, NJ, USA: Lawrence Erlbaum, 1983.

[41] D. Kieras, “Using the Keystroke-Level Model to Estimate Execution Times,” tutorial notes, University of Michigan, Ann Arbor, MI, USA, 2001.

[42] Profitina, “Profitina — AI Visibility (GEO) platform,” accessed Sep. 2026. [Online]. Available: https://profitina.com

Chapter 2

Human Factors: Perception, Cognition, Memory, Fitts's and Hick's Laws

An interface is used by a body and a mind with limits that do not change when the software is updated: eyes that see sharply only in a small spot, attention that follows one thing at a time, a working memory that holds a handful of items for a few seconds, and hands that need time to reach a target. This chapter explains those limits in plain words, gives the evidence behind them, and turns two of the most reliable results of experimental psychology, Fitts's law and Hick's law, into a calculator. By the end of the chapter you will be able to describe the human as an information processor with perceptual, cognitive and motor parts; apply the Gestalt principles, the limits of attention and the limits of working memory to a screen; explain how vision, memory and movement change with age; compute the index of difficulty and the movement time of a target with Fitts's law and the decision time of a menu with Hick's law; fit both laws to your own measurements with least squares; and use the result to compare two layouts of a puskesmas queue kiosk in Java and in Python.

2.1 Bu Siti at the Puskesmas Kiosk

Puskesmas Kenanga, a community health center in a suburb of Sidoarjo, installed a touchscreen queue kiosk in the waiting hall in 2025. A patient touches the name of the clinic they need, and the kiosk prints a numbered ticket. On the first screen are twelve buttons, one per service, arranged in a neat grid of three columns and four rows at the top of the screen, each labeled in small letters: General, Dental, Mother and child, Elderly, Children, Laboratory, Tuberculosis, Nutrition, Immunization, Mental health, Sanitation, Pharmacy.

Bu Siti is sixty-eight. She has come for her monthly blood-pressure check at the elderly clinic, as she has for two years. She stands in front of the kiosk for a long time, leans forward, then backward, because the letters are too small for her distance glasses and too far for her reading glasses. She finds *Elderly* in the second row, reaches up, and touches the button next to it, *Children*, because her finger covers the label she was aiming at and the buttons are close together. The ticket says "A-017, Children". She does not notice until the nurse calls her into the wrong room forty minutes later.

The staff's first explanation is that elderly patients "are not used to technology". The kiosk's supplier offers a training video. But the head of the puskesmas, Dr. Rina, asks a different question: what exactly, in the body and mind of a person like Bu Siti, did the kiosk ask too much of? Her eyes had to read small text at an awkward distance. Her attention had to scan twelve labels that all looked alike. Her memory had to hold the word she was looking for while reading the others. Her hand had to hit a target twenty millimeters tall at arm's length. And the kiosk gave her no second chance to notice the mistake.

Each of these has been measured by psychologists for decades, and several obey laws precise enough to calculate with. This chapter follows Dr. Rina's question through perception, attention,

memory and movement, and ends by computing, for the old kiosk and for a redesigned one, how long a patient needs to make a choice and reach the button.

Bu Siti is not an exception. According to Statistics Indonesia, 11.97% of Indonesians were sixty or older in 2025 [1], and the elderly clinic of a puskesmas exists precisely because they are among its most frequent visitors. A kiosk that fails them fails its most regular users.

2.2 The Idea in Plain Words: a Person as an Information Processor

To reason about what a screen asks of a person, HCI borrows a simple picture from cognitive psychology: the person as a processor of information with three stages. Card, Moran and Newell drew it in 1983 as the *Model Human Processor* [2]; **Figure 2.1** shows it. Information enters through the senses and is held very briefly in sensory stores. The *perceptual processor* turns it into something recognized, such as "a button labeled Elderly". The *cognitive processor* decides what to do, using a small, fast *working memory* and a large, slow *long-term memory*. The *motor processor* moves the hand, the eyes or the voice.

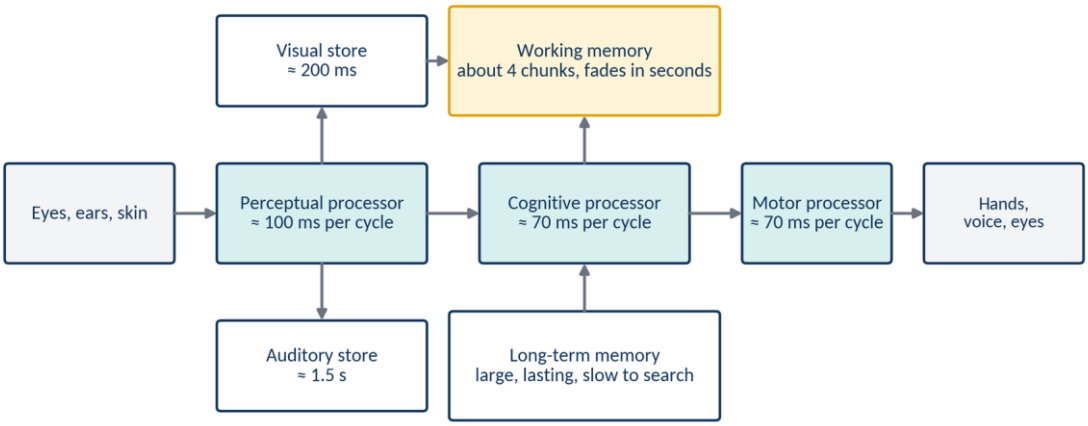


Figure 2.1 The Model Human Processor: perceptual, cognitive and motor processors with typical cycle times, the brief sensory stores and the two memories.

The cycle times in the figure are typical values from Card, Moran and Newell; each has a wide range across people and tasks (the perceptual cycle, for example, from about 50 to 200 ms) [2]. Two consequences are worth remembering. First, even the simplest reaction takes a few hundred milliseconds, because it passes through all three processors at least once; an interface that responds to a touch within about a tenth of a second feels instantaneous, and one that takes a second feels slow, because the person's own loop is far faster than that. Second, the stages have *capacities*: each can be overloaded, and when one is, the others wait. Most bad interfaces overload one stage while the others idle.

The model is a simplification, and modern textbooks on engineering psychology describe richer pictures with parallel resources and feedback loops [3]. For a designer, the three stages remain a good checklist: *Can they see it? Can they decide? Can they reach it?*

2.3 Perception: What Reaches the Mind

2.3.1 Vision

The eye sees sharply only in the *fovea*, a small area at the center of the retina covering about two degrees of the visual field, roughly the width of a thumbnail at arm's length. Everything else is seen in *peripheral vision*, which is good at detecting movement and large shapes but poor at reading. We have the impression of seeing a whole screen at once because the eyes jump several times a second from one point to another, in movements called *saccades*, and the brain assembles the result [3]. For design this means that text is read only where the eyes land, and that a message placed far from where the person is looking, such as an error at the top of a long form while the user is at the bottom, is likely to be missed.

Vision changes with age in ways that matter for every screen. From the forties onward the lens of the eye hardens and can no longer focus on near objects, a condition called *presbyopia*, which is why Bu Siti owns reading glasses. The lens also yellows and lets less light through, so older eyes need more light and more contrast, and are more easily dazzled by glare [4]. Small text, low contrast, thin fonts and glossy screens in bright waiting halls all hit older users first. Czaja and her colleagues' guide to designing for older adults collects these findings into practical rules, such as generous text size, strong contrast and large targets [4].

Color vision differs between people. About 1 in 12 men and 1 in 200 women have some form of color vision deficiency, most often a reduced ability to tell red from green [5]. A queue display that shows called numbers in green and waiting numbers in red is therefore unreadable for a noticeable share of patients. The rule is simple: never let color be the *only* carrier of a meaning; add a word, a shape or a position. WCAG 2.2 makes this a success criterion (1.4.1, "Use of Color") [6], and Chapter 11 adds the contrast ratio that can be computed for any pair of colors.

2.3.2 Grouping: the Gestalt principles

Before we read a single label, perception has already grouped what we see. Early in the twentieth century the Gestalt psychologists described the rules of this grouping, and a century of later research has confirmed and refined them [7]. **Figure 2.2** shows four that matter most for screens.

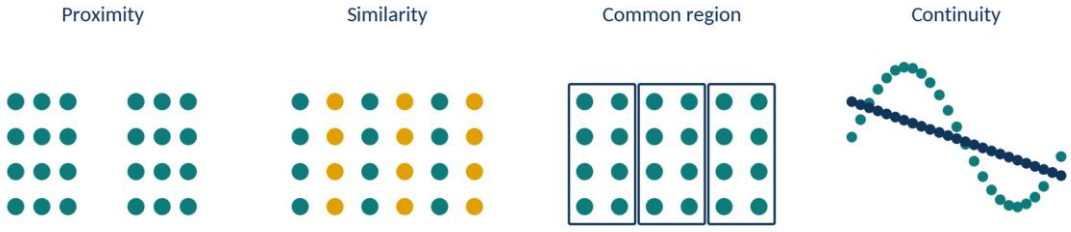


Figure 2.2 Four Gestalt principles: proximity (two groups by distance), similarity (columns by color), common region (pairs by enclosure) and continuity (two smooth lines rather than a zigzag).

Proximity: items close together are seen as a group, so the twelve dots on the left of the figure form two groups of six because of one wider gap. *Similarity*: items that look alike are grouped, so the second panel reads as columns of teal and amber. *Common region*: items inside the same border belong together, however they are colored or spaced. *Continuity*: the eye follows smooth lines, so the last panel shows a wave and a straight line crossing, not two zigzags meeting.

The old kiosk violates all four in the same way: twelve identical buttons at identical distances say, perceptually, that all twelve services are the same kind of thing and equally important. A layout

that separates the six most common services from the rest by space, size or a border lets perception do the first half of the search before the patient reads a word. The same principles explain why a form whose labels sit closer to the field *below* them than to the field they belong to causes wrong entries: proximity beats intention.

2.3.3 What stands out: preattentive features

Some visual differences are noticed almost instantly, whatever the number of other items: a single red item among grey ones, a large item among small ones, a moving item among still ones. Treisman and Gelade's feature-integration theory explains why: simple features such as color, size, orientation and motion are registered in parallel across the whole visual field, while *combinations* of features must be checked item by item [8]. For a designer the lesson is that one highlighted element (the main action, an error) can be found at once, but a highlight that is used everywhere highlights nothing. Material 3 Expressive, Google's 2025 revision of its design language, relies on exactly this effect: Google reports that people found key elements up to four times faster in expressive layouts that use size, shape and color to make the main action stand out, and that the performance gap between older and younger participants narrowed [9].

2.3.4 Hearing and touch

Hearing is the channel for warnings and for people who cannot see a screen; its sensory store holds a sound for a second or two, longer than vision, which is why a spoken ticket number can still be recalled a moment after it was said [2]. It is also the channel that fails in a noisy waiting hall and for the many older people with age-related hearing loss, so a called number must be both shown and announced. Touch carries the feedback of a physical button, which a flat touchscreen lacks: a short vibration or a visible press state replaces the click that fingers used to feel.

2.4 Attention: One Thing at a Time

Attention is the selection of what gets processed further. It can be directed on purpose (looking for *Elderly*) or captured by the world (a flashing advertisement on the kiosk's screen). It is limited: we can focus on one demanding task at a time, and switching between tasks costs time and increases errors [3].

The most striking demonstration of its limits is *inattentional blindness*. In a famous experiment by Simons and Chabris, viewers counting basketball passes in a video very often failed to notice a person in a gorilla costume who walked through the scene and beat their chest [10]. What people do not attend to, they often do not see at all, even when it is in plain view. In interfaces this explains why a warning banner next to a task is missed while the user concentrates on the task, and why a confirmation message that appears where the user is not looking does not confirm anything. Bu Siti, concentrating on finding a word, did not read the ticket that said *Children*.

Three design rules follow. Put feedback where the attention already is: next to the button that was pressed, or in the thing the user is looking at. Do not compete for attention without reason: every animation, badge and banner draws on the same small budget. And when something really needs attention, such as a wrong choice that will cost an hour of waiting, interrupt deliberately and ask, because a passive message will be missed.

2.5 Memory: What Stays, and for How Long

2.5.1 Working memory

Working memory is where we hold what we are thinking about right now: the name of the clinic we are looking for, the four digits of a one-time password, the item we have just decided to order. In 1956 George Miller famously suggested that its span is "seven, plus or minus two" items [11]. Later research showed that Miller's number was inflated by people's ability to group items, and that the real limit is closer to about four meaningful units, or *chunks* [12]. Information in working memory also fades within seconds unless it is rehearsed, and it is easily displaced by new information.

Chunking is the way around the limit. The digits 081234567890 are twelve items, far too many; written as 0812-3456-7890 they become three chunks, and "0812" is recognized as a familiar mobile prefix, which makes it one chunk that costs almost nothing. This is why Indonesian phone numbers, bank account numbers and virtual-account numbers for payments should always be shown grouped, and why a one-time password of six digits is easier to copy as "482 913" than as "482913".

2.5.2 Recognition rather than recall

Long-term memory is large and lasting, but retrieving something from it without a cue, *recall*, is slow and error-prone. Recognizing something that is shown, *recognition*, is much easier: most people cannot list the twelve services of their puskesmas, but they recognize *Elderly* immediately when they see it. Norman describes this as the difference between *knowledge in the head* and *knowledge in the world*: a good interface puts the knowledge the user needs into the world, on the screen, so that it does not have to be in the head [13]. Menus, visible options, recently used items and autocomplete all turn recall into recognition. The keyboard codes of Exercise 1.3 are the opposite: fast for an expert, but they demand recall. Chapter 3 meets this idea again as Nielsen's heuristic "recognition rather than recall".

2.5.3 Mental models

People also carry *mental models*: their own simplified explanation of how a system works. A patient who thinks the kiosk "calls the doctor" may press a second button to be sure, expecting to cancel the first; a system that instead issues a second ticket clashes with the model. Designers cannot install a mental model directly, but they shape it through what the interface shows. A visible queue ("You are number 17 for the Elderly clinic; 6 people before you") builds a correct model; a bare ticket number builds a vague one.

> **KEY INSIGHT:** Design for about four chunks, not seven. If a task requires the user to carry more than a few items from one screen to the next, the design should carry them instead: show them, group them, or fill them in.

2.6 Fitts's Law: How Long It Takes to Reach a Target

2.6.1 The law

In 1954 the psychologist Paul Fitts measured how long people needed to tap alternately between two metal plates of various widths placed at various distances [14]. He found that the time depends neither on the distance alone nor on the width alone, but on their *ratio*. A target twice as

far away is just as quick to hit as before if it is also twice as wide. Scott MacKenzie later adapted the law to computer pointing in the form now used in HCI and in the international standard for evaluating pointing devices, ISO 9241-411 [15], [16].

> **DEFINITION.** *Fitts's law* states that the time to move to a target of width W at distance D from the starting point (both in the same unit) grows linearly with the target's *index of difficulty* ID, which depends only on the ratio D/W. The two quantities are defined by Equation (2.1) and Equation (2.2).

The index of difficulty is

$$ID = \log_2\left(\frac{D}{W} + 1\right) \tag{2.1}$$

measured in bits, and the *movement time* is

$$MT = a + b \cdot ID \tag{2.2}$$

where a (seconds) and b (seconds per bit) are constants that depend on the person, the device and the posture, and are found by measurement.

In words, Equation (2.1) says that difficulty grows with the logarithm of how many target widths the hand must travel. The logarithm is what makes the law so useful: doubling the distance does not double the time, it adds a fixed amount b, because $\log_2(2x) = \log_2 x + 1$ when D/W is large. The constant a is the time needed even for a target that is very close, for starting and stopping. Equation (2.2) is a straight line in ID, which is why the two constants can be found by fitting a line to measured points, as Section 2.8 does.

The reciprocal of the slope is the *throughput*, the rate at which the person-and-device system transmits information:

$$TP = \frac{1}{b} \tag{2.3}$$

in bits per second. A higher throughput means a faster pointing system. (ISO 9241-411 defines throughput more carefully, as the mean of ID_e/MT with an *effective* width computed from the actual spread of taps [16], [17]; Equation (2.3) is the simple version, which is enough for comparing layouts.)

2.6.2 A worked example

Take the practice constants that Section 2.9 obtains from a log of taps: $a = 0.156$ s and $b = 0.147$ s/bit. On the old kiosk, the *General* button is 60 mm wide and 20 mm tall with its center at (45,225) mm, measured from the bottom-left corner of the screen, and the patient's hand rests on the ledge below the screen at $(110, -40)$ mm. The distance is

$$D = \sqrt{(45-110)^2 + (225-(-40))^2} = \sqrt{4225 + 70225} = \sqrt{74450} \\ = 272.9 \text{ mm}.$$

For W we take the *smaller* side of the button, 20 mm, because the finger approaches from below and must stop within the button's height; this is a common and slightly pessimistic simplification for rectangular targets [15]. Then

$$ID = \log_2\left(\frac{272.9}{20} + 1\right) = \log_2 14.645 = 3.872 \text{ bits},$$

$$MT = 0.156 + 0.147 \times 3.872 = 0.725 \text{ s}.$$

On the redesigned kiosk the *General* button is 95 by 60 mm with its center at (57.5,200): $D = 245.7$ mm, $W = 60$ mm, $ID = \log_2(245.7/60 + 1) = 2.349$ bits and $MT = 0.156 + 0.147 \times 2.349 = 0.501$ s. The larger button saves about a quarter of a second of movement, and, more important for Bu Siti, it is three times taller, so a finger that lands a little off still lands on it.

2.6.3 What the law means for design

Fitts's law turns into five everyday rules. *Make frequent targets big. Put frequent targets close to where the hand or pointer already is. Use the edges and corners of a screen with a mouse*, because the pointer stops there and the target is effectively infinitely deep. *Keep dangerous actions small and far from frequent ones*, the reverse use of the same law. And *do not make targets smaller than a person can hit reliably*. Platform guidelines and WCAG 2.2 turn the last rule into numbers, collected in **Table 2.1**.

Table 2.1 Minimum sizes for touch targets in current guidelines.

Source	Minimum target size	Note
WCAG 2.2, criterion 2.5.8 Target Size (Minimum), level AA	24 by 24 CSS pixels	or enough spacing around a smaller target [6]
WCAG 2.2, criterion 2.5.5 Target Size (Enhanced), level AAA	44 by 44 CSS pixels	[6]
Apple Human Interface Guidelines	44 by 44 points	for controls on iPhone and iPad [18]
Android and Material Design	48 by 48 density-independent pixels	about 9 mm on the screen [19]

On a public kiosk used by older people standing at arm's length, these minimums are floors, not goals: the old kiosk's buttons, at 20 mm, were larger than all of them and still too small for Bu Siti.

2.7 Hick's Law: How Long It Takes to Decide

2.7.1 The law

Before the hand moves, the mind must choose. In 1952 William Hick measured how long people needed to react to one of n equally likely lamps by pressing the matching key, and found that the time grows with the logarithm of the number of choices; Ray Hyman showed the next year that what really matters is the *information* in the choice, which is smaller when some options are more likely than others [20], [21]. The *Hick–Hyman law* is

$$RT = a + b\log_2(n + 1) \tag{2.4}$$

for n equally likely options, where the $+1$ accounts for the possibility of not responding at all, and a and b are again constants found by measurement. When the options have probabilities $p_1, p_2, \ldots, p_n$, the number of bits becomes the *entropy*

$$H = \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i} \tag{2.5}$$

and $RT = a + bH$ [21]. For the practice constants of Section 2.9, $a = 0.227$ s and $b = 0.205$ s/bit, a choice among the twelve buttons of the old kiosk takes $0.227 + 0.205 \log_2 13 = 0.227 + 0.205 \times 3.700 = 0.986$ s, and a choice among the seven buttons of the new first screen $0.227 + 0.205 \times 3 = 0.842$ s.

2.7.2 When the law applies, and when it does not

Hick's law describes a person who *knows* the options and their positions and only has to pick one, like a practiced cashier or a patient who comes every month. It does not describe a newcomer *searching* an unfamiliar list of text labels. Such a search is closer to reading the labels one by one, and its time grows roughly in proportion to the number of labels, not with their logarithm [3]. This is why the law must not be used to justify long menus ("the logarithm grows slowly, so forty items are fine"): for first-time users, forty unfamiliar items are slow however they are counted.

The practical lessons are these. Fewer options mean faster decisions, so remove options that almost nobody needs from the first screen. Order and group the rest so that searching becomes recognizing: frequent items first, related items together (Gestalt again). And prefer *broad and shallow* menus over *narrow and deep* ones when every level costs a new decision and a new movement: Exercise 2.2 computes why.

2.8 Building the Calculator Step by Step

Dr. Rina's team now has a proposal: six large buttons for the six services that 89% of patients need, and a wide *Other services* button that leads to a second screen with the remaining six. **Figure 2.3** shows both layouts to scale, with the share of patients for each service taken from the puskesmas's own counts of the previous month.

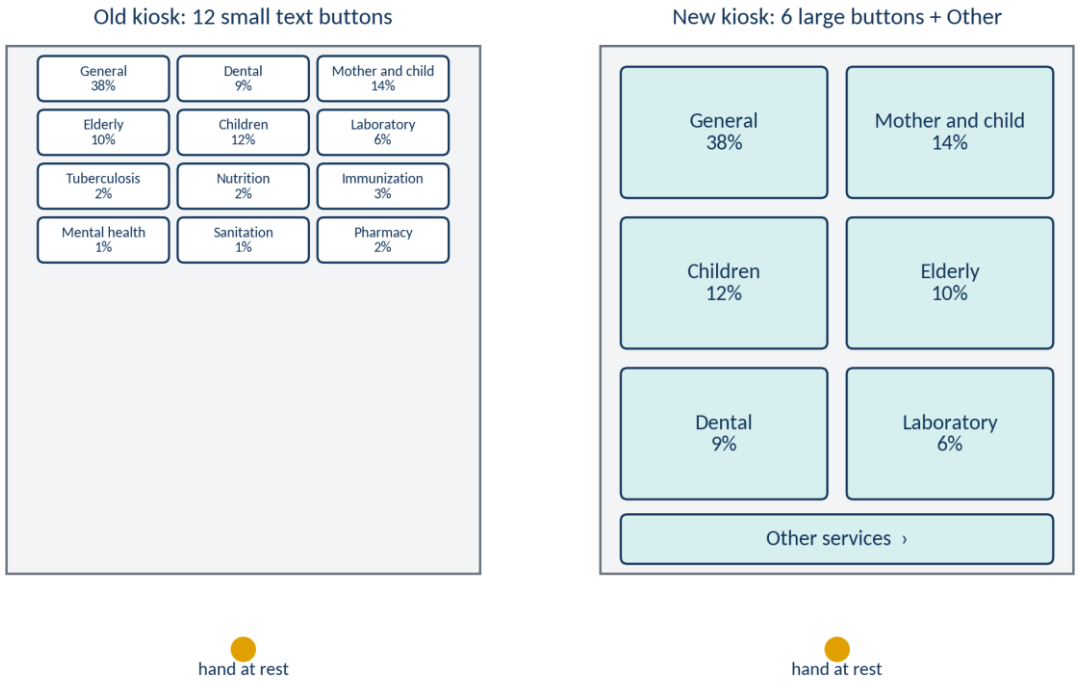


Figure 2.3 The old and the new first screen of the queue kiosk, to scale, with each service's share of patients and the resting point of the hand.

Is the proposal faster, for whom, and by how much? The program of this section answers with the two laws. It does three things: it fits Fitts's law to a log of taps and Hick's law to a log of choices, to obtain the constants a and b; it reads each layout from a small table; and it computes, for every service, the expected time from a ready hand to the pressed button, and the average over all patients weighted by the shares.

2.8.1 The data files

The layouts are CSV files. **Listing 2.1** is the old kiosk: every button with its screen, label, position and size in millimeters (the position is the bottom-left corner), and the share of patients.

Listing 2.1 `data/kiosk_old.csv`: the old layout, twelve buttons on one screen.

```
screen,label,x_mm,y_mm,w_mm,h_mm,share
1,General,15,215,60,20,0.38
1,Dental,80,215,60,20,0.09
1,Mother and child,145,215,60,20,0.14
1,Elderly,15,190,60,20,0.10
1,Children,80,190,60,20,0.12
1,Laboratory,145,190,60,20,0.06
1,Tuberculosis,15,165,60,20,0.02
1,Nutrition,80,165,60,20,0.02
1,Immunization,145,165,60,20,0.03
1,Mental health,15,140,60,20,0.01
1,Sanitation,80,140,60,20,0.01
1,Pharmacy,145,140,60,20,0.02
```

Listing 2.2 is the new layout. The button labeled `Other` has no share of its own; it leads to screen 2.

Listing 2.2 `data/kiosk_new.csv`: the new layout, six large buttons and *Other* on screen 1, six more on screen 2.

```
screen,label,x_mm,y_mm,w_mm,h_mm,share
1,General,10,170,95,60,0.38
1,Mother and child,115,170,95,60,0.14
1,Children,10,100,95,60,0.12
1,Elderly,115,100,95,60,0.10
1,Dental,10,30,95,60,0.09
1,Laboratory,115,30,95,60,0.06
1,Other,10,0,200,22,
2,Immunization,10,170,95,60,0.03
2,Tuberculosis,115,170,95,60,0.02
2,Nutrition,10,100,95,60,0.02
2,Pharmacy,115,100,95,60,0.02
2,Mental health,10,30,95,60,0.01
2,Sanitation,115,30,95,60,0.01
```

The constants a and b must come from measurements of real people with a real device, and ideally from the patients themselves. The chapter's folder contains two *practice* logs, `taps.csv` (60 taps at four distances and three widths) and `choices.csv` (48 choices among 1 to 12 options). They are not measurements: they were generated by the small script `make_practice_logs.py` from the two laws with chosen constants plus random noise, so that the fitting has something realistic to recover, and they stand in for the logs you record yourself in Exercise 2.6. The first lines of the tap log look like this:

```
distance_mm,width_mm,seconds
40,8,0.577
40,8,0.574
```

2.8.2 Fitting a line with least squares

Both laws have the form $y = a + bx$: for Fitts's law x is ID and y is MT; for Hick's law x is $\log_2(n+1)$ and y is RT. Given measured points $(x_1, y_1), \dots, (x_n, y_n)$, the *least-squares* line is the one that makes the sum of squared vertical distances from the points to the line as small as possible. Setting the derivatives of that sum with respect to a and b to zero gives the familiar formulas

$$b = \frac{\sum_i (x_i - \bar{x})\,(y_i - \bar{y})}{\sum_i (x_i - \bar{x})^2}, \qquad a = \bar{y} - b\,\bar{x} \tag{2.6}$$

where $\bar{x}$ and $\bar{y}$ are the means. How well the line fits is measured by

$$r^2 = 1 - \frac{\sum_i (y_i - a - bx_i)^2}{\sum_i (y_i - \bar{y})^2} \tag{2.7}$$

the share of the variation in y that the line explains; 1 is a perfect fit and 0 means the line explains nothing.

2.8.3 The expected time of a layout

For a service on the first screen, the time is a decision among all buttons of screen 1 (Equation (2.4)) plus a movement from the resting hand to the button (Equation (2.2)). For a service on the second screen, it is the decision and movement to *Other*, then a decision among the buttons of screen 2 and a movement from *Other* to the service. The expected time per patient is the sum over services of time multiplied by share. This deliberately ignores the time the kiosk takes to change screens and the time to read a first-time layout, both of which Section 2.9 discusses.

2.8.4 The program

Listing 2.3 is the Python version.

Listing 2.3 `humanfactors.py`: fits Fitts's and Hick's laws to the logs and computes the expected time of each kiosk layout.

```
"""Fitts's and Hick's laws: fit them to logs, then compare two kiosk layouts."""
import csv
import math
import sys

HOME = (110.0, -40.0)      # the hand rests on the ledge below the screen (mm)


def fitts_id(distance, width):
    """Index of difficulty in bits (Shannon form)."""
    return math.log2(distance / width + 1)


def hick_bits(choices):
    """Bits of a choice among n equally likely options (plus 'none')."""
    return math.log2(choices + 1)


def fit_line(xs, ys):
    """Least squares: the a and b of y = a + b*x, and r squared."""
    n = len(xs)
    mx, my = sum(xs) / n, sum(ys) / n
    sxy = sum((x - mx) * (y - my) for x, y in zip(xs, ys))
    sxx = sum((x - mx) ** 2 for x in xs)
    b = sxy / sxx
    a = my - b * mx
    ss_res = sum((y - a - b * x) ** 2 for x, y in zip(xs, ys))
    ss_tot = sum((y - my) ** 2 for y in ys)
    return a, b, 1 - ss_res / ss_tot


def read_rows(path):
    with open(path, newline="", encoding="utf-8") as f:
        return list(csv.DictReader(f))


def centre(b):
    return (float(b["x_mm"]) + float(b["w_mm"]) / 2,
            float(b["y_mm"]) + float(b["h_mm"]) / 2)
```

Listing 2.3 (Continued) *humanfactors.py: fits Fitts's and Hick's laws to the logs and computes the expected time of each kiosk layout.*

```
def move_time(fitts, start, button):
    """Fitts time from a point to a button; the width is its smaller side."""
    cx, cy = centre(button)
    d = math.hypot(cx - start[0], cy - start[1])
    w = min(float(button["w_mm"]), float(button["h_mm"]))
    return fitts[0] + fitts[1] * fitts_id(d, w)


def service_times(layout, fitts, hick):
    """Seconds from the ready hand to the pressed service, per service."""
    screen1 = [b for b in layout if b["screen"] == "1"]
    screen2 = [b for b in layout if b["screen"] == "2"]
    other = next((b for b in screen1 if b["label"] == "Other"), None)
    think1 = hick[0] + hick[1] * hick_bits(len(screen1))
    times = {}
    for b in screen1:
        if b is not other:
            times[b["label"]] = (think1 + move_time(fitts, HOME, b),
                                 float(b["share"]))
    if other is not None:
        via = think1 + move_time(fitts, HOME, other)
        think2 = hick[0] + hick[1] * hick_bits(len(screen2))
        for b in screen2:
            t = via + think2 + move_time(fitts, centre(other), b)
            times[b["label"]] = (t, float(b["share"]))
    return times


def main():
    folder = sys.argv[1] if len(sys.argv) > 1 else "data"
    taps = read_rows(f"{folder}/taps.csv")
    ids = [fitts_id(float(r["distance_mm"]), float(r["width_mm"]))
           for r in taps]
    fitts = fit_line(ids, [float(r["seconds"]) for r in taps])
    print(f"Fitts: MT = {fitts[0]:.3f} + {fitts[1]:.3f} * ID   "
          f"(r2 = {fitts[2]:.3f}, {len(taps)} taps)")
    print(f"       throughput 1/b = {1 / fitts[1]:.2f} bits/s")
    picks = read_rows(f"{folder}/choices.csv")
    hick = fit_line([hick_bits(int(r["choices"])) for r in picks],
                    [float(r["seconds"]) for r in picks])
    print(f"Hick:  RT = {hick[0]:.3f} + {hick[1]:.3f} * log2(n+1)   "
          f"(r2 = {hick[2]:.3f}, {len(picks)} choices)")
    for name in ("kiosk_old", "kiosk_new"):
        times = service_times(read_rows(f"{folder}/{name}.csv"), fitts, hick)
        mean = sum(t * s for t, s in times.values())
        slow = max(times, key=lambda k: times[k][0])
```

Listing 2.3 (Continued) *humanfactors.py: fits Fitts's and Hick's laws to the logs and computes the expected time of each kiosk layout.*

```
        print(f"{name}: expected {mean:.2f} s per patient; "
              f"slowest {slow} {times[slow][0]:.2f} s")
        for label, (t, s) in sorted(times.items(), key=lambda kv: -kv[1][1]):
            print(f"  {label:17}{s:6.0%}{t:7.2f} s")


if __name__ == "__main__":
    main()
```

`fitts_id` and `hick_bits` are Equation (2.1) and the logarithm of Equation (2.4). `fit_line` is Equation (2.6) and Equation (2.7) written out. `read_rows` reads a CSV file into a list of dictionaries, one per row, keyed by the column names of the first line. `centre` finds a button's

center from its corner and size, and `move_time` applies Fitts's law from a starting point to a button, using the smaller side as W. `service_times` implements Section 2.8.3: it splits the layout into the two screens, finds the `Other` button if there is one, and returns, for every service, a pair of time and share. `main` fits both laws, prints the constants, then prints the expected time and the time of every service for both layouts, most frequent first.

Listing 2.4 is the Java version, which prints exactly the same lines.

Listing 2.4 `HumanFactors.java`: the same fitting and layout comparison in Java.

```java
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

/** Fitts's and Hick's laws: fit them to logs, then compare two kiosk layouts. */
public class HumanFactors {
    // The hand rests on the ledge below the screen (mm).
    static final double HOME_X = 110, HOME_Y = -40;

    /** Index of difficulty in bits (Shannon form). */
    static double fittsId(double distance, double width) {
        return Math.log(distance / width + 1) / Math.log(2);
    }

    /** Bits of a choice among n equally likely options (plus "none"). */
    static double hickBits(int choices) {
        return Math.log(choices + 1) / Math.log(2);
    }

    /** Least squares: {a, b, r squared} of y = a + b*x. */
    static double[] fitLine(List<Double> xs, List<Double> ys) {
        int n = xs.size();
        double mx = 0, my = 0;
        for (int i = 0; i < n; i++) { mx += xs.get(i) / n; my += ys.get(i) / n; }
        double sxy = 0, sxx = 0, ssTot = 0;
        for (int i = 0; i < n; i++) {
            sxy += (xs.get(i) - mx) * (ys.get(i) - my);
            sxx += (xs.get(i) - mx) * (xs.get(i) - mx);
            ssTot += (ys.get(i) - my) * (ys.get(i) - my);
        }
        double b = sxy / sxx, a = my - b * mx, ssRes = 0;
        for (int i = 0; i < n; i++) {
            double e = ys.get(i) - a - b * xs.get(i);
            ssRes += e * e;
        }
        return new double[] {a, b, 1 - ssRes / ssTot};
    }
```

Listing 2.4 (Continued) *HumanFactors.java: the same fitting and layout comparison in Java.*

```
    /** A CSV file as a list of rows, each a map from column name to text. */
    static List<Map<String, String>> readRows(String path) throws IOException {
        List<String> lines = Files.readAllLines(Path.of(path));
        String[] head = lines.get(0).split(",");
        List<Map<String, String>> rows = new ArrayList<>();
        for (String line : lines.subList(1, lines.size())) {
            String[] cells = line.split(",", -1);
            Map<String, String> row = new HashMap<>();
            for (int i = 0; i < head.length; i++) row.put(head[i], cells[i]);
            rows.add(row);
        }
        return rows;
    }

    static double num(Map<String, String> row, String key) {
        return Double.parseDouble(row.get(key));
    }

    static double[] centre(Map<String, String> b) {
        return new double[] {num(b, "x_mm") + num(b, "w_mm") / 2,
                             num(b, "y_mm") + num(b, "h_mm") / 2};
    }

    /** Fitts time from a point to a button; the width is its smaller side. */
    static double moveTime(double[] fitts, double[] start, Map<String, String> b) {
        double[] c = centre(b);
        double d = Math.hypot(c[0] - start[0], c[1] - start[1]);
        double w = Math.min(num(b, "w_mm"), num(b, "h_mm"));
        return fitts[0] + fitts[1] * fittsId(d, w);
    }

    /** Seconds from the ready hand to the pressed service: {time, share}. */
    static Map<String, double[]> serviceTimes(List<Map<String, String>> layout,
                                              double[] fitts, double[] hick) {
        List<Map<String, String>> s1 = new ArrayList<>(), s2 = new ArrayList<>();
        Map<String, String> other = null;
        for (var b : layout) {
            (b.get("screen").equals("1") ? s1 : s2).add(b);
            if (b.get("label").equals("Other")) other = b;
        }
        double think1 = hick[0] + hick[1] * hickBits(s1.size());
        double[] home = {HOME_X, HOME_Y};
        Map<String, double[]> times = new LinkedHashMap<>();
        for (var b : s1) {
            if (b != other) {
                times.put(b.get("label"), new double[] {
```

Listing 2.4 (Continued) *HumanFactors.java: the same fitting and layout comparison in Java.*

```
                    think1 + moveTime(fitts, home, b), num(b, "share")});
            }
        }
        if (other != null) {
            double via = think1 + moveTime(fitts, home, other);
            double think2 = hick[0] + hick[1] * hickBits(s2.size());
            for (var b : s2) {
                double t = via + think2 + moveTime(fitts, centre(other), b);
                times.put(b.get("label"), new double[] {t, num(b, "share")});
            }
        }
        return times;
    }

    public static void main(String[] args) throws IOException {
        String folder = args.length > 0 ? args[0] : "data";
        List<Double> ids = new ArrayList<>(), mt = new ArrayList<>();
        var taps = readRows(folder + "/taps.csv");
        for (var r : taps) {
            ids.add(fittsId(num(r, "distance_mm"), num(r, "width_mm")));
            mt.add(num(r, "seconds"));
        }
        double[] fitts = fitLine(ids, mt);
        System.out.printf("Fitts: MT = %.3f + %.3f * ID   (r2 = %.3f, %d taps)%n",
                          fitts[0], fitts[1], fitts[2], taps.size());
        System.out.printf("       throughput 1/b = %.2f bits/s%n", 1 / fitts[1]);
        List<Double> bits = new ArrayList<>(), rt = new ArrayList<>();
        var picks = readRows(folder + "/choices.csv");
        for (var r : picks) {
            bits.add(hickBits(Integer.parseInt(r.get("choices"))));
            rt.add(num(r, "seconds"));
        }
        double[] hick = fitLine(bits, rt);
        System.out.printf("Hick:  RT = %.3f + %.3f * log2(n+1)   "
                          + "(r2 = %.3f, %d choices)%n",
                          hick[0], hick[1], hick[2], picks.size());
        for (String name : new String[] {"kiosk_old", "kiosk_new"}) {
            var times = serviceTimes(readRows(folder + "/" + name + ".csv"),
                                     fitts, hick);
            double mean = 0;
            String slow = null;
            for (var e : times.entrySet()) {
                mean += e.getValue()[0] * e.getValue()[1];
                if (slow == null || e.getValue()[0] > times.get(slow)[0]) {
                    slow = e.getKey();
                }
```

Listing 2.4 (Continued) *HumanFactors.java: the same fitting and layout comparison in Java.*

```
            }
            System.out.printf("%s: expected %.2f s per patient; "
                              + "slowest %s %.2f s%n",
                              name, mean, slow, times.get(slow)[0]);
            var list = new ArrayList<>(times.entrySet());
            list.sort((p, q) -> Double.compare(q.getValue()[1], p.getValue()[1]));
            for (var e : list) {
                System.out.printf("  %-17s%5.0f%%%7.2f s%n", e.getKey(),
                                  100 * e.getValue()[1], e.getValue()[0]);
            }
        }
    }
}
```

Java has no built-in CSV reader, so `readRows` splits each line at commas (the `-1` keeps an empty last cell, such as the missing share of `Other`) and builds a `HashMap` per row. Java's `Math.log` is the natural logarithm, so a base-2 logarithm is `Math.log(x) / Math.log(2)`. A `LinkedHashMap` keeps the services in file order, and the closing sort by share is stable, so services with equal shares keep that order, exactly as in Python.

2.9 Running It

```
$ python3 humanfactors.py
$ java -cp out HumanFactors
```

```
Fitts: MT = 0.156 + 0.147 * ID   (r2 = 0.932, 60 taps)
       throughput 1/b = 6.81 bits/s
Hick:  RT = 0.227 + 0.205 * log2(n+1)   (r2 = 0.922, 48 choices)
kiosk_old: expected 1.70 s per patient; slowest General 1.71 s
  General             38%   1.71 s
  Mother and child    14%   1.71 s
  Children            12%   1.68 s
  Elderly             10%   1.69 s
  Dental               9%   1.70 s
  Laboratory           6%   1.69 s
  Immunization         3%   1.67 s
  Tuberculosis         2%   1.67 s
  Nutrition            2%   1.66 s
  Pharmacy             2%   1.65 s
  Mental health        1%   1.65 s
  Sanitation           1%   1.64 s
kiosk_new: expected 1.44 s per patient; slowest Immunization 2.52 s
  General             38%   1.34 s
  Mother and child    14%   1.34 s
  Children            12%   1.29 s
  Elderly             10%   1.29 s
  Dental               9%   1.22 s
  Laboratory           6%   1.22 s
  Immunization         3%   2.52 s
  Tuberculosis         2%   2.52 s
  Nutrition            2%   2.45 s
  Pharmacy             2%   2.45 s
  Mental health        1%   2.38 s
  Sanitation           1%   2.38 s
```

The first three lines are the fitted laws. For the practice log, Fitts's law is $MT = 0.156 + 0.147\,ID$ with $r^2 = 0.932$, a throughput of 6.81 bits per second, and Hick's law is $RT = 0.227 + 0.205\log_2(n+1)$ with $r^2 = 0.922$. **Figure 2.4** shows the points and the fitted lines: the scatter around each line is the noise that every real measurement has, and the lines recover the chosen constants closely (0.18 and 0.14 for Fitts, 0.25 and 0.20 for Hick) but not exactly, which is what fitting 60 or 48 noisy points should do.

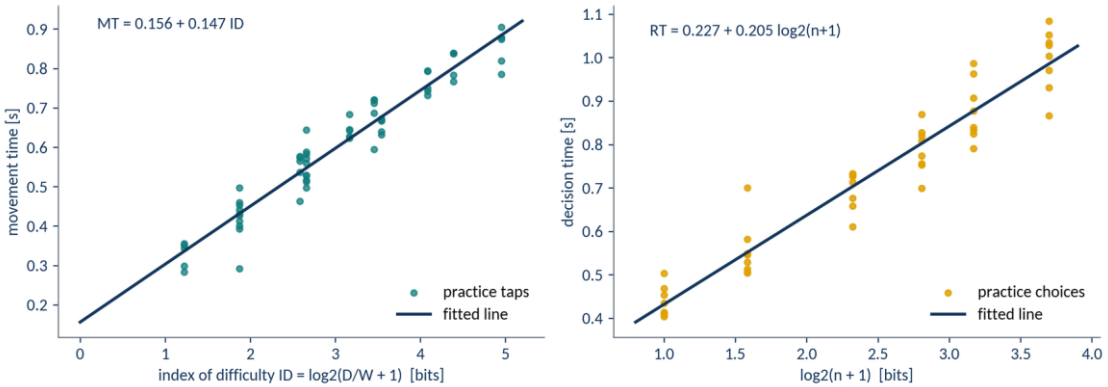


Figure 2.4 The practice logs with the fitted lines: Fitts's law (left) and Hick's law (right).

The two layout blocks answer Dr. Rina's question. The old kiosk needs 1.70 s per patient on average, almost the same for every service, because all buttons are equally small and equally far. The new kiosk needs 1.44 s, 15% less, and the six common services, 89% of patients, each take between 1.22 and 1.34 s. The price is paid by the 11% of patients who need a service on the second screen: they need about 2.4 to 2.5 s, one extra decision and one extra movement.

Two cautions keep the result honest. First, these are times for a person who knows the layout: the model has no term for reading unfamiliar labels, which for a first-time user takes far longer than 1.7 seconds and is where the old kiosk's small text hurt Bu Siti most. The redesign helps there too, with fewer labels, larger text and grouping, but the model cannot measure it; Chapter 7 does, with usability tests. Second, the laws say nothing about *errors*. Bu Siti's wrong ticket came from a target her finger covered; the new buttons, three times taller, make that error much less likely. The most important improvements of a redesign are often the ones that a time model does not see, and a good designer uses the model for what it measures and other methods for the rest.

2.10 Testing It

Listing 2.5 tests the parts whose answers are known exactly: two indices of difficulty ($\log_2(30/10+1) = \log_2 4 = 2$ and $\log_2(70/10+1) = \log_2 8 = 3$), a Hick value ($\log_2 8 = 3$) and a line fitted through three points that lie exactly on $y = 0.3 + 0.2x$.

Listing 2.5 `test_humanfactors.py`: tests of the two laws and of the least-squares fit.

```
"""Tests for Fitts and Hick: python -m unittest test_humanfactors.py"""
import unittest
from humanfactors import fitts_id, hick_bits, fit_line


class HumanFactorsTests(unittest.TestCase):
    def test_index_of_difficulty(self):
        self.assertAlmostEqual(fitts_id(30, 10), 2.0)
        self.assertAlmostEqual(fitts_id(70, 10), 3.0)

    def test_hick_bits(self):
        self.assertAlmostEqual(hick_bits(7), 3.0)

    def test_fit_line_recovers_a_perfect_line(self):
        a, b, r2 = fit_line([1, 2, 3], [0.5, 0.7, 0.9])   # 0.3 + 0.2x
        self.assertAlmostEqual(a, 0.3)
        self.assertAlmostEqual(b, 0.2)
        self.assertAlmostEqual(r2, 1.0)


if __name__ == "__main__":
    unittest.main()
```

Listing 2.6 runs the same six checks in Java.

Listing 2.6 `HumanFactorsTest.java`: the same checks in Java.

```
/** Tests for the Fitts and Hick calculator: java -cp out HumanFactorsTest */
public class HumanFactorsTest {
    static int passed = 0;

    static void near(double got, double wanted, String what) {
        if (Math.abs(got - wanted) > 1e-9) {
            throw new AssertionError(what + ": wanted " + wanted + ", got " + got);
        }
        passed++;
    }

    public static void main(String[] args) throws Exception {
        near(HumanFactors.fittsId(30, 10), 2.0, "ID of D=30, W=10");
        near(HumanFactors.fittsId(70, 10), 3.0, "ID of D=70, W=10");
        near(HumanFactors.hickBits(7), 3.0, "7 choices are 3 bits");
        var xs = java.util.List.of(1.0, 2.0, 3.0);
        var ys = java.util.List.of(0.5, 0.7, 0.9);    // exactly 0.3 + 0.2x
        double[] f = HumanFactors.fitLine(xs, ys);
        near(f[0], 0.3, "intercept");
        near(f[1], 0.2, "slope");
        near(f[2], 1.0, "perfect fit");
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
$ python3 -m unittest test_humanfactors.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
$ java -cp out HumanFactorsTest
All 6 checks passed.
```

2.11 Using It for Real

A human-factors review of any screen, whether a kiosk, an app or a form, can follow the three questions of Section 2.2 as a checklist. **Table 2.2** turns them into questions to ask while looking at a design, ideally together with a person from the intended user group.

Table 2.2 A human-factors checklist for reviewing a screen.

Stage	Question	What to look for
Perception	Can every user see and read it?	Text size and contrast for older eyes; meaning not carried by color alone; related items grouped by proximity or region
Attention	Will the important thing be noticed?	One clear main action; feedback where the user is looking; no competing animation
Memory	Does the task fit in about four chunks?	Recognition instead of recall; long numbers grouped; no carrying of information between screens
Decision	How many options at each step?	Rare options moved away from the first screen; frequent ones first
Movement	Can every target be reached and hit?	Targets at least the sizes of **Table 2.1**, larger for public kiosks; frequent targets close; dangerous ones apart

To use the calculator for your own design, replace the two layout files with your own buttons, measured on the real screen in millimeters (or in pixels, as long as all sizes use the same unit), and replace the shares with real counts. Most important, replace the practice logs with measurements of your users. Exercise 2.6 describes a simple procedure with a phone or a tablet and a helper with a stopwatch app; even twenty taps and twenty choices from three people of the intended age give constants that are more trustworthy for your kiosk than any textbook value.

The same approach works well beyond kiosks. A warung's order page, a laundry's price list on a phone, the pickup screen of an ojek app: each has frequent targets and rare ones, and each can be improved by making the frequent ones big, close and few.

2.12 HCI in Practice: Profitina

Profitina's users are brand owners who open the AI Visibility dashboard between other tasks, often on a phone, and the product's design reflects the limits of this chapter. The dashboard gathers everything a user usually needs next onto one screen, with each notification clickable straight to its section: that removes a decision and several movements for the most frequent path, the reasoning of Hick's and Fitts's laws. The weekly score trend is shown as a line rather than as a table of numbers, so that perception, not working memory, answers the question "am I improving?". And in the community channels, the counter "N new" counts only top-level posts, which keeps a single, meaningful number in view instead of asking members to remember what they had already read.

Memory limits also shape the account screens required by UU PDP. Changing an email address or a password, exporting data, or deleting an account are rare tasks, which means that users meet them without practice: they must rely on recognition, so each is a clearly labeled, separate action with its consequences written out, rather than a code or a setting buried in a long list.

2.13 Common Mistakes, and How to Avoid Them

1. **Designing text and targets for young eyes and hands.** Test with older users and follow the checklist of **Table 2.2**; the users who most need a public service are often the ones with the weakest eyes.
2. **Using color as the only signal.** Add a word, an icon or a position; 1 in 12 men cannot rely on red and green.
3. **Believing "seven plus or minus two".** Design for about four chunks and let the interface carry the rest.
4. **Using Hick's law to defend a long menu.** It holds for practiced choice, not for searching unfamiliar labels; for newcomers, fewer options really are faster.
5. **Plugging textbook constants into Fitts's law.** a and b depend on the people, the device and the posture; measure them, and state where they came from.
6. **Taking the width in the wrong direction.** For a finger that approaches from below, the height of a wide, flat button is the width that matters.
7. **Reading a time model as the whole truth.** It ignores errors, learning and reading; combine it with observation and usability tests.
8. **Placing feedback where the user is not looking.** Put confirmations next to the action or on the result, and interrupt only when a mistake would be costly.

2.14 Chapter Summary

- The Model Human Processor describes a person as perceptual, cognitive and motor processors with sensory stores, a small working memory and a large long-term memory (**Figure 2.1**); each stage can be overloaded.
- Vision is sharp only in the fovea; with age come presbyopia and a need for more light and contrast; 1 in 12 men have a color vision deficiency. The Gestalt principles of proximity, similarity, common region and continuity group what we see before we read.
- Attention is limited and selective; what is not attended is often not seen at all.
- Working memory holds about four chunks for a few seconds; chunking and recognition rather than recall work around it.
- Fitts's law: $MT = a + b\,ID$ with $ID = \log_2(D/W + 1)$ (Equation (2.2), Equation (2.1)); throughput is $1/b$. Targets should be large, close and at least the sizes of **Table 2.1**.
- Hick's law: $RT = a + b\log_2(n + 1)$ (Equation (2.4)), or $a + bH$ with the entropy of unequal choices (Equation (2.5)); it holds for practiced choice, not for search.
- Fitted to the practice logs, the laws predict 1.70 s per patient for the old kiosk and 1.44 s for the new one; the larger buttons also prevent errors that the model does not count.

2.15 Exercises

1. A button 9 mm tall is 120 mm from the resting finger. Using the practice constants $a = 0.156$ s and $b = 0.147$ s/bit, compute its index of difficulty and movement time. How tall must the button be to bring MT below 0.60 s?
2. A menu has 15 equally likely items. Using the practice Hick constants, compare the decision time of one flat menu of 15 items with a two-level menu of 3 groups of 5 items (one decision at each level). Which is faster, and by how much? Ignore movement.

3. Compute the entropy H (Equation (2.5)) of the twelve service shares of the old kiosk, and the Hick–Hyman decision time $a + bH$ with the practice constants. Compare it with the equal-likelihood value $a + b\log_2 13$ and explain the difference.
4. A virtual-account number for a payment is shown as 8808123456789012. Rewrite it for a person who must type it on another phone, and explain your choice in terms of working memory.
5. A colleague suggests keeping the old kiosk's twelve small buttons but moving them 140 mm lower, near the hand. Create a copy of the data folder with this change, run the program on it, and compare the expected time with the old and the new layout. What does the result say about distance and width?
6. Describe a procedure to record your own tap and choice logs for the kiosk with a tablet and a helper, in the formats of `taps.csv` and `choices.csv`. Who should the participants be, and how many taps and choices would you record?
7. The waiting-hall display of Puskesmas Kenanga shows the number being called in green and the next numbers in red, with a short beep. List the human-factors problems of this design for the patients of the elderly clinic and propose a better one.

2.16 Solutions to the Exercises

1. $ID = \log_2(120/9 + 1) = \log_2 14.333 = 3.841$ bits and $MT = 0.156 + 0.147 \times 3.841 = 0.721$ s. For $MT < 0.60$ s we need $ID < (0.60 - 0.156)/0.147 = 3.020$ bits, so $120/W + 1 < 2^{3.020} = 8.11$, that is $W > 120/7.11 = 16.9$ mm. A button 17 mm tall reaches the goal.

2. Flat: $RT = 0.227 + 0.205\log_2 16 = 0.227 + 0.205 \times 4 = 1.047$ s. Two levels: $(0.227 + 0.205\log_2 4) + (0.227 + 0.205\log_2 6) = (0.227 + 0.410) + (0.227 + 0.530) = 1.394$ s. The flat menu is faster by about 0.35 s, because each level pays the intercept a again and adds its own "+1" option. In practice each level also costs a movement and a screen change, so broad and shallow wins more clearly, as long as the flat list stays short enough to be recognized rather than searched.

3. With the shares 0.38, 0.09, 0.14, 0.10, 0.12, 0.06, 0.02, 0.02, 0.03, 0.01, 0.01, 0.02, the entropy is $H = \sum p_i \log_2(1/p_i) = 2.806$ bits, so $RT = 0.227 + 0.205 \times 2.806 = 0.802$ s, against 0.986 s for $\log_2 13 = 3.700$ bits. A practiced patient, for whom General is much more likely than Sanitation, carries less uncertainty than twelve equal options would imply, and the Hick–Hyman law predicts a faster choice. The effect only holds for people who know the frequencies, such as regular patients.

4. Show it as 8808 1234 5678 9012: four groups of four digits, which a person can hold one chunk at a time, copy, and check group by group. Sixteen ungrouped digits exceed working memory many times over, which forces rereading and invites skipped or doubled digits. A copy button next to the number is better still, because it removes the memory task altogether.

5. Copy the folder, for example to `data_low`, subtract 140 from every `y_mm` in `kiosk_old.csv`, and run `python3 humanfactors.py data_low`. The old layout's expected time falls from 1.70 s to 1.56 s (General 1.58 s), still slower than the new layout's 1.44 s. Moving targets closer helps, but the logarithm of D/W is dominated by the small width: the new layout wins mainly because its buttons are three times taller, and because its first screen has fewer options.

6. Recruit at least five people from the real user group, including patients of the elderly clinic, and ask for their consent. On a tablet mounted at the kiosk's height, show one target at a time at a

known distance from a marked resting point, for four distances and three widths, five taps each, in a random order; the app, or the helper with a stopwatch app, records distance, width and time, and wrong taps are noted separately. For choices, show a word and n buttons, n from 1 to 12, eight times each, and record the time to the correct press. That gives 60 taps and 48 choices per person in the formats of `taps.csv` and `choices.csv`; fit each person separately and together, and use the constants of the older participants for the kiosk.

7. Color alone separates called from waiting numbers, which fails for color-deficient viewers; green and red of similar lightness are hard to tell apart even for others at a distance. The numbers may be too small to read from the waiting chairs with age-related vision. A short beep is easily missed with age-related hearing loss or in a noisy hall, and it says nothing about *which* number or clinic. Better: show the called number very large with the word "CALLED" and the clinic name and room, list waiting numbers separately in smaller text, and announce the number and clinic by voice, twice, with a chime before it.

References

[1] Periskop.id, "Jumlah Lansia Indonesia 2025 11,97%, Indonesia Masuk Aging Population," May 5, 2026, reporting Badan Pusat Statistik, *Survei Penduduk Antar Sensus (SUPAS) 2025*. [Online]. Available: https://periskop.id/kependudukan/20260505/jumlah-lansia-indonesia-2025-1197-indonesia-masuk-aging-population

[2] S. K. Card, T. P. Moran, and A. Newell, *The Psychology of Human-Computer Interaction*. Hillsdale, NJ, USA: Lawrence Erlbaum, 1983.

[3] C. D. Wickens, W. S. Helton, J. G. Hollands, and S. Banbury, *Engineering Psychology and Human Performance*, 5th ed. New York, NY, USA: Routledge, 2021.

[4] S. J. Czaja, W. R. Boot, N. Charness, and W. A. Rogers, *Designing for Older Adults: Principles and Creative Human Factors Approaches*, 3rd ed. Boca Raton, FL, USA: CRC Press, 2019.

[5] Colour Blind Awareness, "About Colour Blindness," accessed Sep. 2026. [Online]. Available: https://www.colourblindawareness.org/colour-blindness/

[6] World Wide Web Consortium, *Web Content Accessibility Guidelines (WCAG) 2.2*, W3C Recommendation, success criteria 1.4.1, 2.5.5 and 2.5.8. [Online]. Available: https://www.w3.org/TR/WCAG22/

[7] J. Wagemans, J. H. Elder, M. Kubovy, S. E. Palmer, M. A. Peterson, M. Singh, and R. von der Heydt, "A century of Gestalt psychology in visual perception: I. Perceptual grouping and figure–ground organization," *Psychological Bulletin*, vol. 138, no. 6, pp. 1172–1217, 2012.

[8] A. M. Treisman and G. Gelade, "A feature-integration theory of attention," *Cognitive Psychology*, vol. 12, no. 1, pp. 97–136, 1980.

[9] Google Design, "Expressive Design: Google's UX Research," accessed Sep. 2026. [Online]. Available: https://design.google/library/expressive-material-design-google-research

[10] D. J. Simons and C. F. Chabris, "Gorillas in our midst: Sustained inattentional blindness for dynamic events," *Perception*, vol. 28, no. 9, pp. 1059–1074, 1999.

[11] G. A. Miller, "The magical number seven, plus or minus two: Some limits on our capacity for processing information," *Psychological Review*, vol. 63, no. 2, pp. 81–97, 1956.

[12] N. Cowan, "The magical number 4 in short-term memory: A reconsideration of mental storage capacity," *Behavioral and Brain Sciences*, vol. 24, no. 1, pp. 87–114, 2001.

[13] D. A. Norman, *The Design of Everyday Things*, revised and expanded ed. New York, NY, USA: Basic Books, 2013.

[14] P. M. Fitts, "The information capacity of the human motor system in controlling the amplitude of movement," *Journal of Experimental Psychology*, vol. 47, no. 6, pp. 381–391, 1954.

[15] I. S. MacKenzie, "Fitts' law as a research and design tool in human-computer interaction," *Human-Computer Interaction*, vol. 7, no. 1, pp. 91–139, 1992.

[16] International Organization for Standardization, *ISO 9241-411:2012 Ergonomics of human-system interaction — Part 411: Evaluation methods for the design of physical input devices*. Geneva, Switzerland: ISO, 2012.

[17] R. W. Soukoreff and I. S. MacKenzie, "Towards a standard for pointing device evaluation, perspectives on 27 years of Fitts' law research in HCI," *International Journal of Human-Computer Studies*, vol. 61, no. 6, pp. 751–789, 2004.

[18] Apple Inc., "Accessibility," *Human Interface Guidelines*, accessed Sep. 2026. [Online]. Available: https://developer.apple.com/design/human-interface-guidelines/accessibility

[19] Google, "Touch target size," *Android Accessibility Help*, accessed Sep. 2026. [Online]. Available: https://support.google.com/accessibility/android/answer/7101858

[20] W. E. Hick, "On the rate of gain of information," *Quarterly Journal of Experimental Psychology*, vol. 4, no. 1, pp. 11–26, 1952.

[21] R. Hyman, "Stimulus information as a determinant of reaction time," *Journal of Experimental Psychology*, vol. 45, no. 3, pp. 188–196, 1953.

Chapter 3

Design Principles and Heuristics

Most usability problems are not new. The button that gives no sign of having worked, the form that accepts an impossible date, the error message that says “Error 422”, the price that is hidden until the end: each has been seen, named and solved thousands of times. Design principles and heuristics are the distilled experience of those solutions, short rules that let a designer or an evaluator recognize a known problem before users suffer from it. This chapter presents the principles that matter most, from Norman’s signifiers, mappings, constraints and feedback to Shneiderman’s golden rules, the interaction principles of ISO 9241-110 and Nielsen’s ten usability heuristics; shows how the design languages of 2025 and 2026 apply and sometimes strain them; and teaches heuristic evaluation, the cheapest widely used method for finding usability problems. By the end of the chapter you will be able to explain each principle with an everyday example; carry out a heuristic evaluation with several evaluators and a severity scale; estimate with the Nielsen–Landauer model how many evaluators you need; and turn the evaluators’ findings into a ranked report with a program in Java and in Python, applied to the order page of Profitina Laundry.

3.1 Mbak Dewi’s Laundry Leaves WhatsApp

Profitina Laundry is a small laundry in Surabaya, the teaching case study of this book. Mbak Dewi runs it with two helpers, a motorbike for pickups and a row of washing machines. For years every order came by WhatsApp: a customer sent “Mbak, jemput besok ya”, Mbak Dewi replied with a numbered menu (1 for wash and dry, 2 for wash, dry and iron, 3 for iron only), and a long conversation followed about the day, the address and the price. It worked, but it cost her an hour every evening, and misunderstandings about “2” and “3” were common.

In 2026 a web developer offered her an order page for a modest fee. The first version, on the left of **Figure 3.1**, looked finished and technically worked: every order it sent arrived in her database. But in the first week three customers sent the same order twice because the page just reloaded after *Kirim*, one customer chose a Sunday pickup when the shop is closed, and several asked by WhatsApp how much they would pay, which was exactly what the page was supposed to spare her.

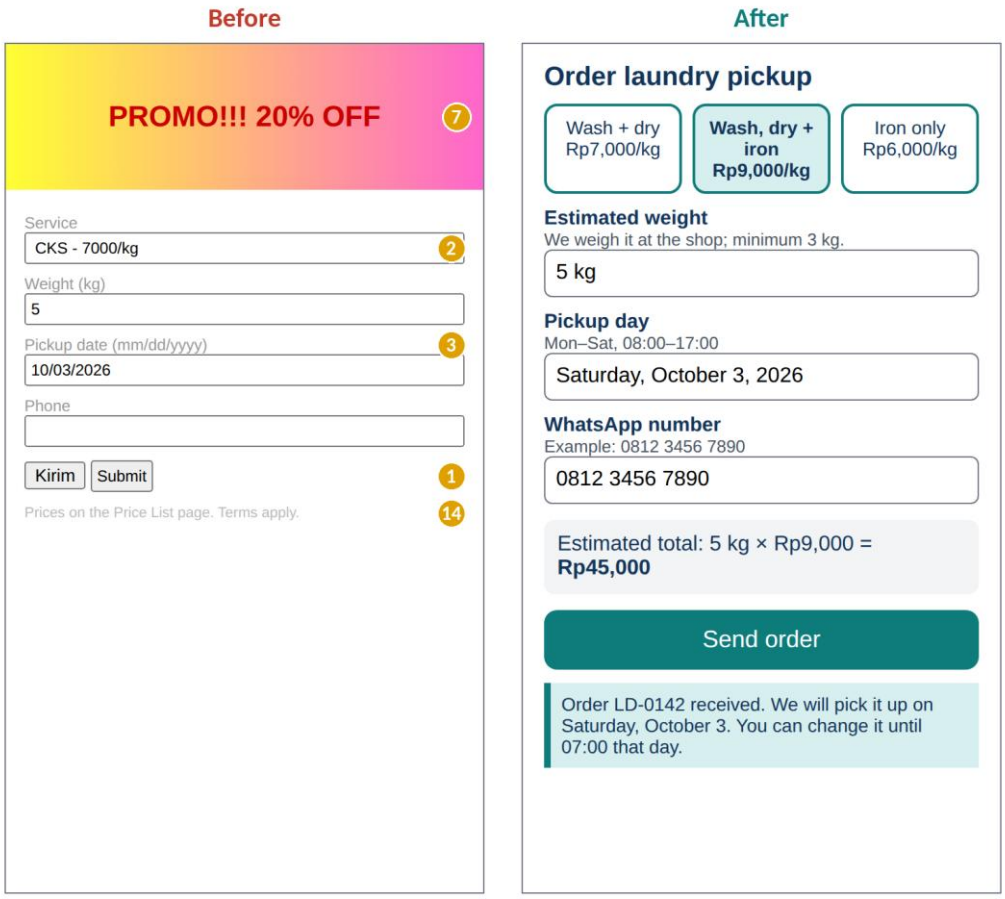


Figure 3.1 The order page of Profitina Laundry before and after a heuristic evaluation, rendered from our own teaching pages; the numbered markers point at problems P01, P02, P03, P07 and P14 of **Listing 3.1**.

Mbak Dewi's niece, an informatics student, suggested something cheaper than guessing and cheaper than a full user study: ask four people who know the principles of interface design to look at the page, each on their own, against a short list of rules, and then combine what they find. This is a *heuristic evaluation*. Within an afternoon the four evaluators had written 26 findings, which turned out to be 14 distinct problems. This chapter explains the rules they used, the method, and the small program that merged their findings into the ranked list that the developer then worked through. The page on the right of the figure is the result.

3.2 The Idea in Plain Words: Principles, Guidelines and Heuristics

Three kinds of design knowledge are often confused.

A *principle* is a general statement about how people and interfaces interact, true across products and years: "people need feedback to know that an action worked." It says *why*, but not exactly *what to do* on a particular screen.

A *guideline* is a specific rule for a specific kind of product: "show a confirmation with the order number after an order is sent," or "touch targets at least 48 by 48 dp." Guidelines are what design systems such as Material Design, Apple's Human Interface Guidelines and Microsoft's Fluent 2 are made of. They are concrete, but they age with the platform.

A *heuristic* is a rule of thumb in between: general enough to apply to any interface, concrete enough to check against a screen. "Visibility of system status: the design should always keep users informed about what is going on, through appropriate feedback within a reasonable amount of

time" is Nielsen's first heuristic [1]. Heuristics are made for evaluation: an evaluator walks through a screen and asks, for each heuristic, whether it is violated and where.

> **DEFINITION.** *Heuristic evaluation* is an inspection method in which a small number of evaluators examine an interface independently, judge its compliance with a set of recognized usability principles (the heuristics), record each violation as a usability problem with its location and a severity, and then merge their findings into one list [2], [3].

The method is called *discount* usability because it needs no users, no laboratory and very little time. That is its strength and its limit: it finds many problems quickly, but it finds the problems that experts can foresee, not the ones that only real users reveal. Chapter 7 adds user testing; a good project uses both.

3.3 Norman's Principles

Don Norman's *The Design of Everyday Things* describes a handful of principles that explain why doors, stoves and screens are easy or hard [4]. They are the foundation on which every heuristic list rests.

3.3.1 Affordances and signifiers

An *affordance* is a relationship between an object and a person: what actions the object makes possible for that person. A flat plate on a door affords pushing; a chair affords sitting for an adult and climbing for a toddler. A *signifier* is a perceivable signal that tells a person what can be done and where: a handle on the pull side of a door, the raised look of a button, the underline of a link, the grey hint text "Example: 0812 3456 7890" in a phone field. On screens almost everything is possible in principle, since any pixel could react to a touch, so what matters is the signifier. Flat design removed many signifiers in the 2010s, and the recurring complaint that users "do not see that it is a button" is a complaint about missing signifiers.

On the first order page, the two send buttons *Kirim* and *Submit* are signifiers that contradict each other (which one should I press?), and the service dropdown shows the code "CKS" but no sign of what the options mean.

3.3.2 Mapping

Mapping is the relationship between controls and their effects. A mapping is *natural* when the arrangement of the controls mirrors the arrangement of what they control. **Figure 3.2** shows Norman's classic example, a stove: four burners in a square and four knobs in a row force the cook to learn or read which knob is which, while four knobs arranged in the same square as the burners need no label at all.

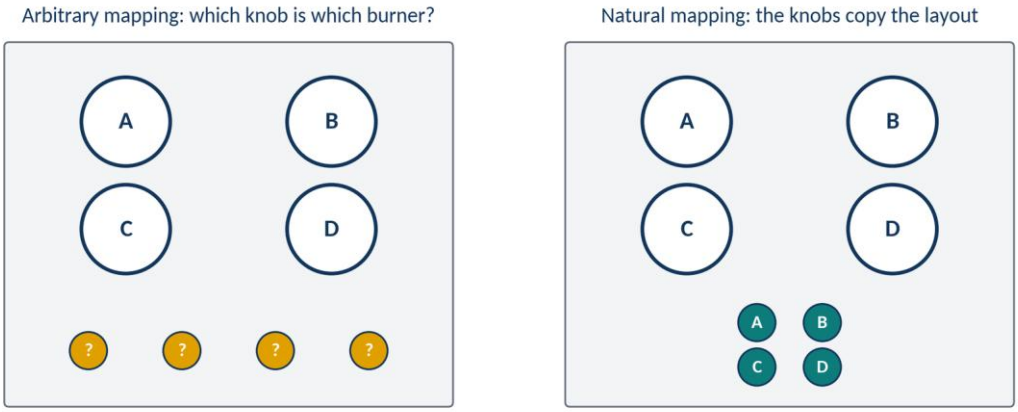


Figure 3.2 Mapping on a stove: knobs in a row give an arbitrary mapping, knobs in the same square as the burners give a natural one.

Screens are full of mappings: a volume slider that goes up for louder, a map where dragging left moves the view left, a seat map in a ticket app that shows seats in their real positions. The three service tiles of the improved order page map the three services to three visible choices with their prices, instead of hiding them behind a code in a list.

3.3.3 Constraints and forcing functions

Constraints limit the actions that are possible, so that wrong actions cannot happen. Physical constraints (a USB-C plug fits either way up, which removed a constraint that users had to learn), logical constraints (a date picker that greys out Sundays when the shop is closed), and cultural constraints (red means stop) all prevent errors before they occur. A *forcing function* is a strong constraint that makes the next step impossible until the previous one is done.

The ATM is the classic case. On early machines the person received the cash first and the card second. Having reached the goal, cash, many people walked away without the card. Psychologists call this a *post-completion error*: the step that comes after the main goal is complete is the one most often forgotten, especially when working memory is busy [5]. The fix was a forcing function: most machines now return the card *before* dispensing the cash, so the goal cannot be reached without first taking the card. No training, no warning sign, no blaming of users: a change of order removed the error.

3.3.4 Feedback and the conceptual model

Feedback tells people what happened as a result of their action, quickly and clearly. A button that changes when pressed, a progress bar, a sound, and above all a confirmation that states the result (“Order LD-0142 received”) close the gulf of evaluation of Chapter 1. Feedback that is late, vague or missing leads to repeated actions, which is exactly why customers sent their laundry order twice.

A *conceptual model* is the explanation of how the system works that the design communicates. The improved order page builds a good one: services have per-kilogram prices, the weight is an estimate that the shop will confirm, and an order has a number and can be changed until a stated time. With that model a customer can predict what will happen and correct mistakes without asking.

Finally, *discoverability* ties the principles together: can a person find out what actions are possible and how to perform them? It is the result of good signifiers, natural mappings, sensible constraints, feedback and a clear conceptual model working together [4].

3.4 Three Families of Rules

Several researchers have turned principles like Norman's into checklists. Three are used everywhere, and they agree far more than they differ.

Shneiderman's eight golden rules of interface design, refined over six editions of *Designing the User Interface*, are: strive for consistency; seek universal usability; offer informative feedback; design dialogs to yield closure; prevent errors; permit easy reversal of actions; keep users in control; and reduce short-term memory load [6]. The second rule, added in later editions, asks designers to serve novices and experts, young and old, people with and without disabilities, a theme Chapter 11 develops.

ISO 9241-110:2020 defines seven *interaction principles* as part of the international ergonomics standard: suitability for the user's tasks; self-descriptiveness; conformity with user expectations; learnability; controllability; use error robustness; and user engagement [7]. Their value is that they are a standard: a public-sector tender or a contract can require compliance with them.

Nielsen's ten usability heuristics were derived in 1994 from a factor analysis of 249 usability problems and have been used almost unchanged since, with wording and examples updated by the Nielsen Norman Group [1]. They are the most widely used list for heuristic evaluation and the one the evaluators of Profitina Laundry used. **Table 3.1** gives each heuristic with a short explanation and an example from the order page.

Table 3.1 Nielsen's ten usability heuristics, in plain words, with examples from the first order page of Profitina Laundry.

Code	Heuristic	In plain words	Example on the first order page
H1	Visibility of system status	Always show what is going on, promptly	No confirmation after sending; no total price
H2	Match between the system and the real world	Speak the users' language and follow real-world conventions	Codes "CKS", "CK", "SL" instead of wash, dry, iron
H3	User control and freedom	Offer a clearly marked way out: undo, cancel, edit	An order cannot be changed or cancelled
H4	Consistency and standards	Same words and actions mean the same thing; follow platform conventions	Two send buttons; month/day date order
H5	Error prevention	Prevent problems, rather than only reporting them	A past date or a Sunday can be chosen
H6	Recognition rather than recall	Make options visible; do not make users remember	Prices are on another page
H7	Flexibility and efficiency of use	Shortcuts for frequent users, invisible to novices	No "repeat my last order"
H8	Aesthetic and minimalist design	Show only what is relevant; every extra element competes	An animated promotion pushes the form down
H9	Help users recognize, diagnose and recover from errors	Plain-language errors that say what happened and what to do	"Error 422"
H10	Help and documentation	Help that is easy to find, focused on the task	No explanation of weighing or the 3 kg minimum

The three families line up closely. Shneiderman's "offer informative feedback", ISO's "self-descriptiveness" and Nielsen's H1 are one idea; "permit easy reversal", "controllability" and H3 are another; "prevent errors", "use error robustness" and H5 a third; "reduce short-term memory load" and H6 a fourth. Choose one list for an evaluation, so that all evaluators speak the same language, and know that the others will not find something fundamentally different.

3.5 Principles Meet the Design Languages of 2025–2026

Design languages change every few years; principles do not. The renewals of 2025 show this well.

Liquid Glass. Apple's design language for iOS 26 and its sibling systems, introduced on June 9, 2025, is a translucent material that "reflects and refracts its surroundings" and changes with the content beneath it [8]. It is beautiful and it makes the content, not the controls, the center of attention, which serves the principle of minimalist design (H8). It also strains another principle: text on a translucent layer over a busy photo can lose contrast, and contrast is what makes text perceivable at all (Chapter 2). After criticism of legibility, especially from older users, Apple added a setting in iOS 26.1, *Settings › Display and Brightness › Liquid Glass*, which offers *Clear* and a more opaque *Tinted* option [9]. The episode is a small lesson in itself: when a style conflicts with a principle, the principle wins in the end, and the fix is to give users control (H3, H7).

Material 3 Expressive. Google's 2025 update to Material Design uses stronger color, shape, size and motion to make the main action stand out. Google reports 46 studies with more than 18,000 participants and elements found up to four times faster in expressive layouts [10]. In the language of this chapter, it strengthens signifiers and visual hierarchy, the preattentive effect of Chapter 2 applied systematically.

Fluent 2. Microsoft's design system for Windows and Microsoft 365 emphasizes consistency across devices and products, with shared components and tokens [11]. It serves H4 across a very large family of applications.

For a designer in Indonesia, where most users are on Android phones and a smaller but significant share on iPhones (Chapter 1), the practical conclusion is to follow the platform's current conventions (H4: users expect a screen to behave like the other apps on their phone) while checking every style choice against the principles, especially contrast and target size for older users.

> **WARNING:** A design system does not guarantee usability. Every component of Material, Apple's guidelines or Fluent 2 can be assembled into a confusing page. The heuristics are the check that the assembly still makes sense.

3.6 Heuristic Evaluation Step by Step

3.6.1 The procedure

A heuristic evaluation has five steps [2], [3].

1. **Prepare.** Choose the heuristics (usually Nielsen's ten), the screens or tasks to cover, and a shared recording template: problem, location, heuristic, severity, and a suggestion.
2. **Evaluate independently.** Each evaluator goes through the interface at least twice, first to get a feel for the flow, then screen by screen against the heuristics, and records every violation. Independence matters: evaluators who talk to each other find the same problems.

3. **Debrief and merge.** The evaluators meet, read out their findings, and decide which findings describe the same problem. Each distinct problem gets an identifier.
4. **Rate severity.** Each evaluator rates each problem they found (or, better, every merged problem) on a severity scale.
5. **Report and prioritize.** Problems are ranked by severity and turned into a work list for the developers.

Nielsen's severity scale, shown in **Table 3.2**, combines how often a problem occurs, how badly it affects the users who meet it, and whether it keeps bothering them once they know it [12].

Table 3.2 Nielsen's severity scale for usability problems.

Rating	Meaning	What to do
0	Not a usability problem	Nothing
1	Cosmetic problem	Fix if time allows
2	Minor usability problem	Fix with low priority
3	Major usability problem	Fix with high priority
4	Usability catastrophe	Fix before release

3.6.2 How many evaluators?

No single evaluator finds everything. Nielsen and Landauer modeled the finding of problems as follows: if each evaluator independently finds a share λ of the N problems in an interface, then i evaluators together find

$$\text{Found}(i) = N(1 - (1 - \lambda)^i) \tag{3.1}$$

problems [13]. The reasoning is short. A given problem is missed by one evaluator with probability $1 - \lambda$, and missed by all i independent evaluators with probability $(1 - \lambda)^i$; it is found by at least one with probability $1 - (1 - \lambda)^i$, and multiplying by N gives the expected number found. Across the studies Nielsen and Landauer analyzed, λ averaged about 0.31 [13].

In practice N is unknown, but λ can be estimated from the evaluation itself. If n evaluators made F findings in total about P distinct problems, the average share found per evaluator is

$$\hat{\lambda} = \frac{F}{n \cdot P} \tag{3.2}$$

This estimate treats the problems found as all the problems, so it is somewhat optimistic; with few evaluators, problems that nobody found are invisible. Solving $1 - (1 - \lambda)^i \geq t$ for i gives the number of evaluators needed to find a target share t:

$$i \geq \frac{\ln(1 - t)}{\ln(1 - \lambda)} \tag{3.3}$$

(both logarithms are negative, so the inequality keeps its direction). With $\lambda = 0.31$ and $t = 0.75$, $i \geq \ln 0.25 / \ln 0.69 = 3.74$, so four evaluators; for $t = 0.90$ we need seven. **Figure 3.3** shows the curve for Nielsen and Landauer's average and for the Profitina Laundry evaluation. The flattening of the curve is the reason behind the usual advice of three to five evaluators: each additional evaluator finds fewer new problems than the one before [3].

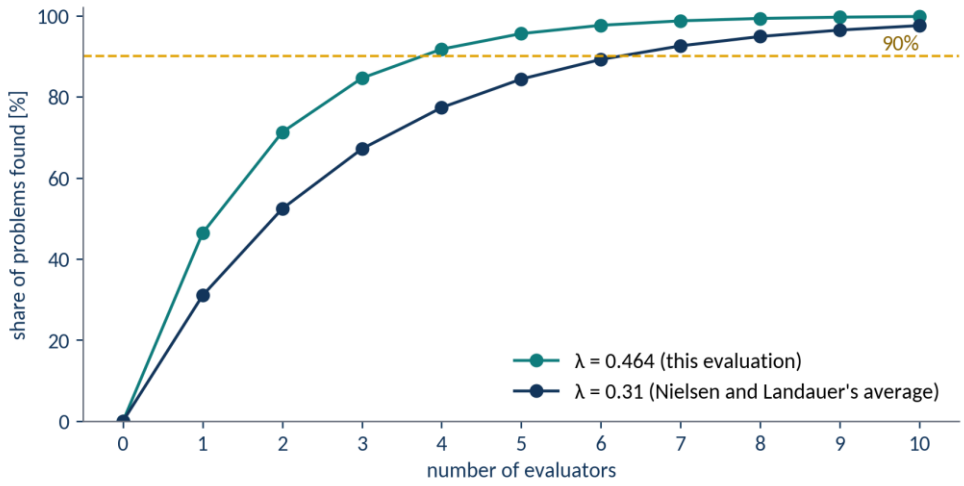


Figure 3.3 The share of problems found by i evaluators, from Equation (3.1), for $\lambda = 0.464$ (this evaluation) and $\lambda = 0.31$ (Nielsen and Landauer's average).

One more caution comes from research on the *evaluator effect*: different evaluators not only find different problems, they also rate the same problem differently, sometimes by two severity levels [14]. Merging and averaging several independent ratings is therefore not a formality; it is what makes the list trustworthy.

3.6.3 Heuristic evaluation and its relatives

Heuristic evaluation is one of several *inspection methods*, in which experts rather than users examine a design. Two relatives are worth knowing. A *cognitive walkthrough* follows one concrete task step by step, such as "a first-time customer orders wash, dry and iron for Saturday", and at every step asks four questions: will the user try to achieve the right effect, will they notice that the correct action is available, will they associate the action with the effect they want, and will they see that progress is being made? It is slower than a heuristic evaluation but better at finding problems of learnability for first-time users, the people the kiosk of Chapter 2 and the laundry's order page most need to serve. A *guideline review* checks a design against a long list of specific rules, such as a platform's guidelines or the success criteria of WCAG 2.2; it is thorough but mechanical, and it finds violations of rules rather than problems of use.

The three methods complement each other. A practical sequence for a small project is a heuristic evaluation of the whole interface first, a cognitive walkthrough of the two or three most important tasks next, and a guideline review for accessibility (Chapter 11) before release, followed by testing with real users (Chapter 7). Each costs hours, not weeks, and together they remove most of the problems that would otherwise reach customers.

3.7 Building the Report Generator

The debriefing of step 3 is a conversation and stays human: only people can decide that "the page reloads after sending" and "I cannot tell whether my order went through" are the same problem. Everything after it, counting, averaging, ranking and estimating, is bookkeeping that a program does better. The program of this section reads two files and prints the report.

3.7.1 The data

Listing 3.1 is the list of distinct problems agreed in the debriefing, with the heuristic each violates and where it occurs. Descriptions that contain commas are quoted, as usual in CSV.

Listing 3.1 `data/problems.csv`: the fourteen distinct problems of the first order page.

```
id,heuristic,location,problem
P01,H1,Send button,No confirmation after sending; the page just reloads
P02,H2,Service list,"Codes CKS, CK and SL instead of wash, dry, iron"
P03,H5,Pickup date,Accepts a past date and Sundays when the shop is closed
P04,H9,Error message,"Shows ""Error 422"" with no cause or remedy"
P05,H6,Prices,Price list is on another page; users must remember it
P06,H4,Buttons,"Two send buttons, ""Kirim"" and ""Submit"", in different styles"
P07,H8,Promo banner,Animated banner pushes the form below the fold
P08,H3,After sending,No way to change or cancel an order
P09,H7,Regular customers,No way to repeat the last order
P10,H10,Weight,No explanation of weighing or the 3 kg minimum
P11,H1,Total price,Total price is not shown before sending
P12,H5,Phone number,Accepts letters; no example of the expected format
P13,H4,Date format,Month/day order unlike every other Indonesian form
P14,H8,Small print,Light grey 11 px text is hard to read
```

Listing 3.2 records who found which problem and the severity they gave it. Evaluator E1 found eight problems, E2 six, E3 five and E4 seven.

Listing 3.2 `data/findings.csv`: 26 findings of four evaluators, each with a severity from 0 to 4.

```
evaluator,id,severity
E1,P01,4
E1,P02,3
E1,P03,3
E1,P04,3
E1,P08,3
E1,P11,4
E1,P12,2
E1,P14,2
E2,P01,3
E2,P02,2
E2,P05,2
E2,P07,2
E2,P11,3
E2,P13,2
E3,P01,4
E3,P03,2
E3,P06,1
E3,P09,2
E3,P11,3
E4,P02,3
E4,P04,2
E4,P08,3
E4,P10,1
E4,P11,4
E4,P12,2
E4,P14,3
```

3.7.2 The program

Listing 3.3 is the Python version.

Listing 3.3 `heuristics.py`: merges the findings, ranks the problems and estimates how many evaluators are needed.

```
"""Merge the findings of a heuristic evaluation into one ranked report."""
import csv
import math
import sys


def read_csv(path):
    with open(path, newline="", encoding="utf-8") as f:
        return list(csv.DictReader(f))


def merge(problems, findings):
    """Per problem: who found it and with which severities (0 to 4)."""
    merged = {p["id"]: dict(p, severities=[]) for p in problems}
    for f in findings:
        merged[f["id"]]["severities"].append(int(f["severity"]))
    return merged


def rank(merged):
    """Most severe first; then found by more evaluators; then by id."""
    def key(p):
        s = p["severities"]
        return (-sum(s) / len(s), -len(s), p["id"])
    return sorted((p for p in merged.values() if p["severities"]), key=key)


def found_share(lam, i):
    """Nielsen and Landauer: share of problems found by i evaluators."""
    return 1 - (1 - lam) ** i


def evaluators_needed(lam, target):
    """The smallest number of evaluators whose predicted share >= target."""
    return math.ceil(math.log(1 - target) / math.log(1 - lam))
```

Listing 3.3 (Continued) *heuristics.py: merges the findings, ranks the problems and estimates how many evaluators are needed.*

```
def main():
    folder = sys.argv[1] if len(sys.argv) > 1 else "data"
    problems = read_csv(f"{folder}/problems.csv")
    findings = read_csv(f"{folder}/findings.csv")
    evaluators = sorted({f["evaluator"] for f in findings})
    ranked = rank(merge(problems, findings))
    n = len(evaluators)
    print(f"Heuristic evaluation: {n} evaluators, {len(ranked)} problems, "
          f"{len(findings)} findings")
    print("rank id   heur found mean max location")
    for r, p in enumerate(ranked, 1):
        s = p["severities"]
        print(f"{r:4} {p['id']:4} {p['heuristic']:4} {len(s)}/{n}  "
              f"{sum(s) / len(s):4.1f} {max(s):3}  {p['location']}")
    urgent = [p["id"] for p in ranked
              if sum(p["severities"]) / len(p["severities"]) >= 3]
    print("Fix before launch (mean severity 3 or more): " + ", ".join(urgent))
    per_h = {}
    for p in ranked:
        per_h[p["heuristic"]] = per_h.get(p["heuristic"], 0) + 1
    order = sorted(per_h, key=lambda h: int(h[1:]))
    print("Problems per heuristic: "
          + ", ".join(f"{h} {per_h[h]}" for h in order))
    lam = len(findings) / (n * len(ranked))
    print(f"Each evaluator found {100 * lam:.1f}% of the problems "
          f"on average (lambda).")
    print("Predicted share found by i evaluators:")
    print("  " + "  ".join(f"{i}: {100 * found_share(lam, i):.0f}%"
                           for i in range(1, 7)))
    print(f"Evaluators needed for 90%: {evaluators_needed(lam, 0.90)}")


if __name__ == "__main__":
    main()
```

`merge` builds, for every problem, a copy of its row with an empty list of severities, then appends each finding's severity to its problem. `rank` sorts by a tuple: the negative mean severity first (so the highest mean comes first), then the negative number of evaluators who found it, then the identifier, so that the order is always the same. `found_share` is Equation (3.1) divided by N, and `evaluators_needed` is Equation (3.3) rounded up with `math.ceil`. `main` prints the ranked table, the list of problems with a mean severity of 3 or more, the number of problems per heuristic, the estimate $\hat{\lambda}$ of Equation (3.2), the predicted share for one to six evaluators and the number needed for 90%.

Listing 3.4 is the Java version, which prints the same report.

Listing 3.4 `Heuristics.java`: the same report in Java.

```java
import java.io.IOException;
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;

/** Merge the findings of a heuristic evaluation into one ranked report. */
public class Heuristics {
    record Problem(String id, String heuristic, String location,
                   List<Integer> severities) {
        double mean() {
            return severities.stream().mapToInt(i -> i).average().orElse(0);
        }
    }

    /** Split one CSV line; commas inside "quoted text" do not split. */
    static List<String> cells(String line) {
        List<String> out = new ArrayList<>();
        StringBuilder cell = new StringBuilder();
        boolean quoted = false;
        for (int i = 0; i < line.length(); i++) {
            char c = line.charAt(i);
            if (c == '"' && quoted && i + 1 < line.length()
                    && line.charAt(i + 1) == '"') {
                cell.append('"');                // "" inside quotes is one "
                i++;
            } else if (c == '"') {
                quoted = !quoted;
            } else if (c == ',' && !quoted) {
                out.add(cell.toString());
                cell.setLength(0);
            } else {
                cell.append(c);
            }
        }
        out.add(cell.toString());
        return out;
    }
```

Listing 3.4 (Continued) *Heuristics.java: the same report in Java.*

```
    static List<List<String>> readCsv(String path) throws IOException {
        List<List<String>> rows = new ArrayList<>();
        List<String> lines = Files.readAllLines(Path.of(path));
        for (String line : lines.subList(1, lines.size())) rows.add(cells(line));
        return rows;
    }

    /** Nielsen and Landauer: share of problems found by i evaluators. */
    static double foundShare(double lambda, int i) {
        return 1 - Math.pow(1 - lambda, i);
    }

    /** The smallest number of evaluators whose predicted share >= target. */
    static int evaluatorsNeeded(double lambda, double target) {
        return (int) Math.ceil(Math.log(1 - target) / Math.log(1 - lambda));
    }

    /** Per problem: who found it and with which severities; most severe first. */
    static List<Problem> rank(List<List<String>> problems,
                              List<List<String>> findings) {
        Map<String, Problem> merged = new LinkedHashMap<>();
        for (var p : problems) {
            merged.put(p.get(0), new Problem(p.get(0), p.get(1), p.get(2),
                                             new ArrayList<>()));
        }
        for (var f : findings) {
            merged.get(f.get(1)).severities().add(Integer.parseInt(f.get(2)));
        }
        List<Problem> ranked = new ArrayList<>();
        for (Problem p : merged.values()) {
            if (!p.severities().isEmpty()) ranked.add(p);
        }
        ranked.sort(Comparator.comparingDouble((Problem p) -> -p.mean())
                .thenComparingInt(p -> -p.severities().size())
                .thenComparing(Problem::id));
        return ranked;
    }
```

Listing 3.4 (Continued) *Heuristics.java: the same report in Java.*

```
    public static void main(String[] args) throws IOException {
        String folder = args.length > 0 ? args[0] : "data";
        var findings = readCsv(folder + "/findings.csv");
        var ranked = rank(readCsv(folder + "/problems.csv"), findings);
        TreeSet<String> evaluators = new TreeSet<>();
        for (var f : findings) evaluators.add(f.get(0));
        int n = evaluators.size();
        System.out.printf("Heuristic evaluation: %d evaluators, %d problems, "
                          + "%d findings%n", n, ranked.size(), findings.size());
        System.out.println("rank id   heur found mean max location");
        List<String> urgent = new ArrayList<>();
        Map<Integer, Integer> perH = new TreeMap<>();
        for (int r = 0; r < ranked.size(); r++) {
            Problem p = ranked.get(r);
            int max = p.severities().stream().mapToInt(i -> i).max().orElse(0);
            System.out.printf("%4d %-4s %-4s %d/%d  %4.1f %3d  %s%n", r + 1,
                              p.id(), p.heuristic(), p.severities().size(), n,
                              p.mean(), max, p.location());
            if (p.mean() >= 3) urgent.add(p.id());
            perH.merge(Integer.parseInt(p.heuristic().substring(1)), 1,
                       Integer::sum);
        }
        System.out.println("Fix before launch (mean severity 3 or more): "
                           + String.join(", ", urgent));
        List<String> counts = new ArrayList<>();
        perH.forEach((h, c) -> counts.add("H" + h + " " + c));
        System.out.println("Problems per heuristic: " + String.join(", ", counts));
        double lambda = (double) findings.size() / (n * ranked.size());
        System.out.printf("Each evaluator found %.1f%% of the problems "
                          + "on average (lambda).%n", 100 * lambda);
        System.out.println("Predicted share found by i evaluators:");
        List<String> shares = new ArrayList<>();
        for (int i = 1; i <= 6; i++) {
            shares.add(String.format("%d: %.0f%%", i, 100 * foundShare(lambda, i)));
        }
        System.out.println("  " + String.join("  ", shares));
        System.out.println("Evaluators needed for 90%: "
                           + evaluatorsNeeded(lambda, 0.90));
    }
}
```

Two points of the Java version deserve a note. A `record` is a compact class whose fields are fixed at construction; `Problem` holds an identifier, a heuristic, a location and a list of severities, and adds a method `mean`. Because the descriptions contain quoted commas, splitting at every comma would break the rows, so the method `cells` walks through each line character by character, switching a flag when it meets a quotation mark and treating two quotation marks inside quotes as one literal quote, which is the CSV rule. The sort uses a chain of comparators that mirrors the Python tuple.

3.8 Running It

```
$ python3 heuristics.py
$ java -cp out Heuristics
```

```
Heuristic evaluation: 4 evaluators, 14 problems, 26 findings
rank id   heur found mean max location
   1 P01  H1   3/4   3.7   4  Send button
   2 P11  H1   4/4   3.5   4  Total price
   3 P08  H3   2/4   3.0   3  After sending
   4 P02  H2   3/4   2.7   3  Service list
   5 P03  H5   2/4   2.5   3  Pickup date
   6 P04  H9   2/4   2.5   3  Error message
   7 P14  H8   2/4   2.5   3  Small print
   8 P12  H5   2/4   2.0   2  Phone number
   9 P05  H6   1/4   2.0   2  Prices
  10 P07  H8   1/4   2.0   2  Promo banner
  11 P09  H7   1/4   2.0   2  Regular customers
  12 P13  H4   1/4   2.0   2  Date format
  13 P06  H4   1/4   1.0   1  Buttons
  14 P10  H10  1/4   1.0   1  Weight
Fix before launch (mean severity 3 or more): P01, P11, P08
Problems per heuristic: H1 2, H2 1, H3 1, H4 2, H5 2, H6 1, H7 1, H8 2, H9 1, H10 1
Each evaluator found 46.4% of the problems on average (lambda).
Predicted share found by i evaluators:
  1: 46%  2: 71%  3: 85%  4: 92%  5: 96%  6: 98%
Evaluators needed for 90%: 4
```

The report tells the developer where to start. Three problems have a mean severity of 3 or more: P01, no confirmation after sending (found by three of four evaluators, mean 3.7); P11, the missing total price (found by all four); and P08, no way to change or cancel an order. These are exactly the problems that produced duplicate orders and the WhatsApp questions of the first week. Next come the codes in the service list (P02), the impossible dates (P03), the "Error 422" message (P04) and the light grey small print (P14). Only one problem, the missing total price, was found by every evaluator, and seven were found by only one: a vivid demonstration of why several independent evaluators are needed.

The estimate $\hat{\lambda} = 26/(4 \times 14) = 0.464$ is higher than Nielsen and Landauer's average of 0.31, which is plausible for a small page with obvious problems; it predicts that four evaluators find about 92% of the problems, and that four are needed for 90%. Because the estimate treats the fourteen found problems as all there are, the true share is somewhat lower, and a fifth evaluator would be a sensible investment for a larger product.

The developer fixed the problems in order of the report. The right-hand page of **Figure 3.1** shows the result: the services are visible tiles with prices in the customer's language (P02, P05), the weight and the minimum are explained where they are entered (P10), the pickup day is a day of the week with the opening hours (P03, P13), the phone field shows an example (P12), the estimated total is shown before sending (P11), there is one clearly labeled send button (P06), and after sending the page states the order number, the pickup day and until when the order can be changed (P01, P08). The promotion and the grey small print are gone (P07, P14). Chapter 7 will show how to check with real customers that the new page is better; the heuristic evaluation made sure that the obvious problems were gone before they were asked.

3.9 Testing It

Listing 3.5 checks the model of Equation (3.1) and Equation (3.3) on a case that is easy by hand ($\lambda = 0.5$: two evaluators find $1 - 0.25 = 75\%$; for 90%, $\ln 0.1 / \ln 0.5 = 3.32$, so four) and checks that ranking puts the more severe problem first.

Listing 3.5 `test_heuristics.py`: tests of the model and of the ranking.

```
"""Tests for the evaluation report: python -m unittest test_heuristics.py"""
import unittest
from heuristics import merge, rank, found_share, evaluators_needed


class HeuristicsTests(unittest.TestCase):
    def test_found_share(self):
        self.assertAlmostEqual(found_share(0.5, 2), 0.75)

    def test_evaluators_needed(self):
        self.assertEqual(evaluators_needed(0.5, 0.90), 4)

    def test_rank_puts_the_more_severe_problem_first(self):
        problems = [{"id": "A"}, {"id": "B"}]
        findings = [{"id": "A", "severity": "2"}, {"id": "B", "severity": "4"}]
        self.assertEqual([p["id"] for p in rank(merge(problems, findings))],
                         ["B", "A"])


if __name__ == "__main__":
    unittest.main()
```

Listing 3.6 adds two checks of the CSV splitting, which Python's `csv` module does for us.

Listing 3.6 `HeuristicsTest.java`: the same checks in Java, plus CSV splitting.

```
import java.util.List;

/** Tests for the heuristic evaluation report: java -cp out HeuristicsTest */
public class HeuristicsTest {
    static int passed = 0;

    static void expect(boolean ok, String what) {
        if (!ok) throw new AssertionError(what);
        passed++;
    }

    public static void main(String[] args) throws Exception {
        expect(Heuristics.cells("P02,H2,x,\"a, b\"").get(3).equals("a, b"),
               "a quoted comma does not split");
        expect(Heuristics.cells("\"say \"\"hi\"\"\"").get(0).equals("say \"hi\""),
               "doubled quotes");
        expect(Math.abs(Heuristics.foundShare(0.5, 2) - 0.75) < 1e-12,
               "two evaluators at lambda 0.5 find 75%");
        expect(Heuristics.evaluatorsNeeded(0.5, 0.90) == 4, "0.5 needs 4 for 90%");
        var ranked = Heuristics.rank(
            List.of(List.of("A", "H1", "a"), List.of("B", "H2", "b")),
            List.of(List.of("E1", "A", "2"), List.of("E1", "B", "4")));
        expect(ranked.get(0).id().equals("B"), "the more severe problem first");
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
$ python3 -m unittest test_heuristics.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
$ java -cp out HeuristicsTest
All 5 checks passed.
```

3.10 Using It for Real

To run a heuristic evaluation of your own product, whether a warung's order page, a puskesmas kiosk or a campus app, you need three to five evaluators, two hours each, and the template of **Table 3.3**. Evaluators need not be professional designers; students who have read this chapter find most of the problems a professional finds, though professionals with knowledge of the domain find more [2].

Table 3.3 A recording template for heuristic evaluation.

Field	What to write	Example
Evaluator	Initials or code	E2
Location	Screen and element	Order page, pickup date field
Problem	What is wrong, observed, in one sentence	Accepts Sundays, when the shop is closed
Heuristic	Code from **Table 3.1**	H5
Severity	0 to 4 from **Table 3.2**	3
Suggestion	A possible fix, optional	Offer only opening days

After the debriefing, write the agreed problems into `problems.csv` and each evaluator's ratings into `findings.csv`, and run the program. Three habits make the report more useful. Describe problems as observations, not solutions ("no confirmation after sending", not "add a toast"), so that developers can choose the best fix. Record positive findings too, so that good parts are not "fixed" away. And repeat the evaluation after the fixes, because fixes introduce new problems.

For AI features, the same method works with an extended list. The guidelines for human-AI interaction of Amershi et al. (Chapter 1) can be used as additional heuristics, such as "make clear what the system can do" and "support efficient correction"; Chapter 13 does exactly that.

3.11 HCI in Practice: Profitina

The real Profitina product shows several heuristics as deliberate features. *Visibility of system status* (H1): posting in the community is optimistic, so the composer clears at once and the post appears immediately, and if sending fails the text is restored with a clear message rather than lost. *Error prevention* (H5) and *help users recover from errors* (H9) appear in Link Safety: every link posted in the community is screened, and a risky link is flagged with an explanation of why, which prevents the error of clicking a scam link and tells the user what to do instead. *User control and freedom* (H3) is built into the account screens required by UU PDP, where data export, password change and full account deletion are self-service actions. And *match with the real world* (H2) is the reason the dashboard explains each AI-visibility signal in plain language rather than in the technical vocabulary of structured data and crawlers.

Channel creation shows *error prevention* through constraints: a user may create at most two channels a day and have five active, and the rule is stated where the button is, so that the limit is a known constraint rather than a surprise.

3.12 Common Mistakes, and How to Avoid Them

1. **Evaluating together from the start.** Evaluators who work as a group find fewer problems. Evaluate independently, then merge.

2. **Using one evaluator.** One person finds roughly a third to a half of the problems. Use three to five.
3. **Writing solutions instead of problems.** "Add a confirmation dialog" hides the problem and fixes the design too early. Write what is wrong and where.
4. **Rating severity alone.** Ratings vary between evaluators; average several independent ratings.
5. **Treating a heuristic evaluation as a user test.** It finds what experts foresee; test with real users as well (Chapter 7).
6. **Following a design system blindly.** A style such as translucent glass can reduce contrast; check every choice against the principles.
7. **Forcing every problem into a heuristic.** Some problems fit none; record them anyway with "other".
8. **Stopping after one round.** Fixes create new problems; evaluate again after the changes.

3.13 Chapter Summary

- Principles are general (why), guidelines are specific to a platform (what), heuristics are rules of thumb for evaluation (check).
- Norman's principles: affordances and signifiers, natural mapping (**Figure 3.2**), constraints and forcing functions (the ATM returns the card before the cash), feedback and a clear conceptual model, together giving discoverability.
- Shneiderman's eight golden rules, the seven interaction principles of ISO 9241-110:2020 and Nielsen's ten heuristics (**Table 3.1**) largely agree.
- Liquid Glass, Material 3 Expressive and Fluent 2 apply the principles and sometimes strain them; Apple's *Tinted* option of iOS 26.1 restored legibility by giving users control.
- Heuristic evaluation: independent evaluation, debriefing, severity ratings 0 to 4 (**Table 3.2**), ranked report. Nielsen–Landauer: i evaluators find $1-(1-\lambda)^i$ of the problems (Equation (3.1)).
- For Profitina Laundry's first order page, four evaluators found 14 problems ($\hat{\lambda} = 0.464$); three of them had a mean severity of 3 or more and caused the duplicate orders and questions of the first week.

3.14 Exercises

1. Assign each problem to one of Nielsen's heuristics and give a severity with a one-line reason: (a) an ojek app shows no driver position after the booking is accepted; (b) a bank app logs the user out without warning after two minutes while they type a transfer description; (c) a campus portal uses the word "Submit" on one page and "Kirim" on the next for the same action; (d) a warung's QRIS page shows the amount in small grey text under a large advertisement.
2. Using Equation (3.3), how many evaluators are needed to find 75% and 90% of the problems when $\lambda = 0.31$? Show the calculation.
3. A fifth evaluator E5 reviews the first order page and reports P01 (severity 4), P05 (3), P11 (3), P13 (2) and P09 (2). Add these findings to a copy of the data folder and run the program. Which lines of the report change, and why does $\hat{\lambda}$ go down although E5 found nothing new?
4. The ATM of Section 3.3.3 prevents a post-completion error with a forcing function. Find a post-completion error in another everyday system in Indonesia (for example, a fuel pump, a

photocopier at a campus copy shop, or a mobile top-up) and propose a forcing function or another fix.

5. Explain, in the terms of Section 3.5, how Liquid Glass serves one heuristic and strains another, and why Apple's *Tinted* option is an application of H3 and H7.
6. Compare the two pages of **Figure 3.1**. For each of the fourteen problems of **Listing 3.1**, say whether the improved page fixes it, and name the problems that a screenshot alone cannot confirm.
7. Extend `heuristics.py` so that it also prints, for every evaluator, how many problems they found and their share of all problems. What does it print for the original data?

3.15 Solutions to the Exercises

1. (a) H1, visibility of system status; severity 3, because the passenger cannot tell whether a driver is coming and may cancel or book twice. (b) H3, user control and freedom (also H1, since there is no warning); severity 3, because the typed transfer is lost and the task must be repeated; a warning with an option to stay signed in fixes it. (c) H4, consistency and standards; severity 1 or 2, a cosmetic inconsistency that can make users wonder whether the actions differ. (d) H8, aesthetic and minimalist design (and H1, since the amount is the status that matters); severity 3, because a payer who cannot read the amount cannot check it before paying.

2. For 75%: $i \geq \ln(1-0.75)/\ln(1-0.31) = \ln 0.25/\ln 0.69 = (-1.386)/(-0.371) = 3.74$, so 4 evaluators. For 90%: $i \geq \ln 0.10/\ln 0.69 = (-2.303)/(-0.371) = 6.21$, so 7 evaluators.

3. Copy the folder and append the five lines `E5,P01,4`, `E5,P05,3`, `E5,P11,3`, `E5,P13,2`, `E5,P09,2` to `findings.csv`. The program prints:

```
Heuristic evaluation: 5 evaluators, 14 problems, 31 findings
rank id   heur found mean max location
   1 P01  H1   4/5   3.8   4  Send button
   2 P11  H1   5/5   3.4   4  Total price
   3 P08  H3   2/5   3.0   3  After sending
   4 P02  H2   3/5   2.7   3  Service list
   5 P03  H5   2/5   2.5   3  Pickup date
   6 P04  H9   2/5   2.5   3  Error message
   7 P05  H6   2/5   2.5   3  Prices
   8 P14  H8   2/5   2.5   3  Small print
   9 P09  H7   2/5   2.0   2  Regular customers
  10 P12  H5   2/5   2.0   2  Phone number
  11 P13  H4   2/5   2.0   2  Date format
  12 P07  H8   1/5   2.0   2  Promo banner
  13 P06  H4   1/5   1.0   1  Buttons
  14 P10  H10  1/5   1.0   1  Weight
Fix before launch (mean severity 3 or more): P01, P11, P08
Problems per heuristic: H1 2, H2 1, H3 1, H4 2, H5 2, H6 1, H7 1, H8 2, H9 1, H10 1
Each evaluator found 44.3% of the problems on average (lambda).
Predicted share found by i evaluators:
  1: 44%  2: 69%  3: 83%  4: 90%  5: 95%  6: 97%
Evaluators needed for 90%: 4
```

Every "found" column now counts out of 5; P01's mean rises to 3.8, P11's falls to 3.4, and P05 rises from rank 9 to rank 7 with a mean of 2.5. $\hat{\lambda}$ falls from 0.464 to $31/(5 \times 14) = 0.443$ because E5 found five of fourteen problems (36%), fewer than the average of the others. The number of evaluators needed for 90% stays at four. E5 found no new problem, which is itself evidence that the list of fourteen is close to complete.

4. Example: a campus copy shop's self-service photocopier. The goal is the copies; after taking them, people often leave the original on the glass, a post-completion error. A forcing function: the machine does not release the last copies until the lid is lifted, or it scans from a feeder that ejects the original together with the copies. Another example is the fuel pump that is paid for after fueling while the fuel cap is left open; a car that cannot start with the cap open would be a forcing function, and a cap tethered to the car is a constraint that prevents losing it.

5. Liquid Glass serves H8, aesthetic and minimalist design, because controls recede and the content becomes the focus. It strains perceivability and H1: text and controls on a translucent layer over busy content can lose contrast, so the status they show is harder to read, especially for older eyes. The *Tinted* option lets users choose a more opaque, higher-contrast version: that is user control (H3), and it is flexibility (H7), because users who need legibility can adapt the interface without affecting those who prefer the clear style.

6. Fixed and visible: P02 (plain service names), P03 and P13 (a named day with opening hours), P05 (prices on the tiles), P06 (one send button), P07 (no promotion), P10 (weighing and minimum explained), P11 (estimated total), P12 (example number), P14 (no grey small print), P01 (confirmation with order number). P08 is partly fixed: the confirmation says the order can be changed until 07:00, but a screenshot cannot show whether the change function works. A screenshot also cannot confirm P03's constraint (whether Sundays really cannot be chosen), P04 (what error messages now look like; no error is shown), or P09 (a repeat-order shortcut for regular customers, which is not visible and was probably not added). Those need a walk-through of the working page.

7. Add to `main`, after reading the findings:

```
    for e in evaluators:
        k = sum(1 for f in findings if f["evaluator"] == e)
        print(f"{e}: {k} problems ({100 * k / len(problems):.0f}%)")
```

For the original data it prints `E1: 8 problems (57%)`, `E2: 6 problems (43%)`, `E3: 5 problems (36%)` and `E4: 7 problems (50%)`. The mean of the four shares is 46.4%, which is $\hat{\lambda}$.

References

[1] J. Nielsen, "10 Usability Heuristics for User Interface Design," Nielsen Norman Group, 1994, updated. [Online]. Available: https://www.nngroup.com/articles/ten-usability-heuristics/

[2] J. Nielsen and R. Molich, "Heuristic evaluation of user interfaces," in *Proc. ACM CHI '90 Conference on Human Factors in Computing Systems*, Seattle, WA, USA, 1990, pp. 249–256.

[3] Nielsen Norman Group, "How to Conduct a Heuristic Evaluation," updated, accessed Sep. 2026. [Online]. Available: https://www.nngroup.com/articles/how-to-conduct-a-heuristic-evaluation/

[4] D. A. Norman, *The Design of Everyday Things*, revised and expanded ed. New York, NY, USA: Basic Books, 2013.

[5] M. D. Byrne and S. Bovair, "A working memory model of a common procedural error," *Cognitive Science*, vol. 21, no. 1, pp. 31–61, 1997.

[6] B. Shneiderman, C. Plaisant, M. Cohen, S. Jacobs, N. Elmqvist, and N. Diakopoulos, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 6th ed. Boston, MA, USA: Pearson, 2016.

[7] International Organization for Standardization, *ISO 9241-110:2020 Ergonomics of human-system interaction — Part 110: Interaction principles*. Geneva, Switzerland: ISO, 2020. [Online]. Available: https://www.iso.org/standard/75258.html

[8] Apple Inc., “Apple introduces a delightful and elegant new software design,” Apple Newsroom, Jun. 9, 2025. [Online]. Available: https://www.apple.com/newsroom/2025/06/apple-introduces-a-delightful-and-elegant-new-software-design/

[9] MacRumors, “Here’s How the iOS 26.1 Transparency Toggle Changes Liquid Glass,” Oct. 20, 2025. [Online]. Available: https://www.macrumors.com/2025/10/20/ios-26-1-transparency-option-liquid-glass/

[10] Google Design, “Expressive Design: Google’s UX Research,” accessed Sep. 2026. [Online]. Available: https://design.google/library/expressive-material-design-google-research

[11] Microsoft, “Fluent 2 Design System,” accessed Sep. 2026. [Online]. Available: https://fluent2.microsoft.design/

[12] J. Nielsen, “Severity Ratings for Usability Problems,” Nielsen Norman Group, 1994. [Online]. Available: https://www.nngroup.com/articles/how-to-rate-the-severity-of-usability-problems/

[13] J. Nielsen and T. K. Landauer, “A mathematical model of the finding of usability problems,” in *Proc. INTERCHI ’93 Conference on Human Factors in Computing Systems*, Amsterdam, The Netherlands, 1993, pp. 206–213.

[14] M. Hertzum and N. E. Jacobsen, “The evaluator effect: A chilling fact about usability evaluation methods,” *International Journal of Human-Computer Interaction*, vol. 13, no. 4, pp. 421–443, 2001.

Chapter 4

User-Centered and Design-Thinking Processes

Knowing the principles of good design (Chapter 3) and the limits of the people who will use it (Chapter 2) is not enough to build the right thing. Many products that follow every heuristic still fail because they solve a problem that the users do not have, or solve it in a way that does not fit their lives. User-centered design is the discipline of organizing a project so that this does not happen: understanding people and their context first, turning that understanding into requirements, designing, and testing the design with those people, again and again. This chapter explains the human-centred design process of ISO 9241-210, the Double Diamond of the UK Design Council and the five modes of design thinking; gives templates for the main working tools, from the affinity diagram and the persona to the journey map, the "how might we" question and the impact-effort matrix; discusses what AI tools can and cannot do in the process in 2026; and builds a small program in Java and in Python that turns a pile of interview notes into themes, a journey with its pain points, design questions and a prioritized list of ideas. By the end of the chapter you will be able to plan a user-centered project, run its synthesis step with real notes, and write measurable user requirements.

4.1 An Ojek Base Wants an App

At the edge of the market of a small town in East Java there is a *pangkalan ojek*, a motorcycle-taxi base: a shelter, a bench, a phone number painted on a board, and about twenty drivers who take turns. Pak Darto, the chairman of the drivers' cooperative, has watched the large ride-hailing apps take away the young passengers and the long trips. His regular passengers are still loyal: market traders, schoolchildren, nurses on the night shift, and many older people who call the base, or ask a grandchild to call, when they need a ride to the market, the puskesmas or the mosque.

Pak Darto asks a team of informatics students for "an app like the big ones, but ours". The students' first instinct is to start drawing screens: a map, a car icon, a booking button, a rating. Their lecturer stops them with a question: *which problem of which person would that map solve?* Nobody knows. The big apps have maps because their passengers are in unfamiliar places with GPS-enabled phones and their drivers do not know them; the base's passengers are mostly regulars, many of whom cannot read a map, and the drivers know the town by its mosques and warungs rather than by street names.

So the team does something slower and much more useful. They sit at the base for three mornings, talk with six passengers and four drivers, write every observation on a sticky note, and only then start to ask what to build. What they find (Section 4.8) is that the most painful moments are not in finding a driver at all, but in *waiting without knowing* and in *agreeing on the price*, and that the drivers' own biggest worry is not the app but the fairness of the turn order at the base. The product they end up proposing is far smaller than "an app like the big ones": a fixed fare table shown before booking, a one-tap "driver on the way" message with the driver's name and photo, a share-my-ride link for night trips, and later an automatic fair queue. It costs a fraction of the original idea and addresses what the people actually said.

This chapter describes the process they followed. The companion program at the end does the bookkeeping of their synthesis step: it groups the notes into themes, measures the pain at each stage of the journey, formulates the design questions, and sorts the ideas by impact and effort.

4.2 The Idea in Plain Words

The idea of user-centered design fits in three sentences written by Gould and Lewis in 1985, which have not been improved since: *focus early on users and tasks*, *measure empirically* how people use simulations and prototypes, and *design iteratively*, fixing what the measurements reveal and measuring again [1]. Every modern process, whatever it is called, is a way of organizing those three habits.

> **DEFINITION.** *User-centered design* (UCD), called *human-centred design* in the ISO standard, is an approach to developing interactive systems that aims to make them usable and useful by focusing on the users, their needs and requirements, and by applying human factors, usability knowledge and techniques [2]. Its core is a cycle of understanding the context of use, specifying user requirements, producing design solutions and evaluating them against the requirements, repeated until the design meets the requirements.

The opposite of user-centered design is not “bad design” but *designing from the inside out*: starting from a technology (“we have a map API”), from a competitor (“the big apps have ratings”), or from the wishes of whoever pays (“an app like the big ones”). Those starting points are not forbidden, but they must be checked against the people who will use the result. Pak Darto knows his passengers far better than the students do, and he is an essential participant; but his *request* is a proposed solution, and the process begins by understanding the problem behind it.

4.3 The Human-Centred Design Process of ISO 9241-210

The international standard ISO 9241-210:2019 describes human-centred design as six principles and four activities [2]. The principles, in **Table 4.1**, say what must be true of a project; the activities say what the team does.

Table 4.1 The six principles of human-centred design in ISO 9241-210:2019, with what they mean at the ojek base.

Principle	What it means in practice	At the ojek base
The design is based upon an explicit understanding of users, tasks and environments	Write down who the users are, what they do and where	Passengers of all ages, many without map skills; drivers at the base; the market street
Users are involved throughout design and development	Not once at the start, but at every stage	Passengers and drivers comment on sketches and test prototypes
The design is driven and refined by user-centred evaluation	Decisions are tested with users, not argued	The fare table is tried with five regular passengers before printing
The process is iterative	Expect to go back; plan for it	The first “driver on the way” message is too long and is shortened
The design addresses the whole user experience	Before, during and after use, including feelings	Feeling safe on night rides matters as much as booking speed
The design team includes multidisciplinary skills and perspectives	Not only programmers	A driver and a passenger join the design sessions; a nurse advises on night trips

The four activities form the cycle of **Figure 4.1**: understand and specify the context of use; specify the user requirements; produce design solutions; and evaluate the designs against the

requirements. When the evaluation shows that the requirements are met, the cycle ends; when it does not, the team goes back to whichever activity the evaluation points to, which may be the very first one, because an evaluation can reveal that the context was misunderstood.

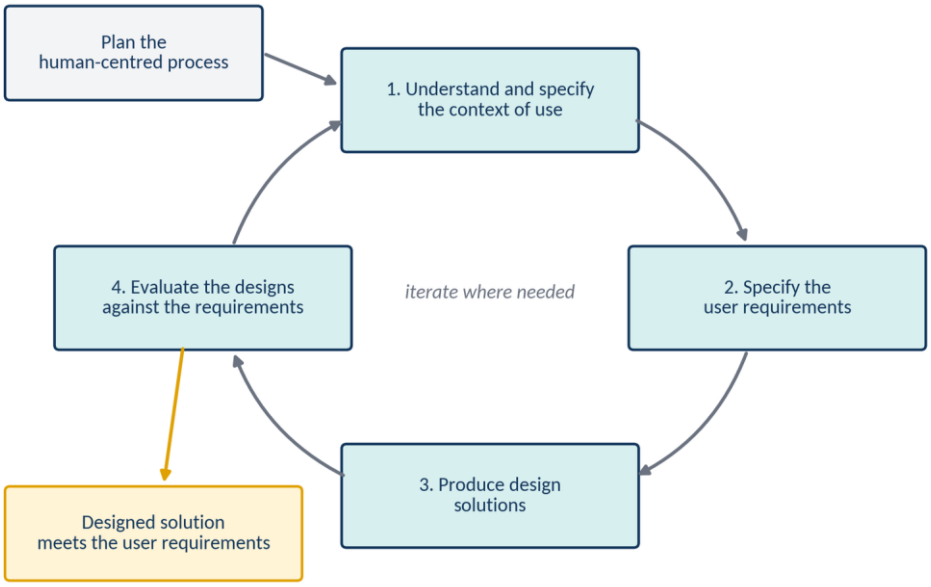


Figure 4.1 The human-centred design activities of ISO 9241-210: plan, then cycle through context, requirements, design and evaluation until the solution meets the user requirements.

The *context of use* is the combination of users, goals, tasks, resources and environments in which a system is used, the same "specified" context of the usability definition in Chapter 1. For the ojek base it includes the passengers who stand at a gate in the sun, the drivers who ride with one hand on a phone they should not touch while riding, the cheap Android phones with limited data, and the market's noise. *User requirements* are statements of what users must be able to do with the system and how well, written so that they can be tested (Section 4.5.7).

4.4 The Double Diamond and Design Thinking

4.4.1 The Double Diamond

In 2004 the UK Design Council studied how leading companies design and summarized their practice as the *Double Diamond*, shown in **Figure 4.2** [3]. The shape encodes an important idea: a design project thinks twice in two directions. In the first diamond, the team *diverges* to discover as much as possible about the people and the problem, then *converges* to define the right problem. In the second diamond, it diverges again to develop many possible solutions, and converges to deliver the one that works, after testing and refining.

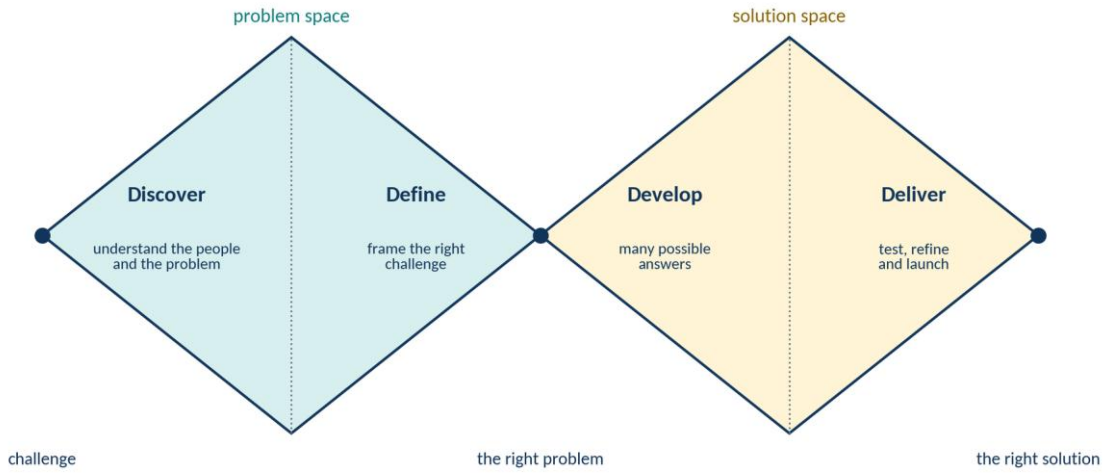


Figure 4.2 The Double Diamond: discover and define in the problem space, develop and deliver in the solution space; each diamond first widens, then narrows.

The most common mistake the diamond warns against is skipping the first diamond: jumping from the brief ("an app like the big ones") straight into developing solutions. The students' lecturer did exactly this when she stopped the screen drawing. In 2019 the Design Council republished the diamond as its *Framework for Innovation*, adding four principles (put people first; communicate visually and inclusively; collaborate and co-create; iterate, iterate, iterate) and a bank of methods, and in 2021 it extended the approach with a *Systemic Design Framework* for problems such as climate and public health that no single product can solve [3], [4].

4.4.2 Design thinking

Design thinking is the name under which these ideas spread beyond design studios into business and education, largely through IDEO and the Hasso Plattner Institute of Design at Stanford (the d.school). Tim Brown of IDEO described it as bringing the designer's methods to the whole of innovation, matching people's needs with what is technologically feasible and economically viable [5]. The d.school describes five *modes*: **empathize** (understand people through observation and conversation), **define** (state the core problem as a point of view), **ideate** (generate many ideas), **prototype** (build quick, cheap things that can be tried) and **test** (try them with people and learn) [6].

The modes are not a one-way sequence. A test often sends a team back to empathize; a prototype often clarifies the definition. The d.school's phrase "a bias toward action" names a real strength of the approach: build something small and learn from it, rather than discuss in the abstract.

4.4.3 Lean UX and agile teams

Most software today is built by agile teams in short cycles, which at first seems to leave no room for research. *Lean UX* reconciles the two: it treats every design decision as a hypothesis ("we believe that showing the fare before booking will reduce haggling; we will know we are right when fewer passengers ask the price by phone"), builds the smallest thing that tests it, and measures the outcome [7]. Many teams run *dual-track* development, in which a discovery track of research and prototyping runs one or two cycles ahead of the delivery track that builds the software.

Table 4.2 lines up the three descriptions. They are not competitors but different vocabularies for the same underlying cycle; a team should choose one vocabulary so that everyone speaks the same language, and know that the others are translations.

Table 4.2 The same cycle in three vocabularies, with typical methods for each step.

ISO 9241-210 activity	Double Diamond	Design thinking mode	Typical methods
Understand and specify the context of use	Discover	Empathize	Interviews, observation, diary studies (Chapter 5)
Specify the user requirements	Define	Define	Affinity diagram, personas, journey map, problem statement, measurable requirements
Produce design solutions	Develop	Ideate, prototype	Brainstorming, sketching, paper and interactive prototypes (Chapter 6)
Evaluate the designs against the requirements	Deliver (test and refine)	Test	Heuristic evaluation (Chapter 3), usability testing (Chapter 7), A/B tests (Chapter 14)

4.5 The Working Tools, with Templates

This section describes the tools the ojek team used between the interviews and the first prototype. Chapter 5 treats the interviews themselves; here we start from the notes.

4.5.1 The affinity diagram

An *affinity diagram* organizes many observations into groups by similarity, from the bottom up. Each observation goes on one note, written as a fact in the participant's terms ("Waits at the gate not knowing if anyone is coming"), not as an interpretation ("needs tracking"). The team then moves notes that seem to belong together next to each other, without deciding the categories in advance, and finally names each group with a short label [8]. The labels that emerge are the *themes* of the research. The ojek team's 37 notes formed eight themes, from "Unknown waiting time" to "Fair turns".

Two numbers describe a theme's weight: how many notes it contains, and how many *different people* contributed to it. The second is more important. Ten notes from one talkative passenger are one person's view; five notes from five people are a pattern. The *reach* of a theme t is the share of all participants who mentioned it,

$$r_t = \frac{|P_t|}{|P|} \tag{4.1}$$

where P_t is the set of participants with at least one note in theme t, P the set of all participants, and $|\cdot|$ the number of elements.

4.5.2 Personas

A *persona* is a short, concrete description of a typical user, built from research, that the team can keep in mind while designing: a name, a photo or drawing, the goals, the context, the frustrations and a characteristic quote [9]. Personas prevent the "elastic user" who bends to justify any feature; the question "would Mbah Sri use this?" has a concrete answer where "would users use this?" does not. **Table 4.3** shows one of the ojek team's personas, built from the notes of participant P4.

Table 4.3 A persona built from research notes, with the design implications it suggests.

Field	Content
Name and context	Mbah Sri, 72, sells vegetables at the market on weekdays; lives with her son's family two kilometres away
Goal	Get to the market by 5:30 and home at noon without depending on her son
Behaviour	Her son usually calls the base for her; knows places only by landmarks; prefers cash
Frustrations	Standing in the sun not knowing how long the driver will take; cannot read a map
Quote	"Just tell me who is coming and when. I know every driver here."
Design implication	No map needed; a clear message with the driver's name and arrival time; landmarks, not addresses

A persona is only as good as its evidence. A persona invented at a desk, or generated by an AI tool from general knowledge (Section 4.6), describes a stereotype, not the users.

4.5.3 The journey map

A *journey map* follows a user through the stages of a task over time and records, for each stage, what the person does, thinks and feels, where the pain is, and which *touchpoints* (phone call, message, shelter, payment) are involved. For the ojek base the stages are *Find* a driver, *Agree* on the place and the price, *Wait* for the driver, *Ride*, and *Pay*. The emotional line of the map, usually drawn as a curve through the stages, shows at a glance where the experience hurts. The team gave every note a pain score from 1 (mild) to 5 (severe); the mean pain of a stage s is

$$\bar{p}_s = \frac{1}{|N_s|} \sum_{n \in N_s} p_n \tag{4.2}$$

where N_s is the set of notes about stage s and p_n the pain score of note n. Section 4.8 draws the curve.

4.5.4 Empathy maps and jobs to be done

Two more tools are often used alongside. An *empathy map* arranges what a persona *says*, *thinks*, *does* and *feels* in four quadrants, which helps a team notice contradictions: Mbah Sri *says* she does not mind waiting, but *does* stand at the gate twenty minutes early. The *jobs-to-be-done* view asks what "job" a person "hires" a product to do: nobody wants an ojek app; people want to "get to the market before the good vegetables are gone" [10]. Framing a need as a job keeps the team from confusing the need with one particular solution.

4.5.5 Problem statements and "how might we" questions

The definition step ends with a *problem statement* that names the user, the need and the insight: "Regular passengers of the base, many of them older people, need to know when their driver will arrive, because waiting without knowing makes them leave home too early and stand in the sun." From it the team derives *how might we* (HMW) questions, which turn a problem into an invitation for ideas: "How might we show passengers when their driver will arrive?" A good HMW question is broad enough to allow many answers and narrow enough to give direction. "How might we build an app?" is too narrow (it names a solution) and says nothing about the problem; "How might we improve transport?" is too broad to act on.

4.5.6 Ideation and the impact-effort matrix

In ideation the team generates many answers to each HMW question without judging them, then selects. A simple, widely used selection tool is the *impact-effort matrix*. Each idea gets an impact score (how much it would reduce the pain found in the research, 1 to 5) and an effort score (how much work it takes, 1 to 5), agreed by the team, and is placed in one of four quadrants: *quick wins* (high impact, low effort), *big bets* (high impact, high effort), *fill-ins* (low impact, low effort) and ideas to *avoid for now* (low impact, high effort). Within a quadrant, the value for effort

$$v = \frac{I}{E} \tag{4.3}$$

with I the impact and E the effort score, orders the ideas, and team votes break ties. The scores are judgments, not measurements; the matrix does not decide for the team, it makes the team's reasoning visible and discussable.

4.5.7 Measurable user requirements

The requirements that close the define step must be testable, in the sense of usability in Chapter 1: user, task, measure and target. Examples for the ojek base: "A regular passenger aged 60 or over books a ride from the saved link in under 30 seconds without help, in 4 of 5 attempts"; "after booking, 9 of 10 passengers can say the driver's name and the expected arrival time"; "in a month of use, calls to the base asking where the driver is fall by half". Each requirement can be checked in a usability test (Chapter 7) or with data (Chapter 14), which is what makes the evaluation step of **Figure 4.1** possible.

4.6 AI in the Process, 2026

AI tools changed the daily work of design teams in 2025 and 2026. At Config 2026, Figma announced an AI agent inside its design canvas with web search, custom skills and connections to external tools, alongside code layers that bring working code onto the canvas [11]; similar assistants exist in other tools. They are genuinely useful for the *develop* half of the Double Diamond: generating layout variants, drafting copy, turning a sketch into a clickable prototype, summarizing a long transcript. Chapter 6 uses them for prototyping.

They are much weaker, and sometimes harmful, in the *discover* half. Tools that offer "synthetic users", AI-generated personas or interview answers, produce plausible statements that reflect what is typical in their training data, not what is true of a particular group of people. The Nielsen Norman Group's review concludes that such output can help prepare research, for example by suggesting questions to ask, but cannot replace research with real people [12]. No language model would have told the ojek team that the drivers' main worry was the fairness of the turn order at their base, or that Mbah Sri knows every driver by name. The rule for this book is simple: AI may help you *work with* what real people told you; it may not *invent* what they would have said.

Working with real people also means working with their personal data. Interview notes, recordings and photos of participants are personal data under Law No. 27 of 2022 on Personal Data Protection, and processing them requires a lawful basis such as explicit consent for a specific purpose [13]; the implementing regulation, Government Regulation No. 33 of 2026, takes effect in January 2027 [14]. In practice: tell participants what the research is for, ask for consent to take notes or record, use codes (P1, D1) rather than names in shared files, and delete recordings when

the analysis is done. Pasting raw transcripts with names into a cloud AI tool is a transfer of personal data to a third party and needs the same care.

4.7 Building the Synthesis Tool

The affinity diagram is made by people moving notes on a wall or on a digital whiteboard; that part stays human, because deciding which notes belong together is interpretation. What comes after, counting notes and people per theme, averaging pain per stage, writing out the HMW questions for the strongest themes, and sorting ideas, is bookkeeping, and the program of this section does it so that the numbers are right and can be recomputed whenever a note is moved.

4.7.1 The data

Three CSV files hold the team's work. **Listing 4.1** lists the eight themes with a short label and the "how might we" phrase agreed for each.

Listing 4.1 `data/themes.csv`: the themes of the affinity diagram and their "how might we" phrases.

```
theme,label,hmw
T1,Unclear price,make the fare clear before the ride starts
T2,Unknown waiting time,show passengers when their driver will arrive
T3,Calling the base,let passengers book without a phone call
T4,Explaining the place,help passengers say where they are without street names
T5,Safety at night,help passengers feel safe on night rides
T6,Big apps feel hard,keep booking as simple as asking a neighbour
T7,Cash and change,make paying easy without exact change
T8,Fair turns,keep the base's turn order fair for every driver
```

Listing 4.2 holds the 37 notes: an identifier, the participant's code (P for passengers, D for drivers), the role, the journey stage, the theme the note was placed in, the pain score and the note itself.

Listing 4.2 `data/notes.csv`: 37 notes from six passengers and four drivers, each with a stage, a theme and a pain score.

```
id,participant,role,stage,theme,pain,note
N01,P1,passenger,Find,T3,4,Calls the base but the phone is busy at 7 am
N02,P1,passenger,Agree,T1,4,Never sure the price is the same as last time
N03,P1,passenger,Wait,T2,5,Waits at the gate not knowing if anyone is coming
N04,P2,passenger,Find,T3,3,Does not have the base number saved on the new phone
N05,P2,passenger,Agree,T4,4,"Explains ""after the mosque, green fence"" twice"
N06,P2,passenger,Pay,T7,3,Driver has no change for a 50 thousand note
N07,P3,passenger,Find,T6,3,Tried a big app; too many screens and promotions
N08,P3,passenger,Wait,T2,4,Asks a neighbour to call again after 15 minutes
N09,P3,passenger,Ride,T5,4,Coming home from the night shift feels unsafe
N10,P3,passenger,Agree,T1,3,Price for the night differs and nobody explains why
N11,P4,passenger,Find,T3,2,Her son usually calls the base for her
N12,P4,passenger,Agree,T4,5,Cannot read a map; knows only landmarks
N13,P4,passenger,Wait,T2,4,Stands in the sun because she does not know how long
N14,P4,passenger,Pay,T7,2,Prefers cash; does not trust QRIS yet
N15,P5,passenger,Find,T6,3,Uninstalled a big app after a surprise fee
N16,P5,passenger,Ride,T5,3,Wants to know the driver's name before getting on
N17,P5,passenger,Pay,T7,2,Would pay by QRIS if the amount is shown first
N18,P5,passenger,Agree,T1,4,Haggles every time; finds it tiring
N19,P6,passenger,Find,T3,3,Sends a WhatsApp voice note; reply comes late
N20,P6,passenger,Wait,T2,3,Wants a message when the driver leaves the base
N21,P6,passenger,Ride,T5,2,Likes riding with drivers she knows
N22,P6,passenger,Agree,T4,3,"Shares location, but the pin is on the wrong street"
N23,D1,driver,Find,T8,4,Arguments about whose turn it is when a call comes
N24,D1,driver,Agree,T4,4,Loses time finding houses without numbers
N25,D1,driver,Pay,T7,3,Needs to keep small notes for change
N26,D2,driver,Find,T6,4,Commission of big apps is too high for short trips
N27,D2,driver,Agree,T1,3,Passengers think the fare is made up on the spot
N28,D2,driver,Find,T8,3,Whoever holds the base phone gets the good trips
N29,D3,driver,Wait,T2,2,Would send a message if it were one tap
N30,D3,driver,Find,T3,3,Misses calls while riding
N31,D3,driver,Ride,T5,2,Wants passengers to know who he is
N32,D4,driver,Find,T8,5,Young drivers skip the queue with their own apps
N33,D4,driver,Agree,T4,3,Old customers explain places by old names
N34,D4,driver,Pay,T7,2,Would accept QRIS if the cooperative helps set it up
N35,P5,passenger,Wait,T2,4,Leaves the house far too early to be safe
N36,P2,passenger,Wait,T2,3,Keeps calling the base to ask where the driver is
N37,D3,driver,Agree,T1,3,Explains the fare again at almost every trip
```

Listing 4.3 lists the ten ideas of the ideation session with the team's impact and effort scores and the number of votes each received from passengers and drivers in a feedback session.

Listing 4.3 `data/ideas.csv`: ten ideas with impact, effort and votes.

```
idea,theme,impact,effort,votes
Fixed fare table by zone shown before booking,T1,5,2,9
Driver-on-the-way message with name and photo,T2,5,2,8
One-tap booking page from a saved WhatsApp link,T3,4,3,7
Pick the pickup point from photos of landmarks,T4,4,4,5
Share-my-ride link for family on night trips,T5,4,2,6
Automatic fair queue for the base,T8,5,3,7
Cooperative QRIS with the amount shown,T7,3,2,4
Own full ride-hailing app like the big ones,T6,3,5,2
Loyalty points for regular passengers,T6,2,3,3
Voice booking by phone with a menu,T3,3,4,4
```

4.7.2 The program

Listing 4.4 is the Python version.

Listing 4.4 `synthesis.py`: themes with their reach, pain per journey stage, "how might we" questions and prioritized ideas.

```
"""From interview notes to themes, a journey, questions and priorities."""
import csv
import sys

STAGES = ["Find", "Agree", "Wait", "Ride", "Pay"]
QUADRANTS = ["Quick win", "Big bet", "Fill-in", "Avoid for now"]


def read_csv(path):
    with open(path, newline="", encoding="utf-8") as f:
        return list(csv.DictReader(f))


def affinity(notes):
    """Group notes by theme: note count, distinct people, roles."""
    board = {}
    for n in notes:
        t = board.setdefault(n["theme"], {"notes": 0, "people": set(),
                                          "roles": set()})
        t["notes"] += 1
        t["people"].add(n["participant"])
        t["roles"].add(n["role"])
    return sorted(board.items(),
                  key=lambda kv: (-len(kv[1]["people"]), -kv[1]["notes"], kv[0]))


def journey(notes):
    """Mean pain (1 mild to 5 severe) and note count per journey stage."""
    result = []
    for stage in STAGES:
        pains = [int(n["pain"]) for n in notes if n["stage"] == stage]
        result.append((stage, len(pains), sum(pains) / len(pains)))
    return result


def quadrant(impact, effort):
    """The impact-effort matrix: high impact is 4 or 5, low effort 1 or 2."""
    if impact >= 4:
        return "Quick win" if effort <= 2 else "Big bet"
    return "Fill-in" if effort <= 2 else "Avoid for now"
```

Listing 4.4 (Continued) *synthesis.py: themes with their reach, pain per journey stage, "how might we" questions and prioritized ideas.*

```
def prioritise(ideas):
    """Quadrant first, then value for effort, then votes."""
    rows = []
    for i in ideas:
        impact, effort = int(i["impact"]), int(i["effort"])
        rows.append((quadrant(impact, effort), impact, effort,
                     impact / effort, int(i["votes"]), i["idea"]))
    return sorted(rows, key=lambda r: (QUADRANTS.index(r[0]), -r[3], -r[4]))


def main():
    folder = sys.argv[1] if len(sys.argv) > 1 else "data"
    notes = read_csv(f"{folder}/notes.csv")
    themes = {t["theme"]: t for t in read_csv(f"{folder}/themes.csv")}
    people = {n["participant"]: n["role"] for n in notes}
    roles = list(people.values())
    print(f"Affinity board: {len(notes)} notes from {len(people)} people "
          f"({roles.count('passenger')} passengers, "
          f"{roles.count('driver')} drivers)")
    print("theme label                  notes people roles")
    board = affinity(notes)
    for theme, t in board:
        who = "+".join(sorted(t["roles"], reverse=True))
        print(f"{theme:5} {themes[theme]['label']:22}{t['notes']:6}"
              f"{len(t['people']):7}  {who}")
    print("Journey: mean pain per stage (1 mild, 5 severe)")
    for stage, count, pain in journey(notes):
        print(f"  {stage:6}{count:3} notes  {pain:.1f}  "
              + "#" * round(2 * pain))
    print("How might we ...")
    for k, (theme, t) in enumerate(board[:3], 1):
        print(f"  {k}. ... {themes[theme]['hmw']}? "
              f"({theme}, {len(t['people'])} people)")
    print("Ideas by impact and effort")
    print("quadrant      imp eff ratio votes idea")
    for q, imp, eff, ratio, votes, idea in prioritise(
            read_csv(f"{folder}/ideas.csv")):
        print(f"{q:14}{imp:3}{eff:4}{ratio:6.2f}{votes:6}  {idea}")


if __name__ == "__main__":
    main()
```

`affinity` builds, for each theme, a count of notes and two sets, of participants and of roles; a set keeps each element once, so its size is the number of *different* people, the $|P_t|$ of Equation (4.1). Themes are sorted by the number of people, then by the number of notes, then by identifier. `journey` computes Equation (4.2) for the five stages in their fixed order. `quadrant` places an idea in the impact-effort matrix, with 4 or 5 counting as high impact and 1 or 2 as low effort, and `prioritise` sorts the ideas by quadrant, then by the value for effort of Equation (4.3), then by votes. `main` prints four blocks: the affinity board, the journey with a small bar of `#` characters (two per point of pain) as a text version of the emotional curve, the HMW questions for the three strongest themes, and the prioritized ideas.

Listing 4.5 is the Java version.

Listing 4.5 `Synthesis.java`: the same synthesis in Java.

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.TreeSet;

/** From interview notes to themes, a journey, questions and priorities. */
public class Synthesis {
    static final List<String> STAGES =
        List.of("Find", "Agree", "Wait", "Ride", "Pay");
    static final List<String> QUADRANTS =
        List.of("Quick win", "Big bet", "Fill-in", "Avoid for now");

    record Theme(String id, int notes, TreeSet<String> people,
                 TreeSet<String> roles) {}

    record Idea(String quadrant, int impact, int effort, double ratio, int votes,
                String text) {}

    /** A CSV file as maps from column name to text; "quoted, text" allowed. */
    static List<Map<String, String>> readCsv(String path) throws IOException {
        List<String> lines = Files.readAllLines(Path.of(path));
        String[] head = lines.get(0).split(",");
        List<Map<String, String>> rows = new ArrayList<>();
        for (String line : lines.subList(1, lines.size())) {
            List<String> cells = new ArrayList<>();
            StringBuilder cell = new StringBuilder();
            boolean quoted = false;
            for (int i = 0; i < line.length(); i++) {
                char c = line.charAt(i);
                if (c == '"' && quoted && i + 1 < line.length()
                        && line.charAt(i + 1) == '"') {
                    cell.append('"');
                    i++;
                } else if (c == '"') {
                    quoted = !quoted;
                } else if (c == ',' && !quoted) {
                    cells.add(cell.toString());
                    cell.setLength(0);
                } else {
                    cell.append(c);
```

Listing 4.5 (Continued) *Synthesis.java: the same synthesis in Java.*

```
                }
            }
            cells.add(cell.toString());
            Map<String, String> row = new HashMap<>();
            for (int i = 0; i < head.length; i++) row.put(head[i], cells.get(i));
            rows.add(row);
        }
        return rows;
    }

    /** Group notes by theme: note count, distinct people, roles. */
    static List<Theme> affinity(List<Map<String, String>> notes) {
        Map<String, Theme> board = new LinkedHashMap<>();
        for (var n : notes) {
            Theme t = board.computeIfAbsent(n.get("theme"),
                k -> new Theme(k, 0, new TreeSet<>(), new TreeSet<>()));
            t.people().add(n.get("participant"));
            t.roles().add(n.get("role"));
            board.put(t.id(),
                      new Theme(t.id(), t.notes() + 1, t.people(), t.roles()));
        }
        List<Theme> list = new ArrayList<>(board.values());
        list.sort(Comparator.comparingInt((Theme t) -> -t.people().size())
                  .thenComparingInt(t -> -t.notes()).thenComparing(Theme::id));
        return list;
    }

    /** The impact-effort matrix: high impact is 4 or 5, low effort 1 or 2. */
    static String quadrant(int impact, int effort) {
        if (impact >= 4) return effort <= 2 ? "Quick win" : "Big bet";
        return effort <= 2 ? "Fill-in" : "Avoid for now";
    }

    /** Quadrant first, then value for effort, then votes. */
    static List<Idea> prioritise(List<Map<String, String>> ideas) {
        List<Idea> rows = new ArrayList<>();
        for (var i : ideas) {
            int impact = Integer.parseInt(i.get("impact"));
            int effort = Integer.parseInt(i.get("effort"));
            rows.add(new Idea(quadrant(impact, effort), impact, effort,
                              (double) impact / effort,
                              Integer.parseInt(i.get("votes")), i.get("idea")));
        }
        rows.sort(Comparator
                  .comparingInt((Idea r) -> QUADRANTS.indexOf(r.quadrant()))
                  .thenComparingDouble(r -> -r.ratio())
```

Listing 4.5 (Continued) *Synthesis.java: the same synthesis in Java.*

```
                .thenComparingInt(r -> -r.votes()));
        return rows;
    }

    public static void main(String[] args) throws IOException {
        String folder = args.length > 0 ? args[0] : "data";
        var notes = readCsv(folder + "/notes.csv");
        Map<String, Map<String, String>> themes = new HashMap<>();
        for (var t : readCsv(folder + "/themes.csv")) themes.put(t.get("theme"), t);
        Map<String, String> people = new LinkedHashMap<>();
        for (var n : notes) people.put(n.get("participant"), n.get("role"));
        long passengers = people.values().stream()
                                .filter(r -> r.equals("passenger")).count();
        System.out.printf("Affinity board: %d notes from %d people "
                          + "(%d passengers, %d drivers)%n", notes.size(),
                          people.size(), passengers, people.size() - passengers);
        System.out.println("theme label                  notes people roles");
        List<Theme> board = affinity(notes);
        for (Theme t : board) {
            String who = String.join("+", t.roles().descendingSet());
            System.out.printf("%-5s %-22s%6d%7d  %s%n", t.id(),
                              themes.get(t.id()).get("label"), t.notes(),
                              t.people().size(), who);
        }
        System.out.println("Journey: mean pain per stage (1 mild, 5 severe)");
        for (String stage : STAGES) {
            int count = 0, sum = 0;
            for (var n : notes) {
                if (n.get("stage").equals(stage)) {
                    count++;
                    sum += Integer.parseInt(n.get("pain"));
                }
            }
            double pain = (double) sum / count;
            System.out.printf("  %-6s%3d notes  %.1f  %s%n", stage, count, pain,
                              "#".repeat((int) Math.round(2 * pain)));
        }
        System.out.println("How might we ...");
        for (int k = 0; k < 3; k++) {
            Theme t = board.get(k);
            System.out.printf("  %d. ... %s? (%s, %d people)%n", k + 1,
                              themes.get(t.id()).get("hmw"), t.id(),
                              t.people().size());
        }
        System.out.println("Ideas by impact and effort");
        System.out.println("quadrant      imp eff ratio votes idea");
```

Listing 4.5 (Continued) *Synthesis.java: the same synthesis in Java.*

```
        for (Idea r : prioritise(readCsv(folder + "/ideas.csv"))) {
            System.out.printf("%-14s%3d%4d%6.2f%6d  %s%n", r.quadrant(), r.impact(),
                              r.effort(), r.ratio(), r.votes(), r.text());
        }
    }
}
```

The Java version uses two records, `Theme` and `Idea`. Because a record's fields cannot be reassigned, `affinity` replaces a theme's record with a new one whose note count is one higher, while the two `TreeSet` objects inside are shared and grow in place; a `TreeSet` also keeps the roles sorted, so that `descendingSet` gives "passenger" before "driver", as Python's `sorted(..., reverse=True)` does. The CSV reader handles quoted fields, since two notes contain commas inside quotation marks.

4.8 Running It

```
$ python3 synthesis.py
$ java -cp out Synthesis
```

```
Affinity board: 37 notes from 10 people (6 passengers, 4 drivers)
theme label                 notes people roles
T2    Unknown waiting time      7      7  passenger+driver
T1    Unclear price             5      5  passenger+driver
T3    Calling the base          5      5  passenger+driver
T4    Explaining the place      5      5  passenger+driver
T7    Cash and change           5      5  passenger+driver
T5    Safety at night           4      4  passenger+driver
T6    Big apps feel hard        3      3  passenger+driver
T8    Fair turns                3      3  driver
Journey: mean pain per stage (1 mild, 5 severe)
  Find   11 notes  3.4  #######
  Agree  10 notes  3.6  #######
  Wait    7 notes  3.6  #######
  Ride    4 notes  2.8  ######
  Pay     5 notes  2.4  #####
How might we ...
  1. ... show passengers when their driver will arrive? (T2, 7 people)
  2. ... make the fare clear before the ride starts? (T1, 5 people)
  3. ... let passengers book without a phone call? (T3, 5 people)
Ideas by impact and effort
quadrant      imp eff ratio votes idea
Quick win       5   2  2.50     9  Fixed fare table by zone shown before booking
Quick win       5   2  2.50     8  Driver-on-the-way message with name and photo
Quick win       4   2  2.00     6  Share-my-ride link for family on night trips
Big bet         5   3  1.67     7  Automatic fair queue for the base
Big bet         4   3  1.33     7  One-tap booking page from a saved WhatsApp link
Big bet         4   4  1.00     5  Pick the pickup point from photos of landmarks
Fill-in         3   2  1.50     4  Cooperative QRIS with the amount shown
Avoid for now   3   4  0.75     4  Voice booking by phone with a menu
Avoid for now   2   3  0.67     3  Loyalty points for regular passengers
Avoid for now   3   5  0.60     2  Own full ride-hailing app like the big ones
```

The affinity board shows one theme clearly ahead: *unknown waiting time* was raised by seven of the ten people, a reach of $r = 7/10 = 70\%$ by Equation (4.1), and by drivers as well as passengers. Four themes follow with five people each (50%). *Fair turns* is raised only by drivers, which is exactly why it would have been missed if the team had interviewed only passengers.

The journey block, drawn as a curve in **Figure 4.3**, puts the pain in the first half of the journey. Agreeing on the place and price and waiting for the driver both average 3.6, and finding a driver 3.4; the ride itself (2.8) and paying (2.4) hurt least. For the stage *Agree*, for example, the ten pain scores are 4, 4, 3, 5, 4, 3, 4, 3, 3 and 3, whose sum is 36, so $\bar{p} = 36/10 = 3.6$. This is a finding worth stating plainly to Pak Darto: the ride, the part the drivers are proud of, is fine; what hurts is everything before it.

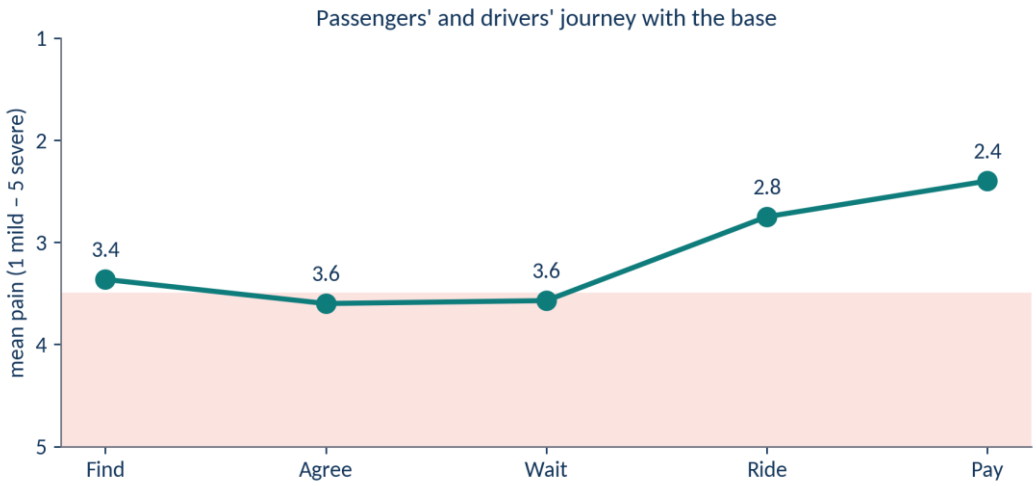


Figure 4.3 The emotional line of the journey map: mean pain per stage from the 37 notes (the axis is reversed, so the curve dips where it hurts; the shaded band marks severe pain).

The three HMW questions give the design sessions their direction: show passengers when the driver will arrive, make the fare clear before the ride, and let passengers book without a phone call. The ideas block and **Figure 4.4** then sort the ten ideas. Three are quick wins: the fixed fare table by zone (impact 5, effort 2, nine votes), the driver-on-the-way message with name and photo, and the share-my-ride link for night trips. Three are big bets worth planning: the automatic fair queue, which addresses the drivers' theme; the one-tap booking page; and choosing the pickup point from photos of landmarks, which answers Mbah Sri's "I know only landmarks". And the original request, "our own full ride-hailing app like the big ones", ends in the last row, with the lowest value for effort and two votes out of the whole group.

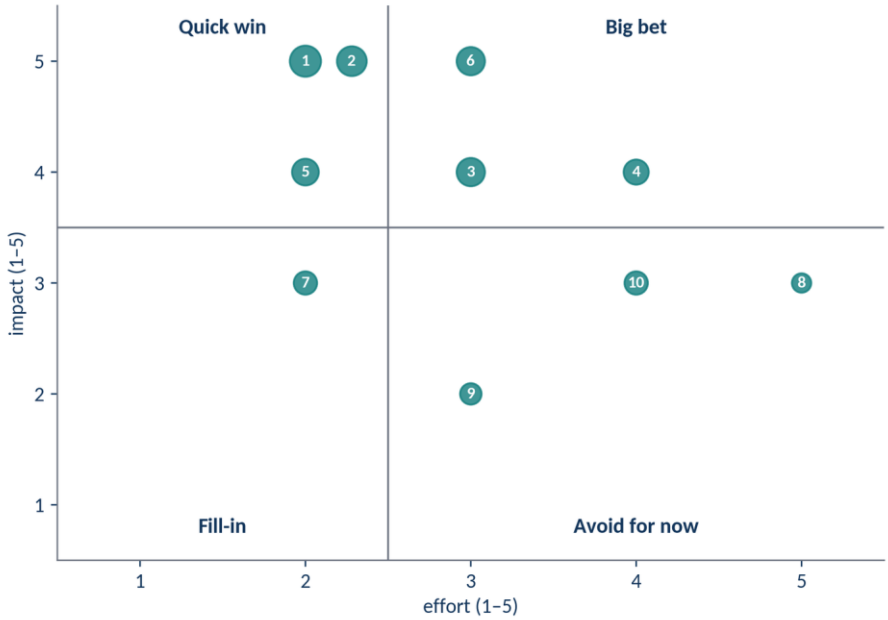


Figure 4.4 The ten ideas in the impact-effort matrix, numbered in the order of **Listing 4.3**; the size of each circle shows its votes.

The team's recommendation to Pak Darto followed directly: start with the three quick wins, which together address the three strongest themes; plan the fair queue with the drivers; prototype the landmark-photo pickup with older passengers; and do not build a full ride-hailing

app. Each recommendation came with measurable requirements of the kind in Section 4.5.7, so that the next cycle of **Figure 4.1** could check whether it worked.

4.9 Testing It

Listing 4.6 checks the four quadrants, checks that a theme mentioned by two people ranks above a theme with the same number of notes from one person, and checks the journey mean on a tiny example.

Listing 4.6 `test_synthesis.py`: tests of the quadrants, the ranking of themes and the journey mean.

```
"""Tests for the research synthesis: python -m unittest test_synthesis.py"""
import unittest
from synthesis import affinity, quadrant, journey


class SynthesisTests(unittest.TestCase):
    def test_quadrants(self):
        self.assertEqual(quadrant(5, 2), "Quick win")
        self.assertEqual(quadrant(4, 3), "Big bet")
        self.assertEqual(quadrant(3, 2), "Fill-in")
        self.assertEqual(quadrant(2, 5), "Avoid for now")

    def test_people_count_more_than_notes(self):
        notes = [dict(theme="A", participant="P1", role="passenger"),
                 dict(theme="A", participant="P1", role="passenger"),
                 dict(theme="B", participant="P1", role="passenger"),
                 dict(theme="B", participant="D1", role="driver")]
        self.assertEqual(affinity(notes)[0][0], "B")

    def test_journey_mean(self):
        notes = [dict(stage=s, pain=p) for s, p in
                 [("Find", "4"), ("Find", "2"), ("Agree", "3"), ("Wait", "5"),
                  ("Ride", "1"), ("Pay", "2")]]
        self.assertEqual(journey(notes)[0], ("Find", 2, 3.0))


if __name__ == "__main__":
    unittest.main()
```

Listing 4.7 runs the corresponding checks in Java.

Listing 4.7 `SynthesisTest.java`: the same checks in Java.

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

/** Tests for the research synthesis: java -cp out SynthesisTest */
public class SynthesisTest {
    static int passed = 0;

    static void expect(boolean ok, String what) {
        if (!ok) throw new AssertionError(what);
        passed++;
    }

    public static void main(String[] args) {
        expect(Synthesis.quadrant(5, 2).equals("Quick win"), "quick win");
        expect(Synthesis.quadrant(4, 3).equals("Big bet"), "big bet");
        expect(Synthesis.quadrant(3, 2).equals("Fill-in"), "fill-in");
        expect(Synthesis.quadrant(2, 5).equals("Avoid for now"), "avoid");
        var notes = List.of(
            Map.of("theme", "A", "participant", "P1", "role", "passenger"),
            Map.of("theme", "A", "participant", "P1", "role", "passenger"),
            Map.of("theme", "B", "participant", "P1", "role", "passenger"),
            Map.of("theme", "B", "participant", "D1", "role", "driver"));
        var board = Synthesis.affinity(notes);
        expect(board.get(0).id().equals("B"), "two people beat two notes");
        System.out.println("All " + passed + " checks passed.");
    }
}
```

```
$ python3 -m unittest test_synthesis.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
$ java -cp out SynthesisTest
All 5 checks passed.
```

4.10 Using It for Real

A small team can run the whole define step of a project in one day with the plan of **Table 4.4**. It assumes that the interviews or observations are done and that every observation is on a note.

Table 4.4 A one-day synthesis workshop plan with its outputs.

Time	Activity	Output
08:00–09:30	Read all notes aloud; group them silently on a wall or whiteboard	Affinity groups
09:30–10:00	Name the groups; tag each note with a journey stage and a pain score	Themes, `notes.csv`, `themes.csv`
10:00–10:30	Run the synthesis program; discuss the reach of each theme and the journey curve	Ranked themes, journey map
10:30–12:00	Write personas and problem statements for the strongest themes; agree the HMW questions	Personas, problem statements
13:00–14:30	Ideate: many ideas per HMW question, no judging	Idea list
14:30–15:30	Score impact and effort; show the ideas to participants and collect votes	`ideas.csv`
15:30–16:30	Run the program again; choose what to prototype; write measurable requirements	Prototype plan, requirements

The files and the program are deliberately general. Replace the stages with those of your own journey (for a laundry: *order*, *pickup*, *wash*, *delivery*, *pay*; for a puskesmas: *arrive*, *register*, *wait*,

consult, *pharmacy*), and the roles with your own (patients and staff, customers and couriers). Two cautions apply. First, the numbers summarize judgments: a pain score of 4 is the team's reading of a note, and the ranking is only as good as the reading; discuss disagreements rather than averaging them away. Second, a theme with low reach may still be critical, for example a safety issue mentioned by one night-shift nurse; the numbers guide attention, they do not replace it.

4.11 HCI in Practice: Profitina

Profitina applies the cycle of **Figure 4.1** to its own product. Its *transparent feedback pipeline* is a user-centered loop made visible: users submit feedback on AI answers, the team curates it, curated items become a dataset, the local Qwen2.5 model is trained and measured on it, and a new version is released, after which the cycle starts again. Users can see where their feedback stands, which makes them participants in the design rather than subjects of it, the second principle of ISO 9241-210.

Product decisions follow evidence about use as well. The Agency subscription tier was removed in August 2026, leaving a single 30-day Pro subscription, a simplification of the kind that the impact-effort matrix of Section 4.5.6 often recommends: fewer options, each clearly understood. And the product's rule that the app advises and the user decides is a requirement in the sense of Section 4.5.7: every AI-generated suggestion must be presented so that a brand owner can evaluate and reject it, and when the local model is unavailable the app says so openly instead of silently substituting a template.

4.12 Common Mistakes, and How to Avoid Them

1. **Starting from the solution.** "An app like the big ones" is a solution. Begin with the first diamond: discover and define before developing.
2. **Treating the client's request as the requirement.** The request is a hypothesis; the people who will use the result define the requirement.
3. **Interviewing only one kind of user.** The drivers' theme of fair turns was invisible in passenger interviews. Include every role that the system touches.
4. **Counting notes instead of people.** One talkative participant can dominate a theme; rank themes by reach (Equation (4.1)).
5. **Inventing personas or users.** A persona without evidence, or a synthetic user from an AI tool, describes a stereotype. Build personas from notes.
6. **Writing requirements that cannot be tested.** "Easy to use" is a wish; "books in under 30 seconds without help in 4 of 5 attempts" is a requirement.
7. **Doing the process once.** User-centered design is a cycle; plan the second iteration before the first ends.
8. **Mishandling participants' data.** Interview notes are personal data under UU PDP; ask for consent, use codes, and delete recordings after analysis.

4.13 Chapter Summary

- User-centered design rests on Gould and Lewis's three habits: early focus on users and tasks, empirical measurement, iterative design.
- ISO 9241-210:2019 gives six principles (**Table 4.1**) and four activities in a cycle: context of use, user requirements, design solutions, evaluation (**Figure 4.1**).

- The Double Diamond (discover, define, develop, deliver; **Figure 4.2**) and the five modes of design thinking (empathize, define, ideate, prototype, test) describe the same cycle (**Table 4.2**); Lean UX fits it into agile teams.
- The working tools: affinity diagram and theme reach (Equation (4.1)), personas (**Table 4.3**), journey map with mean pain per stage (Equation (4.2)), empathy map, jobs to be done, problem statements, HMW questions, the impact-effort matrix with value for effort (Equation (4.3)), and measurable requirements.
- AI tools help to develop and prototype; they cannot replace research with real people. Research data are personal data under UU PDP and PP 33/2026.
- For the ojek base, 37 notes from ten people showed that waiting without knowing (reach 70%) and agreeing on the price hurt most, and turned "an app like the big ones" into three quick wins and three planned bets.

4.14 Exercises

1. Using the notes of participant P1 in **Listing 4.2**, write a persona in the format of **Table 4.3**. Which fields can you fill from evidence, and which would you need to ask about?
2. Compute by hand, with Equation (4.2), the mean pain of the stage *Wait* from the seven notes in **Listing 4.2**, and the reach of theme T5 with Equation (4.1).
3. In a second ideation session, a driver proposes "a printed fare card at the base and in the shops", which the team scores with impact 4, effort 1, and five votes. Add it to a copy of `ideas.csv`, run the program, and say where it lands and why.
4. Improve these "how might we" questions and say what is wrong with each: (a) "How might we build a booking app for the base?"; (b) "How might we improve transport in the town?"; (c) "How might we add a map with GPS?".
5. Describe the ojek project of this chapter in the vocabulary of the Double Diamond and in the vocabulary of ISO 9241-210. Where did the team diverge and converge, and which activity will the next evaluation most likely send them back to?
6. Pak Darto's nephew suggests skipping the interviews: "Ask an AI to generate twenty passenger personas; it's faster." Write a reply of five or six sentences that explains what would be lost and what the AI could usefully do instead.
7. Extend `synthesis.py` so that it also prints the reach of every theme as a percentage and marks the themes raised by only one role. Show the new lines for the original data.

4.15 Solutions to the Exercises

1. From P1's notes: *context*: a regular passenger who calls the base early in the morning (the phone is busy at 7 am); *frustrations*: the phone is busy, the price is not the same each time, and waiting at the gate without knowing if anyone is coming (pain 5, the highest of all notes); *goal* (inferred, to confirm): reach work or school on time in the morning. The quote must come from the interview itself, not from imagination; the notes point to one about waiting at the gate. Age, occupation, destination, phone and payment habits are not in the notes and must be asked; writing them from imagination would turn the persona into a stereotype. *Design implication*: a booking that does not depend on the busy phone and a confirmation of who is coming and when.

2. *Wait* notes: N03 (5), N08 (4), N13 (4), N20 (3), N29 (2), N35 (4), N36 (3). The sum is 25, so $\bar{p} = 25/7 = 3.57$, printed as 3.6. Theme T5 has notes from P3, P5, P6 and D3, four different people out of ten, so $r = 4/10 = 40\%$.

3. Appending the line `Printed fare card at the base and in shops,T1,4,1,5` and running the program gives, in the ideas block:

```
quadrant      imp eff ratio votes idea
Quick win       4   1  4.00     5  Printed fare card at the base and in shops
Quick win       5   2  2.50     9  Fixed fare table by zone shown before booking
Quick win       5   2  2.50     8  Driver-on-the-way message with name and photo
Quick win       4   2  2.00     6  Share-my-ride link for family on night trips
Big bet         5   3  1.67     7  Automatic fair queue for the base
Big bet         4   3  1.33     7  One-tap booking page from a saved WhatsApp link
Big bet         4   4  1.00     5  Pick the pickup point from photos of landmarks
Fill-in         3   2  1.50     4  Cooperative QRIS with the amount shown
Avoid for now   3   4  0.75     4  Voice booking by phone with a menu
Avoid for now   2   3  0.67     3  Loyalty points for regular passengers
Avoid for now   3   5  0.60     2  Own full ride-hailing app like the big ones
```

The card lands at the top: it is a quick win (impact 4 is high, effort 1 is low), and its value for effort $4/1 = 4.00$ is the highest of all ideas. It also serves the theme *unclear price* for passengers without smartphones, which the fare table shown before booking does not, so it complements rather than replaces that idea.

4. (a) Names a solution ("a booking app") and no need; better: "How might we let regular passengers book a ride without calling the base?". (b) Far too broad for a team to act on; better: "How might we help older passengers reach the market before 5:30 without depending on their family?". (c) Names a technology and ignores the finding that many passengers cannot read maps; better: "How might we help passengers say where they are without street names?".

5. Double Diamond: *discover*, three mornings at the base and ten conversations (diverging); *define*, the affinity diagram, personas, journey map and three HMW questions (converging on waiting, price and booking); *develop*, the ideation session with ten ideas (diverging); *deliver*, choosing three quick wins to prototype and test (converging). ISO 9241-210: understanding the context of use (the base, the phones, the passengers), specifying requirements (the measurable statements of Section 4.5.7), producing design solutions (the prototypes to come), and evaluating them. The first evaluation will most likely send the team back to *produce design solutions* for details of the messages, and possibly to *context of use* if the prototype reveals, for example, that older passengers do not read WhatsApp messages while standing in the sun.

6. A generated persona describes what is typical of passengers in general, as represented in the model's training data, not the regulars of this base. It could not have revealed that the drivers' main worry is the fairness of the turn order, that older passengers know places only by landmarks, or that waiting is more painful than booking; those findings came only from listening to these people, and they changed the whole plan. Building on invented users risks spending the cooperative's money on features nobody needs. The AI could usefully help prepare the interview guide, summarize the team's own anonymized notes, or draft variants of the messages once the needs are known. The research itself, three mornings at the base, is cheap compared with building the wrong app.

7. In `main`, after `board = affinity(notes)`, change the theme row to add the reach, and mark single-role themes:

```
    for theme, t in board:
        who = "+".join(sorted(t["roles"], reverse=True))
        reach = 100 * len(t["people"]) / len(people)
        mark = "  <- one role only" if len(t["roles"]) == 1 else ""
        print(f"{theme:5} {themes[theme]['label']:22}{t['notes']:6}"
              f"{len(t['people']):7}  {reach:3.0f}%  {who}{mark}")
```

For the original data the rows then show 70% for T2; 50% for T1, T3, T4 and T7; 40% for T5; 30% for T6; and 30% for T8, the only theme marked “one role only” (drivers).

References

[1] J. D. Gould and C. Lewis, “Designing for usability: Key principles and what designers think,” *Communications of the ACM*, vol. 28, no. 3, pp. 300–311, 1985.

[2] International Organization for Standardization, *ISO 9241-210:2019 Ergonomics of human-system interaction — Part 210: Human-centred design for interactive systems*. Geneva, Switzerland: ISO, 2019. [Online]. Available: https://www.iso.org/standard/77520.html

[3] Design Council, “Framework for Innovation: Design Council’s evolved Double Diamond,” accessed Sep. 2026. [Online]. Available: https://www.designcouncil.org.uk/resources/framework-for-innovation/

[4] Design Council, “Systemic Design Framework,” 2021, accessed Sep. 2026. [Online]. Available: https://www.designcouncil.org.uk/resources/systemic-design-framework/

[5] T. Brown, “Design Thinking,” *Harvard Business Review*, vol. 86, no. 6, pp. 84–92, Jun. 2008.

[6] Hasso Plattner Institute of Design at Stanford, *An Introduction to Design Thinking: Process Guide*. Stanford, CA, USA: d.school, 2010.

[7] J. Gothelf and J. Seiden, *Lean UX: Creating Great Products with Agile Teams*, 3rd ed. Sebastopol, CA, USA: O’Reilly Media, 2021.

[8] K. Holtzblatt and H. Beyer, *Contextual Design: Design for Life*, 2nd ed. Cambridge, MA, USA: Morgan Kaufmann, 2017.

[9] A. Cooper, R. Reimann, D. Cronin, and C. Noessel, *About Face: The Essentials of Interaction Design*, 4th ed. Indianapolis, IN, USA: Wiley, 2014.

[10] C. M. Christensen, T. Hall, K. Dillon, and D. S. Duncan, “Know Your Customers’ ‘Jobs to Be Done’,” *Harvard Business Review*, vol. 94, no. 9, pp. 54–62, Sep. 2016.

[11] Figma, “What’s new from Config 2026,” Figma Help Center, Jun. 2026. [Online]. Available: https://help.figma.com/hc/en-us/articles/39582753756695-What-s-new-from-Config-2026

[12] Nielsen Norman Group, “Synthetic Users: If, When, and How to Use AI-Generated ‘Research’,” accessed Sep. 2026. [Online]. Available: https://www.nngroup.com/articles/synthetic-users/

[13] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. JDIH BPK RI. [Online]. Available: https://peraturan.bpk.go.id/Details/229798/uu-no-27-tahun-2022

[14] Veritask, “Pengaturan Teknis Pelindungan Data Pribadi dan Kewajiban Pengendali serta Prosesor Akhirnya Terbit Lewat PP 33/2026,” 2026. [Online]. Available: https://veritask.ai/id/artikel/pengaturan-teknis-pelindungan-data-pribadi-dan-kewajiban-pengendali-serta-prosesor-akhirnya-terbit-lewat-pp-33-2026

Chapter 5

User Research Methods

A small laundry near a campus is losing customers, and nobody knows why. The owner's nephew wants to build an app; the owner wants to lower the price; a friend says the shop needs a bigger sign. Each of them is guessing. This chapter is about replacing the guess with evidence, cheaply and honestly. By the end of it you will be able to choose a research method from the question you need answered; plan a small study with a research plan, a recruiting screener and a consent script that respects UU PDP 27/2022 and its new implementing regulation PP 33/2026; run interviews and contextual inquiry without leading the participant; write a short survey whose items measure what you think they measure; summarize the survey in Java and Python with means, confidence intervals and top-2-box shares; code interview notes, draw an affinity diagram and decide with a stated rule when you have interviewed enough people; and turn findings into insight statements that a designer can act on. You will also see where generative AI helps a researcher in 2026 and where it quietly does harm.

5.1 Bu Rina's Missing Customers

Bu Rina runs *Laundry Bersih*, a laundry of two washing machines, two dryers and one long ironing table, in a lane off the main road near a university campus in Surabaya. Her customers are students from the boarding houses around the campus, young office workers, a few families and the owners of the boarding houses themselves, who bring the sheets of twenty rooms every Monday. Most customers order by WhatsApp: they send a message, Bu Rina's assistant Dimas rides over on a motorbike to pick up the bag, the clothes are weighed at the shop, and the customer receives a message with the weight and the price. Two days later the clothes come back, folded and wrapped in plastic.

For three years the business grew. This year the number of orders fell for four months in a row, and in her order book Bu Rina counts about a third fewer repeat customers than a year earlier. She has ideas about the reason, and so does everyone around her. Her nephew, who studies informatics, is certain that customers want an app with a map and a tracking screen, as the big ride-hailing and food-delivery apps have. A neighbor thinks that the new laundry at the corner is cheaper. Dimas thinks that customers are annoyed because he sometimes comes late in the evening. Bu Rina herself suspects that the students simply went home for the long holiday and have not all come back.

Every one of these explanations is plausible, and every one of them leads to a different investment. An app costs months of the nephew's time and money for a developer account and a server. A lower price costs margin on every kilogram. A second motorbike costs a salary. If the real reason is something else, each of those investments is wasted, and the customers keep leaving. What Bu Rina needs before she spends anything is to *find out*. That is what user research is: a disciplined way of finding out what people do, what they need and why, before and while you design for them.

The need is not small. In its survey published in May 2026, the Indonesian Internet Service Providers Association (APJII) estimated that 81.7% of Indonesians, about 235 million people out

of 287 million, use the internet, up from 80.66% a year earlier; penetration is 84.75% in cities and 78.18% in villages, and the most common activities are communication and social media, entertainment, information searches and online commerce [1]. Businesses as small as Bu Rina's now meet their customers through a screen, and every decision about that screen is a decision about people she cannot see.

A second number shows why research needs care even when data is plentiful. StatCounter, which counts page views on the websites that carry its tracking code, reported for August 2026 that 60.59% of the page views it saw from Indonesia came from desktop computers and 38.98% from mobile phones [2]. That looks like a contradiction of everything we know about a country where most people first met the internet on a phone. It is not a contradiction; it is a different measurement. APJII asks *people*; StatCounter counts *page views* on a particular set of websites, and page views from office computers, from automated traffic and from a few heavy users weigh as much as anyone else's. Neither number is wrong. Each answers its own question, and a researcher who quotes one of them must know which question it answers. This whole chapter is, in a sense, about that lesson.

5.2 The Idea in Plain Words

5.2.1 Finding out before building

The standard for human-centered design, ISO 9241-210:2019, puts the idea at the start of its process: before designing a solution you must *understand and specify the context of use*, meaning the users, their goals and tasks, and the physical, social and technical environment in which they work [3]. Chapter 4 placed this activity in the design-thinking cycle as the first diamond, the part where the team widens its view of the problem before narrowing on a definition. This chapter supplies the tools for that part.

> **DEFINITION.** *User research* is the systematic study of the people who use (or might use) a product: their behavior, needs, goals, context and attitudes. A *research question* is what the team needs to know, such as "why do repeat customers stop ordering?". A *method* is the procedure used to answer it, such as an interview or a survey. A *participant* is a person who takes part in a study, and a *finding* is something the study showed, stated with its evidence.

A study begins with a research question, never with a method. "Let us do a survey" is not a plan; "we need to know which of four suspected reasons explains most of the lost repeat customers" is. The question then determines the method, the participants and the amount of evidence that is enough.

5.2.2 What people say and what people do

People are poor witnesses of their own behavior. Ask a customer how often she orders laundry and she will tell you "every week"; her order history may show every ten days, with a gap of a month in July. Ask whether she would use an app with order tracking and she will politely say yes, because it sounds modern and saying no seems rude. Neither answer is a lie. People remember the typical case, not the actual one, and they predict their future behavior badly, especially about things they have not tried.

This is why the most useful map of research methods has two axes [4]. The first separates what people *say* (attitudinal methods: interviews, surveys, focus groups) from what people *do* (behavioral methods: observation, usability tests, analytics). The second separates questions of

why and how (qualitative methods, a few participants studied in depth) from questions of *how many and how much* (quantitative methods, many participants measured the same way). **Figure 5.1** places the common methods on this map.

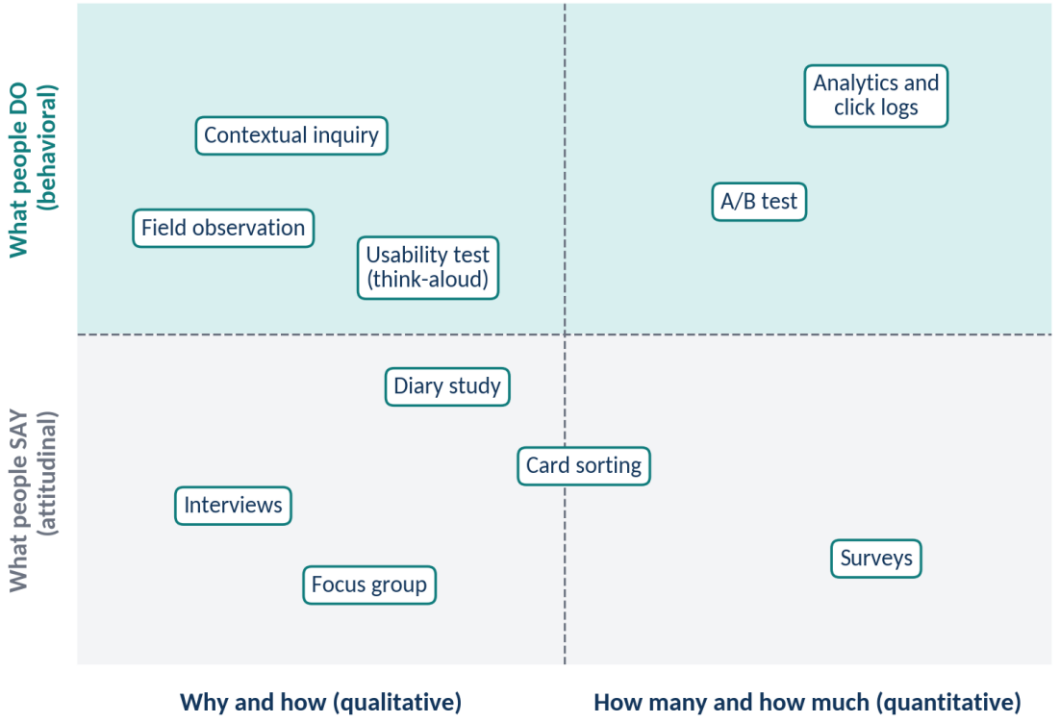


Figure 5.1 The landscape of user research methods: behavioral methods (top) watch what people do, attitudinal methods (bottom) record what people say; qualitative methods (left) explain why, quantitative methods (right) count how many.

No corner is better than the others. Qualitative behavioral methods, such as sitting with Dimas on his pickup round and watching customers hand over their bags, show problems nobody would think to mention. Quantitative attitudinal methods, such as a survey of sixty customers, tell you how widespread a problem is but not why it exists. A good study usually combines two corners, and the combination is called *triangulation*: when an interview, a survey and the order book all point at the same cause, you can act with confidence.

5.2.3 Four questions that choose the method

Before choosing a method, answer four questions in writing.

1. **What decision will this research inform?** Bu Rina must decide where to spend money in the next three months. Research that cannot change the decision is not worth doing.
2. **What do we already know, and how do we know it?** The order book shows fewer repeat customers; that is behavioral and quantitative, but it says nothing about why.
3. **Is the question about why or about how many?** “Why do customers leave?” is qualitative and needs a few deep conversations. “How many customers are affected by late pickups?” is quantitative and needs a count.
4. **What can we afford?** Bu Rina has a few evenings, a list of customers’ phone numbers and a budget for small thank-you vouchers. That rules out a large panel but allows a dozen interviews and a short survey.

Table 5.1 turns the answers into a first choice. It is not a law; it is the kind of table a research team keeps on the wall.

Table 5.1 Choosing a first method from the research question, with typical sample sizes and durations for a small team.

If the question is…	Start with	Typical size	Time for a small team
Why do people do this? What is their day like?	Contextual inquiry, interviews	6–15 people	1–2 weeks
How widespread is a problem or an opinion?	Survey	50–400 answers	1–3 weeks
Can people complete a task with our design?	Usability test (Chapter 7)	5–8 per round	a few days
How do people group our content?	Card sorting (Chapter 8)	15–30 people	1 week
What actually happens over days or weeks?	Diary study	8–15 people	2–4 weeks
Which of two designs performs better?	A/B test (Chapter 7)	hundreds to thousands	1–4 weeks

For Bu Rina the question "why do repeat customers stop ordering?" is a why question, so the study starts with interviews and a morning of contextual inquiry. Once the interviews suggest the likely reasons, a short survey will measure how widespread each one is. That order, qualitative first to discover, quantitative second to measure, is the most common shape of a small study, and it is the one this chapter follows.

5.3 Planning a Study

5.3.1 The research plan on one page

A research plan is one page that anyone in the team can read in two minutes. Its value is not the document; it is the conversation it forces before anyone talks to a customer. **Table 5.2** is the plan for Laundry Bersih.

Table 5.2 A one-page research plan for the Laundry Bersih study.

Part	Laundry Bersih study
Decision	Where to invest in the next three months: price, pickup capacity, communication, or an app
Research questions	Why do repeat customers stop ordering? What happens between the order message and the clothes coming back? What do customers wish they knew or could do?
Methods	12 semi-structured interviews (30–40 min); one morning of contextual inquiry on the pickup round; then a 6-item survey sent to all customers on the WhatsApp list
Participants	Recent customers who stopped ordering, and current regular customers; mix of students, workers, families, boarding-house owners; at least two people aged 60 or more
Recruiting	WhatsApp message from Bu Rina; screener of four questions; voucher for 3 kg of free washing
Ethics and data	Consent script read and agreed before recording; recordings deleted after notes are checked; notes identified only by participant code
Output	Coded notes, affinity diagram, survey summary, five insight statements, a one-page recommendation
Timeline	Week 1 recruit and interview; week 2 finish interviews and analyze; week 3 survey and report

5.3.2 Who to talk to, and how many

The people you most want to hear from are often the hardest to reach. Customers who left are not in the shop, and some of them are annoyed. Yet a study that talks only to happy regulars learns why people stay, not why they leave. Bu Rina's plan therefore asks for both groups on purpose. A

screener, a short list of questions sent before inviting someone, makes sure the mix is right: when did you last order, how often did you order before that, what do you do (student, worker, household, business), and are you willing to talk for about half an hour. People who answer "I have never ordered" are thanked and not invited.

Include the edges. A laundry near a campus is not only used by twenty-year-olds. Retired neighbors, whose children set up WhatsApp for them, may find a small-print price list hard to read or voice notes easier than typing. Customers with disabilities are protected in Indonesia by Law 8/2016 on Persons with Disabilities, which guarantees their right to accessible services and information [5]; a study that leaves them out designs them out. Chapter 11 returns to accessibility; here the rule is simply to recruit on purpose people whose experience differs from the typical customer's.

How many interviews are enough? For a study of one fairly homogeneous group, a widely cited analysis of interview studies found that most of the themes appear within the first six to twelve interviews, and it proposed a simple rule for deciding when to stop that Section 5.7 turns into a program [6]. For a survey, the answer depends on the precision you need, which Section 5.5 computes. For a usability test, Chapter 7 shows why five to eight people per round usually reveal most of the serious problems.

5.3.3 Consent and personal data

Research with people handles personal data: names, phone numbers, voices on a recording, sometimes photographs of a home. In Indonesia, Law 27/2022 on Personal Data Protection (UU PDP) requires a lawful basis for processing personal data, and explicit consent is the basis a research study normally uses [7]. The implementing regulation, Government Regulation (PP) 33/2026, spells out what valid consent looks like: it must be given freely, consciously, specifically and unambiguously; the information must be given *before* consent is asked, concisely and accurately; consent may not be bundled with general terms and conditions or obtained by deceptive methods; a person who withdraws consent must see processing stop within 72 hours of a verified request; and refusing consent may not reduce the quality of a service that does not need the data [8]. Its detailed obligations apply from January 16, 2027 [9], but a study run today should already work this way, because it is also simply the decent way to treat a participant.

In practice a consent script for an interview has six parts, read aloud or sent as a short message and agreed before the recording starts:

1. who is doing the study and why ("to improve the ordering and pickup service of Laundry Bersih");
2. what will happen ("a conversation of about thirty minutes about how you use the laundry");
3. what is recorded and who sees it ("audio only, heard by Bu Rina and one helper, deleted after the notes are checked; the notes use a code, not your name");
4. that taking part is voluntary and can stop at any time, with no effect on the service;
5. the thank-you ("a voucher for three kilograms of washing, whether or not you finish");
6. a question: "Do you agree to take part, and do you agree that we record the audio?", with the two answers kept separate.

The last point matters more than it seems. Agreeing to talk and agreeing to be recorded are two different consents; a person may happily talk but not want a recording. PP 33/2026's requirement that consent be specific means that the two must be asked separately, not folded into one yes [8].

WARNING. Do not collect what you do not need. A study of the ordering experience does not need a participant's address, family details or income. Every extra field is a risk to the participant and a duty for you. Take participant codes (P01, P02, …) from the start, and keep the list that links codes to phone numbers in one place, apart from the notes.

5.4 Interviews and Contextual Inquiry

5.4.1 The semi-structured interview

An interview in user research is not a questionnaire read aloud. It is a guided conversation in which the researcher has a list of topics and a few opening questions, but follows the participant wherever the participant's experience leads. This form is called *semi-structured*. Portigal's advice, distilled from many years of interviewing, is that the researcher's job is to make the participant the expert and to listen far more than talk [10].

A good interview guide for Bu Rina has four parts: a warm-up ("tell me a little about your week, where you live and how laundry fits in"), the story of the last time ("tell me about the last time you sent laundry to us, from the moment you decided"), the specific topics (pickup, price, waiting, getting the clothes back, paying, problems), and a close ("if you could change one thing about sending your laundry, what would it be?", "is there anything I should have asked?").

The heart of the interview is the story of a *specific* recent occasion. Generalities produce generalities; a particular Tuesday produces details. When the participant says "usually it is fine", the researcher asks "tell me about the last time it was not fine".

5.4.2 Asking without leading

The questions themselves are where most interviews go wrong. **Table 5.3** lists the common traps and a better form of each.

Table 5.3 Interview traps and better questions for the laundry study.

Trap	Poor question	Better question
Leading	"Don't you find the pickup times too short?"	"Tell me about arranging the pickup last time."
Asking for a prediction	"Would you use an app with tracking?"	"What did you do while waiting for your clothes last time?"
Asking for a design	"What features should our app have?"	"What was annoying or worrying about the last order?"
Double question	"Was the price clear and fair?"	"How did you find out the price last time?" then "What did you think of it?"
Yes/no closing	"Was the delivery on time?"	"When did the clothes come back, and how did that fit your day?"
Jargon	"How was the order-fulfillment flow?"	"What happened after you sent the message?"

Three habits help more than any list. First, *silence*: after an answer, count to five before speaking; people fill silence with the most interesting part. Second, *echo*: repeat the participant's last words as a question ("the price came later?") to invite more without suggesting anything. Third, *ask for the artifact*: "could you show me that chat?" A WhatsApp conversation on the participant's own phone shows the exact times, the exact words and the exact moment of annoyance, far better than memory.

5.4.3 Contextual inquiry: the master and the apprentice

Contextual inquiry is an interview that takes place where the work happens, while it happens. Holtzblatt and Beyer describe the relationship as that of *master and apprentice*: the participant is the master of the work, the researcher an apprentice who watches, asks and checks understanding, in the real setting [11]. Four principles guide it: *context* (go where the work is), *partnership* (the participant leads), *interpretation* (share your interpretations and let the participant correct them) and *focus* (know which part of the work you are there to learn).

For Laundry Bersih, the inquiry was a morning on the back of Dimas's motorbike and an afternoon at the counter. The morning showed something no interview had mentioned: at three boarding houses, the customer was not at home, and the bag was left with the boarding-house guard, who could not say when it would come back. The afternoon showed the other end: the price message was sent only after weighing, often three or four hours after pickup, and the "ready" message was sent only when Dimas was about to leave, sometimes after nine in the evening. Nobody had complained about this in words, but the chat histories on customers' phones were full of "Bu, sudah selesai belum?" ("Ma'am, is it done yet?").

5.4.4 Field notes and diary studies

Notes are taken during or immediately after each session, in a fixed format so that they can be compared: participant code, date, setting, then one observation per line, each marked as a *quote* (the participant's exact words), an *observation* (what you saw) or an *interpretation* (what you think it means). Keeping these three apart is the single most important habit of a researcher, because the interpretation will later be questioned, and the quote and observation are the evidence.

When the behavior of interest stretches over days, a *diary study* asks participants to record short entries each time something happens: each order, each wait, each problem, perhaps as a voice note or a photo in a WhatsApp group created for the study. Diary studies reveal the ordinary rhythm that a one-hour interview compresses: Bu Rina's boarding-house owners, for instance, recorded that they ordered every Monday but decided the day before, when they saw the pile of sheets, which suggested that a reminder message on Sunday evening might be welcome.

5.5 Surveys and Questionnaires

5.5.1 When a survey is the right tool

After twelve interviews Bu Rina's team had a list of candidate problems: not knowing when the laundry would be ready, the price arriving only after weighing, slow replies in the evening, pickup windows that did not fit working hours. Interviews cannot say how common each problem is among all customers; a survey can. The survey came *after* the interviews, and its items were written in the customers' own words from the interviews. A survey written before any interview asks the team's questions in the team's words, and it measures the team's assumptions.

5.5.2 Writing good items

A survey item is a statement or question with a fixed set of answers. The most common form in user research is the *Likert item*: a statement ("The price is clear before I pay") with five or seven answers from "strongly disagree" to "strongly agree". Good items follow a few rules [12]:

- **One idea per item.** “The price is clear and fair” mixes two ideas; a person who finds it clear but unfair cannot answer.
- **Concrete and recent.** “I know when my laundry will be ready” is about something the customer experiences each time; “I am satisfied with the service” is too vague to act on.
- **Neutral wording.** “Don’t you agree that the pickup is convenient?” pushes toward yes. People tend to agree with statements put to them, which is called *acquiescence bias*; do not add to it.
- **The same scale everywhere,** with the same direction (5 = strongly agree) for every item.
- **Few items.** A customer who answers on a phone gives two minutes, not twenty. Six good items beat thirty mediocre ones.
- **Pilot it.** Before sending, ask three people to answer aloud and say what they think each item means.

The Laundry Bersih survey had six items, listed in `items.csv`, and asked two background questions: age group and whether the customer usually orders by WhatsApp or walks in.

5.5.3 How precise is a survey?

A survey is a sample: sixty customers stand for all of Bu Rina’s customers. Any number computed from the sample, such as “15% agree”, is an estimate with an uncertainty. For a proportion p estimated from n answers, the 95% *margin of error* is

$$E = z\sqrt{\frac{p(1-p)}{n}} \tag{5.1}$$

where $z = 1.96$ is the value that leaves 2.5% of a normal distribution in each tail. The margin is largest when $p = 0.5$, so a researcher planning a survey uses $p = 0.5$ as the worst case. For $n = 60$: $E = 1.96 \times \sqrt{0.25/60} = 1.96 \times 0.0645 = 0.127$, that is, plus or minus 12.7 percentage points. For $n = 400$: $E = 1.96 \times \sqrt{0.25/400} = 1.96 \times 0.025 = 0.049$, about 4.9 points. Quadrupling the sample halves the margin, because n sits under a square root.

Solving Equation (5.1) for n tells you how many answers you need for a desired margin: $n = z^2 p(1-p)/E^2$. With $E = 0.10$ and $p = 0.5$, $n = 3.8416 \times 0.25/0.01 = 96.04$, so 97 answers. Bu Rina’s sixty answers are enough to tell a problem that affects most customers from one that affects few, but not enough to tell 40% from 50%.

For the mean of a Likert item, the uncertainty is expressed as a *confidence interval*. If the item has mean $\bar{x}$, standard deviation s and n answers, the 95% confidence interval is

$$\bar{x} \pm t_{0.975,\,n-1} \cdot \frac{s}{\sqrt{n}} \tag{5.2}$$

where $t_{0.975,\,n-1}$ is the value of Student’s t distribution with $n-1$ degrees of freedom that leaves 2.5% in the upper tail. For large n it approaches 1.96; for $n = 60$ it is about 2.001. Printed tables give it, but a program needs a formula. A classic series expansion around the normal value z is accurate to about three decimals from five degrees of freedom upward:

$$t \approx z + \frac{z^3 + z}{4\nu} + \frac{5z^5 + 16z^3 + 3z}{96\nu^2} \tag{5.3}$$

where $\nu = n - 1$ is the number of degrees of freedom. For $\nu = 59$: $z^3 + z = 7.529 + 1.960 = 9.489$, divided by $4 \times 59 = 236$ gives 0.0402; the second term is $(5 \times 28.925 + 16 \times 7.529 + 3 \times 1.960)/(96 \times 3481) = 270.97/334176 = 0.0008$; so $t \approx 1.960 + 0.040 + 0.001 = 2.001$, matching the table.

The third summary a researcher reports is the *top-2-box* share: the proportion of answers in the two most positive categories (4 and 5). It is easier to explain to a shop owner than a mean ("two customers in three agree that ordering by WhatsApp is easy") and it is a proportion, so Equation (5.1) gives its margin.

5.6 Building the Survey Summarizer

5.6.1 The data

The survey answers are in `data/survey.csv`, one row per respondent: an identifier, the age group, the usual channel, and the six answers from 1 to 5. The item texts are in `data/items.csv`. The first rows look like this:

```
id,age,channel,Q1,Q2,Q3,Q4,Q5,Q6
R01,30-44,walkin,2,2,5,5,4,5
R02,18-29,wa,5,2,2,4,4,4
```

5.6.2 The program in Python

Listing 5.1 reads both files, computes for each item the mean, the confidence interval of Equation (5.2) with the t value of Equation (5.3) and the top-2-box share, then breaks the most worrying item down by age group and prints the margin of error of Equation (5.1) for the whole sample.

Listing 5.1 `survey_stats.py`: summarizes a Likert survey with means, 95% confidence intervals and top-2-box shares.

```
"""Summarize a Likert survey: mean, 95% confidence interval, top-2-box.
Usage: python survey_stats.py [survey.csv] [items.csv]"""
import csv
import math
import sys


def t_critical(df):
    """Two-sided 95% t value; series expansion, accurate for df >= 5."""
    z = 1.959964
    return (z + (z**3 + z) / (4 * df)
            + (5 * z**5 + 16 * z**3 + 3 * z) / (96 * df * df))


def item_stats(values):
    """Return n, mean, sd, half-width of the 95% CI, top-2-box share."""
    n = len(values)
    mean = sum(values) / n
    var = sum((v - mean) ** 2 for v in values) / (n - 1)
    sd = math.sqrt(var)
    half = t_critical(n - 1) * sd / math.sqrt(n)
    top2 = sum(1 for v in values if v >= 4) / n
    return n, mean, sd, half, top2


def margin_of_error(n, p=0.5):
    """Worst-case 95% margin of error of a proportion, in points."""
    return 100 * 1.959964 * math.sqrt(p * (1 - p) / n)


def load(survey_path, items_path):
    with open(items_path, newline="", encoding="utf-8") as f:
        items = [(r["item"], r["text"]) for r in csv.DictReader(f)]
    with open(survey_path, newline="", encoding="utf-8") as f:
        rows = list(csv.DictReader(f))
    return items, rows
```

Listing 5.1 (Continued) *survey_stats.py: summarizes a Likert survey with means, 95% confidence intervals and top-2-box shares.*

```
def main(args):
    survey = args[0] if len(args) > 0 else "../data/survey.csv"
    names = args[1] if len(args) > 1 else "../data/items.csv"
    items, rows = load(survey, names)
    print(f"{len(rows)} respondents, 1 = strongly disagree ... 5 = strongly agree")
    print(f"{'item':<5}{'mean':>6}{'95% CI':>15}{'top-2':>7}  statement")
    for key, text in items:
        n, mean, sd, half, top2 = item_stats([int(r[key]) for r in rows])
        ci = f"[{mean - half:.2f}, {mean + half:.2f}]"
        print(f"{key:<5}{mean:>6.2f}{ci:>15}{top2 * 100:>6.0f}%  {text}")
    print("\nQ2 'I know when my laundry will be ready' by age group:")
    for group in ("18-29", "30-44", "45+"):
        vals = [int(r["Q2"]) for r in rows if r["age"] == group]
        n, mean, sd, half, top2 = item_stats(vals)
        print(f"  {group:<6} n={n:<3} mean {mean:.2f} +/- {half:.2f}")
    print(f"\nMargin of error of a percentage at n={len(rows)}: "
          f"+/-{margin_of_error(len(rows)):.1f} points")


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

Three details deserve a comment. `item_stats` divides by $n-1$, not n, when computing the variance: this is the *sample* standard deviation, the right one when the sixty answers stand for a

larger population. `t_critical` is Equation (5.3) written term for term. And the program reads the item texts from a file rather than hard-coding them, so the same program summarizes the next survey, with different items, unchanged.

5.6.3 The program in Java

Listing 5.2 is the same program in Java. It reads the CSV files with `Files.readAllLines` and splits each line at the commas, which is enough for files like these that contain no quoted commas; a survey exported from a form tool with free-text answers needs a real CSV parser.

Listing 5.2 `SurveyStats.java`: the same survey summary in Java.

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

/** Summarize a Likert survey: mean, 95% confidence interval, top-2-box.
 *  Usage: java SurveyStats [survey.csv] [items.csv] */
public class SurveyStats {
    /** Two-sided 95% t value; series expansion, accurate for df >= 5. */
    static double tCritical(int df) {
        double z = 1.959964;
        return z + (Math.pow(z, 3) + z) / (4.0 * df)
            + (5 * Math.pow(z, 5) + 16 * Math.pow(z, 3) + 3 * z) / (96.0 * df * df);
    }

    /** Returns {n, mean, sd, half-width of the 95% CI, top-2-box share}. */
    static double[] itemStats(List<Integer> values) {
        int n = values.size();
        double sum = 0;
        for (int v : values) sum += v;
        double mean = sum / n, ss = 0;
        int top2 = 0;
        for (int v : values) {
            ss += (v - mean) * (v - mean);
            if (v >= 4) top2++;
        }
        double sd = Math.sqrt(ss / (n - 1));
        double half = tCritical(n - 1) * sd / Math.sqrt(n);
        return new double[] {n, mean, sd, half, (double) top2 / n};
    }

    /** Worst-case 95% margin of error of a proportion, in points. */
    static double marginOfError(int n) {
        return 100 * 1.959964 * Math.sqrt(0.25 / n);
    }

    static List<String[]> readCsv(String path) throws IOException {
        List<String[]> rows = new ArrayList<>();
        List<String> lines = Files.readAllLines(Path.of(path));
        for (String line : lines.subList(1, lines.size()))
            if (!line.isBlank()) rows.add(line.split(",", -1));
        return rows;
    }
```

Listing 5.2 (Continued) *SurveyStats.java: the same survey summary in Java.*

```
    static List<Integer> column(List<String[]> rows, int col, String age) {
        List<Integer> out = new ArrayList<>();
        for (String[] r : rows)
            if (age == null || r[1].equals(age)) out.add(Integer.parseInt(r[col]));
        return out;
    }

    public static void main(String[] args) throws IOException {
        Locale.setDefault(Locale.US);
        String survey = args.length > 0 ? args[0] : "../data/survey.csv";
        String names = args.length > 1 ? args[1] : "../data/items.csv";
        List<String[]> items = readCsv(names), rows = readCsv(survey);
        System.out.println(rows.size()
            + " respondents, 1 = strongly disagree ... 5 = strongly agree");
        System.out.printf("%-5s%6s%15s%7s  statement%n", "item", "mean", "95% CI",
            "top-2");
        for (int i = 0; i < items.size(); i++) {
            double[] s = itemStats(column(rows, 3 + i, null));
            String ci = String.format("[%.2f, %.2f]", s[1] - s[3], s[1] + s[3]);
            System.out.printf("%-5s%6.2f%15s%6.0f%%  %s%n", items.get(i)[0], s[1],
                ci, s[4] * 100, items.get(i)[1]);
        }
        System.out.println(
            "\nQ2 'I know when my laundry will be ready' by age group:");
        for (String group : new String[] {"18-29", "30-44", "45+"}) {
            double[] s = itemStats(column(rows, 4, group));
            System.out.printf("  %-6s n=%-3d mean %.2f +/- %.2f%n", group,
                (int) s[0], s[1], s[3]);
        }
        System.out.printf("%nMargin of error of a percentage at n=%d: "
            + "+/-%.1f points%n", rows.size(), marginOfError(rows.size()));
    }
}
```

`Locale.setDefault(Locale.US)` makes `printf` write a decimal point instead of the decimal comma that an Indonesian system locale would produce, so that both programs print identical output everywhere.

5.6.4 Running it

From the `python` folder, run `python3 survey_stats.py`; from the `java` folder, compile with `javac -d out *.java` and run `java -cp out SurveyStats`. Both print:

```
60 respondents, 1 = strongly disagree ... 5 = strongly agree
item   mean         95% CI  top-2  statement
Q1     3.83   [3.58, 4.09]    67%  Ordering by WhatsApp is easy
Q2     2.47   [2.21, 2.72]    15%  I know when my laundry will be ready
Q3     3.03   [2.78, 3.28]    30%  The price is clear before I pay
Q4     3.95   [3.71, 4.19]    72%  I trust my clothes are handled with care
Q5     3.10   [2.86, 3.34]    33%  The pickup times suit me
Q6     3.57   [3.33, 3.81]    57%  I would recommend the laundry to a friend

Q2 'I know when my laundry will be ready' by age group:
  18-29  n=22  mean 2.68 +/- 0.42
  30-44  n=24  mean 2.46 +/- 0.39
  45+    n=14  mean 2.14 +/- 0.63

Margin of error of a percentage at n=60: +/-12.7 points
```

Figure 5.2 draws the same answers as a *diverging stacked bar chart*, the clearest picture of Likert data: disagreement extends to the left of the center line, agreement to the right, and neutral answers straddle the line.

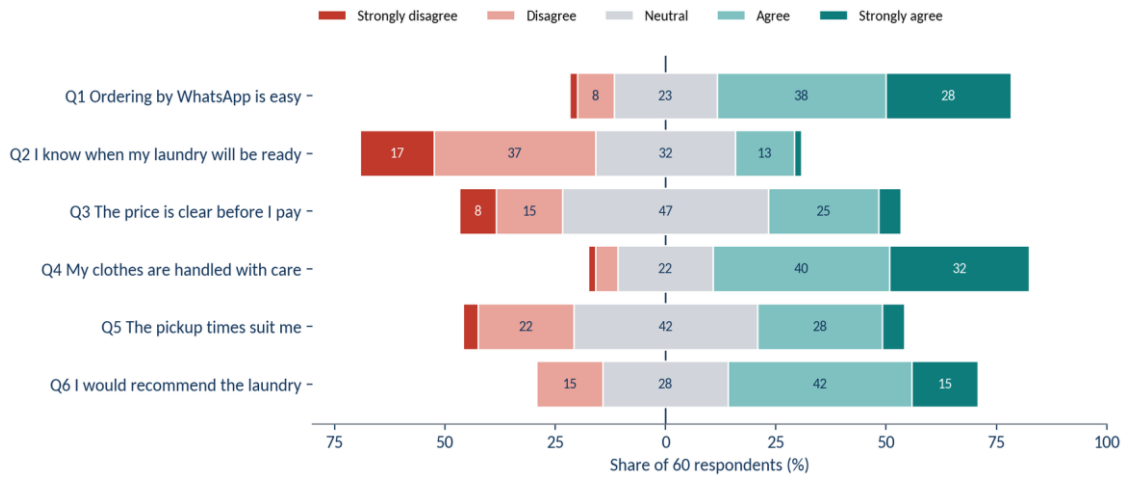


Figure 5.2 The six survey items as a diverging stacked bar chart (60 respondents); numbers are percentages of respondents, and item Q2 leans furthest to the disagreeing side.

5.6.5 Reading the results

The survey confirms what the interviews suggested and adds proportions. Customers find ordering by WhatsApp easy (mean 3.83, two thirds agree) and trust that their clothes are handled with care (3.95, 72% agree): the channel and the core service are not the problem. The weakest item by far is Q2, "I know when my laundry will be ready", with a mean of 2.47 and only 15% agreeing; its confidence interval, 2.21 to 2.72, lies entirely below the neutral midpoint of 3, so the result is not an accident of sampling. Price clarity (Q3) and pickup times (Q5) sit near neutral, with many neutral answers, which usually means "sometimes yes, sometimes no".

The age breakdown of Q2 shows the oldest group at 2.14, the youngest at 2.68. The intervals overlap widely (the oldest group has only fourteen people, so its interval is ±0.63), so the survey alone cannot prove that older customers suffer more. It is a hint, consistent with two interviews in which older customers said they had to ask a grandchild to check the chat, and a hint is what it should be reported as.

Finally, the margin of error: with sixty answers, any percentage in the table is uncertain by up to about 13 points. "15% agree" means "somewhere between a few percent and about a quarter". That is precise enough for Bu Rina's decision, because the other items are far higher, but it would not be precise enough to track a small improvement month by month.

> **KEY INSIGHT.** The interviews told Bu Rina *what* the problems were and *why* they mattered; the survey told her *how widespread* each one was. Neither alone would have been enough: the interviews could have over-weighted one vivid story, and a survey written without interviews would not have contained the item that turned out to matter most.

5.7 Making Sense of Qualitative Data

5.7.1 From notes to codes

Twelve interviews produce perhaps sixty pages of notes. The first step in making sense of them is *coding*: reading the notes line by line and attaching short labels, *codes*, to each passage that says something relevant. "Kalau sore sering nanya sudah selesai belum, dibalasnya malam" ("in the afternoon I often ask whether it's done, and the reply comes at night") gets the codes `ready-time-unknown` and `reply-slow-evening`. Codes are named in a consistent, short form, and the list of codes with a one-line definition of each, the *codebook*, grows as the analysis proceeds.

Braun and Clarke's *thematic analysis* describes the process in six phases: become familiar with the data, generate initial codes, search for themes (groups of codes that belong together), review the themes against the data, define and name them, and write them up with quotes as evidence [13]. The method is flexible, and its central discipline is to go back to the data at every step: a theme that is not supported by several passages from several participants is an opinion, not a finding.

5.7.2 The affinity diagram

The classic way to search for themes as a team is the *affinity diagram*. Each observation or quote is written on a sticky note, one idea per note; the team puts all the notes on a wall and, without talking at first, moves them into groups that feel related; then it names each group with a sentence written from the customer's point of view. For Laundry Bersih the wall ended with five groups:

- "I never know when my clothes will be back" (`ready-time-unknown`, `reply-slow-evening`);
- "I only learn the price after it is too late to change my mind" (`price-after-weighing`);
- "The pickup has to fit around my work" (`pickup-window-short`, `bulk-order-weekly`);
- "I want to be sure my things are safe" (`wants-photo-proof`, `separate-items-worry`, `lost-sock-story`);
- "Ordering must work for me the way I talk" (`voice-note-order`, `small-text-hard`, `child-helps-order`, `pays-by-qris`).

The wording of the group names matters. "Communication problems" is a label; "I never know when my clothes will be back" is a finding someone can design for.

5.7.3 When have we interviewed enough?

Every qualitative study must decide when to stop. The usual answer is *saturation*: stop when new interviews no longer produce new codes. Stated like that, the rule is vague. Guest, Namey and Chen proposed a way to make it precise and to report it [6]. Choose a *base size* b, the number of first interviews whose codes form the base (they suggest 4); a *run length* r, the number of consecutive later interviews considered together (they suggest 2); and a *threshold* τ for new information (they suggest 5%). After the base, look at each run of r interviews. If B is the number of distinct codes found in the base interviews and N_{run} the number of codes that first appear in the run, saturation is reached at the end of the first run for which

$$\frac{N_{\text{run}}}{B} \le \tau \tag{5.4}$$

The ratio compares what the run adds with what the base found. A run that adds nothing, or very little, suggests that further interviews of the same kind would add little too.

The coded notes of the laundry study are in `data/codes.csv`, one line per code mentioned in an interview:

```
interview,participant,code
1,P01 student,ready-time-unknown
1,P01 student,price-after-weighing
1,P01 student,voice-note-order
```

5.7.4 The saturation program

Listing 5.3 counts, for each code, in how many interviews it appears; lists the codes that each interview added for the first time; and applies Equation (5.4) with $b = 4$, $r = 2$ and $\tau = 0.05$.

Listing 5.3 `saturation.py`: code frequencies, new codes per interview and the saturation rule of Guest, Namey and Chen.

```
"""Count interview codes and test for thematic saturation
(base size 4, run length 2, new-information threshold 5%).
Usage: python saturation.py [codes.csv]"""
import csv
import sys


def load(path):
    """Return {interview number: set of codes} in interview order."""
    by_interview = {}
    with open(path, newline="", encoding="utf-8") as f:
        for r in csv.DictReader(f):
            by_interview.setdefault(int(r["interview"]), set()).add(r["code"])
    return dict(sorted(by_interview.items()))


def new_codes_per_interview(by_interview):
    seen, result = set(), []
    for number, codes in by_interview.items():
        fresh = codes - seen
        result.append((number, len(fresh), sorted(fresh)))
        seen |= codes
    return result


def saturation_point(by_interview, base=4, run=2, threshold=0.05):
    """First interview that ends a run whose new codes are <= threshold
    of the codes found in the base interviews; None if never reached."""
    numbers = list(by_interview)
    base_codes = set().union(*(by_interview[n] for n in numbers[:base]))
    seen = set(base_codes)
    fresh_counts = []
    for n in numbers[base:]:
        fresh_counts.append(len(by_interview[n] - seen))
        seen |= by_interview[n]
    for i in range(len(fresh_counts) - run + 1):
        if sum(fresh_counts[i:i + run]) / len(base_codes) <= threshold:
            return numbers[base + i + run - 1], len(base_codes)
    return None, len(base_codes)
```

Listing 5.3 (Continued) *saturation.py: code frequencies, new codes per interview and the saturation rule of Guest, Namey and Chen.*

```
def main(args):
    path = args[0] if args else "../data/codes.csv"
    by_interview = load(path)
    counts = {}
    for codes in by_interview.values():
        for c in codes:
            counts[c] = counts.get(c, 0) + 1
    total = len(by_interview)
    print(f"{total} interviews, {len(counts)} distinct codes")
    for code, k in sorted(counts.items(), key=lambda kv: (-kv[1], kv[0])):
        print(f"  {code:<22}{k:>3} of {total}  {'#' * k}")
    print("\nNew codes per interview:")
    for number, k, fresh in new_codes_per_interview(by_interview):
        print(f"  interview {number:>2}: {k}  {' '.join(fresh)}".rstrip())
    point, base = saturation_point(by_interview)
    if point is None:
        print("\nNo saturation yet: keep interviewing.")
    else:
        print(f"\nBase: {base} codes in interviews 1-4. Saturation reached "
              f"at interview {point} (run of 2 with <= 5% new codes).")


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

Listing 5.4 is the Java version. A `TreeMap` keeps the interviews in order and a `TreeSet` keeps each interview's codes sorted, so that the new codes are listed in the same alphabetical order as in Python.

Listing 5.4 `Saturation.java`: the same analysis in Java.

```java
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.TreeMap;
import java.util.TreeSet;

/** Count interview codes and test for thematic saturation
 *  (base size 4, run length 2, new-information threshold 5%).
 *  Usage: java Saturation [codes.csv] */
public class Saturation {
    /** Returns interview number -> set of codes, in interview order. */
    static TreeMap<Integer, Set<String>> load(String path) throws IOException {
        TreeMap<Integer, Set<String>> byInterview = new TreeMap<>();
        List<String> lines = Files.readAllLines(Path.of(path));
        for (String line : lines.subList(1, lines.size())) {
            if (line.isBlank()) continue;
            String[] f = line.split(",");
            byInterview.computeIfAbsent(Integer.parseInt(f[0]),
                k -> new TreeSet<>()).add(f[2]);
        }
        return byInterview;
    }

    /** Returns {interview that ends the first run with <= threshold new
     *  codes (or -1), number of codes in the base interviews}. */
    static int[] saturationPoint(TreeMap<Integer, Set<String>> byInterview,
                                 int base, int run, double threshold) {
        List<Integer> numbers = new ArrayList<>(byInterview.keySet());
        Set<String> seen = new TreeSet<>();
        for (int i = 0; i < base; i++)
            seen.addAll(byInterview.get(numbers.get(i)));
        int baseCount = seen.size();
        List<Integer> fresh = new ArrayList<>();
        for (int i = base; i < numbers.size(); i++) {
            Set<String> codes = byInterview.get(numbers.get(i));
            int k = 0;
            for (String c : codes) if (!seen.contains(c)) k++;
            fresh.add(k);
            seen.addAll(codes);
        }
        for (int i = 0; i + run <= fresh.size(); i++) {
```

Listing 5.4 (Continued) *Saturation.java: the same analysis in Java.*

```
            int sum = 0;
            for (int j = i; j < i + run; j++) sum += fresh.get(j);
            if ((double) sum / baseCount <= threshold)
                return new int[] {numbers.get(base + i + run - 1), baseCount};
        }
        return new int[] {-1, baseCount};
    }

    public static void main(String[] args) throws IOException {
        String path = args.length > 0 ? args[0] : "../data/codes.csv";
        TreeMap<Integer, Set<String>> byInterview = load(path);
        Map<String, Integer> counts = new HashMap<>();
        for (Set<String> codes : byInterview.values())
            for (String c : codes) counts.merge(c, 1, Integer::sum);
        int total = byInterview.size();
        System.out.println(total + " interviews, " + counts.size()
            + " distinct codes");
        List<Map.Entry<String, Integer>> sorted =
            new ArrayList<>(counts.entrySet());
        sorted.sort((a, b) -> a.getValue().equals(b.getValue())
            ? a.getKey().compareTo(b.getKey()) : b.getValue() - a.getValue());
        for (Map.Entry<String, Integer> e : sorted)
            System.out.printf("  %-22s%3d of %d  %s%n", e.getKey(), e.getValue(),
                total, "#".repeat(e.getValue()));
        System.out.println("\nNew codes per interview:");
        Set<String> seen = new TreeSet<>();
        for (Map.Entry<Integer, Set<String>> e : byInterview.entrySet()) {
            List<String> fresh = new ArrayList<>();
            for (String c : e.getValue()) if (!seen.contains(c)) fresh.add(c);
            seen.addAll(e.getValue());
            String line = String.format("  interview %2d: %d  %s", e.getKey(),
                fresh.size(), String.join(" ", fresh));
            System.out.println(line.stripTrailing());
        }
        int[] sat = saturationPoint(byInterview, 4, 2, 0.05);
        if (sat[0] < 0)
            System.out.println("\nNo saturation yet: keep interviewing.");
        else System.out.printf("%nBase: %d codes in interviews 1-4. Saturation "
            + "reached at interview %d (run of 2 with <= 5%% new codes).%n",
            sat[1], sat[0]);
    }
}
```

Running `python3 saturation.py` or `java -cp out Saturation` prints:

```
12 interviews, 12 distinct codes
  ready-time-unknown     11 of 12  ###########
  price-after-weighing    7 of 12  #######
  reply-slow-evening      5 of 12  #####
  voice-note-order        5 of 12  #####
  pickup-window-short     4 of 12  ####
  pays-by-qris            3 of 12  ###
  wants-photo-proof       3 of 12  ###
  child-helps-order       2 of 12  ##
  small-text-hard         2 of 12  ##
  bulk-order-weekly       1 of 12  #
  lost-sock-story         1 of 12  #
  separate-items-worry    1 of 12  #

New codes per interview:
  interview  1: 4  price-after-weighing ready-time-unknown reply-slow-evening voice-note-
order
  interview  2: 2  pickup-window-short wants-photo-proof
  interview  3: 2  pays-by-qris separate-items-worry
  interview  4: 2  child-helps-order small-text-hard
  interview  5: 1  bulk-order-weekly
  interview  6: 0
  interview  7: 0
  interview  8: 1  lost-sock-story
  interview  9: 0
  interview 10: 0
  interview 11: 0
  interview 12: 0

Base: 10 codes in interviews 1-4. Saturation reached at interview 7 (run of 2 with <= 5%
new codes).
```

The base interviews 1 to 4 found ten codes. Interview 5 added one (`bulk-order-weekly`, from the boarding-house owner), so the run of interviews 5 and 6 added $1/10 = 10\%$, above the threshold. The run of interviews 6 and 7 added nothing, $0/10 = 0\%$, so the rule declares saturation at interview 7. **Figure 5.3** shows the same story: the cumulative number of codes climbs steeply in the base and then flattens.

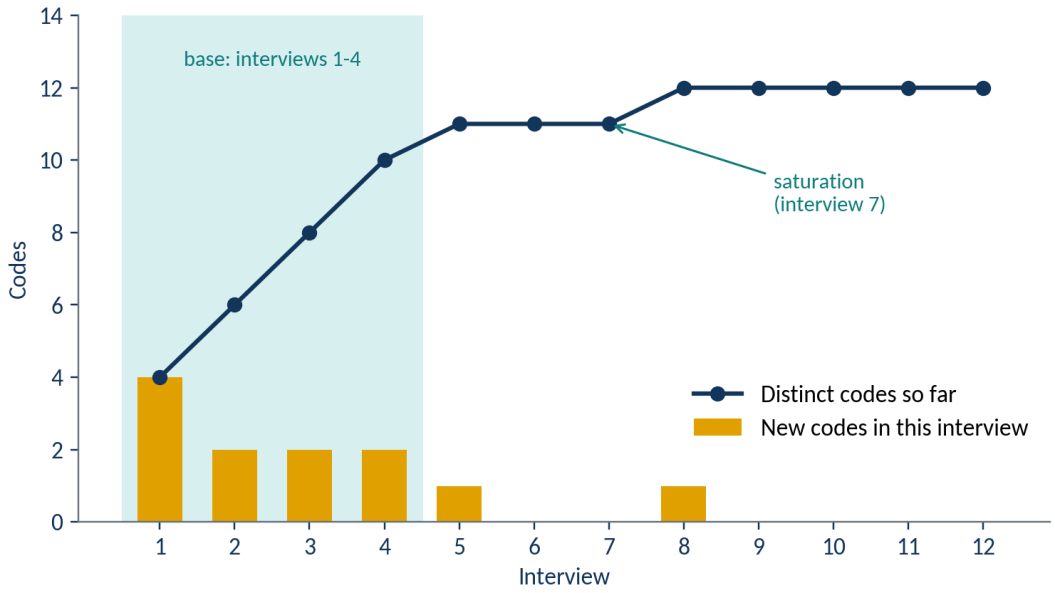


Figure 5.3 New codes per interview (bars) and distinct codes found so far (line) in the laundry study; the rule of Equation (5.4) declares saturation at interview 7.

Notice interview 8. The student who told the story of a lost sock added a new code *after* saturation. That is not a failure of the rule; it is a reminder of what saturation means. It means

that the *common* themes have been found, not that every possible story has been heard. The lost sock appears once in twelve interviews; the unknown ready time appears in eleven. Saturation told the team that it could stop recruiting new kinds of the same participants and start analyzing; it did not promise that rare problems do not exist. A study that must find rare problems, such as a safety study, needs more participants and a different stopping rule.

The frequency table is also a warning against over-reading counts in qualitative data. That `ready-time-unknown` appears in 11 of 12 interviews is strong evidence that the problem is common among the *kinds of people interviewed*. It is not an estimate of a percentage among all customers; that is what the survey is for.

5.8 Testing the Programs

Analysis programs deserve tests, because a wrong formula produces numbers that look perfectly plausible. **Listing 5.5** checks the *t* value against the printed table for 59 and 9 degrees of freedom, a hand-computed mean and top-2-box share, the margin of error for 400 answers (4.9 points, computed in Section 5.5.3), the saturation point of the study data, and the case where every interview brings new codes and saturation is never reached.

Listing 5.5 `test_research.py`: tests of the survey statistics and the saturation rule.

```
"""Tests for survey_stats.py and saturation.py.
Run: python -m unittest test_research.py"""
import unittest
from survey_stats import item_stats, margin_of_error, t_critical
from saturation import load, saturation_point


class ResearchTest(unittest.TestCase):
    def test_t_critical_close_to_table(self):
        self.assertAlmostEqual(t_critical(59), 2.001, places=3)
        self.assertAlmostEqual(t_critical(9), 2.262, places=2)

    def test_item_stats(self):
        n, mean, sd, half, top2 = item_stats([5, 4, 4, 2, 1])
        self.assertEqual(n, 5)
        self.assertAlmostEqual(mean, 3.2)
        self.assertAlmostEqual(top2, 0.6)

    def test_margin_of_error(self):
        self.assertAlmostEqual(margin_of_error(400), 4.9, places=1)

    def test_saturation_on_study_data(self):
        point, base = saturation_point(load("../data/codes.csv"))
        self.assertEqual((point, base), (7, 10))

    def test_no_saturation_when_every_interview_is_new(self):
        data = {i: {f"c{i}a", f"c{i}b"} for i in range(1, 8)}
        self.assertEqual(saturation_point(data)[0], None)


if __name__ == "__main__":
    unittest.main()
```

```
Ran 5 tests in 0.001s

OK
```

Listing 5.6 performs the same checks in Java without a test library, so that it runs with nothing but the JDK.

Listing 5.6 `ResearchTest.java`: the same checks in Java.

```java
import java.io.IOException;
import java.util.List;
import java.util.Set;
import java.util.TreeMap;

/** Tests for SurveyStats and Saturation, without a test library.
 *  Run: java -cp out ResearchTest */
public class ResearchTest {
    static int passed = 0;

    static void check(boolean ok, String name) {
        if (!ok) throw new AssertionError("FAILED: " + name);
        passed++;
    }

    public static void main(String[] args) throws IOException {
        check(Math.abs(SurveyStats.tCritical(59) - 2.001) < 0.0005, "t(59)");
        check(Math.abs(SurveyStats.tCritical(9) - 2.262) < 0.005, "t(9)");
        double[] s = SurveyStats.itemStats(List.of(5, 4, 4, 2, 1));
        check(s[0] == 5 && Math.abs(s[1] - 3.2) < 1e-9, "mean");
        check(Math.abs(s[4] - 0.6) < 1e-9, "top-2-box");
        check(Math.abs(SurveyStats.marginOfError(400) - 4.9) < 0.05, "margin");
        int[] sat = Saturation.saturationPoint(
            Saturation.load("../data/codes.csv"), 4, 2, 0.05);
        check(sat[0] == 7 && sat[1] == 10, "saturation on study data");
        TreeMap<Integer, Set<String>> allNew = new TreeMap<>();
        for (int i = 1; i <= 7; i++)
            allNew.put(i, Set.of("c" + i + "a", "c" + i + "b"));
        check(Saturation.saturationPoint(allNew, 4, 2, 0.05)[0] == -1, "never");
        System.out.println(passed + " checks passed");
    }
}
```

```
7 checks passed
```

5.9 From Findings to Insights

5.9.1 Triangulating the evidence

Bu Rina now had three kinds of evidence. The order book (behavioral, quantitative) showed fewer repeat customers. The interviews and the morning on the motorbike (behavioral and attitudinal, qualitative) showed customers who did not know when their clothes would return, who learned the price hours after pickup, and who wrote “sudah selesai belum?” again and again. The survey (attitudinal, quantitative) showed that this uncertainty was the single most widespread problem. Three corners of **Figure 5.1** pointed at the same place.

What about the explanations everyone had guessed? The cheaper competitor came up in two interviews, both from students, and neither said it was the main reason. The long holiday explained part of the drop among students but not among workers and families. Dimas’s late evening visits were real, but they were a symptom: he came late because the “ready” message went out late, and customers were not home to receive him. And nobody, in twelve interviews, asked for an app; several said they liked WhatsApp *because* it needed no new app.

5.9.2 Writing insight statements

A finding says what is true; an *insight* says why it matters for design. A useful form is: *[who] needs [what] because [why], but today [obstacle]*. Five insights came out of the study:

1. Customers need to know when their laundry will be back *at the time they hand it over*, because they plan their clothes for the week around it, but today they learn it only when Dimas is about to leave.
2. Customers need to know the price before they commit, because students budget to the thousand rupiah, but today the price comes after weighing, hours later.
3. Working customers need a pickup and return time that fits outside working hours, because nobody is home in the day, but today the times are set by the shop's route.
4. Customers need reassurance that nothing was lost, because clothes are personal and some are expensive, but today they receive no count or photo.
5. Older customers need to order by voice and read large, simple messages, because typing and small text are hard for them, but today the price list is a small image.

Each insight can be turned into a *How might we…?* question for the ideation step of Chapter 4: "How might we tell customers the ready time at pickup?", "How might we give a price estimate before weighing?".

5.9.3 What Bu Rina did

The recommendation was not an app. It was four changes to the WhatsApp service, cheap enough to try the next week: a message at pickup with an *estimated* price range (from the bag size and a price per kilogram), a promised ready time and a count of items; a "ready" message sent by 4 p.m. with two return slots to choose from, one of them after 6 p.m.; a photo of the folded laundry with the count before return; and a large-type price list sent as text, not as an image, with a note that voice notes are welcome. Chapter 6 prototypes these messages, and Chapter 7 tests them. The research did not tell Bu Rina what to build; it told her what problem to solve, and which of the tempting solutions would have missed it.

5.10 AI in User Research, as of 2026

Generative AI entered user research quickly, and by 2026 almost every research tool offers it: automatic transcription of recordings, summaries of interviews, suggested codes, clustering of survey comments, and even "interviews" with simulated participants. Some of this is genuinely helpful; some of it is dangerous in a way that is easy to miss.

Where it helps. Transcription saves hours per interview and is now accurate enough in Indonesian for analysis, as long as someone checks names and numbers. A language model can propose a first set of codes for a long transcript, which a researcher then accepts, merges or rejects. It can cluster hundreds of free-text survey answers into candidate groups for a person to review. In a CHI 2026 user study of a locally deployed open-source model for qualitative coding, researchers valued the tool mainly for speed and for surface-level extraction, trusted it less for latent meaning that needs wider context, and asked for outputs that always point back to the source passages and for codebooks that can be fixed so that results are reproducible [14]. Those are good rules for any AI-assisted analysis: every suggested code must link to the passage it came from, and a person decides.

Where it harms: synthetic users. Several products now sell "synthetic users", AI-generated personas that answer interview questions as if they were real customers. The Nielsen Norman Group reviewed such tools and concluded that they may help with desk research, with drafting an interview guide or with a first proto-persona to be checked later, but that they must not replace research with real people or be used to validate a design: the simulated participants are too

agreeable, express enthusiasm about everything without priorities, and have no actual experience of the product [15]. Ask a synthetic laundry customer whether she wants an app with tracking, and she will say yes, enthusiastically, which is exactly the answer that would have sent Bu Rina in the wrong direction.

Privacy. Interview recordings are personal data. Uploading them to a cloud AI service transfers them to another processor, which under UU PDP and PP 33/2026 must be covered by the participant's consent and by an agreement with that processor [7], [8]. A study that uses AI should say so in the consent script, prefer tools that process data locally, and never paste a participant's name, phone number or face into a public chatbot.

> **WARNING.** An AI summary of an interview is an interpretation, not a quote. Never put a sentence from an AI summary in quotation marks in a report, and never let a summary replace reading the notes of at least a few interviews yourself.

5.11 Using It for Real

The same study shape, one page of plan, a handful of interviews and a morning of observation, then a short survey, fits most small services. Three examples show how to adapt it.

A puskesmas queue. The research question is "why do patients arrive two hours before their turn?". Contextual inquiry means sitting in the waiting area at 7 a.m. with permission from the head of the puskesmas; interviews happen while people wait; a survey is short and on paper, read aloud for those who prefer it. Consent must be especially careful: health information is *specific* personal data under UU PDP, so the study asks nothing about illness, only about the queue [7]. If a participant mentions symptoms, the researcher listens politely, does not advise, and suggests speaking to the nurse or doctor on duty.

A warung's ordering by WhatsApp. The question might be "why do regular customers order at the counter instead of by message?". Observation at lunchtime, five interviews, and the warung's own chat history (with the owner's agreement, and only counts and times, not customers' names) are enough.

An ojek base. Drivers are busy and paid by the trip; interview them during waiting time, keep it to fifteen minutes, and pay for the time. The master-apprentice stance matters here: a driver knows the city better than any researcher.

Before starting, go through this checklist:

1. The decision and the research questions are written down.
2. The method follows from the question, not the other way round.
3. The participants include people who left, not only people who stayed, and people at the edges (older, less connected, with disabilities).
4. The consent script separates taking part from recording, and says what happens to the data.
5. The interview guide starts from the last specific occasion and contains no leading or predictive questions.
6. Survey items are written after the interviews, in the participants' words, one idea each.
7. The stopping rule for interviews and the margin of error for the survey are stated.
8. Every finding in the report points to its evidence: quotes, observations, numbers.

5.12 HCI in Practice: Profitina

Profitina is an AI Visibility (GEO) platform: it measures how ready a brand's website is to be cited by AI answer engines and helps fix what it finds. Its users are brand owners and marketers, many of them without a technical background, and the chapter's methods apply to it in three places.

Making a score legible. A readiness score is only useful if a non-technical owner understands what it means and what to do next. The questions a team asks here are exactly those of Section 5.4: not "do you like the dashboard?" but "tell me about the last time you looked at your score; what did you do next?". The honest answer to the second question shows whether the explanation of each signal, which the product gives, actually leads to action.

Consent that is understood. Profitina implements the self-service rights of UU PDP 27/2022: explicit consent, data export, email and password change and a full account purge. Whether a consent screen is *understood*, rather than merely clicked, is a research question: a short interview after sign-up ("what did you agree to just now?") tests it far better than a count of clicks, and PP 33/2026's rule that consent must be informed, specific and not bundled makes that test part of doing the job properly [8].

A feedback pipeline as continuous research. Profitina runs a transparent pipeline in which users submit feedback on AI responses, the submissions are curated into a dataset, and the local model (Qwen2.5, running on Profitina's own server) is trained and measured before a release. From the research point of view this is a permanent, behavioral, qualitative-and-quantitative channel: people show what went wrong in their own words, at the moment it happened. The discipline of Section 5.7 applies unchanged: code the feedback, look for themes, and check each theme against the raw submissions. And because the AI processing is local, the privacy concern of Section 5.10 about sending data to third parties does not arise for that part of the analysis.

Profitina advises and the user decides; research with its users is how the team learns whether the advice is understood.

5.13 Common Mistakes, and How to Avoid Them

- **Starting with a method instead of a question.** "Let's do a survey" produces data nobody uses. Write the decision and the research questions first.
- **Asking people to predict or design.** "Would you use…?" and "what features do you want?" produce polite, useless answers. Ask about the last specific occasion.
- **Talking only to happy customers.** Regulars explain why people stay. Recruit people who left, and people at the edges.
- **Leading questions and helpful interruptions.** Suggesting an answer, or finishing the participant's sentence, replaces their experience with yours. Use silence and echo.
- **Mixing quotes, observations and interpretations in notes.** Later nobody can tell what the participant said from what the researcher thought. Mark each line.
- **Writing the survey before the interviews.** It then measures the team's assumptions. Write items in the participants' words.
- **Reporting a percentage without its margin.** "15% agree" from sixty answers is ±13 points; say so, and do not compare differences smaller than the margin.
- **Treating interview counts as population percentages.** "11 of 12 interviewees" is evidence that a problem is common in the group studied, not an estimate for all customers.

- **Declaring saturation without a rule.** State the base, run and threshold, as Equation (5.4) does, and remember that saturation does not rule out rare problems.
- **Using synthetic users as evidence.** They can help prepare a study; they cannot replace one.
- **One consent for everything.** Taking part, recording, and using an AI service are separate consents; ask them separately.

5.14 Chapter Summary

- User research finds out what people do, need and why, before and while designing. It starts with a decision and research questions, not with a method.
- Methods differ along two axes: what people say versus what they do, and why/how versus how many (**Figure 5.1**). Combining methods from different corners, triangulation, gives confidence.
- A one-page research plan (**Table 5.2**) fixes the decision, questions, methods, participants, ethics and output. Consent under UU PDP 27/2022 and PP 33/2026 must be informed, specific, unbundled and revocable.
- Semi-structured interviews and contextual inquiry reveal behavior and reasons; ask about specific recent occasions and avoid leading, predictive and double questions (**Table 5.3**).
- Surveys measure how widespread something is. The margin of error of a proportion (Equation (5.1)) and the confidence interval of a mean (Equation (5.2), with the t value of Equation (5.3)) say how precise the estimates are.
- Qualitative data is coded, grouped into themes with an affinity diagram, and analyzed until saturation, which the rule of Equation (5.4) makes explicit.
- In the laundry study, not knowing when the laundry would be ready was the most common problem (Q2 mean 2.47, 15% agreement, and 11 of 12 interviews); nobody asked for an app.
- AI helps with transcription and first-pass coding, always with links back to the source and a person deciding; synthetic users do not replace real participants.

5.15 Exercises

1. For each of the following questions, say whether it is about what people say or do, and whether it is qualitative or quantitative, and name a suitable method: (a) What share of Bu Rina's customers pay by QRIS? (b) Why do boarding-house owners order on Mondays? (c) Can a 65-year-old customer find the price list in the chat? (d) Which of two wordings of the "ready" message leads to more same-day collections?
2. Rewrite each of these interview questions so that it does not lead or ask for a prediction: (a) "You'd like to get a photo of your laundry, right?" (b) "Would you pay more for express service?" (c) "Is our pickup convenient and fast?"
3. Using Equation (5.1), compute by hand the 95% margin of error for a proportion near 50% with $n = 150$, and the number of answers needed for a margin of 5 points.
4. Using Equation (5.3), compute t for 9 degrees of freedom and compare it with the printed table value 2.262. Then compute the 95% confidence interval for the ten answers 4, 5, 3, 4, 4, 2, 5, 4, 3, 4.
5. Extend `survey_stats` (in Python or Java) so that it also prints, for every item, the mean for WhatsApp customers and for walk-in customers. Run it and say whether any difference is larger than the uncertainty.

6. Suppose interview 6 of the laundry study had also mentioned `lost-sock-story`. By hand, apply Equation (5.4) with $b = 4$, $r = 2$, $\tau = 5\%$: where is saturation reached now? Check by editing a copy of `codes.csv` and running `saturation.py` on it.
7. Write a consent script of six parts for a diary study in which puskesmas patients send a voice note each time they visit. Which consents must be asked separately, and what must happen if a patient withdraws after two weeks?
8. Turn the finding "Older customers ask a grandchild to read the price message" into an insight statement in the form of Section 5.9.2 and into two *How might we* questions.

5.16 Solutions to the Exercises

1. (a) Do, quantitative: count payment methods in the order book or the payment records; no survey needed, since the behavior is recorded. (b) Say and do, qualitative: interviews with boarding-house owners, ideally a diary study over a few weeks. (c) Do, qualitative: a usability test with older participants (Chapter 7). (d) Do, quantitative: an A/B test, sending each wording to half of the customers at random and counting same-day collections (Chapter 7).

2. (a) "Tell me about the moment you got your laundry back last time. How did you check that everything was there?" (b) "Tell me about a time you needed your clothes back quickly. What did you do?" (c) Split and open it: "Tell me how the pickup went last time", then "How did the timing fit your day?".

3. For $n = 150$: $E = 1.96 \times \sqrt{0.25/150} = 1.96 \times 0.0408 = 0.080$, about ±8.0 points. For $E = 0.05$: $n = 1.96^2 \times 0.25/0.05^2 = 3.8416 \times 0.25/0.0025 = 384.16$, so 385 answers. The program agrees: `margin_of_error(150)` returns 8.0 and `margin_of_error(385)` returns 4.99.

4. With $z = 1.959964$: $z^3 = 7.529$, $z^5 = 28.925$. First term $(7.529 + 1.960)/(4 \times 9) = 9.489/36 = 0.2636$. Second term $(144.62 + 120.46 + 5.88)/(96 \times 81) = 270.97/7776 = 0.0348$. So $t \approx 1.960 + 0.264 + 0.035 = 2.259$, within 0.003 of 2.262; this is why the test allows a tolerance of 0.005 at 9 degrees of freedom. For the ten answers: sum 38, mean 3.8; squared deviations 0.04, 1.44, 0.64, 0.04, 0.04, 3.24, 1.44, 0.04, 0.64, 0.04, total 7.60; $s = \sqrt{7.60/9} = 0.919$; half-width $2.259 \times 0.919/\sqrt{10} = 2.259 \times 0.2906 = 0.656$. The interval is [3.14,4.46].

5. In Python, add after the age breakdown:

```
    print("\nBy channel (mean, n):")
    for key, text in items:
        parts = []
        for ch in ("wa", "walkin"):
            vals = [int(r[key]) for r in rows if r["channel"] == ch]
            n, mean, sd, half, top2 = item_stats(vals)
            parts.append(f"{ch} {mean:.2f} +/- {half:.2f} (n={n})")
        print(f"  {key}: " + "; ".join(parts))
```

In Java, the same loop filters on column 2 before calling `itemStats`. The study has 49 WhatsApp respondents and only 11 walk-in respondents, so the walk-in half-widths are wide, from ±0.56 to ±0.78. The largest difference is on Q2: 2.37 ± 0.29 for WhatsApp customers against 2.91 ± 0.56 for walk-in customers. The two intervals, [2.08, 2.66] and [2.35, 3.47], overlap, so the survey gives no firm evidence of a difference. The difference points in a plausible direction (walk-in customers see the shop and can ask in person), which makes it worth a question in the next round of interviews. The honest conclusion is "no evidence of a difference yet", not "no difference".

6. The base is unchanged: ten codes in interviews 1–4. Interview 5 adds `bulk-order-weekly`; interview 6 now adds `lost-sock-story`. Run (5, 6): $2/10 = 20\%$. Run (6, 7): interview 6 adds 1, interview 7 adds 0, so $1/10 = 10\%$. Run (7, 8): interview 8 no longer adds anything new (the lost sock was already seen), so $0/10 = 0\% \leq 5\%$ and saturation is reached at interview 8. `saturation.py` on the edited file prints "Saturation reached at interview 8".

7. A suitable script: (1) who and why: "the puskesmas and a student team want to shorten waiting"; (2) what happens: "for four weeks, send a short voice note to the study number each time you come, saying when you arrived and when you were called"; (3) what is kept and who hears it: "only the study team; notes use a code; voice notes deleted after they are written down; we will not ask about your illness"; (4) voluntary, stop any time, no effect on your care; (5) thank-you: phone credit each week; (6) separate questions: "Do you agree to take part?", "Do you agree that we keep your voice notes until they are written down?" and, if an AI transcription service is used, "Do you agree that the voice notes are transcribed by that service?". If a patient withdraws, processing of their data must stop within 72 hours of the verified request and the patient is told it has stopped, as PP 33/2026 requires; their existing voice notes are deleted, and their care is unaffected.

8. Insight: "Older customers need to read the price and ready time without help, because they want to manage their own laundry, but today the messages are small images and long texts that they ask a grandchild to read." Questions: "How might we make the price message readable at a glance on any phone?" and "How might we let older customers confirm an order with a single voice note?".

References

[1] ANTARA News, "Survei APJII: Penetrasi internet di Indonesia 2026 capai 81,7 persen," May 21, 2026, reporting Asosiasi Penyelenggara Jasa Internet Indonesia, *Survei Penetrasi Internet Indonesia 2026*. [Online]. Available: https://www.antaranews.com/berita/5576225/survei-apjii-penetrasi-internet-di-indonesia-2026-capai-817-persen

[2] StatCounter Global Stats, "Desktop vs Mobile vs Tablet Market Share Indonesia," data for August 2026, accessed Sep. 2026. [Online]. Available: https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/indonesia

[3] International Organization for Standardization, *ISO 9241-210:2019 Ergonomics of Human-System Interaction — Part 210: Human-Centred Design for Interactive Systems*. Geneva, Switzerland: ISO, 2019.

[4] C. Rohrer, "When to Use Which User-Experience Research Methods," Nielsen Norman Group, updated 2022. [Online]. Available: https://www.nngroup.com/articles/which-ux-research-methods/

[5] Republik Indonesia, *Undang-Undang Nomor 8 Tahun 2016 tentang Penyandang Disabilitas*. Jakarta, Indonesia, 2016.

[6] G. Guest, E. Namey, and M. Chen, "A simple method to assess and report thematic saturation in qualitative research," *PLOS ONE*, vol. 15, no. 5, e0232076, 2020, doi: 10.1371/journal.pone.0232076.

[7] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*. Jakarta, Indonesia, 2022.

[8] Bisnis.com, "Aturan Turunan UU PDP: Persetujuan Data Konsumen Lebih Ketat," Aug. 29, 2026, on Peraturan Pemerintah Nomor 33 Tahun 2026. [Online]. Available: https://teknologi.bisnis.com/read/20260829/84/1999753/aturan-turunan-uu-pdp-persetujuan-data-konsumen-lebih-ketat

[9] Hukumonline Pro, "Pemerintah Akhirnya Terbitkan Kerangka Implementasi Pelindungan Data Pribadi: Berlaku Mulai 16 Januari 2027," 2026. [Online]. Available: https://pro.hukumonline.com/a/lt6a97c3fe4527c/pemerintah-akhirnya-terbitkan-kerangka-implementasi-pelindungan-data-pribadi–berlaku-mulai-16-januari-2027

[10] S. Portigal, *Interviewing Users: How to Uncover Compelling Insights*, 2nd ed. Brooklyn, NY, USA: Rosenfeld Media, 2023.

[11] K. Holtzblatt and H. Beyer, *Contextual Design: Design for Life*, 2nd ed. Cambridge, MA, USA: Morgan Kaufmann, 2017.

[12] J. Sauro and J. R. Lewis, *Quantifying the User Experience: Practical Statistics for User Research*, 2nd ed. Cambridge, MA, USA: Morgan Kaufmann, 2016.

[13] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative Research in Psychology*, vol. 3, no. 2, pp. 77–101, 2006.

[14] T. T. Ngo, D. Nguyen Van, A.-M. Nguyen, P.-A. Do, and A. Nguyen-Quoc, "Qualitative Coding Analysis through Open-Source Large Language Models: A User Study and Design Recommendations," in *Extended Abstracts of the 2026 CHI Conference on Human Factors in Computing Systems (CHI EA '26)*, Barcelona, Spain, 2026, doi: 10.1145/3772363.3798320.

[15] M. Rosala and K. Moran, "Synthetic Users: If, When, and How to Use AI-Generated 'Research'," Nielsen Norman Group, Jun. 21, 2024. [Online]. Available: https://www.nngroup.com/articles/synthetic-users/

Chapter 6

Prototyping: From Paper to Interactive (and AI-Assisted) Prototypes

A prototype is a question you can hold in your hand. Before a health center spends its budget on a touch-screen kiosk, before a laundry changes the messages it sends a thousand times a month, someone should put a rough version in front of real people and watch what happens. This chapter teaches how to do that at every level of polish. By the end of it you will be able to say what question a prototype must answer and choose its fidelity accordingly; run a paper-prototype session with a human "computer"; describe a screen flow in a small text file and check it with a program in Java and Python that finds unreachable screens, dead ends and the fewest taps to a goal; build a clickable, coded prototype of a queue kiosk served by the JDK's built-in HTTP server and by Flask, with large accessible buttons and a log of every screen view; capture screenshots of it automatically with Playwright; and use the AI-assisted prototyping tools of 2026, such as Figma Make and Google Stitch, without letting their speed fool you into skipping the questions that matter.

6.1 The Queue at Dawn

Every weekday at a quarter past six, before the gate of a community health center (*puskesmas*) on the edge of a town in East Java opens, a line forms outside. Older patients come first, because they have learned that the numbers for the general clinic run out by mid-morning. Mothers with babies for the mother-and-child clinic (KIA) come next, then workers who hope to be seen before their shift. At seven thirty the gate opens, a staff member at the front desk writes numbers on slips of paper, and the waiting starts in earnest.

The head of the puskesmas, dr. Wulan, has a plan and a budget line: a touch-screen kiosk at the entrance that prints queue numbers, like those in banks and large hospitals. A vendor has already sent a brochure with a sample screen: a login with the patient's national identity number (NIK), a question about BPJS health insurance, a list of eleven services in small print, and a confirmation page. It looks professional. It would take the vendor two months to install and the puskesmas a large part of its equipment budget to buy.

A student team doing its community-service program at the puskesmas asks for one week before the order is signed. They do not have the vendor's screens; they have paper, scissors, a marker and a cardboard box cut to the size of a kiosk. In that week they learn three things that the brochure could never have told them. Older patients read "BPJS atau Umum?" as a question about which clinic to go to, not about how to pay. The help screen in the vendor's flow has no way back. And the clinic hours, which half the early arrivals want to know, are on a screen nobody can reach. None of these is a hard problem once seen. All of them would have cost weeks of complaints and a change request once the kiosk was installed.

That is the promise of prototyping: find the problems while they are still cheap to fix. This chapter follows the student team from the paper version to a clickable, coded prototype, and shows how the same thinking applies to the laundry messages of Chapter 5 and to the AI tools that, in 2026, can produce a polished screen from a sentence.

6.2 The Idea in Plain Words

6.2.1 A prototype is a question

> **DEFINITION.** A *prototype* is any representation of a design, made before the finished product, that lets people experience or evaluate some aspect of it. Its *fidelity* is how closely it resembles the finished product in appearance, interaction and content. A prototype is *low-fidelity* when it is rough and quick (a sketch, paper screens) and *high-fidelity* when it looks and behaves like the real thing (a clickable mockup, a coded prototype).

Houde and Hill proposed the most useful way to think about prototypes: ask *what the prototype prototypes* [1]. Every prototype explores one or more of three questions. The **role** question asks what the artifact does in a person's life: does a kiosk that prints numbers actually shorten the wait at dawn, or does the problem lie elsewhere? The **look and feel** question asks what it is like to see and touch: are the buttons big enough for a trembling finger, is the text readable at arm's length? The **implementation** question asks whether it can be built: can a small thermal printer be driven from a browser, will the kiosk work when the internet connection drops? A cardboard box with paper screens answers the role question and much of the look and feel; it says nothing about implementation. A working printer connected to a laptop answers implementation and nothing else. A good team knows which question each prototype is for.

6.2.2 Sketches and prototypes; the right thing and the thing right

Buxton drew a sharp line between *sketching* and *prototyping* [2]. Sketches are quick, cheap, plentiful and disposable; they suggest and explore, and their purpose is to ask *are we designing the right thing?* Prototypes are more refined, fewer and more expensive; they test and confirm, and their purpose is to ask *are we designing the thing right?* A team that jumps straight to a polished prototype has answered the second question before the first, and usually for the wrong design.

Figure 6.1 arranges the common forms of prototype along the fidelity axis, from a sketch made in minutes to a coded prototype that takes days. The cost of making a prototype, and of changing it, rises from left to right. So does the risk of a subtle trap: people who see a polished screen comment on colors and fonts, while people who see a pencil sketch comment on whether the idea makes sense.

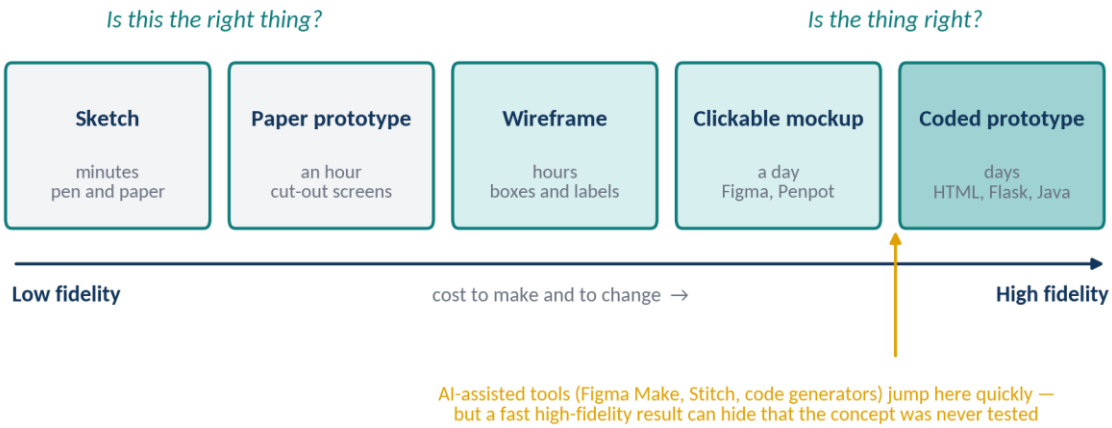


Figure 6.1 The fidelity spectrum: from sketches that ask whether this is the right thing to coded prototypes that ask whether the thing is right; AI-assisted tools reach high fidelity quickly, which makes it easy to skip the left-hand questions.

> **KEY INSIGHT.** Match the fidelity to the question, not to the tools you have. Early questions (is this the right service? in what order?) deserve rough prototypes, because rough prototypes invite honest criticism and cost nothing to throw away. Late questions (is this button big enough? is the wording clear?) deserve refined ones.

6.2.3 Horizontal and vertical prototypes

A second distinction concerns scope. A *horizontal* prototype shows many features, each only as a surface: every screen of the kiosk exists, but pressing "Print" shows a picture of a ticket rather than printing one. A *vertical* prototype implements one path in depth: the "take a number for the dental clinic" path really issues numbers, counts the people ahead and estimates the wait, while the other paths are missing. Most prototypes are a mixture, often called a *T-shape*: broad and shallow across the whole product, deep along the one path that matters most. The coded kiosk prototype of Section 6.6 is exactly that.

6.3 Paper Prototyping

6.3.1 Materials and roles

Paper prototyping is the fastest way to test the flow of an interface with real people. Snyder's practical guide describes it as a technique in which "the computer" is a person who moves paper screens in response to the participant's touches, while a facilitator gives tasks and observers take notes [3]. The materials are cheap: index cards or A5 sheets for screens, sticky notes for pop-ups and changing values, a marker, scissors, removable tape, and something to act as the device frame. The student team cut the front of a cardboard box to the size of a 21-inch kiosk screen and slid paper screens into it from behind.

There are three roles in a session:

- **The facilitator** welcomes the participant, explains that the design is being tested and not the person, reads the tasks, and asks the participant to think aloud (Chapter 7 treats think-aloud in depth).
- **The computer** sits beside or behind the prototype, never speaks, and swaps screens exactly as the design says, including for "wrong" touches. If the participant touches something that has no screen yet, the computer shows a blank card with a question mark; that moment is itself a finding.

- **Observers** write down what the participant does, says and hesitates over, one observation per line, as in Section 5.4.4.

6.3.2 A session at the puskesmas

The team prepared the vendor's flow as eight paper screens and three tasks: "You want to see the general doctor today; get a number", "You want to know until what time the dental clinic is open today", and "You pressed Help by mistake; get back to where you were". With permission from dr. Wulan and a consent script of the kind Section 5.3.3 describes, they tested with eight people over two mornings: five older patients, two mothers and a worker.

The paper version revealed the three problems of Section 6.1 within the first four sessions. It also produced smaller findings: three of the five older participants touched the text of a heading, expecting it to be a button; two participants tried to scroll a list that did not scroll; and several asked, before anything else, how long the wait would be, which the vendor's ticket did not say. None of these required a single line of code to discover.

6.3.3 Wizard of Oz

Sometimes the thing to be prototyped is not a set of screens but a behavior that is hard to build, such as understanding speech. The *Wizard of Oz* technique fakes it: a hidden person, the wizard, produces the system's responses by hand, while the participant believes, or agrees to pretend, that the system produces them. Kelley used it in the early 1980s to design a natural-language office application before the language understanding existed, iterating the design with a human in the loop until the vocabulary and dialogue were right [4].

The student team used it for one idea: a voice option for patients who cannot read the screen. A participant said "saya mau ke poli gigi" ("I want to go to the dental clinic") to the kiosk; a team member hidden behind the box listened and slid in the confirmation screen. The session showed that the voice option was used and welcomed by two of the older participants, and that the confirmation must repeat the service name aloud, because a participant who cannot read the screen cannot read the confirmation either. Wizard of Oz is equally the natural way to prototype an AI assistant before building one, a use Chapter 13 returns to.

6.4 Checking a Flow Before Building It

6.4.1 A flow as a text file

A paper prototype is a graph: screens are nodes, buttons are arrows from one screen to another. Some mistakes in a flow can be found without any participant at all, simply by looking at the graph: a screen with no way in is *unreachable*, a screen with no way out is a *dead end* (unless it is a goal where the task ends), and the number of taps to a goal is the length of the shortest path to it. Nobody draws graphs perfectly on paper, so it is worth writing the flow down in a form that a program can check.

The format is one line per button, `screen -> next : label`, plus a `start` line and one or more `goal` lines; `#` starts a comment. **Listing 6.1**, the file `data/flow_v1.txt`, is the vendor's flow as the team tested it on paper:

Listing 6.1 `flow_v1.txt`: the kiosk flow of the paper test, one line per button.

```
# Paper prototype of the puskesmas queue kiosk, version 1
# screen -> next screen : label on the button
start welcome
goal ticket
welcome -> service : Ambil antrean / Take a number
welcome -> help : Bantuan / Help
service -> bpjs : Umum / General
service -> bpjs : Gigi / Dental
bpjs -> nik : Pakai BPJS / Use BPJS
bpjs -> confirm : Bayar sendiri / Pay myself
nik -> confirm : Lanjut / Next
confirm -> ticket : Cetak / Print
schedule -> service : Kembali / Back
```

6.4.2 The checker

The checker reads the file into a map from each screen to its outgoing buttons, then runs a *breadth-first search* from the start screen. Breadth-first search visits screens in order of distance: first the start, then every screen one tap away, then every screen two taps away, and so on, using a queue. The first time it reaches a screen, it has found a shortest path to it, because every shorter path would have been explored first. After the search, a screen that was never reached is unreachable; a screen with no outgoing button that is not a goal is a dead end. **Listing 6.2** is the Python version.

Listing 6.2 `flow_check.py`: finds unreachable screens, dead ends and the fewest taps to every goal.

```
"""Check a screen-flow file of a prototype: unreachable screens, dead ends,
and the fewest taps from the start to each goal.
Usage: python flow_check.py [flow file]"""
import sys
from collections import deque


def load(path):
    """Return (start, goals, links) where links maps screen -> [(next, label)]."""
    start, goals, links = None, set(), {}
    with open(path, encoding="utf-8") as f:
        for raw in f:
            line = raw.split("#", 1)[0].strip()
            if not line:
                continue
            if line.startswith("start "):
                start = line.split()[1]
            elif line.startswith("goal "):
                goals.add(line.split()[1])
            else:
                left, label = line.split(":", 1)
                src, dst = [s.strip() for s in left.split("->")]
                links.setdefault(src, []).append((dst, label.strip()))
                links.setdefault(dst, [])
    return start, goals, links


def taps_from(start, links):
    """Breadth-first search: fewest taps and the path to every screen."""
    best = {start: [start]}
    queue = deque([start])
    while queue:
        screen = queue.popleft()
        for nxt, _ in links.get(screen, []):
            if nxt not in best:
                best[nxt] = best[screen] + [nxt]
                queue.append(nxt)
    return best
```

Listing 6.2 (Continued) *flow_check.py: finds unreachable screens, dead ends and the fewest taps to every goal.*

```
def check(start, goals, links):
    paths = taps_from(start, links)
    screens = sorted(links)
    unreachable = [s for s in screens if s not in paths]
    dead_ends = [s for s in screens if not links[s] and s not in goals]
    return paths, unreachable, dead_ends


def main(args):
    path = args[0] if args else "../data/flow_v1.txt"
    start, goals, links = load(path)
    paths, unreachable, dead_ends = check(start, goals, links)
    print(f"{path}: {len(links)} screens, "
          f"{sum(len(v) for v in links.values())} buttons")
    for goal in sorted(goals):
        if goal in paths:
            steps = paths[goal]
            print(f"  goal '{goal}': {len(steps) - 1} taps  "
                  + " > ".join(steps))
        else:
            print(f"  goal '{goal}': NOT REACHABLE from '{start}'")
    print("  unreachable screens: " + (", ".join(unreachable) or "none"))
    print("  dead ends (no way on or back): " + (", ".join(dead_ends) or "none"))


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

Listing 6.3 is the Java version. It stores the screens in a `TreeMap` so that the reports come out in alphabetical order, as the `sorted` calls do in Python, and uses an `ArrayDeque` as the queue of the search.

Listing 6.3 `FlowCheck.java`: the same checker in Java.

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.TreeMap;
import java.util.TreeSet;

/** Check a screen-flow file of a prototype: unreachable screens, dead ends,
 *  and the fewest taps from the start to each goal.
 *  Usage: java FlowCheck [flow file] */
public class FlowCheck {
    String start;
    TreeSet<String> goals = new TreeSet<>();
    TreeMap<String, List<String>> links = new TreeMap<>();
    int buttons = 0;

    static FlowCheck load(String path) throws IOException {
        FlowCheck f = new FlowCheck();
        for (String raw : Files.readAllLines(Path.of(path))) {
            String line = raw.split("#", 2)[0].strip();
            if (line.isEmpty()) continue;
            if (line.startsWith("start ")) f.start = line.split("\\s+")[1];
            else if (line.startsWith("goal ")) f.goals.add(line.split("\\s+")[1]);
            else {
                String left = line.split(":", 2)[0];
                String[] ends = left.split("->");
                String src = ends[0].strip(), dst = ends[1].strip();
                f.links.computeIfAbsent(src, k -> new ArrayList<>()).add(dst);
                f.links.computeIfAbsent(dst, k -> new ArrayList<>());
                f.buttons++;
            }
        }
        return f;
    }
```

Listing 6.3 (Continued) *FlowCheck.java: the same checker in Java.*

```
    /** Breadth-first search: fewest taps and the path to every screen. */
    Map<String, List<String>> tapsFrom() {
        Map<String, List<String>> best = new LinkedHashMap<>();
        best.put(start, List.of(start));
        ArrayDeque<String> queue = new ArrayDeque<>(List.of(start));
        while (!queue.isEmpty()) {
            String screen = queue.poll();
            for (String next : links.getOrDefault(screen, List.of()))
                if (!best.containsKey(next)) {
                    List<String> p = new ArrayList<>(best.get(screen));
                    p.add(next);
                    best.put(next, p);
                    queue.add(next);
                }
        }
        return best;
    }

    List<String> unreachable(Map<String, List<String>> paths) {
        List<String> out = new ArrayList<>();
        for (String s : links.keySet()) if (!paths.containsKey(s)) out.add(s);
        return out;
    }

    List<String> deadEnds() {
        List<String> out = new ArrayList<>();
        for (Map.Entry<String, List<String>> e : links.entrySet())
            if (e.getValue().isEmpty() && !goals.contains(e.getKey()))
                out.add(e.getKey());
        return out;
    }

    public static void main(String[] args) throws IOException {
        String path = args.length > 0 ? args[0] : "../data/flow_v1.txt";
        FlowCheck f = load(path);
        Map<String, List<String>> paths = f.tapsFrom();
        System.out.println(path + ": " + f.links.size() + " screens, "
            + f.buttons + " buttons");
        for (String goal : f.goals) {
            if (paths.containsKey(goal)) {
                List<String> steps = paths.get(goal);
                System.out.println("  goal '" + goal + "': " + (steps.size() - 1)
                    + " taps  " + String.join(" > ", steps));
            } else {
                System.out.println("  goal '" + goal + "': NOT REACHABLE from '"
                    + f.start + "'");
```

Listing 6.3 (Continued) *FlowCheck.java: the same checker in Java.*

```
            }
        }
        List<String> un = f.unreachable(paths), dead = f.deadEnds();
        System.out.println("  unreachable screens: "
            + (un.isEmpty() ? "none" : String.join(", ", un)));
        System.out.println("  dead ends (no way on or back): "
            + (dead.isEmpty() ? "none" : String.join(", ", dead)));
    }
}
```

6.4.3 Running it on both versions

Run on the version of the paper test, `python3 flow_check.py ../data/flow_v1.txt` (or `java -cp out FlowCheck ../data/flow_v1.txt`) prints:

```
../data/flow_v1.txt: 8 screens, 9 buttons
  goal 'ticket': 4 taps  welcome > service > bpjs > confirm > ticket
  unreachable screens: schedule
  dead ends (no way on or back): help
```

The program has found two of the three problems the participants found: the clinic hours are unreachable and the help screen is a dead end. It could not find the third, that "BPJS atau Umum?" is misread, because that is a problem of meaning, not of structure. Programs check structure; people check meaning. Both are needed.

After the paper sessions the team made three changes. The BPJS question moved from the kiosk to the front desk, where a person can explain it, which also removed the NIK screen. Help and clinic hours both got a "Back" button and a place on the welcome screen. And the two services missing from the vendor's list that the early arrivals actually needed, KIA and the elderly clinic (*Lansia*), were added. The new flow, `data/flow_v2.txt`, is **Listing 6.4**:

Listing 6.4 `flow_v2.txt`: the flow after the paper test.

```
# Version 2 after the paper test: help returns, schedule reachable,
# the BPJS question moved to the counter
start welcome
goal ticket
welcome -> service : Ambil antrean / Take a number
welcome -> help : Bantuan / Help
welcome -> schedule : Jadwal poli / Clinic hours
help -> welcome : Kembali / Back
schedule -> welcome : Kembali / Back
service -> confirm : Umum / General
service -> confirm : Gigi / Dental
service -> confirm : KIA / Mother and child
service -> confirm : Lansia / Elderly
confirm -> service : Ganti / Change
confirm -> ticket : Cetak / Print
```

```
../data/flow_v2.txt: 6 screens, 11 buttons
  goal 'ticket': 3 taps  welcome > service > confirm > ticket
  unreachable screens: none
  dead ends (no way on or back): none
```

Figure 6.2 draws both graphs. The ticket is now three taps from the welcome screen instead of four (or five for a BPJS patient), every screen can be reached, and every screen has a way on or back.

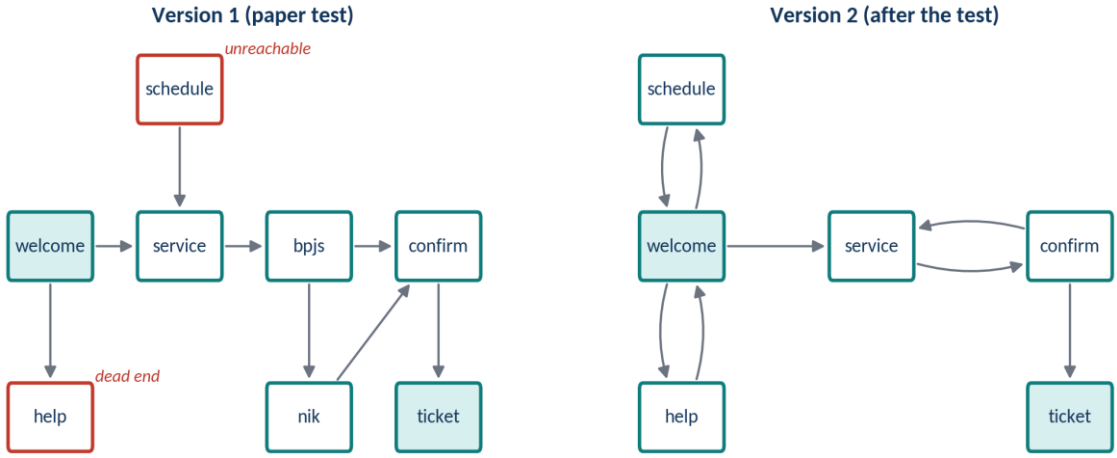


Figure 6.2 The kiosk flow before and after the paper test: version 1 has an unreachable screen and a dead end and needs four or five taps to a ticket; version 2 needs three, and every screen has a way back.

6.5 Wireframes and Clickable Mockups

6.5.1 Wireframes

A *wireframe* is a drawing of a screen that shows structure without style: boxes for content, labels for buttons, lines for text, all in grey. It fixes what is on each screen and where, and deliberately leaves out color, imagery and final wording, so that the conversation stays on layout and hierarchy. Wireframes are where the paper screens become precise enough for a developer to build, and where the visual rules of Chapter 8 (grids, hierarchy, contrast) are first applied.

For the kiosk, the wireframes fixed four decisions that the paper test had motivated: one question per screen; the primary action as a full-width button at least 72 pixels tall, with secondary actions below it in a quieter style; the service letter and name on each service button, so that patients who hear “B, gigi” at the desk find the same words on screen; and a yellow bar across the top that says “PROTOTYPE”, so that nobody mistakes a test session for the real queue.

6.5.2 Clickable mockups and the tools of 2026

A *clickable mockup* links designed screens together so that a participant can tap through them on a real device. In 2026 most teams make these in a collaborative design tool. Figma is the most widely used; at its Config 2025 conference Figma introduced Figma Make, which turns a written description or an existing design into a working prototype or app, alongside Figma Sites for publishing websites, Figma Draw for illustration, Figma Buzz for marketing assets and a CSS-like Grid layout [5]. At Config 2026 it added, among other things, Figma Motion for production-ready animation, *code layers* that bring working code onto the design canvas as an editable layer, and updates to the Figma agent with custom skills, web search and file attachments [6]. Penpot is the main open-source alternative: it runs in the browser, can be self-hosted, and stores designs in open formats, which matters for a public institution such as a puskesmas that may not want patient-facing designs on a foreign vendor’s servers [7].

Whatever the tool, a clickable mockup is still a prototype of look and feel and of flow, not of implementation. It shows that “Print my number” leads to a ticket screen; it does not show that a number is actually issued, that two patients pressing at once get different numbers, or that the wait estimate is sensible. For those questions the team built a coded prototype.

6.6 Building a Coded Prototype

6.6.1 What it must do

A coded prototype is a small program that behaves like the product along its most important path. The kiosk prototype must:

- show the six screens of `flow_v2.txt` in Indonesian or English (`?lang=id` or `?lang=en`);
- issue real, increasing queue numbers per service, starting after the people already waiting, and estimate the wait from the number of people ahead and an average time per patient;
- use large buttons (at least 72 pixels tall, far above the 24 by 24 CSS pixel minimum of WCAG 2.2’s success criterion 2.5.8 [8]), dark text on white, and a visible focus outline;
- carry the “PROTOTYPE” bar on every screen;
- log every screen view with a timestamp, so that a test session can be replayed and timed later (Chapter 7 analyzes such logs).

It does *not* print, does not talk to the puskesmas's patient records, and does not store anything about the person pressing the buttons. The waiting counts are made-up starting values: this is a prototype, and pretending otherwise would be a form of deception towards the participants.

6.6.2 The screens

The screens are produced by one function, `render(screen, lang, code, number)`, that returns a complete HTML page. Keeping all screens in one module, separate from the web server, has two benefits: the tests can check the pages without starting a server, and the Java and Python versions can be compared byte for byte. **Listing 6.5** is the Python module.

Listing 6.5 `kiosk_pages.py`: the texts, services, style and the six screens of the kiosk prototype.

```
"""Screens of the puskesmas queue-kiosk prototype (shared by the server
and the tests). Every screen is plain HTML with large buttons."""
from html import escape

TEXT = {
    "id": {"title": "Antrean Puskesmas", "proto": "PROTOTIPE - bukan antrean asli",
           "welcome": "Selamat datang. Mau berobat ke mana hari ini?",
           "take": "Ambil nomor antrean", "help": "Bantuan",
           "hours": "Jadwal poli", "back": "Kembali", "choose": "Pilih layanan",
           "sure": "Benar layanan ini?", "print": "Cetak nomor",
           "change": "Ganti layanan", "your": "Nomor Anda",
           "ahead": "orang di depan Anda", "wait": "Perkiraan tunggu",
           "min": "menit", "done": "Selesai",
           "helptext": "Tekan tombol hijau besar. "
                       "Petugas di meja depan siap membantu.",
           "hourstext": "Senin-Kamis 07.30-14.00, Jumat 07.30-11.00, "
                        "Sabtu 07.30-12.00"},
    "en": {"title": "Health Center Queue", "proto": "PROTOTYPE - not a real queue",
           "welcome": "Welcome. Which service do you need today?",
           "take": "Take a queue number", "help": "Help",
           "hours": "Clinic hours", "back": "Back", "choose": "Choose a service",
           "sure": "Is this the right service?", "print": "Print my number",
           "change": "Change service", "your": "Your number",
           "ahead": "people ahead of you", "wait": "Estimated wait",
           "min": "min", "done": "Done",
           "helptext": "Press the big green button. "
                       "Staff at the front desk can help.",
           "hourstext": "Mon-Thu 07:30-14:00, Fri 07:30-11:00, Sat 07:30-12:00"},
}
# code: (Indonesian name, English name, people ahead, minutes per patient)
SERVICES = {"A": ("Umum", "General", 14, 6), "B": ("Gigi", "Dental", 3, 15),
            "C": ("KIA", "Mother and child", 6, 8),
            "D": ("Lansia", "Elderly", 4, 7)}
STYLE = """body{font-family:Arial,sans-serif;margin:0;background:#fff;color:#12355B}
main{max-width:720px;margin:auto;padding:16px}
.proto{background:#E0A100;color:#000;text-align:center;padding:6px;font-size:18px}
h1{font-size:34px}p{font-size:24px}
.btn{display:block;min-height:72px;margin:14px 0;padding:18px;font-size:28px;
border-radius:14px;text-decoration:none;text-align:center;box-sizing:border-box}
.go{background:#0F7C7C;color:#fff}.alt{background:#F3F4F6;color:#12355B;
border:2px solid #12355B}.big{font-size:96px;font-weight:bold;margin:8px 0}
a:focus{outline:4px solid #E0A100;outline-offset:3px}"""
```

Listing 6.5 (Continued) *kiosk_pages.py: the texts, services, style and the six screens of the kiosk prototype.*

```
def name(code, lang):
    s = SERVICES[code]
    return s[0] if lang == "id" else s[1]


def button(href, label, kind="go"):
    return f'<a class="btn {kind}" href="{href}">{escape(label)}</a>'


def render(screen, lang="id", code="A", number=0):
    t = TEXT[lang if lang in TEXT else "id"]
    q = f"?lang={lang}"
    if screen == "welcome":
        body = (f"<h1>{t['welcome']}</h1>" + button("/service" + q, t["take"])
                + button("/help" + q, t["help"], "alt")
                + button("/schedule" + q, t["hours"], "alt"))
    elif screen == "service":
        body = f"<h1>{t['choose']}</h1>" + "".join(
            button(f"/confirm{q}&s={c}", f"{c} - {name(c, lang)}")
            for c in SERVICES) + button("/" + q, t["back"], "alt")
    elif screen == "confirm":
        body = (f"<h1>{t['sure']}</h1><p class='big'>{name(code, lang)}</p>"
                + button(f"/ticket{q}&s={code}", t["print"])
                + button("/service" + q, t["change"], "alt"))
    elif screen == "ticket":
        ahead, per = SERVICES[code][2], SERVICES[code][3]
        body = (f"<h1>{t['your']}</h1><p class='big'>{code}-{number:03d}</p>"
                f"<p>{name(code, lang)}: {ahead} {t['ahead']}</p>"
                f"<p>{t['wait']}: {ahead * per} {t['min']}</p>"
                + button("/" + q, t["done"]))
    elif screen == "help":
        body = f"<h1>{t['help']}</h1><p>{t['helptext']}</p>" + button(
            "/" + q, t["back"], "alt")
    else:
        body = f"<h1>{t['hours']}</h1><p>{t['hourstext']}</p>" + button(
            "/" + q, t["back"], "alt")
    return (f'<!doctype html><html lang="{lang}"><head><meta charset="utf-8">'
            f'<meta name="viewport" content="width=device-width,initial-scale=1">'
            f"<title>{t['title']}</title><style>{STYLE}</style></head><body>"
            f'<div class="proto">{t["proto"]}</div><main>{body}</main>'
            "</body></html>")
```

Three details are worth pointing out. The texts of both languages live in one dictionary, so a translation is a change of data, not of code. Button labels pass through `escape`, so that a label containing `<` or `&` cannot break the page. And the page declares its language in `<html lang="...">`, which lets a screen reader pronounce Indonesian as Indonesian, a point Chapter 11 develops.

6.6.3 The server in Python

Listing 6.6 serves the screens with Flask [9]. Each route calls `show`, which reads the language and service code from the query string, issues a number for the ticket screen, logs the view and returns the page.

Listing 6.6 `kiosk_proto.py`: the kiosk prototype served by Flask on port 4060.

```
"""Clickable prototype of the puskesmas queue kiosk, served with Flask.
Every screen view is logged to ../data/events.csv for the test session.
Usage: python kiosk_proto.py   then open http://localhost:4060/?lang=en"""
import time
from flask import Flask, request
from kiosk_pages import SERVICES, render

app = Flask(__name__)
next_number = {code: s[2] + 1 for code, s in SERVICES.items()}
LOG = "../data/events.csv"


def log(screen, lang):
    with open(LOG, "a", encoding="utf-8") as f:
        f.write(f"{time.time():.3f},{lang},{screen}\n")


def show(screen):
    lang = request.args.get("lang", "id")
    code = request.args.get("s", "A")
    if code not in SERVICES:
        code = "A"
    number = 0
    if screen == "ticket":
        number = next_number[code]
        next_number[code] += 1
    log(screen, lang)
    return render(screen, lang, code, number)


app.add_url_rule("/", "welcome", lambda: show("welcome"))
for s in ("service", "confirm", "ticket", "help", "schedule"):
    app.add_url_rule("/" + s, s, lambda s=s: show(s))

if __name__ == "__main__":
    app.run(port=4060)
```

The loop at the bottom registers five routes with one line each; the default argument `s=s` in the `lambda` captures the current screen name, a standard Python idiom without which every route would show the last screen of the loop.

6.6.4 The same prototype in Java

Listing 6.7 is the Java version of the screens, producing exactly the same HTML as the Python module.

Listing 6.7 `KioskPages.java`: the same screens in Java.

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

/** Screens of the puskesmas queue-kiosk prototype (shared by the server
 *  and the tests). Every screen is plain HTML with large buttons. */
public class KioskPages {
    static final Map<String, Map<String, String>> TEXT = Map.of(
        "id", Map.ofEntries(Map.entry("title", "Antrean Puskesmas"),
            Map.entry("proto", "PROTOTIPE - bukan antrean asli"),
            Map.entry("welcome",
                "Selamat datang. Mau berobat ke mana hari ini?"),
            Map.entry("take", "Ambil nomor antrean"), Map.entry("help", "Bantuan"),
            Map.entry("hours", "Jadwal poli"), Map.entry("back", "Kembali"),
            Map.entry("choose", "Pilih layanan"),
            Map.entry("sure", "Benar layanan ini?"),
            Map.entry("print", "Cetak nomor"), Map.entry("change", "Ganti layanan"),
            Map.entry("your", "Nomor Anda"),
            Map.entry("ahead", "orang di depan Anda"),
            Map.entry("wait", "Perkiraan tunggu"), Map.entry("min", "menit"),
            Map.entry("done", "Selesai"),
            Map.entry("helptext", "Tekan tombol hijau besar. "
                + "Petugas di meja depan siap membantu."),
            Map.entry("hourstext", "Senin-Kamis 07.30-14.00, Jumat 07.30-11.00, "
                + "Sabtu 07.30-12.00")),
        "en", Map.ofEntries(Map.entry("title", "Health Center Queue"),
            Map.entry("proto", "PROTOTYPE - not a real queue"),
            Map.entry("welcome", "Welcome. Which service do you need today?"),
            Map.entry("take", "Take a queue number"), Map.entry("help", "Help"),
            Map.entry("hours", "Clinic hours"), Map.entry("back", "Back"),
            Map.entry("choose", "Choose a service"),
            Map.entry("sure", "Is this the right service?"),
            Map.entry("print", "Print my number"),
            Map.entry("change", "Change service"), Map.entry("your", "Your number"),
            Map.entry("ahead", "people ahead of you"),
            Map.entry("wait", "Estimated wait"), Map.entry("min", "min"),
            Map.entry("done", "Done"),
            Map.entry("helptext", "Press the big green button. "
                + "Staff at the front desk can help."),
            Map.entry("hourstext", "Mon-Thu 07:30-14:00, Fri 07:30-11:00, "
                + "Sat 07:30-12:00")));
```

Listing 6.7 (Continued) *KioskPages.java: the same screens in Java.*

```
    /** code -> {Indonesian name, English name, people ahead, minutes each}. */
    static final Map<String, Object[]> SERVICES = new LinkedHashMap<>();
    static {
        SERVICES.put("A", new Object[] {"Umum", "General", 14, 6});
        SERVICES.put("B", new Object[] {"Gigi", "Dental", 3, 15});
        SERVICES.put("C", new Object[] {"KIA", "Mother and child", 6, 8});
        SERVICES.put("D", new Object[] {"Lansia", "Elderly", 4, 7});
    }
    static final String STYLE = String.join("\n",
        "body{font-family:Arial,sans-serif;margin:0;background:#fff;color:#12355B}",
        "main{max-width:720px;margin:auto;padding:16px}",
        ".proto{background:#E0A100;color:#000;text-align:center;padding:6px;"
            + "font-size:18px}",
        "h1{font-size:34px}p{font-size:24px}",
        ".btn{display:block;min-height:72px;margin:14px 0;padding:18px;"
            + "font-size:28px;",
        "border-radius:14px;text-decoration:none;text-align:center;"
            + "box-sizing:border-box}",
        ".go{background:#0F7C7C;color:#fff}.alt{background:#F3F4F6;color:#12355B;",
        "border:2px solid #12355B}.big{font-size:96px;font-weight:bold;"
            + "margin:8px 0}",
        "a:focus{outline:4px solid #E0A100;outline-offset:3px}");

    static String name(String code, String lang) {
        return (String) SERVICES.get(code)[lang.equals("id") ? 0 : 1];
    }

    static String escape(String s) {
        return s.replace("&", "&amp;").replace("<", "&lt;").replace(">", "&gt;")
            .replace("\"", "&quot;").replace("'", "&#x27;");
    }

    static String button(String href, String label, String kind) {
        return "<a class=\"btn " + kind + "\" href=\"" + href + "\">"
            + escape(label) + "</a>";
    }
```

Listing 6.7 (Continued) *KioskPages.java: the same screens in Java.*

```
    static String render(String screen, String lang, String code, int number) {
        if (!TEXT.containsKey(lang)) lang = "id";
        Map<String, String> t = TEXT.get(lang);
        String q = "?lang=" + lang, body;
        switch (screen) {
            case "welcome" -> body = "<h1>" + t.get("welcome") + "</h1>"
                + button("/service" + q, t.get("take"), "go")
                + button("/help" + q, t.get("help"), "alt")
                + button("/schedule" + q, t.get("hours"), "alt");
            case "service" -> {
                StringBuilder b =
                    new StringBuilder("<h1>" + t.get("choose") + "</h1>");
                for (String c : SERVICES.keySet())
                    b.append(button("/confirm" + q + "&s=" + c,
                        c + " - " + name(c, lang), "go"));
                body = b + button("/" + q, t.get("back"), "alt");
            }
            case "confirm" -> body = "<h1>" + t.get("sure")
                + "</h1><p class='big'>" + name(code, lang) + "</p>"
                + button("/ticket" + q + "&s=" + code, t.get("print"), "go")
                + button("/service" + q, t.get("change"), "alt");
            case "ticket" -> {
                int ahead = (int) SERVICES.get(code)[2];
                int per = (int) SERVICES.get(code)[3];
                body = "<h1>" + t.get("your") + "</h1><p class='big'>"
                    + String.format("%s-%03d", code, number) + "</p>"
                    + "<p>" + name(code, lang) + ": " + ahead + " "
                    + t.get("ahead") + "</p><p>" + t.get("wait") + ": "
                    + ahead * per + " " + t.get("min") + "</p>"
                    + button("/" + q, t.get("done"), "go");
            }
            case "help" -> body = "<h1>" + t.get("help") + "</h1><p>"
                + t.get("helptext") + "</p>"
                + button("/" + q, t.get("back"), "alt");
            default -> body = "<h1>" + t.get("hours") + "</h1><p>"
                + t.get("hourstext") + "</p>"
                + button("/" + q, t.get("back"), "alt");
        }
        return "<!doctype html><html lang=\"" + lang + "\"><head>"
            + "<meta charset=\"utf-8\"><meta name=\"viewport\" "
            + "content=\"width=device-width,initial-scale=1\">"
            + "<title>" + t.get("title") + "</title><style>" + STYLE
            + "</style></head><body><div class=\"proto\">" + t.get("proto")
            + "</div><main>" + body + "</main></body></html>";
    }
}
```

Listing 6.8 serves them with `com.sun.net.httpserver.HttpServer`, the small HTTP server that has been part of the JDK since Java 6 and needs no library [10]. One handler serves every path: it maps the path to a screen name, rejects unknown paths with status 404, parses the query string, and writes the page. The counter of issued numbers and the log file are guarded with `synchronized`, because the JDK server may handle two requests at the same time, and two patients must never receive the same number.

Listing 6.8 `KioskProto.java`: the kiosk prototype served by the JDK's built-in HTTP server on port 4061.

```
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;
import java.io.FileWriter;
import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.nio.charset.StandardCharsets;
import java.util.HashMap;
import java.util.Locale;
import java.util.Map;

/** Clickable prototype of the puskesmas queue kiosk, served with the JDK's
 *  built-in HTTP server. Every screen view is logged to ../data/events.csv.
 *  Usage: java -cp out KioskProto   then open http://localhost:4061/?lang=en */
public class KioskProto {
    static final Map<String, Integer> nextNumber = new HashMap<>();

    static Map<String, String> query(HttpExchange ex) {
        Map<String, String> m = new HashMap<>();
        String raw = ex.getRequestURI().getRawQuery();
        if (raw != null)
            for (String pair : raw.split("&")) {
                String[] kv = pair.split("=", 2);
                if (kv.length == 2) m.put(kv[0], kv[1]);
            }
        return m;
    }

    static synchronized void log(String screen, String lang) throws IOException {
        try (FileWriter w = new FileWriter("../data/events.csv", true)) {
            w.write(String.format(Locale.US, "%.3f,%s,%s%n",
                System.currentTimeMillis() / 1000.0, lang, screen));
        }
    }

    static void handle(HttpExchange ex) throws IOException {
        String path = ex.getRequestURI().getPath();
        String screen = path.equals("/") ? "welcome" : path.substring(1);
        byte[] out;
        int status = 200;
        if (!screen.matches("welcome|service|confirm|ticket|help|schedule")) {
            status = 404;
            out = "Not found".getBytes(StandardCharsets.UTF_8);
        } else {
            Map<String, String> q = query(ex);
            String lang = q.getOrDefault("lang", "id");
```

Listing 6.8 (Continued) *KioskProto.java: the kiosk prototype served by the JDK's built-in HTTP server on port 4061.*

```
            String code = q.getOrDefault("s", "A");
            if (!KioskPages.SERVICES.containsKey(code)) code = "A";
            int number = 0;
            if (screen.equals("ticket")) {
                synchronized (nextNumber) {
                    number = nextNumber.merge(code, 1, Integer::sum);
                }
            }
            log(screen, lang);
            out = KioskPages.render(screen, lang, code, number)
                .getBytes(StandardCharsets.UTF_8);
        }
        ex.getResponseHeaders().set("Content-Type", "text/html; charset=utf-8");
        ex.sendResponseHeaders(status, out.length);
        try (OutputStream os = ex.getResponseBody()) {
            os.write(out);
        }
    }

    public static void main(String[] args) throws IOException {
        for (Map.Entry<String, Object[]> e : KioskPages.SERVICES.entrySet())
            nextNumber.put(e.getKey(), (int) e.getValue()[2]);
        HttpServer server = HttpServer.create(new InetSocketAddress(4061), 0);
        server.createContext("/", KioskProto::handle);
        server.start();
        System.out.println("Kiosk prototype on http://localhost:4061/?lang=en");
    }
}
```

6.6.5 Running it and taking screenshots

Start the Python server from the `python` folder with `python3 kiosk_proto.py` (install Flask first with `pip install -r ../requirements.txt`), or the Java server from the `java` folder with `java -cp out KioskProto`. Flask reports:

```
 * Serving Flask app 'kiosk_proto'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a
production WSGI server instead.
 * Running on http://127.0.0.1:4060
Press CTRL+C to quit
```

and the Java server prints `Kiosk prototype on http://localhost:4061/?lang=en`. Open that address in a browser and click through.

For a report, and for comparing versions, it is useful to take screenshots automatically, walking through the prototype exactly as a patient would. **Listing 6.9** does that with Playwright, a browser-automation library that drives a real Chromium browser [11]. It opens the welcome screen at a kiosk-like size, takes a number for the dental clinic and saves a screenshot at each step.

Listing 6.9 `shoot.py`: walks through the running prototype with Playwright and saves screenshots.

```
"""Walk through the running prototype like a patient and save screenshots.
Start the server first (python kiosk_proto.py), then:
    python shoot.py [base URL] [lang]
e.g. python shoot.py http://localhost:4060 en
Needs: pip install playwright   and   python -m playwright install chromium"""
import sys
from playwright.sync_api import sync_playwright


def main(args):
    base = args[0] if args else "http://localhost:4060"
    lang = args[1] if len(args) > 1 else "en"
    with sync_playwright() as p:
        browser = p.chromium.launch()
        page = browser.new_page(viewport={"width": 540, "height": 860})
        page.goto(f"{base}/?lang={lang}")
        page.screenshot(path=f"../screens/welcome_{lang}.png")
        page.click("a.btn.go")                          # take a number
        page.screenshot(path=f"../screens/service_{lang}.png")
        page.click("text=B - ")                         # dental
        page.click("a.btn.go")                          # print my number
        page.screenshot(path=f"../screens/ticket_{lang}.png")
        print("title:", page.title())
        print("ticket:", page.inner_text("p.big"))
        print("saved 3 screenshots in ../screens/")
        browser.close()


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

With the Flask server running, `python3 shoot.py http://localhost:4060 en` prints:

```
title: Health Center Queue
ticket: B-004
saved 3 screenshots in ../screens/
```

Running it against the Java server on port 4061 prints the same three lines, because the two servers produce the same pages and start the same counters. **Figure 6.3** shows the three screenshots. The dental clinic had three people waiting, so the first new ticket is B-004 and the estimated wait is $3 \times 15 = 45$ minutes.

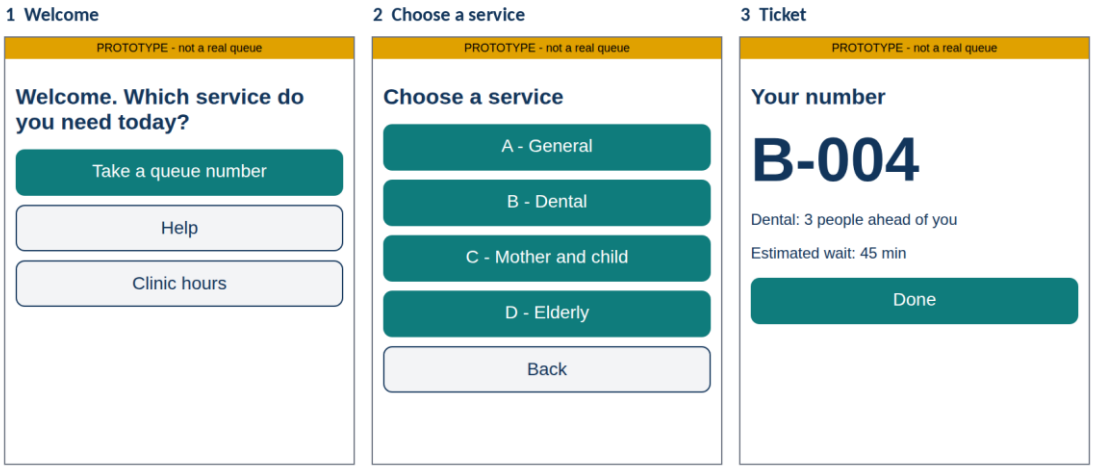


Figure 6.3 Three screens of the running kiosk prototype, captured by `shoot.py`: the welcome screen, the service choice and the ticket with the number of people ahead and the estimated wait.

Each run also appends to `data/events.csv`, one line per screen view:

```
1790349318.935,en,welcome
1790349319.117,en,service
1790349319.241,en,confirm
```

The first column is the time in seconds since 1970, so the difference between two lines is the time spent on a screen. A real test session produces this log with a participant's taps instead of Playwright's, and Chapter 7 turns such logs into task times.

6.7 Testing the Prototype Code

A prototype is disposable, but it still must not mislead: a prototype that gives two patients the same number will produce a test session about the wrong problem. **Listing 6.10** checks the two flows, the language attribute and button count of a page, and, through Flask's test client, that consecutive tickets increase and that each view is logged. The test redirects the log to a temporary file so that running the tests does not pollute the study data.

Listing 6.10 `test_proto.py`: tests of the flow checker and the prototype.

```
"""Tests for the kiosk prototype and the flow checker.
Run: python -m unittest test_proto.py"""
import os
import tempfile
import unittest
import flow_check
import kiosk_proto
from kiosk_pages import render


class ProtoTest(unittest.TestCase):
    def test_flow_v1_problems(self):
        paths, unreachable, dead = flow_check.check(
            *flow_check.load("../data/flow_v1.txt"))
        self.assertEqual(unreachable, ["schedule"])
        self.assertEqual(dead, ["help"])
        self.assertEqual(len(paths["ticket"]) - 1, 4)

    def test_flow_v2_clean_and_shorter(self):
        paths, unreachable, dead = flow_check.check(
            *flow_check.load("../data/flow_v2.txt"))
        self.assertEqual((unreachable, dead), ([], []))
        self.assertEqual(len(paths["ticket"]) - 1, 3)

    def test_pages_are_labelled_and_big(self):
        html = render("service", "en")
        self.assertIn('<html lang="en">', html)
        self.assertEqual(html.count('class="btn'), 5)
        self.assertIn("min-height:72px", html)

    def test_ticket_through_the_server(self):
        kiosk_proto.LOG = os.path.join(tempfile.mkdtemp(), "events.csv")
        client = kiosk_proto.app.test_client()
        first = client.get("/ticket?lang=en&s=B").get_data(as_text=True)
        second = client.get("/ticket?lang=en&s=B").get_data(as_text=True)
        self.assertIn("B-004", first)
        self.assertIn("B-005", second)
        self.assertIn("Estimated wait: 45 min", first)
        with open(kiosk_proto.LOG, encoding="utf-8") as f:
            self.assertEqual(len(f.readlines()), 2)


if __name__ == "__main__":
    unittest.main()
```

```
Ran 4 tests in 0.012s

OK
```

Listing 6.11 makes the same checks in Java, except the server test, since the Java server's counter lives inside `KioskProto`.

Listing 6.11 `ProtoTest.java`: the same checks in Java.

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

/** Tests for the kiosk pages and the flow checker, without a test library.
 *  Run: java -cp out ProtoTest */
public class ProtoTest {
    static int passed = 0;

    static void check(boolean ok, String name) {
        if (!ok) throw new AssertionError("FAILED: " + name);
        passed++;
    }

    static int count(String s, String part) {
        return s.split(java.util.regex.Pattern.quote(part), -1).length - 1;
    }

    public static void main(String[] args) throws IOException {
        FlowCheck v1 = FlowCheck.load("../data/flow_v1.txt");
        Map<String, List<String>> p1 = v1.tapsFrom();
        check(v1.unreachable(p1).equals(List.of("schedule")), "v1 unreachable");
        check(v1.deadEnds().equals(List.of("help")), "v1 dead end");
        check(p1.get("ticket").size() - 1 == 4, "v1 four taps");
        FlowCheck v2 = FlowCheck.load("../data/flow_v2.txt");
        Map<String, List<String>> p2 = v2.tapsFrom();
        check(v2.unreachable(p2).isEmpty() && v2.deadEnds().isEmpty(), "v2 clean");
        check(p2.get("ticket").size() - 1 == 3, "v2 three taps");
        String html = KioskPages.render("service", "en", "A", 0);
        check(html.contains("<html lang=\"en\">"), "lang attribute");
        check(count(html, "class=\"btn") == 5, "five buttons");
        String ticket = KioskPages.render("ticket", "en", "B", 4);
        check(ticket.contains("B-004") && ticket.contains("Estimated wait: 45 min"),
            "ticket text");
        System.out.println(passed + " checks passed");
    }
}
```

```
8 checks passed
```

6.8 AI-Assisted Prototyping in 2026

6.8.1 What the tools do

By 2026, a single sentence can produce a high-fidelity screen. Figma Make turns a prompt or an existing design into a working prototype [5]. Google's Stitch, introduced by Google Labs in May 2025, turns text prompts and images, including hand-drawn sketches and wireframes, into UI designs and front-end code that can be exported to Figma, using the multimodal capabilities of the Gemini models [12]. General-purpose coding assistants produce a Flask or React app from a description. The student team tried a prompt: "a touch-screen queue kiosk for an Indonesian community health center, large buttons, for elderly users". In under a minute a tool produced four attractive screens.

The screens were instructive, in both directions. They were visually clean, with sensible spacing and colors. They also contained a login with the NIK, a BPJS question, a carousel of health tips and a small "Help" link in a corner: the tool had reproduced the typical kiosk, including every problem that the paper test had just removed. It had no way to know that older patients misread the BPJS question; that knowledge existed only in the observers' notes.

6.8.2 Using AI without skipping the questions

The lesson is not that AI prototyping is bad; it is that it is fast at the right-hand end of **Figure 6.1**, and speed at that end makes it tempting to skip the left-hand end. A few rules keep it useful:

1. **Sketch and test first.** Use paper or a whiteboard for the questions of role and flow; bring AI in when the flow is settled and you need screens to test look and feel.
2. **Prompt with your research.** Put the findings into the prompt: "no login; one question per screen; primary button full width at least 72 px; four services A–D with these names; show people ahead and estimated wait on the ticket". An AI tool can only follow knowledge it is given.
3. **Check every generated screen against your flow file and your accessibility checklist.** Generated designs often contain small text, low contrast, icons without labels and components that look clickable but are not. Chapter 8's contrast checker and Chapter 11's checklist apply to them exactly as to hand-made screens.
4. **Treat generated code as a prototype.** Code produced from a prompt is not reviewed, tested or secure by default; in this chapter the coded prototype has tests, and a generated one needs them too.
5. **Mind what you upload.** A prompt with real patient data, internal documents or unreleased designs sends them to a third party. For a puskesmas, prefer tools that can run locally or a self-hosted design tool, and use invented example data.
6. **Test with people.** A generated prototype is still a prototype: put it in front of participants, with the same tasks and the same observation as the paper version.

> **WARNING.** A polished prototype produced in a minute looks finished, and people react to finished-looking things with polite approval instead of honest criticism. When you test an AI-generated prototype early, consider showing it in grayscale or printing it on paper, so that participants talk about the idea instead of the colors.

6.9 Prototyping the Laundry Messages

Not every interface is a screen. The four changes recommended to Bu Rina in Section 5.9.3 are *messages*: a pickup message with an estimated price range, a ready time and an item count; a "ready" message with two return slots; a photo with the count; and a large-type price list sent as text. A message is prototyped the same way as a screen, at the lowest fidelity that answers the question.

The team wrote each message on a card and, with six customers from the interviews, played the conversation: one person acted as the laundry, sending the cards as WhatsApp messages from a spare phone, while the customer answered as they would. This is a Wizard of Oz prototype of a messaging service: no automation, only the right words at the right moment. Two findings came out at once. A price range of "Rp28.000–35.000" was welcome, but customers asked what would make it higher, so the message gained a line ("extra for blankets and bed covers"). And the two return slots had to be numbered ("reply 1 for 16.00–18.00 or 2 for 19.00–21.00"), because customers answered "the evening one" and Dimas could not tell which evening.

The messages were then used for real for two weeks, with Bu Rina sending them by hand from templates in WhatsApp Business. Only after that, when the words had settled, was it worth automating anything. The order matters: prototype the words, then the behavior, then the automation.

6.10 Using It for Real

A prototype plan fits on a card:

1. **The question.** Write down which question the prototype answers: role, look and feel, or implementation, and what you will decide from the answer.
2. **The fidelity.** Choose the lowest fidelity that answers it. Paper for flow, wireframes for layout, a clickable mockup for look and feel on a real device, code for behavior and implementation.
3. **The flow file.** Write the flow as `screen -> next : label` lines and run `flow_check` before every test round. Fix unreachable screens and dead ends before participants find them.
4. **The tasks.** Three to five realistic tasks, written in the users' words, with a clear end ("you have a number in your hand").
5. **The participants.** Five to eight per round, including the people at the edges: older users, first-time users, people who read slowly.
6. **The honesty.** Label the prototype as a prototype, get consent, and never collect real personal data in a prototype.
7. **The iteration.** After each round, change the design and test again. Two or three short rounds beat one long one.

The same card works well beyond clinics. For an **ATM** of a regional bank, a cardboard front with paper screens and a real keypad tests the order of the withdrawal steps; the implementation question (does the cash dispenser respond in time?) belongs to the bank's engineers. For an **ojek app**, a clickable mockup on a phone, tested at a driver base, tests whether the driver sees the pickup point before accepting. For a **warung's ordering by WhatsApp**, message cards played with regular customers, as in Section 6.9, are enough.

6.11 HCI in Practice: Profitina

Profitina's product decisions show the three prototype questions in practice.

Prototyping a behavior: instant, optimistic posting. In Profitina's real-time community, the composer clears at once and the post appears immediately, while it is still being sent; if sending fails, the text is restored so nothing is lost. Whether that feels fast and trustworthy is a look-and-feel question that no static mockup can answer, because it is about *timing*. A coded prototype with an artificial delay and an artificial failure is the right fidelity: participants post, the prototype sometimes fails on purpose, and observers watch whether people notice the restored text and understand what happened.

A quality gate for generated content. Profitina's Content Engine drafts content with a local model (Qwen2.5) and serves an AI draft only when it scores genuinely better than a built-in template; otherwise the template is served. That is the same discipline this chapter recommends for AI-generated prototypes: a generated result must beat a known baseline before it is used, and "generated" is not a synonym for "better".

Implementation prototypes for the widget. The community widget is embedded across the marketing site and in the app and streams updates by Server-Sent Events. Whether a widget of that kind keeps up with a busy feed on a slow phone connection is an implementation question, answered by a small coded prototype long before visual polish. Profitina advises and the user decides; its prototypes, like any others, are there to find out before committing.

6.12 Common Mistakes, and How to Avoid Them

- **Starting at high fidelity.** A polished first prototype attracts comments on colors and hides problems of flow. Start with paper for the questions of role and order.
- **Prototyping without a question.** "Let's make a prototype" produces a demo. Write down the question and the decision it informs.
- **Defending the prototype.** A facilitator who explains the design during a session tests the explanation, not the design. Say nothing; let the participant struggle; take notes.
- **Unreachable screens and dead ends.** They are easy to create on paper and in design tools. Write the flow down and check it with `flow_check` before each round.
- **Forgetting to label a prototype.** A realistic prototype in a public place can be mistaken for the real service. Keep a visible "PROTOTYPE" bar.
- **Real data in prototypes.** A prototype that asks for a NIK or stores names collects personal data without need. Use invented examples.
- **Trusting AI output as research.** A generated screen reproduces the typical design, including its typical problems. Prompt with findings, then test with people.
- **Treating prototype code as product code.** Prototype code is written to answer questions quickly; before it becomes a product it needs review, security, tests and accessibility checks.
- **Automating before the words work.** For messaging services, prototype the messages by hand first; automate only when they have settled.

6.13 Chapter Summary

- A prototype is a representation of a design made to answer a question. Houde and Hill's three questions are role, look and feel, and implementation.
- Fidelity should match the question (**Figure 6.1**): sketches and paper prototypes ask whether this is the right thing; clickable mockups and coded prototypes ask whether the thing is right.
- Paper prototyping uses a facilitator, a silent human "computer" and observers; Wizard of Oz fakes a hard behavior, such as voice, with a hidden person.
- A flow is a graph. `flow_check` in Java and Python (**Listing 6.2**, **Listing 6.3**) uses breadth-first search to find unreachable screens, dead ends and the fewest taps to a goal; the paper test and the checker together reduced the kiosk's path to a ticket from four or five taps to three (**Figure 6.2**).
- A coded prototype implements the most important path deeply: the kiosk issues real numbers, estimates the wait, logs every view, uses large accessible buttons and is labeled as a prototype, in Flask and with the JDK's HTTP server (**Figure 6.3**).
- AI-assisted tools of 2026 (Figma Make, Stitch, coding assistants) reach high fidelity in minutes. Use them after the flow is settled, prompt them with research findings, check their output, and test it with people.

6.14 Exercises

1. For each prototype, say which of Houde and Hill's questions (role, look and feel, implementation) it mainly answers: (a) a laptop connected to a thermal printer that prints "A-001" when a key is pressed; (b) a cardboard kiosk with paper screens; (c) a Figma mockup shown on a tablet at the puskesmas entrance; (d) a morning in which a staff member stands at the entrance with a clipboard and hands out numbers that tell patients the estimated wait.
2. Add a screen `language` to `flow_v2.txt`, reachable from `welcome` with the button "English / Bahasa Indonesia", which returns to `welcome`. Run `flow_check` before and after adding the way back, and explain the difference in the output.
3. The vendor's full flow had a BPJS patient go `welcome > service > bpjs > nik > confirm > ticket`. How many taps is that, and why does `flow_check` report four taps to the ticket for `flow_v1.txt`? What does that tell you about reporting only the shortest path?
4. Extend `flow_check` (in Python or Java) so that it also reports, for each goal, the *longest* simple path from the start (the path without repeated screens with the most taps). Run it on `flow_v1.txt`.
5. Change the kiosk prototype so that the ticket screen also shows the time at which the patient should come back, assuming the current time is 07:30 and the wait estimate of the ticket. Which functions must change, and why is it better to pass the current time in as a parameter than to read the clock inside `render`?
6. Write a prompt for an AI prototyping tool that produces the kiosk of `flow_v2.txt`, using at least five findings from Sections 6.3 and 6.4. Then list four things you would check in the generated screens before showing them to a participant.
7. Plan a Wizard of Oz test of a voice option for the laundry of Chapter 5, in which a customer sends a voice note to order. Who is the wizard, what does the customer believe or agree to, what are the tasks, and what must the consent script say?

6.15 Solutions to the Exercises

1. (a) Implementation: it tells you whether a number can be printed, nothing about whether patients understand the kiosk. (b) Role and much of look and feel (sizes, order, wording), no implementation. (c) Look and feel on a realistic screen at the real place, and some role; no implementation. (d) Role: a human "kiosk" tests whether telling patients the estimated wait changes when they arrive and whether they leave and come back, before anything is built. This last kind is sometimes called a *service prototype*.

2. Adding only `welcome -> language : English / Bahasa Indonesia` makes `language` a dead end: the checker reports `dead ends (no way on or back): language`. Adding `language -> welcome : Kembali / Back` removes it, and the report returns to `none`. The number of screens rises to 7 and the number of buttons to 12 and 13, and the path to the ticket stays at three taps.

3. Five taps. `flow_check` reports four because it reports the *shortest* path, which goes through `bpjs -> confirm` (the "pay myself" button) and skips the NIK screen. A BPJS patient, who in that puskesmas was the majority of the early arrivals, would take the longer path every time. The shortest path is a lower bound; a report should also consider the path that the most common users actually take, which is what Exercise 4 begins to address.

4. A depth-first search that tries every path without repeated screens and keeps the longest one to each goal:

```
def longest(start, goal, links, path=None):
    path = (path or []) + [start]
    if start == goal:
        return path
    best = None
    for nxt, _ in links.get(start, []):
        if nxt not in path:
            p = longest(nxt, goal, links, path)
            if p and (best is None or len(p) > len(best)):
                best = p
    return best
```

For `flow_v1.txt`, `longest("welcome", "ticket", links)` returns `welcome > service > bpjs > nik > confirm > ticket`, five taps. The search is exponential in the worst case, which is fine for a prototype with ten screens and not for a website with thousands of pages. In Java the same recursion uses a `List<String>` for the path.

5. Only the ticket branch of `render` changes, plus its callers: `render` gains a parameter `now` (minutes since midnight, for example 450 for 07:30), and the ticket shows `now + ahead * per` formatted as hours and minutes; for the dental ticket of Section 6.6.5, $450 + 45 = 495$ minutes, that is 08:15. The server passes the real time; the tests pass a fixed one. Reading the clock inside `render` would make the output change from minute to minute, so the tests could not check it and the Java and Python pages could no longer be compared byte for byte.

6. A possible prompt: “Design four screens for a touch-screen queue kiosk at an Indonesian community health center. No login and no insurance question. One question per screen. Welcome screen: one full-width primary button ‘Ambil nomor antrean’ at least 72 px tall, and two secondary buttons ‘Bantuan’ and ‘Jadwal poli’. Service screen: four full-width buttons ‘A - Umum’, ‘B - Gigi’, ‘C - KIA’, ‘D - Lansia’ and a ‘Kembali’ button. Confirmation screen repeating the chosen service in very large text. Ticket screen with the number in very large text, people ahead and estimated wait in minutes. Dark navy text on white, minimum text size 24 px, visible focus outline, a yellow ‘PROTOTIPE’ bar at the top. Language Indonesian.” Checks: every screen has a way on or back (write its flow file and run `flow_check`); text contrast at least 4.5:1 (Chapter 8’s checker); no small icon-only controls; no extra features the research removed (login, BPJS, carousels).

7. The wizard is Bu Rina (or Dimas), who listens to the voice note and replies by hand with the prototype messages of Section 6.9; nothing is automated. The customer agrees to try “a new way to order by voice note” and is told that a person listens and replies. Tasks: “order a pickup for tomorrow evening by voice note”, “change the pickup to the day after”, “ask how much a blanket costs”. The consent script says that voice notes are listened to by the laundry staff only, are deleted after the order is written down, and that the customer can stop at any time and order as before; and it asks separately whether voice notes may be kept for the length of the study.

References

[1] S. Houde and C. Hill, “What do prototypes prototype?,” in *Handbook of Human-Computer Interaction*, 2nd ed., M. Helander, T. K. Landauer, and P. Prabhu, Eds. Amsterdam, The Netherlands: Elsevier, 1997, pp. 367–381.

[2] B. Buxton, *Sketching User Experiences: Getting the Design Right and the Right Design*. San Francisco, CA, USA: Morgan Kaufmann, 2007.

[3] C. Snyder, *Paper Prototyping: The Fast and Easy Way to Design and Refine User Interfaces*. San Francisco, CA, USA: Morgan Kaufmann, 2003.

[4] J. F. Kelley, “An iterative design methodology for user-friendly natural language office information applications,” *ACM Transactions on Office Information Systems*, vol. 2, no. 1, pp. 26–41, 1984.

[5] Figma, “Config 2025 Launches Deepen Figma’s Design Capabilities As Its Platform Expands,” press release, May 7, 2025. [Online]. Available: https://www.figma.com/blog/config-2025-press-release/

[6] Figma Learn, “What’s new from Config 2026,” Help Center, Jun. 2026. [Online]. Available: https://help.figma.com/hc/en-us/articles/39582753756695-What-s-new-from-Config-2026

[7] Penpot, “Release Notes,” accessed Sep. 2026. [Online]. Available: https://penpot.app/release-notes

[8] W3C, *Web Content Accessibility Guidelines (WCAG) 2.2*, W3C Recommendation, Oct. 5, 2023, latest version accessed Sep. 2026. [Online]. Available: https://www.w3.org/TR/WCAG22/

[9] Pallets, “Flask Documentation (3.1.x),” accessed Sep. 2026. [Online]. Available: https://flask.palletsprojects.com/

[10] 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

[11] Microsoft, “Playwright for Python,” documentation, accessed Sep. 2026. [Online]. Available: https://playwright.dev/python/

[12] Google for Developers, “From idea to app: Introducing Stitch, a new way to design UIs,” May 20, 2025. [Online]. Available: https://developers.googleblog.com/stitch-a-new-way-to-design-uis/

Chapter 7
Usability Evaluation and Testing

A design is a claim that people can use it. Evaluation is how you check the claim. This chapter teaches the methods that working teams use in 2026 to find out whether an interface is usable, how usable, and what to fix first. By the end of it you will be able to run a heuristic evaluation and rate the severity of what you find; plan and moderate a think-aloud usability test with realistic tasks; measure task success, time and errors, and report success rates with the adjusted-Wald confidence interval and times with the geometric mean; score the System Usability Scale (SUS) and the two-item UMUX-Lite and interpret them against published benchmarks; explain why five to eight participants per round find most problems, and when they do not; and run and analyze an A/B test with a two-proportion z-test, a confidence interval for the difference and a sample-size calculation. Every calculation is done by hand first and then by programs in Java and Python.

7.1 Pre-Orders at Warung Bu Tini

Warung Bu Tini sells *nasi campur*, rice with a choice of side dishes, from a narrow front room near an office block and a campus in Surabaya. Between twelve and one o'clock the queue reaches the street, and office workers with forty-five minutes for lunch often give up and walk to a fast-food chain. Bu Tini's son Arif, a second-year informatics student, built a small web page so that customers can order in the morning and pick up at a chosen time: choose dishes, choose a quantity, choose a pickup time, pay by QRIS or at the counter, and receive a confirmation.

Arif tested it himself many times, and it worked. His friends tried it and said it was nice. Bu Tini was ready to print a sign with a QR code for the page. Before she did, Arif's lecturer asked him a simple question: *how do you know that her customers can use it?* Arif did not know. He knew that *he* could use it, and that his friends had not complained. Neither is evidence about a sixty-year-old retired teacher who buys lunch for her husband every day, or about an office worker ordering on a small phone while walking between meetings.

This chapter follows Arif as he finds out. First two classmates inspect the page against a list of principles, a *heuristic evaluation*. Then eight real customers try four tasks while thinking aloud, and Arif measures what happens. The answer turns out to be more interesting than "it works" or "it doesn't": most customers finish most tasks, the average satisfaction score is almost exactly that of an average product, and hidden inside that average are two older customers for whom the page nearly fails. After fixes, a two-week A/B test on the label of the order button shows how to measure the effect of a change on hundreds of customers who never meet the designer.

7.2 The Idea in Plain Words

7.2.1 What "usable" means

> **DEFINITION.** *Usability* is the extent to which a system can be used by specified users to achieve specified goals with *effectiveness* (accuracy and completeness), *efficiency* (resources such as time and effort spent) and *satisfaction* (the users' responses, from comfort to

> acceptance) in a specified context of use [1]. *Usability evaluation* is any systematic activity that assesses a design against these criteria.

The definition from ISO 9241-11:2018 has two lessons in it. First, usability is not one number: a design can be effective but slow, or fast but frustrating, and each needs its own measure. Second, usability exists only for *specified users, goals and context*: the same ordering page can be very usable for a student at a desk and nearly unusable for an older customer on a small phone in the sun. Every evaluation must therefore say whom it tested, doing what, and where.

7.2.2 Two families of evaluation

Evaluations come in two families. **Inspection methods** use experts instead of users: evaluators go through the design looking for violations of known principles. They are fast and cheap and find many problems, but they also report some that real users never meet, and they miss problems that come from users' own knowledge and habits. **Empirical methods** use real users doing real tasks: usability tests, questionnaires, analytics and experiments. They are slower and cost more per finding, but what they find is real.

A second distinction concerns purpose. **Formative** evaluation happens during design and asks *what is wrong and how do we fix it?*; its main output is a list of problems. **Summative** evaluation happens at a milestone and asks *how good is it?*; its main output is numbers, often compared with a target, a competitor or an earlier version. Arif's test is formative; the A/B test at the end of the chapter is summative. Most of this chapter's numbers serve both purposes.

7.3 Heuristic Evaluation

7.3.1 The method

Heuristic evaluation was introduced by Nielsen and Molich as a "discount" method: a few evaluators examine an interface and judge its compliance with a small set of recognized usability principles, the *heuristics* [2]. Chapter 3 presented Nielsen's ten heuristics (visibility of system status, match with the real world, user control and freedom, consistency and standards, error prevention, recognition rather than recall, flexibility and efficiency, aesthetic and minimalist design, helping users recover from errors, help and documentation). Here we use them as an evaluation tool.

The procedure has four steps:

1. **Brief the evaluators** on the users, their goals and the context. Arif's classmates were told: customers are office workers and residents of all ages, most on phones, ordering between 7 and 11 a.m. for pickup between 11:30 and 13:30.
2. **Inspect independently.** Each evaluator goes through the interface at least twice, once to get a feel for the flow and once to examine each screen, and writes down every problem with the heuristic it violates and where it occurs. Working independently matters: evaluators who talk too early converge on the same few problems.
3. **Merge.** The evaluators combine their lists into one, removing duplicates.
4. **Rate severity.** Each evaluator rates each problem independently, and the ratings are averaged.

Nielsen and Molich found that a single evaluator finds a modest fraction of the problems and that aggregating several evaluators finds far more, which is why three to five evaluators are the usual recommendation [2].

7.3.2 Severity ratings

A list of problems is useless without priorities. The common severity scale, also from Nielsen, runs from 0 to 4 [3]:

Table 7.1 The severity scale for usability problems, with examples from the ordering page.

Rating	Meaning	Laundry, warung or clinic example
0	Not a usability problem	A color the evaluator dislikes
1	Cosmetic: fix only if time allows	Menu photos load slowly on a slow connection
2	Minor: low priority	The price updates only after leaving the quantity field
3	Major: important to fix	The QRIS code is too small to scan from another phone
4	Catastrophe: must fix before release	The back button empties the cart

Severity combines three factors: *frequency* (how many users meet it), *impact* (how hard it is to overcome) and *persistence* (whether users meet it once and learn, or again and again). A problem that everyone meets once and overcomes easily may be less severe than one that a few users meet every time and cannot overcome.

Arif's two classmates found six problems in an afternoon, including the unlabeled menu icon that hid the "change order" link (violating *recognition rather than recall*) and the error message "Invalid input" (violating *help users recognize, diagnose and recover from errors*). They missed three that the user test later found, including the fact that the confirmation did not say when the food would be ready, which only customers who actually planned their lunch around it noticed.

7.4 Usability Testing with Real Users

7.4.1 Planning the test

A usability test puts representative users in front of the design with realistic tasks and watches what they do. Rubin and Chisnell's handbook, still the standard guide, organizes the plan around the purpose, the research questions, the participants, the method, the task list, the environment, the data to collect and the report [4]. For the warung the plan was short:

- **Research questions.** Can customers order, change an order and pay without help? Where do they hesitate or fail? Is the page equally usable for older customers?
- **Participants.** Eight regular customers recruited at the counter: four office workers, two students and two retired residents over sixty (P3 and P6), each thanked with a free lunch. The consent script followed Section 5.3.3, with screen and voice recording asked separately.
- **Setting.** Each participant used their own phone, in the warung after the lunch rush, so that screen size, connection and habits were real.
- **Tasks.** Four tasks, each with a clear end:
  - T1: *Find out what is on today's menu and how much a portion of nasi campur with chicken costs.*
  - T2: *Order two portions of nasi campur with chicken for pickup at 12:00.*
  - T3: *You have just learned a colleague wants lunch too. Change your order to three portions.*
  - T4: *Pay for your order with QRIS.* (A test QRIS code was used; no money moved.)
- **Measures.** Success or failure of each task, time on task, number of errors, a satisfaction questionnaire at the end (SUS and UMUX-Lite), and the problems observed.

Tasks are written as goals in the customer's words, never as instructions in the interface's words. "Change your order to three portions" is a goal; "tap the menu icon and choose Edit" is a set of instructions that tests nothing except reading.

7.4.2 Thinking aloud

In a think-aloud test the participant says what they are thinking while they work: what they are looking for, what they expect, what surprises them. The method rests on Ericsson and Simon's analysis of verbal reports, which showed that people can report the content of their working memory while doing a task without greatly changing how they do it, provided they are asked to *verbalize* rather than to *explain* [5]. That is why the facilitator's instruction is "please say whatever you are thinking as you go", not "please explain why you do things".

The facilitator's job is mostly to stay quiet. When a participant falls silent, a neutral prompt ("what are you thinking now?") is enough. When a participant asks for help ("where do I change it?"), the facilitator turns the question back ("where would you expect to find it?") and gives help only when the participant is stuck and the rest of the session would otherwise be lost, noting the task as a failure. Leading, explaining or defending the design, even with a sigh, ruins the data.

7.4.3 What happened

Four moments from the eight sessions show what think-aloud reveals:

- P3, a retired teacher, on T3: "Change… there's no 'change'. Maybe I have to order again? Then I'll pay twice." She ordered a second time, which counted as a failure of T3 and would have produced a double charge.
- P6, a retired civil servant, on T2: "The time list starts at seven in the morning… it's lunch, why seven?" He scrolled for twenty seconds through times nobody would choose.
- P1, an office worker, on T3: "I pressed back to fix the number and my cart is empty. Great." The back button of the browser discarded the cart.
- P5, a student, finished every task quickly and said at the end that the page was "fine, like any other". Participants like P5 show that a design works for confident users; they cannot show that it works for everyone.

7.5 Measuring Performance

7.5.1 Task success and its confidence interval

The most basic measure is whether the task was completed. The *completion rate* of a task is the number of participants who succeeded, x, divided by the number who tried, n. With small samples the ordinary confidence interval for a proportion behaves badly: for 8 successes out of 8 it gives the absurd interval "100% to 100%". Sauro and Lewis recommend the *adjusted-Wald* interval for completion rates in usability tests, because it has good coverage even for very small samples and for rates near 0% or 100% [6]. It adds $z^2/2$ successes and z^2 trials before computing:

$$\hat{p}_{\text{adj}} = \frac{x + z^2/2}{n + z^2}, \qquad \hat{p}_{\text{adj}} \pm z\sqrt{\frac{\hat{p}_{\text{adj}}(1-\hat{p}_{\text{adj}})}{n + z^2}} \tag{7.1}$$

where $z = 1.96$ for 95% confidence, and the interval is clipped to the range 0 to 1. With $z^2 = 3.8415$, adding $z^2/2 \approx 1.92$ successes and $z^2 \approx 3.84$ trials is almost the same as adding two successes and two failures, which is why the method is also known as "add two successes and two failures".

Worked by hand for T1, where all 8 participants succeeded: $\hat{p}_{\text{adj}} = (8 + 1.921)/(8 + 3.842) = 9.921/11.842 = 0.838$; half-width $1.96 \times \sqrt{0.838 \times 0.162/11.842} = 1.96 \times 0.1071 = 0.210$; the interval is $0.838 \pm 0.210 = [0.628, 1.048]$, clipped to $[62.8\%, 100\%]$. So "8 out of 8" means only that, with 95% confidence, at least about 63% of customers like these would complete the task. For T3, where 4 of 8 succeeded: $\hat{p}_{\text{adj}} = 5.921/11.842 = 0.500$, half-width $1.96 \times \sqrt{0.25/11.842} = 0.285$, interval $[21.5\%, 78.5\%]$. Small samples give wide intervals; that is not a defect of the method but an honest statement of how much eight people can tell you.

7.5.2 Time on task and the geometric mean

Task times are skewed: most participants take a similar time, a few take much longer. The arithmetic mean is pulled up by the slow few, and with small samples one very slow participant can dominate it. For samples smaller than about 25, Sauro and Lewis recommend the *geometric mean*, the n-th root of the product of the times, computed via logarithms [6]:

$$\bar{t}_{\text{geo}} = \exp\left(\frac{1}{n}\sum_{i=1}^{n} \ln t_i\right) \tag{7.2}$$

For T4, the seven successful times are 70, 41, 96, 55, 38, 47 and 62 seconds. Their natural logarithms are 4.248, 3.714, 4.564, 4.007, 3.638, 3.850 and 4.127, with sum 28.148 and mean 4.021; $e^{4.021} = 55.8$ seconds. The arithmetic mean is $409/7 = 58.4$ seconds; the difference is small here because no time is extreme, but for T2, with one success at 140 seconds, it grows. By convention the times of failed attempts are left out of the time measure, since a participant who gave up early did not "take" that time to complete the task; the failures are already counted in the completion rate.

7.5.3 Errors

An *error* is any action that does not lead toward the goal and that the participant has to undo or work around: tapping a wrong button, entering an invalid value, going back. Counting errors per task shows *where* a design misleads, which is often more useful for fixing it than the time.

7.5.4 The results

`data/sessions.csv` holds one line per participant and task: success (1 or 0), seconds, errors. The first part of the program in Section 7.7 prints the task summary:

```
Task  success  95% CI (adj. Wald)  geo-mean time  errors
T1    8/8     62.8% - 100.0%       37.4 s     0.25
T2    7/8     50.8% -  99.9%       82.9 s     1.25
T3    4/8     21.5% -  78.5%       97.9 s     1.75
T4    7/8     50.8% -  99.9%       55.8 s     0.50
```

T3, changing the order, is the clear problem: only half the participants succeeded, and those who did took the longest and made the most errors. The failures were P1 (the back button emptied the cart), P3 (ordered again), P6 (gave up) and P8 (could not find the change link at all).

7.6 Measuring Satisfaction: SUS and UMUX-Lite

7.6.1 The System Usability Scale

The System Usability Scale, published by Brooke in 1996 as a “quick and dirty” usability scale, is the most widely used standardized usability questionnaire [7]. It has ten statements, answered from 1 (strongly disagree) to 5 (strongly agree), alternating between positive and negative wording:

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

For the warung, “system” was replaced by “ordering page”, and an Indonesian translation was checked by reading it back into English with a second person. The score is computed by converting every answer to a contribution from 0 to 4 and scaling the sum to 0–100. If a_i is the answer to item i:

$$\text{SUS} = 2.5 \left(\sum_{i \in \{1,3,5,7,9\}} (a_i - 1) + \sum_{i \in \{2,4,6,8,10\}} (5 - a_i) \right) \tag{7.3}$$

The odd items are positive, so agreeing contributes more; the even items are negative, so disagreeing contributes more. Worked for P1, whose answers were 4, 2, 4, 1, 3, 2, 4, 2, 4, 2: odd contributions $3 + 3 + 2 + 3 + 3 = 14$; even contributions $3 + 4 + 3 + 3 + 3 = 16$; total 30; $\text{SUS} = 2.5 \times 30 = 75$.

A SUS score is *not* a percentage, and 70 does not mean “70% usable”. Its meaning comes from comparison with other products. In a large collection of SUS studies, the average score was 68 [8], and Sauro and Lewis turned the distribution of those scores into a *curved grading scale* in which, for example, a mean from 65.0 to 71.0 earns a C, from 80.8 to 84.0 an A, and 84.1 or above an A+ [6].

7.6.2 UMUX-Lite

When there is no time for ten items, for instance in a pop-up after a transaction, the two-item UMUX-Lite of Lewis, Utesch and Maher is a well-validated alternative [9]. Its items, answered on a 7-point scale, are “[This system’s] capabilities meet my requirements” and “[This system] is easy to use”. With answers c and e from 1 to 7, the raw score is scaled to 0–100, and a regression equation puts it on the SUS scale so that the two can be compared [10]:

$$\text{UMUX}_{\text{raw}} = \frac{(c-1)+(e-1)}{12} \times 100, \qquad \widehat{\text{SUS}} = 0.65\,\text{UMUX}_{\text{raw}} + 22.9 \tag{7.4}$$

For P1, $c = 6$ and $e = 5$: raw $= (5+4)/12 \times 100 = 75.0$, and on the SUS scale $0.65 \times 75 + 22.9 = 71.65$.

7.6.3 The results

The program of Section 7.7 continues:

```
SUS per participant:
  P1:  75.0
  P2:  95.0
  P3:  37.5
  P4:  77.5
  P5: 100.0
  P6:  40.0
  P7:  77.5
  P8:  60.0
SUS mean 70.3 (95% CI 51.1 to 89.5), grade C; the average product scores 68
UMUX-Lite raw mean 74.0, on the SUS scale 71.0
```

The confidence interval is the t-interval of Section 5.5.3, computed from the eight scores. **Figure 7.1** plots the eight scores against the grade bands. The mean, 70.3, is a C, just above the average product; UMUX-Lite, mapped onto the SUS scale, agrees at 71.0. If Arif reported only the mean, he would say "about average". The picture says something more important: six participants gave scores from 60 to 100, and the two older participants, P3 and P6, gave 37.5 and 40, well inside F. They are the same two who failed T3 and struggled with T2. An average can hide a group for whom the design fails, and it usually hides exactly the people at the edges of Chapter 5.

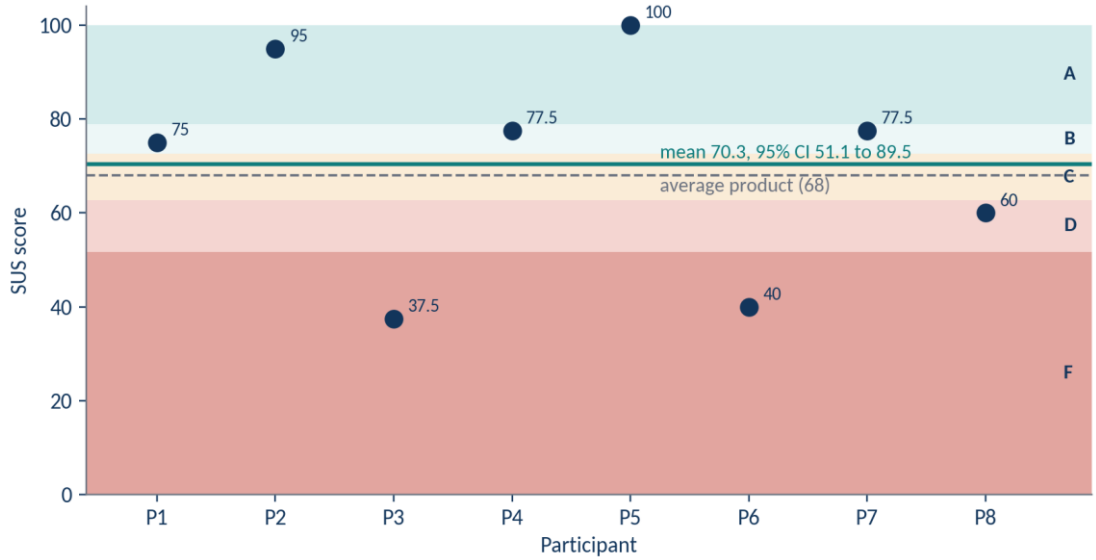


Figure 7.1 SUS scores of the eight participants against the curved grading bands; the mean (70.3) is just above the average product (68), but the two older participants score in the F band.

> **KEY INSIGHT.** Report distributions, not only averages. With eight participants, a plot of all eight scores takes no more space than a mean and shows at once whether everyone had a similar experience or whether a group was left behind.

7.7 Building the Metrics Program

Listing 7.1 computes everything in Sections 7.5 and 7.6 from the three data files, plus the problem-discovery estimate of Section 7.8. Each formula is a small function with the name of its equation: `adjusted_wald` is Equation (7.1), `geometric_mean` is Equation (7.2), `sus_score` is Equation (7.3), `umux_lite` is Equation (7.4), and `discovery` is Equation (7.5) below.

Listing 7.1 `usability_metrics.py`: task success with adjusted-Wald intervals, geometric mean times, SUS with grade, UMUX-Lite and problem discovery.

```
"""Usability-test metrics: task success (adjusted-Wald CI), geometric mean
time, SUS with grade, UMUX-Lite, and the problem-discovery estimate.
Usage: python usability_metrics.py [data folder]"""
import csv
import math
import sys

Z = 1.959964
# Curved grading scale for SUS (lower bound of each grade)
GRADES = [(84.1, "A+"), (80.8, "A"), (78.9, "A-"), (77.2, "B+"), (74.1, "B"),
          (72.6, "B-"), (71.1, "C+"), (65.0, "C"), (62.7, "C-"), (51.7, "D"),
          (0.0, "F")]


def rows(path):
    with open(path, newline="", encoding="utf-8") as f:
        return list(csv.DictReader(f))


def adjusted_wald(successes, n):
    """95% adjusted-Wald interval for a completion rate."""
    p = (successes + Z * Z / 2) / (n + Z * Z)
    half = Z * math.sqrt(p * (1 - p) / (n + Z * Z))
    return max(0.0, p - half), min(1.0, p + half)


def geometric_mean(values):
    return math.exp(sum(math.log(v) for v in values) / len(values))


def sus_score(answers):
    """answers: the ten 1-5 responses in questionnaire order."""
    total = 0
    for i, a in enumerate(answers):
        total += (a - 1) if i % 2 == 0 else (5 - a)   # odd items positive
    return total * 2.5


def grade(score):
    for low, letter in GRADES:
        if score >= low:
            return letter
    return "F"
```

Listing 7.1 (Continued) *usability_metrics.py: task success with adjusted-Wald intervals, geometric mean times, SUS with grade, UMUX-Lite and problem discovery.*

```
def umux_lite(capabilities, easy):
    """Raw UMUX-Lite (0-100) and its regression onto the SUS scale."""
    raw = (capabilities + easy - 2) * 100 / 12
    return raw, 0.65 * raw + 22.9


def t_critical(df):
    z = Z
    return (z + (z**3 + z) / (4 * df)
            + (5 * z**5 + 16 * z**3 + 3 * z) / (96 * df * df))


def mean_ci(values):
    n = len(values)
    mean = sum(values) / n
    sd = math.sqrt(sum((v - mean) ** 2 for v in values) / (n - 1))
    return mean, t_critical(n - 1) * sd / math.sqrt(n)


def discovery(p, n):
    """Expected share of problems found by n participants."""
    return 1 - (1 - p) ** n


def main(args):
    folder = args[0] if args else "../data"
    sessions = rows(folder + "/sessions.csv")
    print("Task  success  95% CI (adj. Wald)  geo-mean time  errors")
    for task in sorted({r["task"] for r in sessions}):
        rs = [r for r in sessions if r["task"] == task]
        ok = [r for r in rs if r["success"] == "1"]
        low, high = adjusted_wald(len(ok), len(rs))
        gm = geometric_mean([float(r["seconds"]) for r in ok])
        err = sum(int(r["errors"]) for r in rs) / len(rs)
        print(f"{task:<6}{len(ok)}/{len(rs)}    {low * 100:5.1f}% - "
              f"{high * 100:5.1f}%   {gm:8.1f} s   {err:6.2f}")
```

Listing 7.1 (Continued) *usability_metrics.py: task success with adjusted-Wald intervals, geometric mean times, SUS with grade, UMUX-Lite and problem discovery.*

```
    scores = []
    print("\nSUS per participant:")
    for r in rows(folder + "/sus.csv"):
        s = sus_score([int(r[f"q{i}"]) for i in range(1, 11)])
        scores.append(s)
        print(f"  {r['participant']}: {s:5.1f}")
    mean, half = mean_ci(scores)
    print(f"SUS mean {mean:.1f} (95% CI {mean - half:.1f} to {mean + half:.1f}),"
          f" grade {grade(mean)}; the average product scores 68")

    raws, preds = [], []
    for r in rows(folder + "/umux.csv"):
        raw, pred = umux_lite(int(r["capabilities"]), int(r["easy"]))
        raws.append(raw)
        preds.append(pred)
    print(f"UMUX-Lite raw mean {sum(raws) / len(raws):.1f}, "
          f"on the SUS scale {sum(preds) / len(preds):.1f}")

    problems = rows(folder + "/problems.csv")
    n = len(scores)
    hits = sum(len(r["participants"].split()) for r in problems)
    p = hits / (len(problems) * n)
    print(f"\n{len(problems)} problems, {hits} sightings by {n} participants:"
          f" p = {p:.3f}")
    print("participants  found (p=0.31)  found (this study)")
    for k in (1, 2, 3, 5, 8, 10):
        print(f"{k:>12}  {discovery(0.31, k) * 100:13.0f}%"
              f"  {discovery(p, k) * 100:17.0f}%")


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

The curved grading scale is a list of lower bounds from the highest grade down; `grade` returns the first grade whose lower bound the score reaches. `sus_score` uses the position `i` in the list: even positions (0, 2, ...) are the odd-numbered, positive items.

Listing 7.2 is the Java version. It reads each CSV line with `split(",")`, which is enough because none of the fields contains a comma.

Listing 7.2 `UsabilityMetrics.java`: the same metrics in Java.

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

/** Usability-test metrics: task success (adjusted-Wald CI), geometric mean
 *  time, SUS with grade, UMUX-Lite, and the problem-discovery estimate.
 *  Usage: java UsabilityMetrics [data folder] */
public class UsabilityMetrics {
    static final double Z = 1.959964;
    /** Curved grading scale for SUS (lower bound of each grade). */
    static final double[] LOW = {84.1, 80.8, 78.9, 77.2, 74.1, 72.6, 71.1, 65.0,
        62.7, 51.7, 0.0};
    static final String[] LETTER = {"A+", "A", "A-", "B+", "B", "B-", "C+", "C",
        "C-", "D", "F"};

    static List<String[]> rows(String path) throws IOException {
        List<String> lines = Files.readAllLines(Path.of(path));
        List<String[]> out = new ArrayList<>();
        for (String line : lines.subList(1, lines.size()))
            if (!line.isBlank()) out.add(line.split(","));
        return out;
    }

    /** 95% adjusted-Wald interval for a completion rate. */
    static double[] adjustedWald(int successes, int n) {
        double p = (successes + Z * Z / 2) / (n + Z * Z);
        double half = Z * Math.sqrt(p * (1 - p) / (n + Z * Z));
        return new double[] {Math.max(0, p - half), Math.min(1, p + half)};
    }

    static double geometricMean(List<Double> values) {
        double sum = 0;
        for (double v : values) sum += Math.log(v);
        return Math.exp(sum / values.size());
    }
```

Listing 7.2 (Continued) *UsabilityMetrics.java: the same metrics in Java.*

```
    /** answers: the ten 1-5 responses in questionnaire order. */
    static double susScore(int[] answers) {
        int total = 0;
        for (int i = 0; i < answers.length; i++)
            total += (i % 2 == 0) ? answers[i] - 1 : 5 - answers[i];
        return total * 2.5;
    }

    static String grade(double score) {
        for (int i = 0; i < LOW.length; i++) if (score >= LOW[i]) return LETTER[i];
        return "F";
    }

    /** Raw UMUX-Lite (0-100) and its regression onto the SUS scale. */
    static double[] umuxLite(int capabilities, int easy) {
        double raw = (capabilities + easy - 2) * 100.0 / 12;
        return new double[] {raw, 0.65 * raw + 22.9};
    }

    static double tCritical(int df) {
        return Z + (Math.pow(Z, 3) + Z) / (4.0 * df)
            + (5 * Math.pow(Z, 5) + 16 * Math.pow(Z, 3) + 3 * Z) / (96.0 * df * df);
    }

    /** Expected share of problems found by n participants. */
    static double discovery(double p, int n) {
        return 1 - Math.pow(1 - p, n);
    }

    public static void main(String[] args) throws IOException {
        Locale.setDefault(Locale.US);
        String folder = args.length > 0 ? args[0] : "../data";
        List<String[]> sessions = rows(folder + "/sessions.csv");
        System.out.println(
            "Task  success  95% CI (adj. Wald)  geo-mean time  errors");
        TreeSet<String> tasks = new TreeSet<>();
        for (String[] r : sessions) tasks.add(r[1]);
        for (String task : tasks) {
            int n = 0, ok = 0, errors = 0;
            List<Double> times = new ArrayList<>();
            for (String[] r : sessions) {
                if (!r[1].equals(task)) continue;
                n++;
                errors += Integer.parseInt(r[4]);
                if (r[2].equals("1")) {
                    ok++;
```

Listing 7.2 (Continued) *UsabilityMetrics.java: the same metrics in Java.*

```
                    times.add(Double.parseDouble(r[3]));
                }
            }
            double[] ci = adjustedWald(ok, n);
            System.out.printf("%-6s%d/%d    %5.1f%% - %5.1f%%   %8.1f s   %6.2f%n",
                task, ok, n, ci[0] * 100, ci[1] * 100, geometricMean(times),
                (double) errors / n);
        }
        System.out.println("\nSUS per participant:");
        List<Double> scores = new ArrayList<>();
        for (String[] r : rows(folder + "/sus.csv")) {
            int[] a = new int[10];
            for (int i = 0; i < 10; i++) a[i] = Integer.parseInt(r[i + 1]);
            double s = susScore(a);
            scores.add(s);
            System.out.printf("  %s: %5.1f%n", r[0], s);
        }
        int n = scores.size();
        double mean = 0, ss = 0;
        for (double s : scores) mean += s / n;
        for (double s : scores) ss += (s - mean) * (s - mean);
        double half = tCritical(n - 1) * Math.sqrt(ss / (n - 1)) / Math.sqrt(n);
        System.out.printf("SUS mean %.1f (95%% CI %.1f to %.1f), grade %s; "
            + "the average product scores 68%n", mean, mean - half, mean + half,
            grade(mean));
        double rawSum = 0, predSum = 0;
        List<String[]> umux = rows(folder + "/umux.csv");
        for (String[] r : umux) {
            double[] u = umuxLite(Integer.parseInt(r[1]), Integer.parseInt(r[2]));
            rawSum += u[0];
            predSum += u[1];
        }
        System.out.printf("UMUX-Lite raw mean %.1f, on the SUS scale %.1f%n",
            rawSum / umux.size(), predSum / umux.size());
        List<String[]> problems = rows(folder + "/problems.csv");
        int hits = 0;
        for (String[] r : problems) hits += r[2].strip().split("\\s+").length;
        double p = (double) hits / (problems.size() * n);
        System.out.printf("%n%d problems, %d sightings by %d participants: "
            + "p = %.3f%n", problems.size(), hits, n, p);
        System.out.println("participants  found (p=0.31)  found (this study)");
        for (int k : new int[] {1, 2, 3, 5, 8, 10})
            System.out.printf("%12d  %13.0f%%  %17.0f%%%n", k,
                discovery(0.31, k) * 100, discovery(p, k) * 100);
    }
}
```

Running `python3 usability_metrics.py` or `java -cp out UsabilityMetrics` prints the complete report, whose parts were shown in Sections 7.5.4 and 7.6.3:

```
Task  success  95% CI (adj. Wald)  geo-mean time  errors
T1    8/8     62.8% - 100.0%       37.4 s     0.25
T2    7/8     50.8% -  99.9%       82.9 s     1.25
T3    4/8     21.5% -  78.5%       97.9 s     1.75
T4    7/8     50.8% -  99.9%       55.8 s     0.50

SUS per participant:
  P1:  75.0
  P2:  95.0
  P3:  37.5
  P4:  77.5
  P5: 100.0
  P6:  40.0
  P7:  77.5
  P8:  60.0
SUS mean 70.3 (95% CI 51.1 to 89.5), grade C; the average product scores 68
UMUX-Lite raw mean 74.0, on the SUS scale 71.0

9 problems, 25 sightings by 8 participants: p = 0.347
participants  found (p=0.31)  found (this study)
           1             31%                 35%
           2             52%                 57%
           3             67%                 72%
           5             84%                 88%
           8             95%                 97%
          10             98%                 99%
```

7.8 How Many Participants?

7.8.1 The problem-discovery model

Why eight participants, and not fifty? Nielsen and Landauer modeled the discovery of usability problems as follows [11]. Suppose each participant, independently, meets a given problem with probability p. Then the probability that a problem is met by at least one of n participants is one minus the probability that all of them miss it:

$$P(\text{found}) = 1 - (1 - p)^n \tag{7.5}$$

and, averaged over problems, this is the expected share of problems that n participants reveal. Across the projects they studied, the average p was about 0.31, and with $p = 0.31$ five participants reveal $1 - 0.69^5 = 1 - 0.156 = 0.844$, about 85% of the problems. That calculation is the source of Nielsen's well-known advice that the best return comes from testing with about five users, fixing what they find and testing again, rather than testing once with many [12].

7.8.2 Estimating p from the study

The data of Arif's test allow an estimate of p for this design. `data/problems.csv` lists the nine problems found, with a severity and the participants who met each one. **Figure 7.2** draws it as a problem-by-participant matrix: each row is a problem, each dot a participant who met it.

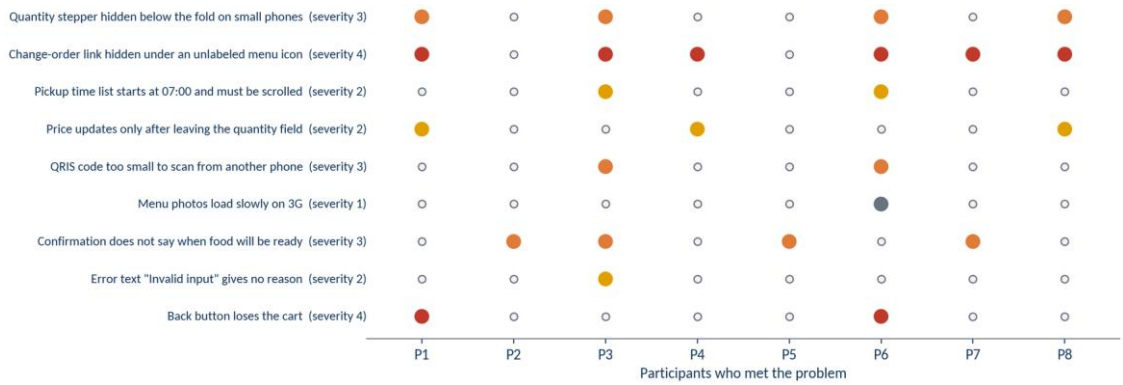


Figure 7.2 The nine problems of the warung test and the participants who met each; larger colored dots mark a sighting, colored by severity, and P3 and P6 meet the most problems.

There are $9 \times 8 = 72$ cells and 25 sightings, so $\hat{p} = 25/72 = 0.347$. **Figure 7.3** plots Equation (7.5) for this value, for Nielsen and Landauer's 0.31, and for a design whose problems are rarer, $p = 0.15$.

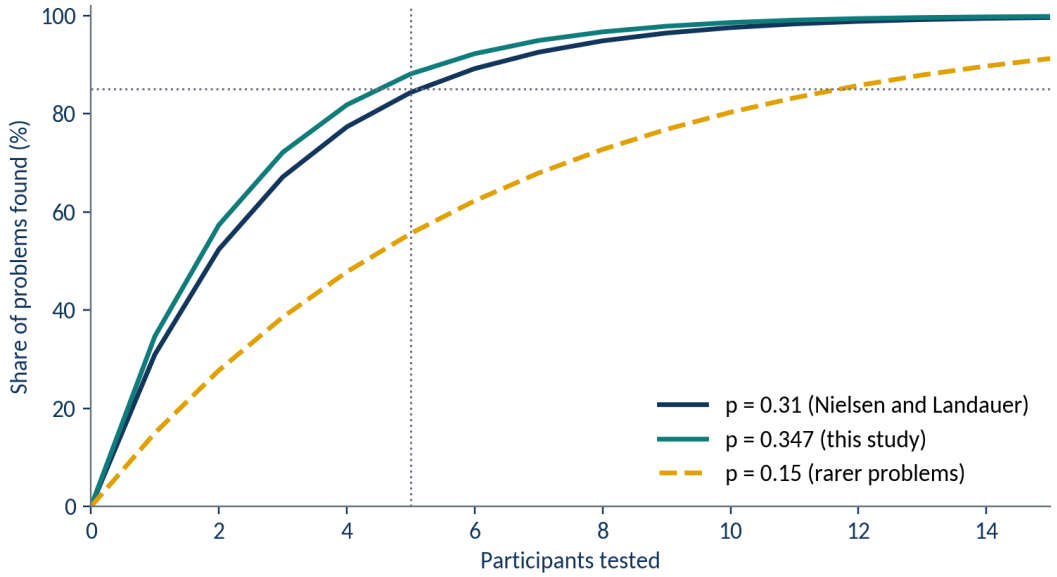


Figure 7.3 Expected share of problems found as participants are added, from $1 - (1 - p)^n$: with p around 0.3, five participants find about 85%; with rarer problems ($p = 0.15$), five find only about 56%.

The model comes with three warnings. First, $\hat{p}$ estimated from a small study is biased upward, because problems that nobody met are not in the matrix at all; Lewis proposed adjustments for this [13]. Second, the model assumes that all participants are alike. Look at the matrix: P3 and P6 met far more problems than P5. If Arif had tested five confident office workers, p for *them* might have looked high and the test "complete", while the problems of older customers stayed hidden. Five users find most problems *of the kind of user tested*. When a product serves distinct groups, test each group. Third, rarer problems need more participants: with $p = 0.15$, five participants find only $1 - 0.85^5 = 0.556$ of the problems. That is why the advice is "five, fix, and test again", not "five and done".

7.9 Testing the Programs

Listing 7.3 checks the formulas on cases whose answers are known: the best and worst SUS answers give 100 and 0, all threes give 50; the grade boundaries; the top UMUX-Lite score; the adjusted-Wald interval for 8 of 8 worked by hand in Section 7.5.1; a geometric mean that must be

100; the 84% of Equation (7.5) for five participants; the normal distribution at 1.96; and the A/B test of Section 7.10.

Listing 7.3 `test_eval.py`: tests of the usability metrics and the A/B test.

```
"""Tests for usability_metrics.py and ab_test.py.
Run: python -m unittest test_eval.py"""
import unittest
from ab_test import phi, sample_size, two_proportion_test
from usability_metrics import (adjusted_wald, discovery, geometric_mean,
                               grade, sus_score, umux_lite)


class EvalTest(unittest.TestCase):
    def test_sus_extremes_and_neutral(self):
        self.assertEqual(sus_score([5, 1] * 5), 100.0)
        self.assertEqual(sus_score([1, 5] * 5), 0.0)
        self.assertEqual(sus_score([3] * 10), 50.0)

    def test_grade_boundaries(self):
        self.assertEqual(grade(68.0), "C")
        self.assertEqual(grade(84.1), "A+")
        self.assertEqual(grade(51.6), "F")

    def test_umux_lite_top_score(self):
        self.assertEqual(umux_lite(7, 7), (100.0, 87.9))

    def test_adjusted_wald_all_successes(self):
        low, high = adjusted_wald(8, 8)
        self.assertAlmostEqual(low, 0.628, places=3)
        self.assertEqual(high, 1.0)

    def test_geometric_mean(self):
        self.assertAlmostEqual(geometric_mean([10, 1000]), 100.0)

    def test_discovery(self):
        self.assertAlmostEqual(discovery(0.31, 5), 0.8436, places=4)

    def test_normal_cdf(self):
        self.assertAlmostEqual(phi(1.959964), 0.975, places=6)

    def test_ab_on_study_data(self):
        p1, p2, z, p, ci = two_proportion_test(157, 1210, 198, 1187)
        self.assertAlmostEqual(z, 2.554, places=3)
        self.assertLess(p, 0.05)
        self.assertEqual(sample_size(0.13, 0.16), 2162)


if __name__ == "__main__":
    unittest.main()
```

```
Ran 8 tests in 0.001s

OK
```

Listing 7.4 runs the same checks in Java.

Listing 7.4 `EvalTest.java`: the same checks in Java.

```java
/** Tests for UsabilityMetrics and AbTest, without a test library.
 *  Run: java -cp out EvalTest */
public class EvalTest {
    static int passed = 0;

    static void check(boolean ok, String name) {
        if (!ok) throw new AssertionError("FAILED: " + name);
        passed++;
    }

    static boolean near(double a, double b, double tol) {
        return Math.abs(a - b) < tol;
    }

    public static void main(String[] args) {
        check(UsabilityMetrics.susScore(new int[] {5, 1, 5, 1, 5, 1, 5, 1, 5, 1})
            == 100.0, "SUS best");
        check(UsabilityMetrics.susScore(new int[] {3, 3, 3, 3, 3, 3, 3, 3, 3, 3})
            == 50.0, "SUS neutral");
        check(UsabilityMetrics.grade(68.0).equals("C"), "grade C");
        check(UsabilityMetrics.grade(51.6).equals("F"), "grade F");
        double[] u = UsabilityMetrics.umuxLite(7, 7);
        check(u[0] == 100.0 && near(u[1], 87.9, 1e-9), "UMUX-Lite top");
        double[] ci = UsabilityMetrics.adjustedWald(8, 8);
        check(near(ci[0], 0.628, 0.0005) && ci[1] == 1.0, "adjusted Wald");
        check(near(UsabilityMetrics.geometricMean(java.util.List.of(10.0, 1000.0)),
            100.0, 1e-9), "geometric mean");
        check(near(UsabilityMetrics.discovery(0.31, 5), 0.8436, 0.00005),
            "discovery");
        check(near(AbTest.phi(1.959964), 0.975, 1e-6), "normal CDF");
        double[] r = AbTest.twoProportionTest(157, 1210, 198, 1187);
        check(near(r[2], 2.554, 0.0005) && r[3] < 0.05, "A/B z");
        check(AbTest.sampleSize(0.13, 0.16) == 2162, "sample size");
        System.out.println(passed + " checks passed");
    }
}
```

```
11 checks passed
```

7.10 A/B Testing

7.10.1 From eight people to hundreds

After the test, Arif fixed the catastrophes and major problems: the back button kept the cart, a labeled “Change order” button appeared on the confirmation, the pickup times started at 11:30, the QRIS code was shown full-width with a “save image” option, and the confirmation said “Ready at 12:00”. A second round with five new participants, including two older customers, found no failure on T3.

One question remained that eight or five participants cannot answer: which wording of the order button makes more visitors actually place an order? The old label was *Kirim* (“Send”). The think-aloud sessions suggested that customers were unsure what “Send” committed them to, so Arif wrote a second version: *Pesan sekarang – ambil jam 12.00* (“Order now – pick up at 12:00”, with the chosen time filled in). The difference between two such versions is small, so it can only be measured on many people, and the people who use the real page are the right ones.

> **DEFINITION.** An *A/B test* (an online controlled experiment) randomly assigns each visitor to one of two versions of a design, A (the control) or B (the treatment), and compares a pre-

chosen metric between the groups. Because assignment is random, any difference larger than chance can be attributed to the change [14].

Kohavi, Tang and Xu, drawing on experience at large web companies, stress three rules that matter even for a warung: choose the metric *before* the test starts (here, the share of visitors to the menu page who place an order); fix the sample size or duration in advance rather than stopping when the result looks good; and check that the two groups are the same size as planned, since an unexpected imbalance usually means a bug in the assignment [14].

7.10.2 The two-proportion z-test

After two weeks, `data/ab.csv` held the counts: version A was shown to 1,210 visitors and received 157 orders; version B to 1,187 visitors and received 198. The completion rates are $\hat{p}_A = 157/1210 = 0.1298$ and $\hat{p}_B = 198/1187 = 0.1668$. Is the difference of 3.7 points real, or could it arise by chance if the labels made no difference?

The *two-proportion z-test* answers this. Under the hypothesis that the labels make no difference, both groups share one rate, estimated by pooling: $\bar{p} = (x_A + x_B)/(n_A + n_B)$. The test statistic divides the observed difference by its standard error under that hypothesis:

$$z = \frac{\hat{p}_B - \hat{p}_A}{\sqrt{\bar{p}(1-\bar{p})\left(\frac{1}{n_A} + \frac{1}{n_B}\right)}} \tag{7.6}$$

and the two-sided *p-value* is the probability that a standard normal variable is at least $|z|$ away from zero, $2(1 - \Phi(|z|))$, where Φ is the standard normal cumulative distribution function.

By hand: $\bar{p} = 355/2397 = 0.1481$; $\bar{p}(1-\bar{p}) = 0.1481 \times 0.8519 = 0.1262$; $1/1210 + 1/1187 = 0.000826 + 0.000842 = 0.001669$; the standard error is $\sqrt{0.1262 \times 0.001669} = \sqrt{0.0002106} = 0.01451$; $z = 0.0370/0.01451 = 2.55$. A z of 2.55 leaves about 0.5% in each tail, so $p \approx 0.011$: if the labels made no difference, a gap this large would appear only about once in a hundred such tests. The 95% confidence interval for the difference uses the unpooled standard error, $\sqrt{\hat{p}_A(1-\hat{p}_A)/n_A + \hat{p}_B(1-\hat{p}_B)/n_B}$, and gives about +0.9 to +6.6 points.

7.10.3 How many visitors are needed?

Before a test starts, one should know how many visitors are needed to detect the smallest difference worth acting on. For a test at the 5% level with 80% *power* (an 80% chance of detecting a true difference of the chosen size), the number of visitors per version to detect a change from p_1 to p_2 is

$$n = \frac{\left(z_{\alpha/2}\sqrt{2\bar{p}(1-\bar{p})} + z_{\beta}\sqrt{p_1(1-p_1) + p_2(1-p_2)}\right)^2}{(p_2 - p_1)^2} \tag{7.7}$$

where $\bar{p} = (p_1 + p_2)/2$, $z_{\alpha/2} = 1.96$ and $z_{\beta} = 0.8416$. The denominator is the square of the difference, so halving the difference to be detected quadruples the sample.

7.10.4 The program

Listing 7.5 implements Equation (7.6) and Equation (7.7). The standard normal distribution function Φ is computed with the rational approximation 7.1.26 of Abramowitz and Stegun, whose

error is below 1.5×10^{-7} [15]; Python has `math.erf`, but Java has no built-in error function, and using the same approximation in both keeps their outputs identical.

Listing 7.5 `ab_test.py`: the two-proportion z-test, the confidence interval of the difference and the sample size.

```
"""Compare the completion rates of two designs (A/B test): two-proportion
z-test, confidence interval of the difference, and the sample size needed.
Usage: python ab_test.py [ab.csv]"""
import csv
import math
import sys

Z95, Z80 = 1.959964, 0.841621   # two-sided 5% level; 80% power


def phi(x):
    """Standard normal CDF (Abramowitz-Stegun 7.1.26, error < 1.5e-7)."""
    t = 1 / (1 + 0.3275911 * abs(x) / math.sqrt(2))
    poly = t * (0.254829592 + t * (-0.284496736 + t * (1.421413741
           + t * (-1.453152027 + t * 1.061405429))))
    erf = 1 - poly * math.exp(-x * x / 2)
    return 0.5 * (1 + erf) if x >= 0 else 0.5 * (1 - erf)


def two_proportion_test(x1, n1, x2, n2):
    """Return p1, p2, z, two-sided p-value, 95% CI of p2 - p1."""
    p1, p2 = x1 / n1, x2 / n2
    pooled = (x1 + x2) / (n1 + n2)
    se0 = math.sqrt(pooled * (1 - pooled) * (1 / n1 + 1 / n2))
    z = (p2 - p1) / se0
    p_value = 2 * (1 - phi(abs(z)))
    se = math.sqrt(p1 * (1 - p1) / n1 + p2 * (1 - p2) / n2)
    diff = p2 - p1
    return p1, p2, z, p_value, (diff - Z95 * se, diff + Z95 * se)


def sample_size(p1, p2):
    """Visitors per variant to detect p1 -> p2 (5% level, 80% power)."""
    pbar = (p1 + p2) / 2
    top = (Z95 * math.sqrt(2 * pbar * (1 - pbar))
           + Z80 * math.sqrt(p1 * (1 - p1) + p2 * (1 - p2)))
    return math.ceil(top * top / ((p2 - p1) ** 2))
```

Listing 7.5 (Continued) *ab_test.py: the two-proportion z-test, the confidence interval of the difference and the sample size.*

```
def main(args):
    path = args[0] if args else "../data/ab.csv"
    with open(path, newline="", encoding="utf-8") as f:
        a, b = list(csv.DictReader(f))
    x1, n1 = int(a["orders"]), int(a["visitors"])
    x2, n2 = int(b["orders"]), int(b["visitors"])
    p1, p2, z, p_value, (low, high) = two_proportion_test(x1, n1, x2, n2)
    print(f"A '{a['label']}': {x1}/{n1} = {p1 * 100:.2f}%")
    print(f"B '{b['label']}': {x2}/{n2} = {p2 * 100:.2f}%")
    print(f"difference B - A: {(p2 - p1) * 100:+.2f} points, "
          f"95% CI {low * 100:+.2f} to {high * 100:+.2f}")
    print(f"z = {z:.3f}, two-sided p = {p_value:.4f}")
    verdict = "significant" if p_value < 0.05 else "not significant"
    print(f"at the 5% level the difference is {verdict}")
    for target in (0.16, 0.14):
        print(f"visitors per variant to detect 13% -> {target * 100:.0f}%: "
              f"{sample_size(0.13, target)}")


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

Listing 7.6 is the Java version.

Listing 7.6 `AbTest.java`: the same A/B analysis in Java.

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

/** Compare the completion rates of two designs (A/B test): two-proportion
 *  z-test, confidence interval of the difference, and the sample size needed.
 *  Usage: java AbTest [ab.csv] */
public class AbTest {
    static final double Z95 = 1.959964, Z80 = 0.841621;

    /** Standard normal CDF (Abramowitz-Stegun 7.1.26, error < 1.5e-7). */
    static double phi(double x) {
        double t = 1 / (1 + 0.3275911 * Math.abs(x) / Math.sqrt(2));
        double poly = t * (0.254829592 + t * (-0.284496736 + t * (1.421413741
            + t * (-1.453152027 + t * 1.061405429))));
        double erf = 1 - poly * Math.exp(-x * x / 2);
        return x >= 0 ? 0.5 * (1 + erf) : 0.5 * (1 - erf);
    }

    /** Returns {p1, p2, z, two-sided p-value, CI low, CI high}. */
    static double[] twoProportionTest(int x1, int n1, int x2, int n2) {
        double p1 = (double) x1 / n1, p2 = (double) x2 / n2;
        double pooled = (double) (x1 + x2) / (n1 + n2);
        double se0 = Math.sqrt(pooled * (1 - pooled) * (1.0 / n1 + 1.0 / n2));
        double z = (p2 - p1) / se0;
        double pValue = 2 * (1 - phi(Math.abs(z)));
        double se = Math.sqrt(p1 * (1 - p1) / n1 + p2 * (1 - p2) / n2);
        double diff = p2 - p1;
        return new double[] {p1, p2, z, pValue, diff - Z95 * se, diff + Z95 * se};
    }

    /** Visitors per variant to detect p1 -> p2 (5% level, 80% power). */
    static int sampleSize(double p1, double p2) {
        double pbar = (p1 + p2) / 2;
        double top = Z95 * Math.sqrt(2 * pbar * (1 - pbar))
            + Z80 * Math.sqrt(p1 * (1 - p1) + p2 * (1 - p2));
        return (int) Math.ceil(top * top / ((p2 - p1) * (p2 - p1)));
    }
```

Listing 7.6 (Continued) *AbTest.java: the same A/B analysis in Java.*

```
    public static void main(String[] args) throws IOException {
        Locale.setDefault(Locale.US);
        String path = args.length > 0 ? args[0] : "../data/ab.csv";
        List<String> lines = Files.readAllLines(Path.of(path));
        String[] a = lines.get(1).split(","), b = lines.get(2).split(",");
        int x1 = Integer.parseInt(a[3]), n1 = Integer.parseInt(a[2]);
        int x2 = Integer.parseInt(b[3]), n2 = Integer.parseInt(b[2]);
        double[] r = twoProportionTest(x1, n1, x2, n2);
        System.out.printf("A '%s': %d/%d = %.2f%%%n", a[1], x1, n1, r[0] * 100);
        System.out.printf("B '%s': %d/%d = %.2f%%%n", b[1], x2, n2, r[1] * 100);
        System.out.printf("difference B - A: %+.2f points, "
            + "95%% CI %+.2f to %+.2f%n", (r[1] - r[0]) * 100, r[4] * 100,
            r[5] * 100);
        System.out.printf("z = %.3f, two-sided p = %.4f%n", r[2], r[3]);
        System.out.println("at the 5% level the difference is "
            + (r[3] < 0.05 ? "significant" : "not significant"));
        for (double target : new double[] {0.16, 0.14})
            System.out.printf("visitors per variant to detect 13%% -> %.0f%%: %d%n",
                target * 100, sampleSize(0.13, target));
    }
}
```

`python3 ab_test.py` and `java -cp out AbTest` print:

```
A 'Kirim': 157/1210 = 12.98%
B 'Pesan sekarang - ambil jam 12.00': 198/1187 = 16.68%
difference B - A: +3.71 points, 95% CI +0.86 to +6.55
z = 2.554, two-sided p = 0.0107
at the 5% level the difference is significant
visitors per variant to detect 13% -> 16%: 2162
visitors per variant to detect 13% -> 14%: 18330
```

The new label won: +3.71 points, significant at the 5% level. But look at the last two lines. To detect a lift from 13% to 16% with 80% power, the test needed about 2,162 visitors per version; the warung's two weeks gave about 1,200. The result is significant, but the test was underpowered for a 3-point effect, and the confidence interval is wide: the true lift could be as small as 0.9 points or as large as 6.6. Arif's honest report was: "B is very likely better; by how much we do not know precisely." And detecting a 1-point lift (13% to 14%) would need 18,330 visitors per version, many months of the warung's traffic, so small refinements of this page are better judged by a usability test than by an A/B test.

> **WARNING.** Do not peek and stop. If you check the p-value every day and stop the first time it drops below 0.05, you will declare far more "winners" than the 5% error rate promises, because random fluctuations cross the line by chance along the way. Fix the duration or the sample size in advance, and analyze once.

7.11 Evaluation in 2026: Remote, Unmoderated and AI-Assisted

In 2026 much usability testing is *remote*: the facilitator and participant meet by video call, or the participant does the tasks alone in an *unmoderated* test in which a platform records the screen and voice and asks the questionnaire at the end. Remote testing reaches people outside the city and saves travel; unmoderated testing reaches many participants cheaply. Both lose something: the facilitator cannot see the participant's hands, and nobody can ask "what were you expecting there?" at the right moment.

Research platforms now offer AI features on top: automatic transcripts, highlights of moments where a participant hesitated, summaries of sessions, and even AI moderators that ask follow-up questions. The advice of Section 5.10 applies: use AI to find moments to watch, not to replace watching them; check every AI summary against the recording; and treat recordings of faces, voices and screens as personal data under UU PDP, with consent that names any third-party service that will process them. An AI summary that "users found the page easy" would have hidden P3 and P6 exactly as the average SUS score did.

7.12 Using It for Real

The methods of this chapter scale down to the smallest business and up to national services. A one-page evaluation plan:

1. **Purpose.** Formative (find problems) or summative (measure against a target)? Name the decision.
2. **Inspection first.** Two or three people run a heuristic evaluation independently and rate severity with **Table 7.1**. Fix the catastrophes before users see them.
3. **Participants.** Five to eight per round and per distinct user group; include the edges (older users, low vision, first-time users, slow connections).

4. **Tasks.** Three to six realistic goals in users' words, each with a clear end and a success criterion.
5. **Measures.** Completion with adjusted-Wald intervals, geometric mean times, errors, SUS (or UMUX-Lite when time is short), and the problem list with severities.
6. **Report distributions.** Plot every participant's score; look for groups left behind.
7. **Iterate.** Fix, then test again with new participants.
8. **Experiments only for big-traffic questions.** Use an A/B test when many people use the real product and the question is "which version performs better on a metric"; compute the sample size first with Equation (7.7).

A puskesmas kiosk (Chapter 6) is evaluated with the same tasks on the real kiosk, with older patients over-represented, and the log of screen views gives times without a stopwatch. **An ojek app** is tested with drivers on their own phones, outdoors, because sunlight and one-handed use are part of the context. **A bank's ATM** redesign is evaluated with a heuristic review against accessibility guidelines, a test with blind and older customers using the audio jack, and completion rates per transaction type from the machine logs.

7.13 HCI in Practice: Profitina

A score that must be understood. Profitina's central screen shows a brand's AI-visibility score and explains each signal behind it. A formative test of such a screen uses tasks like those of Section 7.4: "Find out why your score dropped this week", "Find the one change that would help most". Completion and think-aloud show whether a non-technical owner can move from number to action; a satisfaction questionnaire alone would not.

A release gate built on tests. Profitina releases only when its full automated suite, well over a thousand tests including JavaScript UI guards, is green, and a staged release reports success only after the deployed version answers its health check. Automated tests check that the product does what it is built to do; they cannot check whether people understand it. The two kinds of evaluation are complementary: automated tests keep yesterday's usability fixes from breaking, and usability tests find tomorrow's.

Measuring AI answers with users. Profitina's feedback pipeline lets users mark and correct AI responses; each release of the local model is measured before it is served. That is a summative evaluation built into the product, and it follows this chapter's rules: a fixed metric chosen in advance, a comparison against the current version, and a release only when the new one is genuinely better. Profitina advises and the user decides; evaluation is how the team learns whether its advice helps.

7.14 Common Mistakes, and How to Avoid Them

- **Testing with friends and colleagues.** They know too much and are too kind. Recruit real users, including the edges.
- **Tasks that give away the answer.** "Tap Edit to change the order" tests reading. Write goals in users' words.
- **Helping during the test.** Every hint turns a failure into a false success. Stay quiet; turn questions back; note assisted tasks as failures.
- **Reporting "8 out of 8" as 100%.** Give the adjusted-Wald interval (Equation (7.1)); with eight participants it starts around 63%.

- **Averaging skewed times arithmetically.** Use the geometric mean (Equation (7.2)) for small samples.
- **Reading SUS as a percentage.** A SUS of 70 is about average, a C, not "70% usable". Compare with benchmarks, and plot individual scores.
- **Scoring SUS without reversing the negative items.** Items 2, 4, 6, 8 and 10 count $5 - a$, not $a - 1$ (Equation (7.3)).
- **"Five users are always enough."** Five users of one kind find most problems of that kind. Test each distinct group, and test again after fixing.
- **Peeking at an A/B test and stopping early.** Fix the sample size in advance with Equation (7.7), then analyze once.
- **Declaring a significant result precise.** A p-value below 0.05 says the difference is probably real, not how large it is. Report the confidence interval.
- **Letting AI summaries replace watching.** Summaries flatten exactly the minority experiences that matter.

7.15 Chapter Summary

- Usability is effectiveness, efficiency and satisfaction for specified users, goals and context [1]. Evaluation is formative (find problems) or summative (measure), by inspection or with users.
- Heuristic evaluation by a few independent evaluators finds many problems cheaply; severity ratings from 0 to 4 (**Table 7.1**) set priorities.
- A think-aloud usability test with realistic tasks shows where and why users struggle. The facilitator mostly stays quiet.
- Report completion with the adjusted-Wald interval (Equation (7.1)), times with the geometric mean (Equation (7.2)), and errors per task. In the warung test, changing an order succeeded for only 4 of 8 participants (interval 21.5% to 78.5%).
- SUS (Equation (7.3)) and UMUX-Lite (Equation (7.4)) measure perceived usability; 68 is the average product. The warung page scored 70.3 (C), but the two older participants scored in the F band (**Figure 7.1**).
- The problem-discovery model (Equation (7.5), **Figure 7.3**) explains why five to eight participants per round find most problems of the group tested, and why distinct groups and rare problems need more.
- An A/B test compares two versions on many real users; the two-proportion z-test (Equation (7.6)) and a sample size fixed in advance (Equation (7.7)) make it trustworthy. The new button label lifted orders from 12.98% to 16.68% ($p = 0.011$), with a wide interval of +0.9 to +6.6 points.

7.16 Exercises

1. Compute by hand the SUS score of P3, whose answers were 3, 4, 2, 4, 3, 3, 3, 4, 2, 3, and check it against the program's output.
2. Compute by hand the adjusted-Wald 95% interval for 4 successes out of 5, and compare it with the simple interval $\hat{p} \pm 1.96\sqrt{\hat{p}(1-\hat{p})/n}$. Which one makes sense, and why?
3. For T2, the successful times were 95, 62, 77, 58, 140, 81 and 90 seconds. Compute the arithmetic mean and the geometric mean, and explain the difference.

4. Using Equation (7.5), how many participants are needed to find, on average, at least 90% of the problems when $p = 0.31$? And when $p = 0.15$?
5. Two classmates rated the nine problems of `problems.csv` independently. Where their ratings differ by 2 or more, what should they do before averaging, and why?
6. A second A/B test compared two layouts of the menu page: A 96 orders from 800 visitors, B 108 from 790. Write the file `ab2.csv`, run `ab_test` on it, and interpret the result, including the sample size lines.
7. Extend `usability_metrics` (in Python or Java) to print the SUS mean separately for participants aged 60 and over (P3 and P6) and for the others, and discuss what can and cannot be concluded from two participants.
8. The warung wants a single question after each order to track satisfaction month by month. Would you use SUS, UMUX-Lite or something else, and why? What must stay fixed from month to month for the numbers to be comparable?

7.17 Solutions to the Exercises

1. Odd items (1, 3, 5, 7, 9) with answers 3, 2, 3, 3, 2 contribute $2 + 1 + 2 + 2 + 1 = 8$. Even items (2, 4, 6, 8, 10) with answers 4, 4, 3, 4, 3 contribute $1 + 1 + 2 + 1 + 2 = 7$. Total 15; SUS $= 2.5 \times 15 = 37.5$, as the program prints.

2. Adjusted Wald: $\hat{p}_{\text{adj}} = (4 + 1.921)/(5 + 3.842) = 5.921/8.842 = 0.670$; half-width $1.96 \times \sqrt{0.670 \times 0.330/8.842} = 1.96 \times 0.1581 = 0.310$; interval [0.360,0.980], that is 36.0% to 98.0%. Simple: $\hat{p} = 0.8$, half-width $1.96 \times \sqrt{0.8 \times 0.2/5} = 1.96 \times 0.179 = 0.351$; interval [0.449,1.151], which exceeds 100% and is too narrow at the bottom. The simple interval relies on a normal approximation that fails for small n and rates near 0 or 1; the adjusted interval corrects this, which is why it is recommended for usability tests.

3. Arithmetic mean $603/7 = 86.1$ seconds. Logarithms: 4.554, 4.127, 4.344, 4.060, 4.942, 4.394, 4.500; sum 30.921; mean 4.4173; $e^{4.4173} = 82.9$ seconds, as the program prints. The arithmetic mean is higher because the single slow time of 140 seconds (P6, who found the pickup list confusing) pulls it up; the geometric mean is less affected by one extreme value and is a better estimate of the typical time for small skewed samples.

4. Solve $1 - (1 - p)^n \geq 0.90$, that is $n \geq \ln 0.10/\ln(1 - p)$. For $p = 0.31$: $-2.3026/-0.3711 = 6.2$, so 7 participants. For $p = 0.15$: $-2.3026/-0.1625 = 14.2$, so 15 participants.

5. They should discuss each such problem, returning to the evidence (who met it, how often, whether they could overcome it) and then rate again independently. A difference of two points usually means the evaluators understood the problem differently, for instance one thought it blocked the task and the other did not; averaging “4” and “2” into “3” would hide a disagreement about the facts rather than resolve it.

6. `ab2.csv`:

```
variant,label,visitors,orders
A,Menu list,800,96
B,Menu cards,790,108
```

The program prints rates of 12.00% and 13.67%, a difference of +1.67 points with a 95% interval of −1.62 to +4.96, $z = 0.996$ and $p = 0.32$: not significant. The sample-size lines are the same as before (they depend only on 13% and the target), and show that about 2,162 visitors per version would be needed even for a 3-point lift. The honest conclusion is “no evidence of a difference with

this sample", not "the layouts perform the same". If the layout matters, run the test longer, until the planned sample is reached, or judge it with a usability test instead.

7. In Python, after computing `scores`:

```
older = [s for r, s in zip(rows(folder + "/sus.csv"), scores)
         if r["participant"] in ("P3", "P6")]
others = [s for r, s in zip(rows(folder + "/sus.csv"), scores)
          if r["participant"] not in ("P3", "P6")]
print(f"SUS aged 60+: {sum(older) / len(older):.1f} (n={len(older)}); "
      f"others: {sum(others) / len(others):.1f} (n={len(others)})")
```

This prints 38.8 for the two older participants and 80.8 for the six others. With two participants, no confidence interval is meaningful and no percentage of older customers can be estimated. What *can* be concluded is qualitative and still important: the design failed for both older participants tested, for identifiable reasons (hidden change link, long time list, small QRIS code). The right next step is a round with five or more older participants after the fixes.

8. UMUX-Lite is the natural choice: two items, validated, comparable with SUS through Equation (7.4). SUS is too long for a post-order pop-up and would be skipped. A single "how easy was it?" question is possible, but it is not comparable with published benchmarks. To compare month by month, keep fixed the wording and language of the items, the scale (1 to 7), the moment of asking (right after the order confirmation) and who is asked (every n-th order, or everyone), and report the number of answers with each month's mean.

References

[1] International Organization for Standardization, *ISO 9241-11:2018 Ergonomics of Human-System Interaction — Part 11: Usability: Definitions and Concepts*. Geneva, Switzerland: ISO, 2018.

[2] J. Nielsen and R. Molich, "Heuristic evaluation of user interfaces," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '90)*, Seattle, WA, USA, 1990, pp. 249–256.

[3] J. Nielsen, "Severity Ratings for Usability Problems," Nielsen Norman Group, 1994, updated. [Online]. Available: https://www.nngroup.com/articles/how-to-rate-the-severity-of-usability-problems/

[4] J. Rubin and D. Chisnell, *Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests*, 2nd ed. Indianapolis, IN, USA: Wiley, 2008.

[5] K. A. Ericsson and H. A. Simon, *Protocol Analysis: Verbal Reports as Data*, rev. ed. Cambridge, MA, USA: MIT Press, 1993.

[6] J. Sauro and J. R. Lewis, *Quantifying the User Experience: Practical Statistics for User Research*, 2nd ed. Cambridge, MA, USA: Morgan Kaufmann, 2016.

[7] J. Brooke, "SUS: A 'quick and dirty' usability scale," in *Usability Evaluation in Industry*, P. W. Jordan, B. Thomas, B. A. Weerdmeester, and I. L. McClelland, Eds. London, U.K.: Taylor & Francis, 1996, pp. 189–194.

[8] J. Sauro, "Measuring Usability with the System Usability Scale (SUS)," MeasuringU, Feb. 2, 2011. [Online]. Available: https://measuringu.com/sus/

[9] J. R. Lewis, B. S. Utesch, and D. E. Maher, "UMUX-LITE: When there's no time for the SUS," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*, Paris, France, 2013, pp. 2099–2102.

[10] J. Sauro, "Measuring Usability: From the SUS to the UMUX-Lite," MeasuringU, Oct. 11, 2017. [Online]. Available: https://measuringu.com/umux-lite/

[11] J. Nielsen and T. K. Landauer, "A mathematical model of the finding of usability problems," in *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, Amsterdam, The Netherlands, 1993, pp. 206–213.

[12] J. Nielsen, "Why You Only Need to Test with 5 Users," Nielsen Norman Group, Mar. 18, 2000. [Online]. Available: https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/

[13] J. R. Lewis, "Evaluation of procedures for adjusting problem-discovery rates estimated from small samples," *International Journal of Human-Computer Interaction*, vol. 13, no. 4, pp. 445–479, 2001.

[14] R. Kohavi, D. Tang, and Y. Xu, *Trustworthy Online Controlled Experiments: A Practical Guide to A/B Testing*. Cambridge, U.K.: Cambridge University Press, 2020.

[15] M. Abramowitz and I. A. Stegun, Eds., *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Washington, DC, USA: National Bureau of Standards, 1964.

Chapter 8

Visual Design, Design Systems and Information Architecture

People do not read screens; they scan them, looking for groups, for what is most important and for the word that matches what they want. Visual design decides what they see first and what belongs together; information architecture decides where things live and what they are called; a design system makes those decisions once and reuses them everywhere. By the end of this chapter you will be able to use the Gestalt principles and visual hierarchy to make structure visible; choose type sizes from a modular scale and lay out screens on a spacing grid; compute the WCAG 2.2 contrast ratio from a color's hex code by hand and with programs in Java and Python that read a W3C Design Tokens file and suggest a compliant color when a pair fails; explain what a design system is and what the design languages of 2026 (Material 3 Expressive, Apple's Liquid Glass and Microsoft's Fluent 2) mean for a small team; and run and analyze an open card sort, with co-occurrence similarities and average-linkage clustering, to propose a menu that matches how customers think.

8.1 A Rural Bank's App

BPR Tani Makmur is a small rural bank (*bank perkreditan rakyat*) in a regency of East Java. Most of its customers are farmers, traders at the district market, teachers and retired civil servants; many of them are over fifty. Five years ago the bank launched a mobile app built by a contractor. It works, and it is used: customers check balances, transfer money and buy electricity tokens with it. But the branch staff hear the same complaints every week. "The letters are too small." "I can't read the white writing on the green." "Where do I pay the loan installment?" "I pressed something and it sent me to the settings." At the counter, the most common request is for a staff member to "just do it on my phone".

The app's home screen explains much of this. Fifteen icons sit in a five-by-three grid, each the same size, each in a different bright color, most without a label; the balance is in small grey text at the top; the "Transfer" and "Pay" functions sit next to "Change PIN" and "Branch locations" with nothing to show which are everyday tasks and which are rare. Nothing on the screen is wrong in isolation. Together, it offers no structure: every item shouts equally, so nothing is heard.

The bank asked a small team to redesign the app in three steps that this chapter follows. First, make structure visible with grouping, hierarchy, type and color, and check every color pair against the accessibility standard. Second, write the decisions down as a small design system of tokens and components, so that the next contractor cannot undo them. Third, rebuild the menu from how customers themselves group the bank's functions, using a card sort with twenty customers. Along the way, the team found that customers think of a loan installment as a *bill*, not as part of their loan, which is the kind of finding that no designer would have guessed.

8.2 The Idea in Plain Words

Three ideas run through the chapter.

DEFINITION. *Visual design* is the arrangement of type, color, space and imagery so that the structure and priorities of an interface are visible at a glance. *Information architecture* (IA) is the structural design of shared information environments: how content and functions are organized, labeled, navigated and searched [1]. A *design system* is a documented, shared set of design decisions (*design tokens* for values such as colors, sizes and spacing; components such as buttons and cards; and the rules for using them) that lets many people build consistent interfaces.

Visual design answers *what do I see first, and what goes with what?* Information architecture answers *where is it, and what is it called?* A design system answers *how do we make the same decision the same way every time?* A beautiful screen with the wrong structure fails, and so does a sound structure that nobody can read. The team at the bank needed all three.

8.3 Seeing Groups: Gestalt and Visual Hierarchy

8.3.1 The Gestalt principles

Early in the twentieth century, Gestalt psychologists described how people see configurations rather than separate marks: the visual system groups elements before we consciously look at them. A century of later research has refined and extended these principles and confirmed that they reflect how perception actually organizes a scene [2]. Four of them do most of the work in interface design, and **Figure 8.1** shows each.

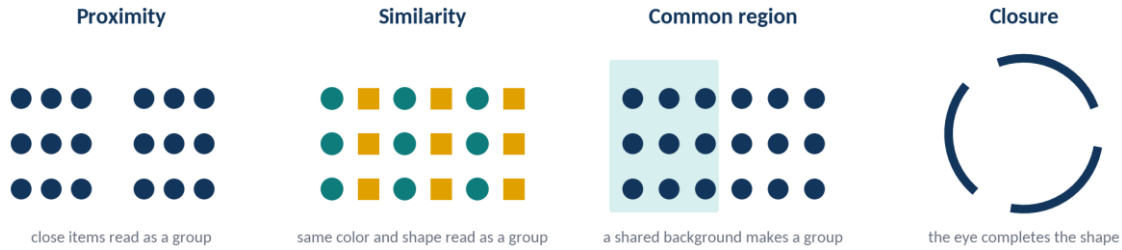


Figure 8.1 Four Gestalt principles used in interface design: proximity, similarity, common region and closure.

- **Proximity.** Elements that are close together are seen as a group. The gap between two groups must be clearly larger than the gaps inside a group; otherwise the eye sees one block.
- **Similarity.** Elements that share color, shape or size are seen as related. This is why all tappable things in an app should look alike, and why a random color for each icon, as in the bank's old home screen, destroys grouping instead of creating it.
- **Common region.** Elements inside a shared boundary or background are seen as a group, even when they are not close. A card with a light background groups its contents more strongly than spacing alone [2].
- **Closure.** The eye completes incomplete shapes. Icons and logos use it; so does a list whose last item is cut off at the bottom of the screen, which tells the user "there is more below".

The redesign of the bank's home screen used proximity and common region together: the balance and the two everyday actions (transfer, pay) sit in one card at the top; less frequent functions sit below in a second card; settings and help move to a separate tab. Similarity does the rest: every tappable element is the same teal, and nothing that is not tappable is teal.

8.3.2 Visual hierarchy

Visual hierarchy is the order in which the eye takes in a screen. It is created by contrast in size, weight, color and position: the largest, darkest, most isolated element is seen first. The rule is that the visual order should match the order of importance for the user's most common task. On the bank's home screen, the most common task is checking the balance, then transferring or paying; so the balance becomes the largest text on the screen, the two main actions become the only filled buttons, and everything else is visually quieter.

A quick test of hierarchy is the *squint test*: look at the screen with half-closed eyes, or blur a screenshot, and see what remains. On the old screen, fifteen equal colored blobs remained. On the new one, a large number, two buttons and a quiet list remained, which is exactly the order of importance.

8.4 Typography and Layout

8.4.1 Type that can be read

Text is most of what an interface shows, so type decisions matter more than any other visual choice. Four rules served the bank's audience:

- **Size.** Body text of at least 16 CSS pixels on phones, larger for primary information. Older eyes need larger text, and the platforms' own text-size settings (Dynamic Type on iOS, font scale on Android) must be respected so that a customer who has enlarged text on the phone gets enlarged text in the app.
- **Line length.** Long lines are hard to follow back to the start; about 45 to 75 characters per line is comfortable for reading text.
- **Weight and case.** Use weight (regular, bold) rather than capitals for emphasis; long runs of capitals are slower to read, because words lose their shape.
- **Numbers.** Show money in the format customers use every day: *Rp1.250.000*, with dots between thousands, not *1250000* or *IDR 1,250,000*. For a bank, numbers are the most important text on the screen.

8.4.2 A modular type scale

Choosing sizes one by one leads to many sizes that differ by one or two pixels, which the eye sees as mistakes rather than as levels. A *modular scale* chooses a base size b and a ratio r and uses only the sizes

$$s_k = b \cdot r^k, \quad k = \dots, -1, 0, 1, 2, \dots \tag{8.1}$$

With $b = 16$ px and $r = 1.25$ (a ratio known in music as a major third), the sizes are $16 \times 1.25^{-1} = 12.8$, 16, 20, 25, 31.25 and 39.06 px. Rounded, the bank used 13 for captions, 16 for body text, 20 for list titles, 25 for section headings and 39 for the balance. Each step is 25% larger than the one below, enough to be seen as a different level.

8.4.3 Space and grids

Space is how proximity is expressed. The bank's design adopted the common practice of a spacing scale built on multiples of 4 or 8 pixels (4, 8, 12, 16, 24, 32, 48), so that every gap is a deliberate choice from a short list. Inside a group, elements are 8 px apart; between groups, 24 px; around the edge of the screen, 16 px. A layout *grid* (on a phone, usually four columns with 16 px margins

and gutters) aligns elements into columns, which makes a screen look ordered and lets the eye move down a column instead of zigzagging.

Touch targets need space too. WCAG 2.2 requires a target of at least 24 by 24 CSS pixels, or enough spacing around a smaller one (success criterion 2.5.8), and platform guidelines recommend larger targets (about 44 to 48 points or density-independent pixels) for primary controls [3]. Chapter 10 returns to touch and thumbs.

8.5 Color and Contrast

8.5.1 Why contrast is measured, not judged

Text must stand out from its background enough to be read by people with low vision, in bright sunlight and on cheap screens. “Enough” cannot be judged by eye on the designer’s good monitor, so the Web Content Accessibility Guidelines define it as a number. WCAG 2.2, a W3C Recommendation since October 2023, requires a contrast ratio of at least 4.5:1 for normal text and 3:1 for large text (at least 18 point, or 14 point bold) at level AA (success criterion 1.4.3), and at least 3:1 for the visual boundaries of controls, icons and focus indicators that are needed to understand them (success criterion 1.4.11) [3]. In October 2025 WCAG 2.2 was also approved as the international standard ISO/IEC 40500:2025, which strengthens its standing in procurement and law [4]. Its successor, WCAG 3.0, is still a Working Draft; the W3C published a new draft for review in September 2026 [5], and it is not yet a standard anyone must meet. Laws in force today, including the European Accessibility Act applied since June 2025 and Indonesian accessibility rules discussed in Chapter 11, point to WCAG 2.x.

A second rule is independent of contrast: *color must never be the only way of conveying information* (success criterion 1.4.1) [3]. The bank’s old app showed failed transactions in red and successful ones in green, with no text or icon; red-green color vision deficiency is common enough, several percent of men in many populations, that some customers could not tell the two apart. The new design writes “Berhasil” or “Gagal” beside the color and adds an icon.

8.5.2 Relative luminance and the contrast ratio

The contrast ratio compares the *relative luminance* of two colors, their brightness as perceived, on a scale from 0 (black) to 1 (white). A screen color is given as three sRGB channels, red, green and blue, each from 0 to 255 (in hex, `#RRGGBB`). The computation has three steps [3].

First, each channel value c_8 is divided by 255 and converted from the sRGB encoding, which stores brightness non-linearly, to linear light:

$$c = \frac{c_8}{255}, \qquad C = \begin{cases} c/12.92 & \text{if } c \le 0.04045 \\ \left(\frac{c+0.055}{1.055}\right)^{2.4} & \text{otherwise} \end{cases} \tag{8.2}$$

Second, the three linear channels R, G, B are weighted by how strongly the eye responds to each; green counts most:

$$L = 0.2126\,R + 0.7152\,G + 0.0722\,B \tag{8.3}$$

Third, if L_1 is the luminance of the lighter color and L_2 that of the darker, the contrast ratio is

$$\mathrm{CR} = \frac{L_1 + 0.05}{L_2 + 0.05} \tag{8.4}$$

The constant 0.05 models the light reflected from a screen in a normal room; it keeps the ratio finite for pure black and sets the scale from 1:1 (identical colors) to 21:1 (black on white, $(1 + 0.05)/(0 + 0.05) = 21$).

Worked by hand for the bank's navy text `#12355B` on white. The channels are $12_{16} = 18$, $35_{16} = 53$ and $5\mathrm{B}_{16} = 91$. Divided by 255: 0.0706, 0.2078, 0.3569, all above 0.04045, so each uses the power formula of Equation (8.2): $R = (0.1256/1.055)^{2.4} = 0.1190^{2.4} = 0.00605$, $G = 0.2493^{2.4} = 0.0356$, $B = 0.3904^{2.4} = 0.1046$. By Equation (8.3), $L = 0.2126 \times 0.00605 + 0.7152 \times 0.0356 + 0.0722 \times 0.1046 = 0.0013 + 0.0255 + 0.0076 = 0.0343$. White has $L = 1$. By Equation (8.4), $\mathrm{CR} = 1.05/0.0843 = 12.46$, far above 4.5.

For white text on the house amber `#E0A100`, the amber's luminance is 0.4134, and $\mathrm{CR} = 1.05/0.4634 = 2.27$: below even the 3:1 for large text. Amber is a good color for a highlight or an icon on navy; it is not a background for white text.

8.6 Building the Contrast Checker

8.6.1 Colors as design tokens

The team kept the app's colors in one file in the format of the W3C Design Tokens Community Group, whose Design Tokens Format Module reached its first stable version, 2025.10, on October 28, 2025 [6]. A token file is JSON; each token has a `$value` and a `$type`, and tokens can be grouped. In the 2025.10 format a color value is an object that names its color space and components, with an optional `hex` for sRGB colors. The start of `data/tokens.json`:

```
{
  "color": {
    "$type": "color",
    "$description": "BPR Tani Makmur app palette",
    "navy": {
      "$value": {
        "colorSpace": "srgb",
        "components": [0.0706, 0.2078, 0.3569],
        "hex": "#12355B"
      }
    },
```

The file holds the new palette plus two colors of the old app, `green-old` and `sky-old`, so that old and new can be checked side by side. A second file, `data/pairs.csv`, lists the pairs that are actually used on screen, with their use: `text` for body text, `large` for large text, `ui` for icons and borders.

8.6.2 The program in Python

Listing 8.1 reads the tokens, computes Equation (8.2), Equation (8.3) and Equation (8.4) for every pair, compares the result with the minimum for its use, and, for each failing pair, suggests a fix: it darkens the darker of the two colors in steps of 1% (mixing it with black) until the pair reaches the minimum.

Listing 8.1 `contrast.py`: checks token color pairs against WCAG 2.2 contrast minimums and suggests a compliant color.

```
"""Check color pairs of a design-token file against WCAG 2.2 contrast
minimums and suggest a darker foreground where a pair fails.
Usage: python contrast.py [tokens.json] [pairs.csv]"""
import csv
import json
import sys

# minimum ratio for AA by use: body text, large text, UI parts and icons
MINIMUM = {"text": 4.5, "large": 3.0, "ui": 3.0}


def load_tokens(path):
    """Return {token name: '#RRGGBB'} from a Design Tokens (2025.10) file."""
    with open(path, encoding="utf-8") as f:
        group = json.load(f)["color"]
    return {name: t["$value"]["hex"].upper() for name, t in group.items()
            if not name.startswith("$")}


def channel(c8):
    """sRGB channel 0-255 to linear light."""
    c = c8 / 255
    return c / 12.92 if c <= 0.04045 else ((c + 0.055) / 1.055) ** 2.4


def luminance(hex_color):
    r, g, b = (int(hex_color[i:i + 2], 16) for i in (1, 3, 5))
    return 0.2126 * channel(r) + 0.7152 * channel(g) + 0.0722 * channel(b)


def ratio(fg, bg):
    l1, l2 = sorted((luminance(fg), luminance(bg)), reverse=True)
    return (l1 + 0.05) / (l2 + 0.05)


def darken(hex_color, share):
    """Mix a color with black: share 0 keeps it, 1 gives black."""
    r, g, b = (int(hex_color[i:i + 2], 16) for i in (1, 3, 5))
    return "#{:02X}{:02X}{:02X}".format(
        *(round(c * (1 - share)) for c in (r, g, b)))
```

Listing 8.1 (Continued) *contrast.py: checks token color pairs against WCAG 2.2 contrast minimums and suggests a compliant color.*

```
def fix(fg, bg, target):
    """Darken the darker color of the pair in 1% steps until the pair
    reaches the target; return (which one, new color)."""
    which = "foreground" if luminance(fg) < luminance(bg) else "background"
    dark, light = (fg, bg) if which == "foreground" else (bg, fg)
    for step in range(101):
        candidate = darken(dark, step / 100)
        if ratio(candidate, light) >= target:
            return which, candidate
    return which, None


def main(args):
    tokens = load_tokens(args[0] if args else "../data/tokens.json")
    pairs_path = args[1] if len(args) > 1 else "../data/pairs.csv"
    with open(pairs_path, newline="", encoding="utf-8") as f:
        pairs = list(csv.DictReader(f))
    print(f"{len(tokens)} color tokens, {len(pairs)} pairs checked")
    print(f"{'foreground':<11}{'background':<12}{'use':<6}{'ratio':>7}  result")
    failures = 0
    for p in pairs:
        fg, bg = tokens[p["foreground"]], tokens[p["background"]]
        r, need = ratio(fg, bg), MINIMUM[p["use"]]
        verdict = "AA pass" if r >= need else f"FAIL (needs {need}:1)"
        print(f"{p['foreground']:<11}{p['background']:<12}{p['use']:<6}"
              f"{r:>6.2f}:1  {verdict}")
        if r < need:
            failures += 1
            which, better = fix(fg, bg, need)
            old = fg if which == "foreground" else bg
            other = bg if which == "foreground" else fg
            print(f"{'':<29}try {which} {better} instead of {old}: "
                  f"{ratio(better, other):.2f}:1")
    print(f"{failures} pair(s) fail WCAG 2.2 AA")


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

Darkening the *darker* color of the pair is the simplest fix that always moves the ratio in the right direction: the lighter color stays, the darker gets darker, and the difference in luminance grows. For white text on amber, that means a darker amber background; for grey text on light grey, a darker grey text. A designer may prefer another fix, such as a different hue, but the suggestion shows how far the color must move.

8.6.3 The program in Java

Listing 8.2 is the Java version. The JDK has no JSON parser, so `loadTokens` finds each token with a regular expression that matches a name followed by a `$value` object containing a `hex` field. That is enough for a flat file of the shape above; a design system with nested groups, aliases and other color spaces should be read with a real JSON library.

Listing 8.2 `Contrast.java`: the same checker in Java.

```java
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Locale;
import java.util.Map;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/** Check color pairs of a design-token file against WCAG 2.2 contrast
 *  minimums and suggest a darker color where a pair fails.
 *  Usage: java Contrast [tokens.json] [pairs.csv] */
public class Contrast {
    static final Map<String, Double> MINIMUM =
        Map.of("text", 4.5, "large", 3.0, "ui", 3.0);

    /** Reads {name: "#RRGGBB"} from the flat color group of a Design Tokens
     *  (2025.10) file. A regular expression is enough for this file shape;
     *  use a JSON library for anything more general. */
    static Map<String, String> loadTokens(String path) throws IOException {
        String json = Files.readString(Path.of(path));
        Pattern p = Pattern.compile(
            "\"([\\w-]+)\"\\s*:\\s*\\{\\s*\"\\$value\"[^}]*?"
            + "\"hex\"\\s*:\\s*\"(#[0-9A-Fa-f]{6})\"", Pattern.DOTALL);
        Map<String, String> tokens = new LinkedHashMap<>();
        Matcher m = p.matcher(json);
        while (m.find()) tokens.put(m.group(1), m.group(2).toUpperCase());
        return tokens;
    }

    /** sRGB channel 0-255 to linear light. */
    static double channel(int c8) {
        double c = c8 / 255.0;
        return c <= 0.04045 ? c / 12.92 : Math.pow((c + 0.055) / 1.055, 2.4);
    }

    static int part(String hex, int i) {
        return Integer.parseInt(hex.substring(1 + 2 * i, 3 + 2 * i), 16);
    }

    static double luminance(String hex) {
        return 0.2126 * channel(part(hex, 0)) + 0.7152 * channel(part(hex, 1))
            + 0.0722 * channel(part(hex, 2));
    }
```

Listing 8.2 (Continued) *Contrast.java: the same checker in Java.*

```
    static double ratio(String fg, String bg) {
        double a = luminance(fg), b = luminance(bg);
        return (Math.max(a, b) + 0.05) / (Math.min(a, b) + 0.05);
    }

    /** Mix a color with black: share 0 keeps it, 1 gives black. */
    static String darken(String hex, double share) {
        int[] c = new int[3];
        for (int i = 0; i < 3; i++)
            c[i] = (int) Math.round(part(hex, i) * (1 - share));
        return String.format("#%02X%02X%02X", c[0], c[1], c[2]);
    }

    /** Darken the darker color of the pair in 1% steps until the pair
     *  reaches the target; returns {which one, new color}. */
    static String[] fix(String fg, String bg, double target) {
        boolean fgDarker = luminance(fg) < luminance(bg);
        String dark = fgDarker ? fg : bg, light = fgDarker ? bg : fg;
        String which = fgDarker ? "foreground" : "background";
        for (int step = 0; step <= 100; step++) {
            String candidate = darken(dark, step / 100.0);
            if (ratio(candidate, light) >= target)
                return new String[] {which, candidate};
        }
        return new String[] {which, null};
    }

    public static void main(String[] args) throws IOException {
        Locale.setDefault(Locale.US);
        Map<String, String> tokens =
            loadTokens(args.length > 0 ? args[0] : "../data/tokens.json");
        List<String> lines = Files.readAllLines(
            Path.of(args.length > 1 ? args[1] : "../data/pairs.csv"));
        List<String> pairs = lines.subList(1, lines.size());
        System.out.println(tokens.size() + " color tokens, " + pairs.size()
            + " pairs checked");
        System.out.printf("%-11s%-12s%-6s%7s  result%n", "foreground",
            "background", "use", "ratio");
        int failures = 0;
        for (String line : pairs) {
            String[] f = line.split(",");
            String fg = tokens.get(f[0]), bg = tokens.get(f[1]);
            double r = ratio(fg, bg), need = MINIMUM.get(f[2]);
            String verdict = r >= need ? "AA pass" : "FAIL (needs " + need + ":1)";
            System.out.printf("%-11s%-12s%-6s%6.2f:1  %s%n", f[0], f[1], f[2], r,
                verdict);
```

Listing 8.2 (Continued) *Contrast.java: the same checker in Java.*

```
            if (r < need) {
                failures++;
                String[] fx = fix(fg, bg, need);
                boolean isFg = fx[0].equals("foreground");
                System.out.printf("%-29stry %s %s instead of %s: %.2f:1%n", "",
                    fx[0], fx[1], isFg ? fg : bg, ratio(fx[1], isFg ? bg : fg));
            }
        }
        System.out.println(failures + " pair(s) fail WCAG 2.2 AA");
    }
}
```

8.6.4 Running it

`python3 contrast.py` (or `java -cp out Contrast`) prints:

```
10 color tokens, 10 pairs checked
foreground background  use     ratio  result
navy       white       text   12.46:1  AA pass
grey       white       text    4.83:1  AA pass
grey       grey-light  text    4.39:1  FAIL (needs 4.5:1)
                              try foreground #69707D instead of #6B7280: 4.53:1
white      teal        text    5.01:1  AA pass
white      amber       large   2.27:1  FAIL (needs 3.0:1)
                              try background #C18A00 instead of #E0A100: 3.04:1
white      green-old   text    2.78:1  FAIL (needs 4.5:1)
                              try background #3A853D instead of #4CAF50: 4.56:1
white      sky-old     large   1.87:1  FAIL (needs 3.0:1)
                              try background #629CB1 instead of #7EC8E3: 3.04:1
amber      white       ui      2.27:1  FAIL (needs 3.0:1)
                              try foreground #C18A00 instead of #E0A100: 3.04:1
teal       teal-light  text    4.18:1  FAIL (needs 4.5:1)
                              try foreground #0E7676 instead of #0F7C7C: 4.52:1
red        white       text    5.44:1  AA pass
6 pair(s) fail WCAG 2.2 AA
```

Figure 8.2 shows the same pairs as swatches. Three results deserve comment. The old app's white-on-green buttons reach only 2.78:1, which explains "I can't read the white writing on the green"; a green dark enough to pass, `#3A853D`, is suggested. Grey body text passes on white (4.83:1) but *fails* on the light grey card background (4.39:1), a common trap: a color pair is checked, then the same text is later placed on a slightly different background. And the house teal on light teal, a combination that looks calm and readable on a good monitor, fails at 4.18:1; the fix is a barely darker teal, `#0E7676`, that nobody would notice as a change of color.



Figure 8.2 The ten color pairs of the bank app as swatches, with their use, contrast ratio and WCAG 2.2 AA result.

> **WARNING.** Check the pairs that are actually used, including text on cards, on images and on disabled or hover states, not only the main text on the main background. Most contrast failures in real products come from secondary combinations that nobody checked.

8.7 Design Systems and Tokens

8.7.1 From decisions to a system

A design system turns decisions like those above into something that outlives the people who made them. It usually has three layers. *Design tokens* hold the raw values: colors, type sizes, spacing steps, corner radii, durations of animation. *Components* combine tokens into reusable parts: a primary button, a list row, a card, a form field, each with its states (normal, pressed, disabled, focused, error). *Guidelines* say when to use which component and why, with examples of correct and incorrect use.

Tokens themselves come in two levels, as **Figure 8.3** shows. *Primitive* tokens name raw values (`color.teal = #0F7C7C`). *Semantic* tokens name purposes and point to primitives (`button.primary.background = {color.teal}`). Components use only semantic tokens. The indirection pays off when something changes: if a contrast check shows that the teal must be darker, one primitive changes and every component that uses it follows; if the bank adds a high-contrast theme, the semantic tokens point to different primitives in that theme.

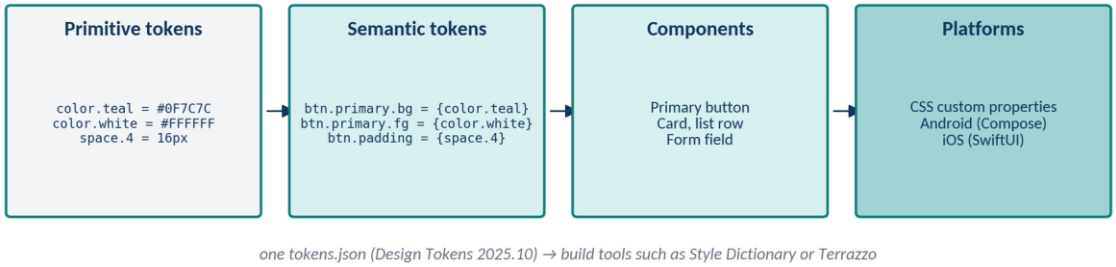


Figure 8.3 Design tokens in a design system: primitive tokens hold values, semantic tokens name purposes, components use semantic tokens, and build tools turn one token file into code for each platform.

The value of the W3C format is that one file can feed every tool. The Design Tokens Community Group announced the 2025.10 specification with support for theming and multiple brands, modern color spaces such as Display P3 and Oklch, aliases between tokens, and generation of code for iOS, Android, the web and Flutter; its reference implementations include Style Dictionary, Tokens Studio and Terrazzo, and design tools including Figma, Penpot, Sketch and Framer were adopting it [6]. For the bank, that means the same `tokens.json` that the contrast checker reads can generate the CSS for the web version and the resource files of the Android app.

8.7.2 The design languages of 2026

The three large platform design systems all changed direction in 2025, and a small team designing for their platforms needs to know how.

Material 3 Expressive. Google introduced Material 3 Expressive in May 2025 as an evolution of its Material Design system for Android and the web, with more varied shapes, bolder type, richer color and physics-based motion. Google described it as the result of 46 research studies with more than 18,000 participants; in eye-tracking studies across ten apps, participants spotted key interface elements up to four times faster in the expressive designs, and older participants found key interactive elements as quickly as younger ones [7]. Those are Google's own findings about its own system, and they are worth reading with that in mind; they are also a reminder that "expressive" need not mean "less usable" when size, contrast and hierarchy are designed on purpose.

Liquid Glass. Apple announced Liquid Glass on June 9, 2025 as a new software material for iOS 26, iPadOS 26, macOS Tahoe 26, watchOS 26 and tvOS 26: a translucent material that reflects and refracts what lies behind it and adapts its color to the content, used for controls, tab bars, sidebars and system surfaces [8]. Translucency is exactly the situation in which contrast can no longer be checked once, because the background behind a control changes as the user scrolls. After criticism of legibility in the early releases, iOS 26.1 added a setting in Display & Brightness to choose between the original *Clear* appearance and a *Tinted* one that increases opacity and contrast, alongside the existing Reduce Transparency accessibility option [9]. The lesson for any designer is general: when text or icons sit on a changing background, guarantee contrast with a backing layer or tint, and test on real content, not on a plain background.

Fluent 2. Microsoft's Fluent 2 design system covers Windows, the web, iOS and Android apps such as those of Microsoft 365, and publishes its tokens, components and accessibility guidance openly [10]. For a team building a business app that must feel at home next to Microsoft's tools, it is a ready set of decisions.

8.7.3 What a small team should do

A rural bank does not need its own Material. It needs a *small* system: a token file (perhaps twenty colors, a type scale, a spacing scale, three radii), a dozen components with their states, and one page of rules, all built on the platform's own conventions so that customers find things where their phone has taught them to look. The team at BPR Tani Makmur kept the platform's navigation patterns and controls, applied its own tokens for color, type and spacing, and wrote a one-page guide whose first rule was "every text and icon pair passes `contrast.py` before it ships".

8.8 Information Architecture

8.8.1 Organizing, labeling, navigating, searching

Rosenfeld, Morville and Arango divide information architecture into four systems [1]:

- **Organization systems** decide how content is grouped: by task ("pay", "send"), by topic ("savings", "loans"), by audience ("for farmers", "for teachers"), alphabetically or by time. Most apps mix schemes, and most confusion comes from mixing them without noticing.
- **Labeling systems** decide what the groups and items are called. A label is a promise about what is behind it; "Layanan" ("Services") promises nothing specific, "Bayar tagihan" ("Pay bills") promises something a customer can recognize.
- **Navigation systems** decide how people move: tabs at the bottom of a phone screen for the three to five most important destinations, lists and cards within them, and a clear way back.
- **Search systems** let people who know what they want skip the navigation, and they fail when the search words are the designers' rather than the users'.

The bank's old app had one flat organization (fifteen icons), labels in the bank's internal vocabulary ("Mutasi", "Inquiry", "Setor Angsuran") and no search. The design team could have invented a new structure; instead, it asked customers.

8.8.2 Card sorting and tree testing

In a *card sort*, participants group cards, each naming one item of content or one function, into piles that make sense to them. In an *open* card sort they also name the piles; in a *closed* sort they place cards into given categories. Open sorts discover how people think about a domain and what

words they use; closed sorts check a proposed structure [11]. Card sorts can be done with paper cards at the branch or with an online tool; fifteen to thirty participants of one audience usually give a stable picture.

The reverse test is a *tree test*: participants see only the text of a proposed menu hierarchy, without any visual design, and are asked where they would look for something ("Where would you pay your loan installment?"). The share of participants who reach the right place, and the paths they take, show whether the structure works before any screen is designed.

The bank's team ran an open card sort with twenty customers at two branches, with the fifteen functions of the app on paper cards. Each participant sorted the cards into piles and wrote a name on each pile. `data/cardsort.csv` records the result, one line per participant and card, with the name the participant gave the pile:

```
participant,card,group
U01,Transfer to a BPR account,Transfer
U01,Transfer to another bank,Transfer
U01,Scheduled transfer,Bills
```

8.9 Analyzing a Card Sort

8.9.1 Similarity between cards

The first question is simple: how often was each pair of cards put in the same pile? If N participants sorted the cards, and n_{ij} of them put cards i and j in the same pile, the *co-occurrence similarity* of the pair is

$$S_{ij} = \frac{n_{ij}}{N} \tag{8.5}$$

a number from 0 (never together) to 1 (always together). The pile names do not matter for this measure, only whether two cards share a pile; one participant's "Pay" and another's "Bills" are both simply "the same pile as the other bill cards".

8.9.2 Average-linkage clustering

The similarities of fifteen cards form a table of 105 pairs, too many to read. *Hierarchical agglomerative clustering* summarizes them. It starts with every card in its own cluster and repeatedly merges the two most similar clusters, until one cluster contains everything. The similarity of two clusters A and B, each a set of cards, is defined by *average linkage* as the mean similarity over all pairs of cards, one from each:

$$S(A,B) = \frac{1}{|A|\,|B|} \sum_{i \in A} \sum_{j \in B} S_{ij} \tag{8.6}$$

where $|A|$ is the number of cards in A. Average linkage is a compromise between joining clusters as soon as *any* two members are similar (which chains unrelated cards together) and only when *all* members are similar (which is very strict). The record of merges, drawn as a tree, is a *dendrogram*; cutting the tree at a chosen similarity gives groups in which, on average, at least that share of participants agreed.

8.9.3 The program

Listing 8.3 computes Equation (8.5) for all pairs, clusters with Equation (8.6), and cuts the result at a chosen level (0.5 by default: groups on which at least half of the participants agreed, on average).

Listing 8.3 `cardsort.py`: co-occurrence similarities and average-linkage clustering of an open card sort.

```
"""Analyze an open card sort: how often each pair of cards was grouped
together, and average-linkage clustering into a proposed menu.
Usage: python cardsort.py [cardsort.csv] [cut]"""
import csv
import sys


def load(path):
    """Return (cards in first-seen order, {participant: {card: group}})."""
    cards, sorts = [], {}
    with open(path, newline="", encoding="utf-8") as f:
        for r in csv.DictReader(f):
            if r["card"] not in cards:
                cards.append(r["card"])
            sorts.setdefault(r["participant"], {})[r["card"]] = r["group"]
    return cards, sorts


def similarity(cards, sorts):
    """sim[i][j] = share of participants who put cards i and j together."""
    n = len(cards)
    sim = [[0.0] * n for _ in range(n)]
    for groups in sorts.values():
        for i in range(n):
            for j in range(n):
                if groups[cards[i]] == groups[cards[j]]:
                    sim[i][j] += 1 / len(sorts)
    return sim


def link(a, b, sim):
    """Average linkage: mean similarity over all pairs across two clusters."""
    total = 0.0
    for i in a:
        for j in b:
            total += sim[i][j]
    return total / (len(a) * len(b))
```

Listing 8.3 (Continued) *cardsort.py: co-occurrence similarities and average-linkage clustering of an open card sort.*

```
def cluster(sim):
    """Merge the two most similar clusters until one is left.
    Returns the list of merges (similarity, cluster a, cluster b)."""
    clusters = [[i] for i in range(len(sim))]
    merges = []
    while len(clusters) > 1:
        best = None
        for x in range(len(clusters)):
            for y in range(x + 1, len(clusters)):
                s = link(clusters[x], clusters[y], sim)
                if best is None or s > best[0] + 1e-12:
                    best = (s, x, y)
        s, x, y = best
        merges.append((s, clusters[x], clusters[y]))
        clusters = ([c for k, c in enumerate(clusters) if k not in (x, y)]
                    + [clusters[x] + clusters[y]])
    return merges


def cut(n, merges, level):
    """Groups formed by all merges at or above the similarity level."""
    groups = [[i] for i in range(n)]
    for s, a, b in merges:
        if s >= level:
            ga = next(g for g in groups if a[0] in g)
            gb = next(g for g in groups if b[0] in g)
            groups.remove(gb)
            ga.extend(gb)
    return groups


def main(args):
    path = args[0] if args else "../data/cardsort.csv"
    level = float(args[1]) if len(args) > 1 else 0.5
    cards, sorts = load(path)
    sim = similarity(cards, sorts)
    print(f"{len(sorts)} participants sorted {len(cards)} cards")
    pairs = sorted(((sim[i][j], cards[i], cards[j]) for i in range(len(cards))
                    for j in range(i + 1, len(cards))), key=lambda t: -t[0])
    print("Most often together:")
    for s, a, b in pairs[:5]:
        print(f"  {s * 100:3.0f}%  {a} + {b}")
    print("Merges (average linkage), each creating a new group G1, G2, ...:")
    names = {}                          # frozenset of card indexes -> name
```

Listing 8.3 (Continued) *cardsort.py: co-occurrence similarities and average-linkage clustering of an open card sort.*

```
    def label(members):
        return names.get(frozenset(members), cards[members[0]])

    for k, (s, a, b) in enumerate(cluster(sim), 1):
        print(f"  {s:.2f}  G{k} = {label(a)} + {label(b)}")
        names[frozenset(a + b)] = f"G{k}"
    print(f"Proposed groups at similarity >= {level}:")
    for g in cut(len(cards), cluster(sim), level):
        print("  - " + "; ".join(cards[i] for i in g))


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

The clustering is written for clarity, not speed: at each step it compares every pair of clusters with `link`, which is fine for the dozens of cards of a card sort and would be too slow for thousands of items. When two pairs are equally similar, the first one found wins; the small tolerance `1e-12` makes sure that the Python and Java versions, which add the same numbers in the same order, make the same choice.

Listing 8.4 is the Java version. A `record` holds each merge; a map from the set of card indexes to a name such as `G3` lets the report refer to earlier groups.

Listing 8.4 `CardSort.java`: the same analysis in Java.

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Locale;
import java.util.Map;
import java.util.Set;

/** Analyze an open card sort: how often each pair of cards was grouped
 *  together, and average-linkage clustering into a proposed menu.
 *  Usage: java CardSort [cardsort.csv] [cut] */
public class CardSort {
    List<String> cards = new ArrayList<>();
    Map<String, Map<String, String>> sorts = new LinkedHashMap<>();
    double[][] sim;

    record Merge(double sim, List<Integer> a, List<Integer> b) {}

    static CardSort load(String path) throws IOException {
        CardSort cs = new CardSort();
        List<String> lines = Files.readAllLines(Path.of(path));
        for (String line : lines.subList(1, lines.size())) {
            if (line.isBlank()) continue;
            String[] f = line.split(",");
            if (!cs.cards.contains(f[1])) cs.cards.add(f[1]);
            cs.sorts.computeIfAbsent(f[0], k -> new HashMap<>()).put(f[1], f[2]);
        }
        int n = cs.cards.size();
        cs.sim = new double[n][n];
        for (Map<String, String> groups : cs.sorts.values())
            for (int i = 0; i < n; i++)
                for (int j = 0; j < n; j++)
                    if (groups.get(cs.cards.get(i))
                            .equals(groups.get(cs.cards.get(j))))
                        cs.sim[i][j] += 1.0 / cs.sorts.size();
        return cs;
    }
```

Listing 8.4 (Continued) *CardSort.java: the same analysis in Java.*

```
    /** Average linkage: mean similarity over all pairs across two clusters. */
    double link(List<Integer> a, List<Integer> b) {
        double total = 0;
        for (int i : a) for (int j : b) total += sim[i][j];
        return total / (a.size() * b.size());
    }

    List<Merge> cluster() {
        List<List<Integer>> clusters = new ArrayList<>();
        for (int i = 0; i < cards.size(); i++) clusters.add(List.of(i));
        List<Merge> merges = new ArrayList<>();
        while (clusters.size() > 1) {
            double best = -1;
            int bx = -1, by = -1;
            for (int x = 0; x < clusters.size(); x++)
                for (int y = x + 1; y < clusters.size(); y++) {
                    double s = link(clusters.get(x), clusters.get(y));
                    if (bx < 0 || s > best + 1e-12) {
                        best = s;
                        bx = x;
                        by = y;
                    }
                }
            List<Integer> a = clusters.get(bx), b = clusters.get(by);
            merges.add(new Merge(best, a, b));
            List<List<Integer>> next = new ArrayList<>();
            for (int k = 0; k < clusters.size(); k++)
                if (k != bx && k != by) next.add(clusters.get(k));
            List<Integer> joined = new ArrayList<>(a);
            joined.addAll(b);
            next.add(joined);
            clusters = next;
        }
        return merges;
    }
```

Listing 8.4 (Continued) *CardSort.java: the same analysis in Java.*

```
    /** Groups formed by all merges at or above the similarity level. */
    List<List<Integer>> cut(List<Merge> merges, double level) {
        List<List<Integer>> groups = new ArrayList<>();
        for (int i = 0; i < cards.size(); i++)
            groups.add(new ArrayList<>(List.of(i)));
        for (Merge m : merges) {
            if (m.sim() < level) continue;
            List<Integer> ga = null, gb = null;
            for (List<Integer> g : groups) {
                if (g.contains(m.a().get(0))) ga = g;
                if (g.contains(m.b().get(0))) gb = g;
            }
            groups.remove(gb);
            ga.addAll(gb);
        }
        return groups;
    }

    String names(List<Integer> idx) {
        List<String> out = new ArrayList<>();
        for (int i : idx) out.add(cards.get(i));
        return String.join("; ", out);
    }

    public static void main(String[] args) throws IOException {
        Locale.setDefault(Locale.US);
        CardSort cs = load(args.length > 0 ? args[0] : "../data/cardsort.csv");
        double level = args.length > 1 ? Double.parseDouble(args[1]) : 0.5;
        int n = cs.cards.size();
        System.out.println(cs.sorts.size() + " participants sorted " + n
            + " cards");
        List<double[]> pairs = new ArrayList<>();
        for (int i = 0; i < n; i++)
            for (int j = i + 1; j < n; j++)
                pairs.add(new double[] {cs.sim[i][j], i, j});
        pairs.sort((p, q) -> Double.compare(q[0], p[0]));   // stable sort
        System.out.println("Most often together:");
        for (double[] p : pairs.subList(0, 5))
            System.out.printf("  %3.0f%%  %s + %s%n", p[0] * 100,
                cs.cards.get((int) p[1]), cs.cards.get((int) p[2]));
        System.out.println(
            "Merges (average linkage), each creating a new group G1, G2, ...:");
        Map<Set<Integer>, String> labels = new HashMap<>();
        List<Merge> merges = cs.cluster();
        for (int k = 0; k < merges.size(); k++) {
            Merge m = merges.get(k);
```

Listing 8.4 (Continued) *CardSort.java: the same analysis in Java.*

```
            String la = labels.getOrDefault(new HashSet<>(m.a()),
                cs.cards.get(m.a().get(0)));
            String lb = labels.getOrDefault(new HashSet<>(m.b()),
                cs.cards.get(m.b().get(0)));
            System.out.printf("  %.2f  G%d = %s + %s%n", m.sim(), k + 1, la, lb);
            Set<Integer> joined = new HashSet<>(m.a());
            joined.addAll(m.b());
            labels.put(joined, "G" + (k + 1));
        }
        System.out.println("Proposed groups at similarity >= " + level + ":");
        for (List<Integer> g : cs.cut(merges, level))
            System.out.println("  - " + cs.names(g));
    }
}
```

8.9.4 Reading the result

`python3 cardsort.py` (or `java -cp out CardSort`) prints:

```
20 participants sorted 15 cards
Most often together:
  100%  Transfer to a BPR account + Transfer to another bank
  100%  PLN electricity token + PDAM water bill
  100%  PLN electricity token + BPJS Health contribution
  100%  PDAM water bill + BPJS Health contribution
  100%  Savings balance + Apply for a loan
Merges (average linkage), each creating a new group G1, G2, ...:
  1.00  G1 = Transfer to a BPR account + Transfer to another bank
  1.00  G2 = PLN electricity token + PDAM water bill
  1.00  G3 = BPJS Health contribution + G2
  1.00  G4 = Savings balance + Apply for a loan
  1.00  G5 = Branch and ATM locations + Contact customer service
  0.70  G6 = Phone credit + G3
  0.70  G7 = Account statement + G4
  0.65  G8 = Pay by QRIS + G1
  0.55  G9 = Change PIN + G5
  0.53  G10 = Scheduled transfer + G8
  0.51  G11 = Loan installment + G6
  0.22  G12 = G10 + G11
  0.22  G13 = G7 + G9
  0.03  G14 = G12 + G13
Proposed groups at similarity >= 0.5:
  - Scheduled transfer; Pay by QRIS; Transfer to a BPR account; Transfer to another bank
  - Account statement; Savings balance; Apply for a loan
  - Loan installment; Phone credit; BPJS Health contribution; PLN electricity token; PDAM
water bill
  - Change PIN; Branch and ATM locations; Contact customer service
```

Figure 8.4 draws the merges as a dendrogram: the further to the right two cards join, the more participants put them together. Cutting at 0.5 gives four groups.

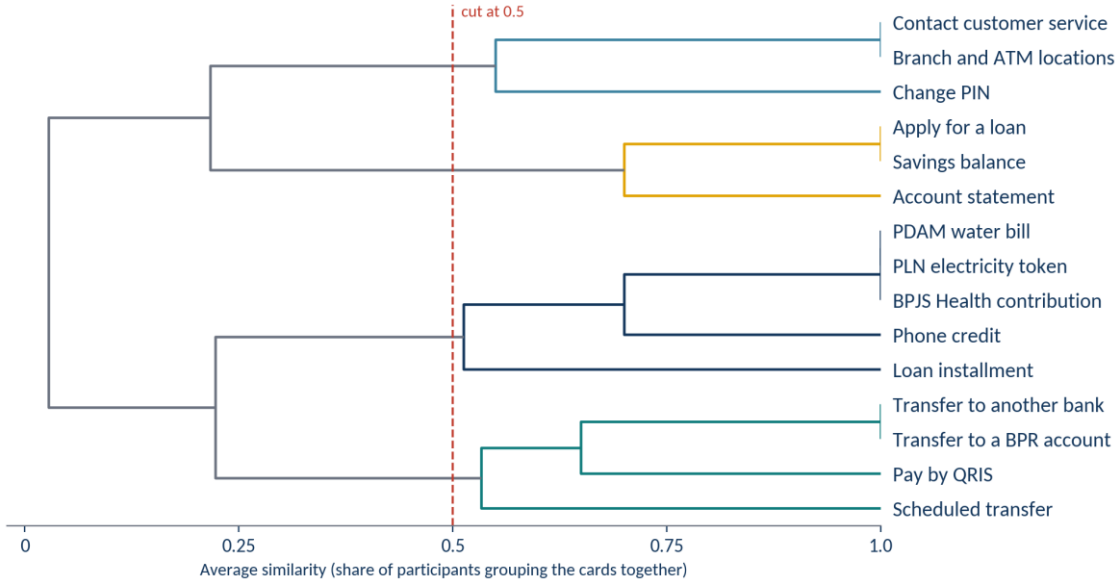


Figure 8.4 Dendrogram of the open card sort (20 participants, 15 cards, average linkage); cutting at a similarity of 0.5 gives four groups, and the loan installment joins the bills.

Three of the groups are what a banker would expect: transfers (including QRIS payment and scheduled transfers), savings and loans (balance, statement, loan application), and help (PIN, locations, customer service). The fourth holds the surprise: *Loan installment* joined the bills (electricity, water, BPJS, phone credit) at a similarity of 0.51, not the loan group. A closer look at

the similarities shows the split: the installment was grouped with the water bill by 55% of participants and with the savings balance by 45%. For many customers, an installment is something you *pay* every month, like electricity; the loan is something you *have*. The bank's old app put "Setor Angsuran" under "Pinjaman", where more than half of the customers would not look for it.

The team drew three design decisions from the sort:

1. The bottom navigation got four tabs matching the four groups, with the participants' own words as labels: *Kirim uang* (Send money), *Bayar* (Pay), *Tabungan & pinjaman* (Savings & loans) and *Bantuan* (Help). The pile names show which words customers used; "Bayar tagihan" and "Bills" were the most common names for the bill pile, and the shorter *Bayar* was chosen to fit a tab.
2. *Loan installment* appears under *Bayar*, and also as a link on the loan's own page, because 45% of customers will look there. Cross-listing an item that participants split between groups is a standard IA remedy.
3. The labels of cards that caused hesitation during sorting were renamed: "Mutasi" became "Riwayat transaksi" (transaction history) in the interface, while "mutasi" was kept as a search keyword because long-time customers know it.

A tree test with twenty new customers then checked the structure: for the task "pay your loan installment", 18 of 20 found it on the first try, most of them under *Bayar*.

> **KEY INSIGHT.** A card sort does not design the menu; it shows how customers think. The clusters propose groups, the pile names propose labels, and the items that participants split between groups show where a cross-link is needed. Designers still decide, and a tree test checks the decision.

8.10 Testing the Programs

Listing 8.5 checks the contrast formulas at their limits (21:1 for black on white in either order, 1:1 for identical colors), the token reader and the navy-on-white ratio computed by hand, that the suggested fix reaches its target, and the card-sort analysis: fourteen merges for fifteen cards, four groups at 0.5 with the installment among the bills, and two groups at 0.2.

Listing 8.5 `test_design.py`: tests of the contrast checker and the card-sort analysis.

```
"""Tests for contrast.py and cardsort.py.
Run: python -m unittest test_design.py"""
import unittest
import cardsort
from contrast import darken, fix, load_tokens, ratio


class DesignTest(unittest.TestCase):
    def test_ratio_limits(self):
        self.assertAlmostEqual(ratio("#000000", "#FFFFFF"), 21.0)
        self.assertAlmostEqual(ratio("#777777", "#777777"), 1.0)
        self.assertAlmostEqual(ratio("#FFFFFF", "#000000"), 21.0)

    def test_tokens_file(self):
        tokens = load_tokens("../data/tokens.json")
        self.assertEqual(tokens["navy"], "#12355B")
        self.assertAlmostEqual(ratio(tokens["navy"], tokens["white"]), 12.46,
                               places=2)

    def test_fix_reaches_target(self):
        which, color = fix("#FFFFFF", "#E0A100", 3.0)
        self.assertEqual(which, "background")
        self.assertGreaterEqual(ratio("#FFFFFF", color), 3.0)
        self.assertEqual(darken("#FFFFFF", 1.0), "#000000")

    def test_card_sort_groups(self):
        cards, sorts = cardsort.load("../data/cardsort.csv")
        sim = cardsort.similarity(cards, sorts)
        merges = cardsort.cluster(sim)
        self.assertEqual(len(merges), len(cards) - 1)
        groups = cardsort.cut(len(cards), merges, 0.5)
        self.assertEqual(len(groups), 4)
        bills = next(g for g in groups
                     if cards.index("PDAM water bill") in g)
        self.assertIn(cards.index("Loan installment"), bills)
        self.assertEqual(len(cardsort.cut(len(cards), merges, 0.2)), 2)


if __name__ == "__main__":
    unittest.main()
```

```
Ran 4 tests in 0.002s

OK
```

Listing 8.6 makes the same checks in Java.

Listing 8.6 `DesignTest.java`: the same checks in Java.

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

/** Tests for Contrast and CardSort, without a test library.
 *  Run: java -cp out DesignTest */
public class DesignTest {
    static int passed = 0;

    static void check(boolean ok, String name) {
        if (!ok) throw new AssertionError("FAILED: " + name);
        passed++;
    }

    public static void main(String[] args) throws IOException {
        check(Math.abs(Contrast.ratio("#000000", "#FFFFFF") - 21.0) < 1e-9, "21:1");
        check(Math.abs(Contrast.ratio("#777777", "#777777") - 1.0) < 1e-9, "1:1");
        Map<String, String> tokens = Contrast.loadTokens("../data/tokens.json");
        check(tokens.get("navy").equals("#12355B"), "token read");
        check(Math.abs(Contrast.ratio(tokens.get("navy"), tokens.get("white"))
            - 12.46) < 0.005, "navy on white");
        String[] fx = Contrast.fix("#FFFFFF", "#E0A100", 3.0);
        check(fx[0].equals("background")
            && Contrast.ratio("#FFFFFF", fx[1]) >= 3.0, "fix");
        check(Contrast.darken("#FFFFFF", 1.0).equals("#000000"), "darken");
        CardSort cs = CardSort.load("../data/cardsort.csv");
        List<CardSort.Merge> merges = cs.cluster();
        check(merges.size() == cs.cards.size() - 1, "merge count");
        List<List<Integer>> groups = cs.cut(merges, 0.5);
        check(groups.size() == 4, "four groups");
        int bill = cs.cards.indexOf("PDAM water bill");
        int loan = cs.cards.indexOf("Loan installment");
        boolean together = false;
        for (List<Integer> g : groups)
            if (g.contains(bill) && g.contains(loan)) together = true;
        check(together, "installment with bills");
        check(cs.cut(merges, 0.2).size() == 2, "two groups at 0.2");
        System.out.println(passed + " checks passed");
    }
}
```

```
10 checks passed
```

8.11 Using It for Real

The same three steps work for any small service.

For visual design: list the three most important things on each screen and make the visual order match; group with proximity and common region; use one color for “tappable”; choose type sizes from a modular scale (Equation (8.1)) and spacing from a 4- or 8-pixel scale; squint.

For color: keep every color in a token file; list every pair actually used; run `contrast.py` on every change; never use color alone for meaning; test translucent or image backgrounds with real content.

For a design system: start small (tokens, a dozen components, one page of rules), build on the platform’s conventions, and give the system an owner, because a system without an owner decays into a folder of old files.

For information architecture: run an open card sort with 15 to 30 real users; analyze it with Equation (8.5) and Equation (8.6); take labels from the participants’ pile names; cross-list items that participants split; check the result with a tree test before designing screens.

Three examples. **A puskesmas website** can card-sort its services with patients in the waiting room; patients will likely group "vaccination", "posyandu" and "KIA" together whatever the organization chart says. **A warung's menu board and ordering page** (Chapter 7) can use the Gestalt principles directly: dishes grouped by type in common regions, prices aligned in one column, the day's specials in the one accent color. **An ojek app's settings** can be tree-tested with drivers to find where they expect "change my vehicle plate" and "stop receiving orders".

8.12 HCI in Practice: Profitina

Profitina's product illustrates each of the chapter's three ideas.

A dashboard as a cockpit. Profitina's dashboard gathers, in one request, everything that needs the user's attention: notifications of replies, likes, reposts, direct messages, pins, feedback status and PROFIT purchases, followed channels, rank and activity, each item clickable to its own section. That is an information-architecture decision, organizing by *what needs me now* rather than by *which module produced it*, and a visual-design problem: many kinds of item must be scannable at once, so grouping, a consistent item layout and a clear hierarchy between unread and read items matter more than decoration.

Counters that mean something. Channels show "N new" counters, and those counters count only top-level posts, not every reply. The choice of *what* is counted is a labeling decision: a number next to a channel promises "there is this much new to see", and counting replies would inflate the promise and train users to ignore it.

Architecture for machines as well as people. Profitina's marketing site is bilingual (Indonesian and English) and publishes `/llms.txt`, a plain-text map of the site for AI answer engines, alongside the structured data (schema.org) that the product helps its customers add. Information architecture in 2026 has two audiences: the people who navigate the site and the AI systems that read it on their behalf. Clear organization and honest labels serve both. Profitina advises and the user decides; a legible structure is what lets the user decide well.

8.13 Common Mistakes, and How to Avoid Them

- **Everything equally loud.** When every element is bold, colored or large, nothing stands out. Decide the order of importance and let most things be quiet.
- **Random colors for icons.** Color should signal meaning (tappable, success, warning), not decorate. Similar things should look similar.
- **Gaps that do not group.** If the space between groups is no larger than the space within them, proximity fails. Use a spacing scale with clearly different steps.
- **Too many font sizes.** Sizes that differ by a pixel look like mistakes. Use a modular scale.
- **Checking contrast by eye.** Compute it (Equation (8.4)); check every pair in use, including text on cards and translucent surfaces.
- **Color as the only signal.** Always add text or an icon.
- **Copying a platform's style without its accessibility settings.** Respect system text size, reduce-transparency and dark-mode settings.
- **A design system without tokens or an owner.** Hard-coded values drift; a system nobody maintains is abandoned.
- **Organizing by the org chart.** Customers do not know the bank's departments. Ask them with a card sort.

- **Internal labels.** “Mutasi”, “Inquiry” and “Setor” are the bank’s words. Use the customers’ words and keep the old ones as search keywords.
- **Forcing every item into one place.** When participants split an item between groups, cross-list it.

8.14 Chapter Summary

- Visual design makes structure visible. The Gestalt principles of proximity, similarity, common region and closure (**Figure 8.1**) create groups; contrast in size, weight, color and position creates hierarchy.
- Type should be large enough, in comfortable lines, and chosen from a modular scale (Equation (8.1)); space follows a 4- or 8-pixel scale and a grid.
- WCAG 2.2 (also ISO/IEC 40500:2025) requires 4.5:1 for body text and 3:1 for large text and for UI parts. The contrast ratio follows from linearized sRGB channels (Equation (8.2)), relative luminance (Equation (8.3)) and Equation (8.4). WCAG 3.0 remains a draft.
- `contrast.py` and `Contrast.java` read a W3C Design Tokens 2025.10 file, check every pair and suggest fixes; in the bank app, six of ten pairs failed, including the old white-on-green buttons (2.78:1) (**Figure 8.2**).
- A design system combines tokens, components and guidelines; primitive and semantic tokens (**Figure 8.3**) make change and theming safe. Material 3 Expressive, Liquid Glass and Fluent 2 are the platform languages of 2026; translucent materials make contrast a continuous concern.
- Information architecture organizes, labels, navigates and searches. An open card sort, analyzed with co-occurrence similarity (Equation (8.5)) and average linkage (Equation (8.6)), proposed four groups for the bank app and showed that customers file the loan installment under bills (**Figure 8.4**).

8.15 Exercises

1. Compute by hand the contrast ratio of grey `#6B7280` on white, and check it against the program’s 4.83:1. Does it pass for body text? For large text?
2. With a modular scale of base 16 px and ratio 1.2, list the sizes for $k = -1$ to 4, rounded to whole pixels. Compare them with the scale of ratio 1.25 in Section 8.4.2: which gives more distinct levels on a small phone screen?
3. Add the pair `amber,navy,ui` (amber icons on a navy bar) to `pairs.csv` and run the checker. Explain the result using Equation (8.4).
4. The contrast checker darkens the darker color. Write a variant that *lightens* the lighter color (mixing it with white) and compare the two suggestions for `grey` on `grey-light`. When would a designer prefer each?
5. Run `cardsort.py` with a cut of 0.6 and of 0.7. How many groups do you get, and which cards leave their group? What does that say about the stability of those cards’ placement?
6. Explain in your own words why single linkage (the similarity of the *most* similar pair) could join the transfer and bill groups early, using the card *Scheduled transfer* as an example. Modify `link` to use single linkage and check your explanation.
7. Design a tree test for the bank’s new four-tab structure: write five tasks, including one about the loan installment and one about the old word “mutasi”, and say what result would make you change the structure.

8. A designer proposes placing white text on a Liquid-Glass-style translucent bar over a photo of rice fields. Describe how you would check and guarantee contrast for this bar.

8.16 Solutions to the Exercises

1. Channels 0x6B = 107, 0x72 = 114, 0x80 = 128; divided by 255: 0.4196, 0.4471, 0.5020. Linearized with Equation (8.2): $(0.4746/1.055)^{2.4} = 0.4499^{2.4} = 0.147$; $(0.5021/1.055)^{2.4} = 0.4759^{2.4} = 0.168$; $(0.5570/1.055)^{2.4} = 0.5279^{2.4} = 0.216$. $L = 0.2126 \times 0.147 + 0.7152 \times 0.168 + 0.0722 \times 0.216 = 0.0313 + 0.1202 + 0.0156 = 0.167$. CR $= 1.05/0.217 = 4.83$. It passes for body text (4.5) and for large text (3), but with little margin: on any background slightly darker than white it fails, as the pair with `grey-light` shows.

2. 16×1.2^k for $k = -1, \ldots, 4$: 13.3, 16, 19.2, 23.0, 27.6, 33.2, rounded 13, 16, 19, 23, 28, 33. The 1.25 scale gives 13, 16, 20, 25, 31, 39. On a small screen, 1.25 gives fewer but more clearly distinct levels and a larger top size for the balance; 1.2 gives more levels, which helps text-heavy screens but risks sizes that look alike (16 and 19).

3. Add the line `amber,navy,ui`. The program reports a ratio of 5.50:1, `AA pass`. By Equation (8.4), $(0.4134 + 0.05)/(0.0343 + 0.05) = 0.4634/0.0843 = 5.50$. Amber, which fails on white, passes on navy: contrast is a property of the pair, never of one color.

4. A `lighten(hex, share)` function computes `round(c + (255 - c) * share)` for each channel, and the fix loop lightens the lighter color until the ratio is reached. For grey text on `grey-light` (`#F3F4F6`), lightening the background reaches 4.5:1 near pure white (`#FDFDFE` or similar), which changes the card's look; darkening the text to `#69707D` is barely visible. A designer lightens the background when the card color is not important to the design and darkens the text when the background is part of the brand or the text color must stay consistent across screens.

5. At 0.6, *Scheduled transfer* (joined at 0.53), *Loan installment* (0.51) and *Change PIN* (0.55) become groups of their own, giving seven groups; at 0.7, *Pay by QRIS* (0.65) also separates and the groups become eight. Those four cards are the ones whose placement participants disagreed about. They are candidates for cross-listing, clearer labels or a closer look in a tree test, while the cards that joined at 1.00 can be placed with confidence.

6. Single linkage takes the *largest* similarity between any card of one cluster and any card of the other. *Scheduled transfer* was placed with the bills by several participants (they saw it as "paying regularly"), so its similarity to a bill card is high; once it is in the transfer cluster, single linkage treats the whole transfer cluster as close to the bills, and the two groups merge at a high level, even though most transfer cards are never grouped with bills. Replacing `link` by the maximum of `sim[i][j]` over pairs shows the transfer and bill clusters merging at 0.55, far above the 0.22 of average linkage. This is the *chaining* weakness of single linkage.

7. Tasks: "Pay this month's loan installment", "See the money that came into your account last week" (tests "Riwayat transaksi" and the old word "mutasi"), "Send Rp200.000 to your son's account at another bank", "Buy an electricity token for Rp50.000", "Find the opening hours of the nearest branch". Success is reaching the right item on the first path; record the first tab chosen. A task with less than about 80% first-try success, or a single wrong tab chosen by many participants, suggests a change: a cross-link, a renamed tab or a moved item. If most participants look for "mutasi" under *Bantuan*, for example, the history needs a clearer label or a link from the balance.

8. Contrast over a photo cannot be computed once, because the background varies. Guarantee it by design: put a backing layer under the text (a tint of navy at an opacity high enough that the lightest part of any photo behind it still gives at least 4.5:1 with white text), or a text shadow or outline, and respect the system's Reduce Transparency and Tinted settings. Check by computing the ratio of the text color against the *lightest* composite background, the tint blended over pure white, and by testing screenshots with real, bright photos. The same approach is what iOS 26.1's Tinted option applies at the system level.

References

[1] L. Rosenfeld, P. Morville, and J. Arango, *Information Architecture: For the Web and Beyond*, 4th ed. Sebastopol, CA, USA: O'Reilly Media, 2015.

[2] J. Wagemans, J. H. Elder, M. Kubovy, S. E. Palmer, M. A. Peterson, M. Singh, and R. von der Heydt, "A century of Gestalt psychology in visual perception: I. Perceptual grouping and figure–ground organization," *Psychological Bulletin*, vol. 138, no. 6, pp. 1172–1217, 2012.

[3] W3C, *Web Content Accessibility Guidelines (WCAG) 2.2*, W3C Recommendation, Oct. 5, 2023, latest version accessed Sep. 2026. [Online]. Available: https://www.w3.org/TR/WCAG22/

[4] W3C Web Accessibility Initiative, "WCAG 2.2 Approved as an ISO Standard," Oct. 21, 2025. [Online]. Available: https://www.w3.org/WAI/news/2025-10-21/wcag22-iso

[5] W3C Web Accessibility Initiative, "For Review: WCAG 3 Working Draft – September 2026," Sep. 10, 2026. [Online]. Available: https://www.w3.org/WAI/news/2026-09-10/wcag3/

[6] W3C Design Tokens Community Group, "Design Tokens specification reaches first stable version," Oct. 28, 2025. [Online]. Available: https://www.w3.org/community/design-tokens/2025/10/28/design-tokens-specification-reaches-first-stable-version/

[7] Google Design, "Expressive Design: Google's UX Research," 2025. [Online]. Available: https://design.google/library/expressive-material-design-google-research

[8] Apple Newsroom, "Apple introduces a delightful and elegant new software design," Jun. 9, 2025. [Online]. Available: https://www.apple.com/newsroom/2025/06/apple-introduces-a-delightful-and-elegant-new-software-design/

[9] Engadget, "How to adjust the Liquid Glass effect in iOS 26.1," Nov. 9, 2025. [Online]. Available: https://www.engadget.com/mobile/smartphones/how-to-adjust-the-liquid-glass-effect-in-ios-261-203634681.html

[10] Microsoft, "Fluent 2 Design System," accessed Sep. 2026. [Online]. Available: https://fluent2.microsoft.design/

[11] D. Spencer, *Card Sorting: Designing Usable Categories*. Brooklyn, NY, USA: Rosenfeld Media, 2009.

Chapter 9

Interaction Styles: GUI, Touch, Voice, Gesture

A glass of iced tea can be ordered in many ways: by calling across the counter, by pointing at a picture, by ticking a paper form, by typing a command, by tapping a phone or by speaking to a speaker on the table. Each way is an *interaction style*, and each one makes some things easy and other things hard. This chapter follows a small warung in Surabaya that wants to take orders from a tablet on the counter, and uses it to explain the main interaction styles of today: command languages, menus and forms, direct manipulation in graphical user interfaces, touch, gestures, voice and the new spatial interfaces. By the end of the chapter you will be able to name the styles and say what each costs the user; explain the gulfs of execution and evaluation and why direct manipulation narrows them; use Fitts's law and its finger-touch version to reason about target sizes; apply the target-size rules of WCAG 2.2 and of the Apple and Google platforms; design a voice exchange that asks for confirmation when it is unsure; and build, in Java and in Python, one order core with three front ends — a command line, a typed "voice" line and a touch kiosk — and a program that measures every tappable target of the kiosk in a real browser.

9.1 Bu Tini's Counter

Bu Tini runs a warung near a campus in Surabaya: five dishes, one cook, one helper, and at noon a crowd of students, drivers and office workers who all want to order at once. Watch the counter for ten minutes and you see every interaction style in this chapter at work. A regular shouts "*Es teh dua, Bu!*" from the door, and Bu Tini answers "*Dua, ya?*" to be sure — a voice command with a spoken confirmation. A new customer who does not know the menu points at the laminated card with photos — a menu selection, done by gesture. A driver in a hurry holds up two fingers and nods at the tea jar, and that works too, because Bu Tini can see both him and the jar. The office workers who order for a whole room write the order on a slip of paper with a line for each dish — a form. And the students ask whether they can order and pay from their phones.

Bu Tini's nephew Dimas, a student of informatics, offers to build a tablet kiosk for the counter. His first idea is simple: put the menu on the screen, let people tap. Then the questions begin. Should the kiosk also understand typed or spoken words, for customers who would rather say "nasi goreng dua" than hunt for buttons? How big must a button be so that a thumb with sambal on it hits the right dish? What should the kiosk do when it hears something it does not understand — guess, or ask? And what happens to Bu Tini's older customers, some of whom have never touched a tablet?

These are the questions of this chapter. They matter well beyond one warung. According to the survey of the Indonesian Internet Service Providers Association (APJII) released in May 2026, 81.72% of Indonesians — about 235 million people — now use the internet [1]. Almost every one of them meets many interaction styles in a single day: the ATM's menu of buttons beside the screen, the ride-hailing app's map that is dragged with one finger, the voice note in a chat, the QR code scanned at the till. A designer who knows only one style will force every task into it, and many users will pay for that choice.

9.2 The Idea in Plain Words

9.2.1 What an interaction style is

An **interaction style** is the way a person tells a computer what to do and sees what the computer has done. It answers two small questions. *How do I say it?* — by typing a word, by choosing from a list, by filling in blanks, by pointing at and moving an object, by touching, by gesturing, by speaking. *How do I see the result?* — as a line of text, as a changed picture, as a sound, as a vibration.

The classic textbook of the field, Shneiderman and colleagues' *Designing the User Interface*, groups the ways of "saying it" into five primary styles: direct manipulation, menu selection, form fill-in, command language and natural language [2]. Touch, gesture, voice and spatial interaction are not new styles so much as new *channels* for the old ones: a phone app mostly offers direct manipulation and menus through touch; a voice assistant offers natural language through speech; a headset such as Apple Vision Pro offers direct manipulation through the eyes and a pinch of the fingers.

> **DEFINITION.** An *interaction style* is a family of ways in which the user expresses an intention to a system and perceives its response. The five primary styles are *command language* (the user types commands with a fixed syntax), *menu selection* (the user chooses from listed options), *form fill-in* (the user fills labeled fields), *direct manipulation* (the user acts on visible objects and sees the effect at once) and *natural language* (the user writes or speaks as to a person). A *modality* is the sensory channel used: keyboard, pointer, touch, gesture, voice, gaze.

9.2.2 A map of the styles

Table 9.1 lays the styles side by side, with the question that decides between them: what must the user *remember*, and what must the user *see*? A command language asks the user to recall the right word from memory; a menu lets the user recognize it in a list. Recognition is easier than recall, which is why menus suit occasional users; recall is faster once learned, which is why experts love commands and keyboard shortcuts.

Table 9.1 The five primary interaction styles, what each asks of the user, and where each appears at Bu Tini's counter.

Style	The user…	Strong when	Weak when	At the warung
Command language	types a word with a fixed syntax	experts, repeated tasks, scripting	novices; errors in syntax	`add esteh 2` on a laptop
Menu selection	picks from a list	occasional users; few options	many options, deep trees	the laminated card, the ATM
Form fill-in	fills labeled blanks	structured data (address, order slip)	fields the user does not understand	the office order slip
Direct manipulation	acts on visible objects	seeing and undoing results at once	abstract actions, many objects	tapping a dish on the kiosk
Natural language	speaks or writes freely	hands busy, eyes busy, no menu known	ambiguity, noise, privacy	*"Es teh dua, Bu!"*

No style wins everywhere. The ATM uses menus because a bank cannot train its customers; a programmer uses commands because the same twenty words are typed a thousand times a day; the kiosk uses direct manipulation because the dishes are few and pictures of them are clear; the regular at the door uses natural language because his hands hold a helmet. Good systems often

combine styles: the same order can be given in more than one way, and the system shows the result in one shared picture.

9.2.3 A worked example: one glass of tea, five ways

Take the order "two iced teas" and count what each style asks.

1. **Command.** Type `add esteh 2` and press Enter: 13 keystrokes. The user must know the code `esteh`, the verb `add` and the order of the words. A typo such as `add esteh x` gives an error.
2. **Menu.** On a numbered menu, press 1 (drinks), 1 (iced tea), then 2 for the quantity: three presses. The user must read the lists, but need not remember anything.
3. **Form.** Find the line "Iced tea" on the slip and write 2 in its box: one field. The form also shows everything else that could be ordered.
4. **Direct manipulation.** Tap the "Iced tea" button twice and watch the order line say "2 x Iced tea". The result is visible after every tap, and a minus button undoes a mistake.
5. **Natural language.** Say "*es teh dua*". Three words — but the system must now *understand* them, and it may hear "*es teh*" as "*estate*", or "*dua*" as "*juga*".

The count shows the trade: styles that ask less of the user's memory ask more of the screen, and styles that ask the least of the user (speaking) ask the most of the system. The rest of this chapter makes that trade precise.

9.3 The Theory with Evidence

9.3.1 Two gulfs, and why direct manipulation narrows them

Norman describes every use of a device as a crossing of two gulfs [3]. The **gulf of execution** is the distance between what the user wants ("two teas") and what the system lets the user do (type a command? find a button?). The **gulf of evaluation** is the distance between what the system shows and what the user needs to know ("did it take my order?"). A good interface bridges both: it makes the possible actions visible and it shows the state of the system clearly after each action. Chapter 1 met these gulfs in general; here they explain the styles.

A command language leaves the gulf of execution wide — the user must translate the wish into a syntax — and often leaves the gulf of evaluation wide too, because a command may succeed silently. Shneiderman coined the term **direct manipulation** for interfaces that narrow both gulfs at once [2]: the objects of interest are continuously visible; the user acts on them by physical actions (pointing, tapping, dragging) instead of complex syntax; and actions are rapid, incremental and reversible, with their effect visible immediately. Spreadsheets, drawing programs, the map in a ride-hailing app and the kiosk of this chapter are all direct manipulation. The price is screen space and the difficulty of expressing abstract actions: "every Friday at noon, order ten teas for the office" is easy to type as a command and awkward to express by dragging.

9.3.2 Events, state and the speed of feedback

Every graphical, touch or voice interface is built the same way underneath: it waits for **events** — a key press, a tap, a finished utterance, a network reply — and runs a **handler** for each event, which changes the program's **state** and redraws what the user sees. This event-driven structure is why an interface never "runs from top to bottom" like a batch program; it sits in a loop, and the user decides what happens next. Our kiosk has a tiny state (the cart) and three events (add,

remove, pay); our voice line has one more piece of state, an order *waiting for confirmation*, which Section 9.5 introduces.

How fast must a handler show its result? Research going back to Miller in 1968 and summarized by Nielsen gives three limits that are still used today [4]. About **0.1 s** is the limit for the user to feel that the system reacts *instantaneously*, as when a button changes color under the finger; about **1 s** is the limit for the user's flow of thought to stay uninterrupted; about **10 s** is the limit for keeping the user's attention on the task at all. Past one second, show that something is happening; past ten, show progress and let the user do something else. These limits are not laws of nature but they are a reliable guide, and they are why well-made apps react to a tap at once and do the slow work afterwards.

9.3.3 Touch: a finger is not a mouse pointer

A mouse pointer is one pixel wide; a finger pad touches the glass with an area several millimeters across and hides the very target it presses. Two consequences follow: targets must be larger, and the speed–accuracy trade-off changes.

Fitts's law predicts the time to point at a target from its distance and its width. In the form used in HCI (the "Shannon formulation"), the time MT to hit a target of width W at distance D is

$$MT = a + b\log_2\left(\frac{D}{W} + 1\right) \tag{9.1}$$

where a and b are constants measured for a device and a person, and the logarithm, called the **index of difficulty** ID and measured in bits, captures how hard the target is [5]. Chapter 2 derived the law for the mouse. Equation (9.1) says that halving the width of a target costs about as much as doubling its distance: small targets are slow targets.

On a touch screen the law needs one correction. Bi, Li and Zhai showed that finger touch has an *absolute* imprecision that does not shrink however slowly the user moves: the finger's contact point scatters around the aimed point even with all the care in the world [6]. Their **FFitts law** (finger Fitts) splits the observed spread of touch points, with standard deviation σ, into a part due to the speed of the movement and a part σ_a due to the finger itself, and keeps only the first part in the index of difficulty:

$$ID_f = \log_2\left(\frac{D}{\sqrt{2\pi e\,(\sigma^2 - \sigma_a^2)}} + 1\right) \tag{9.2}$$

The practical meaning of Equation (9.2) is simple: below a certain size, making a touch target smaller does not merely slow the user down, it produces *errors* that no amount of care prevents. This is why every touch platform sets a minimum target size.

> **EXAMPLE.** On the 360-pixel-wide kiosk page, the "Pay" button and the language link are both about $D = 300$ CSS pixels from where the thumb rests. The first version of the page drew the language link as a line of small text only 13 pixels high; the menu buttons are 64 pixels high. With Equation (9.1): for the link, $ID = \log_2(300/13 + 1) = \log_2(24.08) = 4.59$ bits; for a menu button, $ID = \log_2(300/64 + 1) = \log_2(5.69) = 2.51$ bits. The link is almost two bits harder — and by Equation (9.2) a 13-pixel target is also close to the size at which the finger's own scatter causes misses.

Table 9.2 collects the minimum sizes in force in September 2026. The **Web Content Accessibility Guidelines (WCAG) 2.2** of the W3C, approved as the international standard

ISO/IEC 40500:2025 in October 2025, added success criterion 2.5.8 *Target Size (Minimum)* at level AA: a pointer target must be at least 24 by 24 CSS pixels, unless it is spaced so that a 24-pixel circle centered on it does not touch another target, or it is inline in a sentence, or one of a few other exceptions applies [7]. The older criterion 2.5.5 at level AAA asks for 44 by 44. Apple's Human Interface Guidelines recommend a hit region of at least 44 by 44 points on iPhone and iPad [8]; Google's Material Design asks for at least 48 by 48 density-independent pixels, about 9 mm, for touch targets on Android [9].

Table 9.2 Minimum sizes for touch and pointer targets as of September 2026.

Source	Minimum target	Level or status
WCAG 2.2, SC 2.5.8 *Target Size (Minimum)*	24 × 24 CSS px, or spacing exception	AA (required for most laws)
WCAG 2.2, SC 2.5.5 *Target Size (Enhanced)*	44 × 44 CSS px	AAA
Apple Human Interface Guidelines	44 × 44 pt	platform guideline
Material Design 3 (Android)	48 × 48 dp (about 9 mm)	platform guideline

9.3.4 Gestures: powerful, and invisible

A **gesture** is a movement that carries a command: a swipe to delete a chat, a pinch to zoom a map, a long press for a menu, a pull down to refresh. Gestures are fast for those who know them and invisible to those who do not — there is nothing on the screen that says "you can swipe here". Norman's term for the cue that tells the user an action is possible is a **signifier** [3], and gestures usually have none. Designers therefore follow three rules. First, a gesture should be a shortcut, not the only way: the chat app that deletes by swipe also offers a delete item in a menu. Second, gestures should follow the platform's conventions, because users learn them once per phone, not once per app. Third, gestures must not exclude people who cannot perform them.

That last rule is now part of WCAG 2.2. Success criterion 2.5.1 *Pointer Gestures* requires that anything done with a multipoint or path-based gesture (a pinch, a swipe along a path) can also be done with a single pointer without a path, such as a tap; criterion 2.5.7 *Dragging Movements*, new in WCAG 2.2, requires that anything done by dragging can also be done by single taps or clicks [7]. A map that can only be zoomed by pinching, or a list that can only be reordered by dragging, fails; add plus and minus buttons, or up and down arrows, and it passes. Chapter 11 returns to these criteria from the side of users with motor disabilities.

9.3.5 Voice: natural, and ambiguous

Voice is the oldest interface of all and, for computers, one of the hardest. It suits moments when the hands and eyes are busy — cooking, driving, carrying a helmet — and people who find screens hard, including many older adults and people with low vision. The main voice assistants are changing fast: in August 2026 Google announced that, starting September 4, 2026, Gemini would gradually replace Google Assistant on Android phones, tablets, Wear OS watches and Android Auto [10], and assistants built on large language models now accept long, loosely phrased requests instead of fixed command phrases. Research at CHI 2026 has explored how voice assistants can support adults with intellectual disabilities in doing everyday tasks independently [11]; the older finding that voice helps some users and frustrates others remains true.

Whatever engine sits underneath, a **voice user interface** (VUI) faces three problems that a touch screen does not. It has *no visible menu*: users do not know what they may say. It is *uncertain*: the recognizer may mishear, and the words may be ambiguous — "*es teh dua*" could mean two teas or

one tea for table two. And it is *linear and fleeting*: a spoken list of twenty dishes is forgotten before it ends. The classic design answers are these.

- **Confirm in proportion to risk and doubt.** An *implicit* confirmation repeats what was understood while going ahead ("Two iced teas — anything else?"); an *explicit* confirmation stops and asks ("Did you mean one coffee? Say yes to confirm"). Use explicit confirmation when the recognizer is unsure or when an error is costly (paying, sending money, deleting).
- **Say what can be said.** When the system does not understand, it should not say only "Sorry?"; it should give an example of a good request ("Please say a menu name, such as *nasi goreng*").
- **Keep a way back.** Every spoken action should be undoable by voice ("*batal*", cancel) and visible on a screen if there is one.

Bu Tini's own "*Dua, ya?*" is exactly an explicit confirmation. Section 9.5 builds it into the program with a number, the **confidence** c of an interpretation: the share of the words in the utterance that the parser understood,

$$c = \frac{k}{m} \tag{9.3}$$

where m is the number of words heard and k the number that matched a dish, a number, an intent word or a harmless filler word. Real recognizers report a confidence from their acoustic and language models; Equation (9.3) is a simple, explainable stand-in with the same role. When c is below a threshold, the system asks instead of acting.

> **KEY INSIGHT:** A voice interface that never asks is guessing, and a voice interface that always asks is tiring. The skill is in choosing when to ask: when the system is unsure, and when a wrong guess is expensive.

9.3.6 Today's design languages, and interaction in space

The large platforms publish **design languages** that fix how their controls look and behave, and they changed noticeably in 2025. On September 15, 2025, Apple released iOS 26 and its siblings with **Liquid Glass**, a translucent material for controls and navigation that reflects and refracts what lies behind it and gives the content more of the screen [12]. Google's **Material 3 Expressive**, announced in May 2025, adds more shapes, color, type contrast and springy motion; Google reports that it rests on 46 studies with more than 18,000 participants, in which people spotted key interface elements up to four times faster in the expressive designs [13]. Microsoft's **Fluent 2** is the shared design system of Windows and Microsoft 365 [14]. For the designer of a small app the lesson is modest but real: follow the platform's controls and gestures, because users have learned them; check the new translucent and colorful styles for legibility, because beautiful glass can lower contrast (Chapter 11 measures contrast).

Spatial computing carries direct manipulation into three dimensions. In visionOS, the user *looks* at a control to target it and *pinches* thumb and finger to activate it; the hands can rest in the lap [15]. This is Fitts's law with the eyes as the pointer, and Apple's guidance accordingly asks for generous target regions and spacing in space. Chapter 15 studies spatial, wearable and ambient interfaces in depth; for now it is enough to see that the same two gulfs and the same trade between recall and recognition apply there too.

9.4 A Hands-On Method: Choosing and Checking a Style

When you design a new feature, choose its style deliberately, then check it. **Table 9.3** is a one-page worksheet that does both; fill it in for each important task.

Table 9.3 A worksheet for choosing and checking an interaction style, filled in for the warung kiosk.

Step	Question	Bu Tini's kiosk
1. Users	Who does the task, how often, with what skill?	Students (daily), drivers (in a hurry), older regulars (rarely touch screens)
2. Context	Hands, eyes, noise, light, privacy?	One hand often busy; noisy at noon; greasy fingers
3. Task shape	Few options or many? Concrete or abstract?	Five dishes, concrete: suits direct manipulation
4. Primary style	Which style fits steps 1–3 best?	Touch kiosk with big picture buttons
5. Second path	Which alternative serves users the primary style fails?	Spoken or typed order with confirmation; the helper at the counter
6. Feedback	What does the user see within 0.1 s, 1 s, 10 s?	Order line updates at once; total always visible
7. Errors	How is a mistake seen and undone?	Minus button per line; explicit confirmation when unsure
8. Check	Target sizes, gesture alternatives, contrast measured?	Audit every target at 24, 44 and 48 px (Section 9.5.4)

Steps 1 to 3 come from the user research of Chapter 5; step 8 is where the program of this chapter helps, because target sizes can be measured automatically in a real browser.

9.5 Building It Step by Step

The program has one **order core** and three front ends, as **Figure 9.1** shows. The core knows the menu, keeps the cart, computes totals and interprets two kinds of input: strict commands and free words. The front ends only translate between a person and the core. This separation is the most important design decision in the chapter: whatever the style, the prices, the rules and the total are the same, and a new style — a real speech recognizer, a WhatsApp bot — can be added without touching the core.

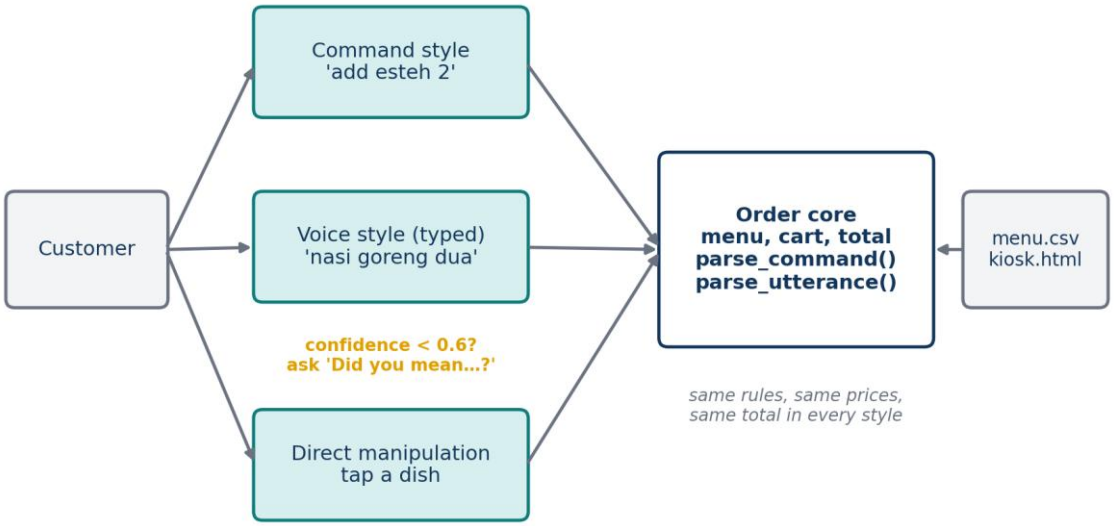


Figure 9.1 One order core with three front ends: the command style, the typed voice style with its confirmation rule, and the direct-manipulation kiosk all use the same menu, cart and total.

9.5.1 The menu and the order core

The menu is a small CSV file, **Listing 9.1**. Besides the code, the two names and the price, each dish lists the *aliases* customers use for it — the vocabulary of the voice style:

Listing 9.1 `menu.csv`: the five dishes, their prices and the words customers use for them.

```
code,name_en,name_id,price,aliases
esteh,Iced tea,Es teh,5000,es teh|teh|iced tea|tea
kopi,Coffee,Kopi,6000,kopi|coffee
nasgor,Fried rice,Nasi goreng,15000,nasi goreng|nasgor|fried rice
mie,Fried noodles,Mi goreng,13000,mi goreng|mie goreng|mie|noodles
tempe,Fried tempeh,Tempe goreng,2000,tempe goreng|tempe|tempeh
```

The Python core in **Listing 9.2** has three parts. `load_menu` reads the CSV. The class `Cart` holds quantities per dish and refuses unknown dishes and quantities outside 1 to 20. Two parsers turn input into actions: `parse_command` accepts the strict syntax `verb code [quantity]`, and `parse_utterance` reads free words. The utterance parser scans the words from left to right; at each position it looks for the *longest* alias that starts there (so “nasi goreng” wins over a shorter alias), takes a number word just before or just after the dish as its quantity, and marks each word as understood or not. Words like “*batal*” or “cancel” switch the intent to remove; filler words like “*tolong*” (please) and “*ya*” are understood but carry no meaning. The confidence of Equation (9.3) is the share of understood words.

Listing 9.2 `order_core.py`: the menu, the cart and the two parsers for the command and voice styles.

```
"""Warung order core shared by the command, voice and kiosk front ends."""
import csv
from pathlib import Path

DATA = Path(__file__).resolve().parent.parent / "data"
NUMBERS = {"satu": 1, "dua": 2, "tiga": 3, "empat": 4, "lima": 5,
           "one": 1, "two": 2, "three": 3, "four": 4, "five": 5}
ADD = {"pesan", "tambah", "mau", "minta", "order", "add", "want"}
REMOVE = {"hapus", "batal", "kurangi", "remove", "cancel"}
TOTAL = {"berapa", "total", "much"}
FILLER = {"tolong", "saya", "aku", "ya", "dong", "sama", "dan", "yang",
          "please", "i", "a", "and", "the", "me", "give", "how"}
CONFIRM_BELOW = 0.6


def load_menu(path=DATA / "menu.csv"):
    with open(path, encoding="utf-8") as f:
        rows = list(csv.DictReader(f))
    for r in rows:
        r["price"] = int(r["price"])
        r["aliases"] = r["aliases"].split("|")
    return {r["code"]: r for r in rows}


class Cart:
    def __init__(self, menu):
        self.menu = menu
        self.qty = {}

    def add(self, code, n=1):
        if code not in self.menu:
            raise ValueError("unknown item: " + code)
        if not 1 <= n <= 20:
            raise ValueError("quantity must be 1..20")
        self.qty[code] = self.qty.get(code, 0) + n

    def remove(self, code, n=1):
        left = self.qty.get(code, 0) - n
        if left > 0:
            self.qty[code] = left
        else:
            self.qty.pop(code, None)
```

Listing 9.2 (Continued) *order_core.py: the menu, the cart and the two parsers for the command and voice styles.*

```
    def lines(self, lang="en"):
        out = []
        for code, n in self.qty.items():
            item = self.menu[code]
            out.append((code, item["name_" + lang], n, n * item["price"]))
        return out

    def total(self):
        return sum(n * self.menu[c]["price"] for c, n in self.qty.items())


def rupiah(x):
    return "Rp" + format(x, ",").replace(",", ".")


def parse_command(line):
    """Command style: 'add esteh 2', 'remove kopi', 'show'."""
    parts = line.split()
    if not parts:
        raise ValueError("empty command")
    verb, args = parts[0], parts[1:]
    if verb in ("add", "remove") and 1 <= len(args) <= 2:
        n = 1
        if len(args) == 2:
            if not args[1].isdigit():
                raise ValueError("quantity must be a number: " + args[1])
            n = int(args[1])
        return verb, args[0], n
    if verb in ("show", "total") and not args:
        return verb, None, 0
    raise ValueError("unknown command: " + line)


def parse_utterance(text, menu):
    """Voice style: free words -> (intent, [(code, n)], confidence)."""
    words = text.lower().replace(",", " ").split()
    known = [False] * len(words)
    intent = "add"
    items = []
    i = 0
    while i < len(words):
        match = None
        for code, item in menu.items():
            for alias in item["aliases"]:
                a = alias.split()
                if words[i:i + len(a)] == a and (
```

Listing 9.2 (Continued) *order_core.py: the menu, the cart and the two parsers for the command and voice styles.*

```
                    match is None or len(a) > len(match[1])):
                match = (code, a)
        if match:
            n, j = 1, i + len(match[1])
            if i > 0 and words[i - 1] in NUMBERS:
                n = NUMBERS[words[i - 1]]
            elif j < len(words) and words[j] in NUMBERS:
                n = NUMBERS[words[j]]
                known[j] = True
                j += 1
            items.append((match[0], n))
            for k in range(i, i + len(match[1])):
                known[k] = True
            i = j
            continue
        w = words[i]
        if w in REMOVE:
            intent = "remove"
        elif w in TOTAL:
            intent = "total"
        if w in ADD | REMOVE | TOTAL | FILLER or w in NUMBERS:
            known[i] = True
        i += 1
    conf = sum(known) / len(known) if known else 0.0
    return intent, items, round(conf, 2)
```

The Java core in **Listing 9.3** is the same design. Java records hold a menu item, a cart line and an interpretation (`Heard`); the cart uses a `LinkedHashMap` so that lines stay in the order they were added, exactly as the Python dictionary does.

Listing 9.3 `OrderCore.java`: the same menu, cart and parsers in Java.

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.*;

/** Warung order core shared by the command, voice and kiosk front ends. */
public class OrderCore {
    static final Path DATA = Path.of("..", "data");
    static final Map<String, Integer> NUMBERS = Map.of("satu", 1, "dua", 2,
        "tiga", 3, "empat", 4, "lima", 5, "one", 1, "two", 2, "three", 3,
        "four", 4, "five", 5);
    static final Set<String> ADD = Set.of("pesan", "tambah", "mau", "minta",
        "order", "add", "want");
    static final Set<String> REMOVE = Set.of("hapus", "batal", "kurangi",
        "remove", "cancel");
    static final Set<String> TOTAL = Set.of("berapa", "total", "much");
    static final Set<String> FILLER = Set.of("tolong", "saya", "aku", "ya",
        "dong", "sama", "dan", "yang", "please", "i", "a", "and", "the", "me",
        "give", "how");
    static final double CONFIRM_BELOW = 0.6;

    record Item(String code, String nameEn, String nameId, int price,
                List<String> aliases) {
        String name(String lang) { return lang.equals("id") ? nameId : nameEn; }
    }

    record Line(String code, String name, int qty, int subtotal) {}

    record Heard(String intent, List<Map.Entry<String, Integer>> items,
                 double confidence) {}

    static Map<String, Item> loadMenu() throws IOException {
        Map<String, Item> menu = new LinkedHashMap<>();
        List<String> rows = Files.readAllLines(DATA.resolve("menu.csv"),
                                               StandardCharsets.UTF_8);
        for (String row : rows.subList(1, rows.size())) {
            String[] f = row.split(",");
            menu.put(f[0], new Item(f[0], f[1], f[2], Integer.parseInt(f[3]),
                                    List.of(f[4].split("\\|"))));
        }
        return menu;
    }
```

Listing 9.3 (Continued) *OrderCore.java: the same menu, cart and parsers in Java.*

```
    static class Cart {
        final Map<String, Item> menu;
        final Map<String, Integer> qty = new LinkedHashMap<>();

        Cart(Map<String, Item> menu) { this.menu = menu; }

        void add(String code, int n) {
            if (!menu.containsKey(code))
                throw new IllegalArgumentException("unknown item: " + code);
            if (n < 1 || n > 20)
                throw new IllegalArgumentException("quantity must be 1..20");
            qty.merge(code, n, Integer::sum);
        }

        void remove(String code, int n) {
            int left = qty.getOrDefault(code, 0) - n;
            if (left > 0) qty.put(code, left); else qty.remove(code);
        }

        List<Line> lines(String lang) {
            List<Line> out = new ArrayList<>();
            qty.forEach((c, n) -> out.add(new Line(c, menu.get(c).name(lang), n,
                                                   n * menu.get(c).price())));
            return out;
        }

        int total() {
            int t = 0;
            for (var e : qty.entrySet())
                t += e.getValue() * menu.get(e.getKey()).price();
            return t;
        }
    }

    static String rupiah(int x) {
        return "Rp" + String.format(Locale.US, "%,d", x).replace(',', '.');
    }
```

Listing 9.3 (Continued) *OrderCore.java: the same menu, cart and parsers in Java.*

```
    /** Command style: "add esteh 2", "remove kopi", "show". */
    static String[] parseCommand(String line) {
        String[] p = line.trim().split("\\s+");
        String verb = p[0];
        if ((verb.equals("add") || verb.equals("remove"))
                && p.length >= 2 && p.length <= 3) {
            String n = "1";
            if (p.length == 3) {
                if (!p[2].matches("\\d+"))
                    throw new IllegalArgumentException(
                        "quantity must be a number: " + p[2]);
                n = p[2];
            }
            return new String[] {verb, p[1], n};
        }
        if ((verb.equals("show") || verb.equals("total")) && p.length == 1)
            return new String[] {verb, null, "0"};
        throw new IllegalArgumentException("unknown command: " + line);
    }

    /** Voice style: free words -> intent, items and confidence. */
    static Heard parseUtterance(String text, Map<String, Item> menu) {
        String[] w = text.toLowerCase().replace(",", " ").trim().split("\\s+");
        boolean[] known = new boolean[w.length];
        String intent = "add";
        List<Map.Entry<String, Integer>> items = new ArrayList<>();
        int i = 0;
        while (i < w.length) {
            String code = null;
            int len = 0;
            for (Item item : menu.values())
                for (String alias : item.aliases()) {
                    String[] a = alias.split(" ");
                    if (i + a.length <= w.length && a.length > len
                        && Arrays.equals(Arrays.copyOfRange(w, i, i + a.length),
                                         a)) {
                        code = item.code();
                        len = a.length;
                    }
                }
            if (code != null) {
                int n = 1, j = i + len;
                if (i > 0 && NUMBERS.containsKey(w[i - 1])) {
                    n = NUMBERS.get(w[i - 1]);
                } else if (j < w.length && NUMBERS.containsKey(w[j])) {
                    n = NUMBERS.get(w[j]);
```

Listing 9.3 (Continued) *OrderCore.java: the same menu, cart and parsers in Java.*

```
                    known[j] = true;
                    j++;
                }
                items.add(Map.entry(code, n));
                for (int k = i; k < i + len; k++) known[k] = true;
                i = j;
                continue;
            }
            if (REMOVE.contains(w[i])) intent = "remove";
            else if (TOTAL.contains(w[i])) intent = "total";
            if (ADD.contains(w[i]) || REMOVE.contains(w[i])
                || TOTAL.contains(w[i]) || FILLER.contains(w[i])
                || NUMBERS.containsKey(w[i]))
                known[i] = true;
            i++;
        }
        int k = 0;
        for (boolean b : known) if (b) k++;
        double conf = Math.round(100.0 * k / w.length) / 100.0;
        return new Heard(intent, items, conf);
    }
}
```

9.5.2 The command line and the typed voice line

The first front end reads a session file line by line. A line that starts with `say` is treated as speech (typed, so that the program runs anywhere and its output can be checked); every other line is a command. The interesting part is the dialogue state `pending`. When an utterance names a dish but the confidence is below `CONFIRM_BELOW` (0.6), the front end does not act: it stores the interpretation and asks “Did you mean …? Say ‘yes’ to confirm.” If the next utterance begins with *ya*, *iya* or *yes*, the stored order is carried out; anything else discards it. When no dish is found at all, the front end asks for a menu name. **Listing 9.4** and **Listing 9.5** show the two versions.

Listing 9.4 `cli.py`: the command style and the typed voice style, with explicit confirmation below the confidence threshold.

```
"""Command style and voice style (typed) front end for the warung order."""
import sys
from order_core import (CONFIRM_BELOW, Cart, load_menu, parse_command,
                        parse_utterance, rupiah)


def run(lines):
    menu = load_menu()
    cart = Cart(menu)
    pending = None  # an order heard with low confidence, waiting for "yes"
    for line in lines:
        line = line.strip()
        if not line:
            continue
        print(">", line)
        if line.startswith("say "):
            words = line[4:].lower().split()
            if pending and words and words[0] in ("ya", "yes", "iya"):
                intent, items = pending
                pending = None
            else:
                pending = None
                intent, items, conf = parse_utterance(line[4:], menu)
                heard = ", ".join(f"{n} x {menu[c]['name_en']}"
                                  for c, n in items)
                print(f"  heard: {intent} [{heard or 'nothing'}] "
                      f"confidence {conf}")
                if not items:
                    print("  -> Sorry, which menu item? Please say its name.")
                    continue
                if conf < CONFIRM_BELOW:
                    # Low confidence: ask before acting (explicit confirm).
                    pending = (intent, items)
                    print(f"  -> Did you mean {heard}? Say 'yes' to confirm.")
                    continue
            for c, n in items:
                if intent == "remove":
                    cart.remove(c, n)
                else:
                    cart.add(c, n)
            print(f"  -> OK. Total {rupiah(cart.total())}")
            continue
        try:
            verb, code, n = parse_command(line)
            if verb == "add":
                cart.add(code, n)
```

Listing 9.4 (Continued) *cli.py: the command style and the typed voice style, with explicit confirmation below the confidence threshold.*

```
            elif verb == "remove":
                cart.remove(code, n)
            else:
                for _, name, q, sub in cart.lines():
                    print(f"  {q:>2} x {name:<14}{rupiah(sub):>10}")
            print(f"  total {rupiah(cart.total())}")
        except ValueError as e:
            print("  error:", e)


if __name__ == "__main__":
    src = open(sys.argv[1], encoding="utf-8") if len(sys.argv) > 1 else sys.stdin
    run(src)
```

Listing 9.5 `Cli.java`: the same command and voice front end in Java.

```
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.*;

/** Command style and voice style (typed) front end for the warung order. */
public class Cli {
    public static void main(String[] args) throws Exception {
        var menu = OrderCore.loadMenu();
        var cart = new OrderCore.Cart(menu);
        OrderCore.Heard pending = null; // low-confidence order, awaits "yes"
        for (String raw : Files.readAllLines(Path.of(args[0]),
                                             StandardCharsets.UTF_8)) {
            String line = raw.strip();
            if (line.isEmpty()) continue;
            System.out.println("> " + line);
            if (line.startsWith("say ")) {
                String first = line.substring(4).toLowerCase().split("\\s+")[0];
                OrderCore.Heard h;
                if (pending != null
                        && List.of("ya", "yes", "iya").contains(first)) {
                    h = pending;
                    pending = null;
                } else {
                    pending = null;
                    h = OrderCore.parseUtterance(line.substring(4), menu);
                    List<String> parts = new ArrayList<>();
                    for (var e : h.items())
                        parts.add(e.getValue() + " x "
                                + menu.get(e.getKey()).nameEn());
                    String heard = String.join(", ", parts);
                    System.out.println("  heard: " + h.intent() + " ["
                        + (heard.isEmpty() ? "nothing" : heard) + "] confidence "
                        + h.confidence());
                    if (h.items().isEmpty()) {
                        System.out.println(
                            "  -> Sorry, which menu item? Please say its name.");
                        continue;
                    }
                    if (h.confidence() < OrderCore.CONFIRM_BELOW) {
                        // Low confidence: ask before acting (explicit confirm).
                        pending = h;
                        System.out.println("  -> Did you mean " + heard
                            + "? Say 'yes' to confirm.");
                        continue;
                    }
```

Listing 9.5 (Continued) *Cli.java: the same command and voice front end in Java.*

```
                }
                for (var e : h.items()) {
                    if (h.intent().equals("remove"))
                        cart.remove(e.getKey(), e.getValue());
                    else
                        cart.add(e.getKey(), e.getValue());
                }
                System.out.println("  -> OK. Total "
                                   + OrderCore.rupiah(cart.total()));
                continue;
            }
            try {
                String[] c = OrderCore.parseCommand(line);
                int n = Integer.parseInt(c[2]);
                if (c[0].equals("add")) cart.add(c[1], n);
                else if (c[0].equals("remove")) cart.remove(c[1], n);
                else for (var l : cart.lines("en"))
                    System.out.printf("  %2d x %-14s%10s%n", l.qty(), l.name(),
                                      OrderCore.rupiah(l.subtotal()));
                System.out.println("  total " + OrderCore.rupiah(cart.total()));
            } catch (IllegalArgumentException e) {
                System.out.println("  error: " + e.getMessage());
            }
        }
    }
}
```

The session file mixes both styles and includes three deliberate troubles: a command with a bad quantity, an utterance with no dish in it, and an utterance whose dish is clear but whose other words are not.

Listing 9.6 `session.txt`: a scripted session with commands, utterances and three deliberate troubles.

```
add esteh 2
add nasgor
add kopi x
say tolong nasi goreng dua sama es teh satu ya
say minta yang anget anget dong
say kopi item yang kentel banget
say ya
remove esteh 1
show
```

9.5.3 The touch kiosk

The kiosk is a web page with one large button per dish and a minus button per order line. Every button is a tiny HTML form that posts to `/add`, `/remove` or `/pay`; the server changes the cart and redirects back to the page (the *post/redirect/get* pattern, which stops a reload from ordering twice). Because it is plain HTML with no script, it works on the cheapest phone, with a keyboard and with a screen reader. The page template, shared by both servers, sets the sizes that matter: menu buttons at least 64 pixels high, minus buttons 48 by 48, the pay button 56 high, and a visible amber focus ring for keyboard users. The first version, **Listing 9.7**, also has a small language link at the bottom, drawn at 70% of the text size — a very common kind of target on real sites, and a deliberate defect that the audit will find.

Listing 9.7 `kiosk_v1.html`: the first version of the kiosk page template, with the placeholders that the servers fill in.

```
<!DOCTYPE html>
<html lang="{{LANG}}">
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>{{TITLE}}</title>
<style>
body { font-family: "DejaVu Sans", sans-serif; margin: 0; color: #1f2937;
       background: #f3f4f6; }
header { background: #12355B; color: #fff; padding: 12px 16px; }
h1 { font-size: 1.3rem; margin: 0; }
main { padding: 12px 16px; }
.menu { display: grid; grid-template-columns: 1fr 1fr; gap: 10px; }
.menu button { width: 100%; min-height: 64px;
       border: 2px solid #0F7C7C; border-radius: 12px; background: #fff; }
.menu button span { display: block; color: #0F7C7C; font-weight: bold; }
ul { list-style: none; padding: 0; }
li { display: flex; justify-content: space-between; align-items: center;
     background: #fff; margin: 6px 0; padding: 8px; border-radius: 8px; }
.minus { min-width: 48px; min-height: 48px; font-size: 1.4rem;
         border: 2px solid #C0392B; border-radius: 50%; background: #fff; }
.total { font-size: 1.2rem; font-weight: bold; }
.pay { width: 100%; min-height: 56px; font-size: 1.1rem; color: #fff;
       background: #0F7C7C; border: 0; border-radius: 12px; }
.tiny { font-size: 0.7rem; }
button { font: inherit; }
button:focus-visible { outline: 3px solid #E0A100; outline-offset: 2px; }
</style>
</head>
<body>
<header><h1>{{TITLE}}</h1></header>
<main>
<h2>{{MENU}}</h2>
<div class="menu">{{ITEMS}}</div>
<h2>{{ORDER}}</h2>
<ul>{{LINES}}</ul>
<p class="total">{{TOTAL}}</p>
<form method="post" action="/pay"><input type="hidden" name="lang" value="{{LANG}}">
<button class="pay" type="submit">{{PAY}}</button></form>
<p><a class="tiny" href="/?lang={{OTHER}}">{{SWITCH}}</a></p>
</main>
</body>
</html>
```

The Flask server in **Listing 9.8** fills the template: it builds the menu buttons and the order lines, escapes every name, and swaps English and Indonesian texts with the `lang` parameter. Each minus button carries an `aria-label` such as “Remove one: Iced tea”, so that a screen reader announces more than a bare “minus”.

Listing 9.8 `kiosk.py`: the direct-manipulation kiosk served by Flask on port 4090.

```
"""Direct-manipulation kiosk: tap a dish to add it, tap minus to remove."""
import sys
from html import escape
from flask import Flask, redirect, request
from order_core import DATA, Cart, load_menu, rupiah

TEXT = {"en": {"TITLE": "Warung Bu Tini", "MENU": "Tap to add",
               "ORDER": "Your order", "PAY": "Pay", "TOTAL": "Total",
               "SWITCH": "Bahasa Indonesia", "EMPTY": "Nothing yet",
               "LESS": "Remove one"},
        "id": {"TITLE": "Warung Bu Tini", "MENU": "Ketuk untuk menambah",
               "ORDER": "Pesanan Anda", "PAY": "Bayar", "TOTAL": "Total",
               "SWITCH": "English", "EMPTY": "Belum ada pesanan",
               "LESS": "Kurangi satu"}}
app = Flask(__name__)
menu = load_menu()
cart = Cart(menu)
name = sys.argv[1] if len(sys.argv) > 1 else "kiosk.html"
template = (DATA / name).read_text(encoding="utf-8")


def page(lang):
    t = TEXT[lang]
    items = "".join(
        f'<form method="post" action="/add"><input type="hidden" name="lang" '
        f'value="{lang}"><button name="code" value="{c}">'
        f'{escape(m["name_" + lang])}<span>{rupiah(m["price"])}</span>'
        f"</button></form>" for c, m in menu.items())
    lines = "".join(
        f'<li><span>{n} x {escape(name)} = {rupiah(sub)}</span>'
        f'<form method="post" action="/remove"><input type="hidden" '
        f'name="lang" value="{lang}"><button class="minus" name="code" '
        f'value="{c}" aria-label="{t["LESS"]}: {escape(name)}">-</button>'
        f"</form></li>" for c, name, n, sub in cart.lines(lang))
    html = template
    for key, val in {**t, "LANG": lang, "OTHER": "en" if lang == "id" else "id",
                     "ITEMS": items, "LINES": lines or f"<li>{t['EMPTY']}</li>",
                     "TOTAL": f"{t['TOTAL']}: {rupiah(cart.total())}"}.items():
        html = html.replace("{{" + key + "}}", val)
    return html


def lang_of(values):
    return "id" if values.get("lang") == "id" else "en"
```

Listing 9.8 (Continued) *kiosk.py: the direct-manipulation kiosk served by Flask on port 4090.*

```
@app.get("/")
def home():
    return page(lang_of(request.args))


@app.post("/add")
def add():
    cart.add(request.form["code"])
    return redirect("/?lang=" + lang_of(request.form), code=303)


@app.post("/remove")
def remove():
    cart.remove(request.form["code"])
    return redirect("/?lang=" + lang_of(request.form), code=303)


@app.post("/pay")
def pay():
    cart.qty.clear()
    return redirect("/?lang=" + lang_of(request.form), code=303)


if __name__ == "__main__":
    app.run(port=4090)
```

The Java server in **Listing 9.9** uses only the standard library: `com.sun.net.httpserver.HttpServer`, a small form decoder and the same template. It listens on port 4091, and for the same cart it produces byte-for-byte the same page as the Python server — a fact the tests of Section 9.7 rely on.

Listing 9.9 `Kiosk.java`: the same kiosk with the JDK's built-in HTTP server on port 4091.

```
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.URLDecoder;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.util.*;

/** Direct-manipulation kiosk: tap a dish to add it, tap minus to remove. */
public class Kiosk {
    static final Map<String, Map<String, String>> TEXT = Map.of(
        "en", Map.of("TITLE", "Warung Bu Tini", "MENU", "Tap to add",
            "ORDER", "Your order", "PAY", "Pay", "TOTAL", "Total",
            "SWITCH", "Bahasa Indonesia", "EMPTY", "Nothing yet",
            "LESS", "Remove one"),
        "id", Map.of("TITLE", "Warung Bu Tini", "MENU", "Ketuk untuk menambah",
            "ORDER", "Pesanan Anda", "PAY", "Bayar", "TOTAL", "Total",
            "SWITCH", "English", "EMPTY", "Belum ada pesanan",
            "LESS", "Kurangi satu"));
    static Map<String, OrderCore.Item> menu;
    static OrderCore.Cart cart;
    static String template;

    static String esc(String s) {
        return s.replace("&", "&amp;").replace("<", "&lt;")
                .replace(">", "&gt;").replace("\"", "&quot;");
    }

    static String page(String lang) {
        var t = TEXT.get(lang);
        StringBuilder items = new StringBuilder(), lines = new StringBuilder();
        for (var m : menu.values())
            items.append("<form method=\"post\" action=\"/add\">"
                + "<input type=\"hidden\" name=\"lang\" value=\"" + lang
                + "\"><button name=\"code\" value=\""
                + m.code() + "\">" + esc(m.name(lang)) + "<span>"
                + OrderCore.rupiah(m.price()) + "</span></button></form>");
        for (var l : cart.lines(lang))
            lines.append("<li><span>" + l.qty() + " x " + esc(l.name()) + " = "
                + OrderCore.rupiah(l.subtotal()) + "</span><form method=\"post\" "
                + "action=\"/remove\"><input type=\"hidden\" name=\"lang\" value=\""
                + lang + "\"><button class=\"minus\" name=\"code\" value=\""
                + l.code() + "\" aria-label=\"" + t.get("LESS") + ": "
                + esc(l.name()) + "\">-</button></form></li>");
        Map<String, String> v = new HashMap<>(t);
```

Listing 9.9 (Continued) *Kiosk.java: the same kiosk with the JDK's built-in HTTP server on port 4091.*

```
        v.put("LANG", lang);
        v.put("OTHER", lang.equals("id") ? "en" : "id");
        v.put("ITEMS", items.toString());
        v.put("LINES", lines.length() > 0 ? lines.toString()
                                          : "<li>" + t.get("EMPTY") + "</li>");
        v.put("TOTAL", t.get("TOTAL") + ": " + OrderCore.rupiah(cart.total()));
        String html = template;
        for (var e : v.entrySet())
            html = html.replace("{{" + e.getKey() + "}}", e.getValue());
        return html;
    }

    static Map<String, String> form(String body) {
        Map<String, String> f = new HashMap<>();
        if (body == null || body.isEmpty()) return f;
        for (String pair : body.split("&")) {
            String[] kv = pair.split("=", 2);
            f.put(URLDecoder.decode(kv[0], StandardCharsets.UTF_8),
                  kv.length > 1 ? URLDecoder.decode(kv[1], StandardCharsets.UTF_8)
                                : "");
        }
        return f;
    }

    static void handle(HttpExchange ex) throws IOException {
        String path = ex.getRequestURI().getPath();
        if (ex.getRequestMethod().equals("POST")) {
            var f = form(new String(ex.getRequestBody().readAllBytes(),
                                    StandardCharsets.UTF_8));
            String lang = "id".equals(f.get("lang")) ? "id" : "en";
            switch (path) {
                case "/add" -> cart.add(f.get("code"), 1);
                case "/remove" -> cart.remove(f.get("code"), 1);
                case "/pay" -> cart.qty.clear();
                default -> { }
            }
            ex.getResponseHeaders().set("Location", "/?lang=" + lang);
            ex.sendResponseHeaders(303, -1);  // back to the page after a POST
            ex.close();
            return;
        }
        var q = form(ex.getRequestURI().getRawQuery());
        byte[] body = page("id".equals(q.get("lang")) ? "id" : "en")
                          .getBytes(StandardCharsets.UTF_8);
        ex.getResponseHeaders().set("Content-Type", "text/html; charset=utf-8");
        ex.sendResponseHeaders(200, body.length);
```

Listing 9.9 (Continued) *Kiosk.java: the same kiosk with the JDK's built-in HTTP server on port 4091.*

```
        ex.getResponseBody().write(body);
        ex.close();
    }

    public static void main(String[] args) throws IOException {
        menu = OrderCore.loadMenu();
        cart = new OrderCore.Cart(menu);
        String name = args.length > 0 ? args[0] : "kiosk.html";
        template = Files.readString(OrderCore.DATA.resolve(name));
        HttpServer server = HttpServer.create(new InetSocketAddress(4091), 0);
        server.createContext("/", Kiosk::handle);
        server.start();
        System.out.println("Kiosk on http://127.0.0.1:4091/");
    }
}
```

9.5.4 Measuring every target in a real browser

Target sizes should be measured, not guessed, because CSS, fonts and the phone's width decide the size a finger actually meets. The audit program in **Listing 9.10** uses Playwright to open the kiosk in a real Chromium browser with a 360-pixel-wide viewport (a common Android width), taps "Iced tea" twice and "Fried rice" once as a customer would, and then asks the browser for the rectangle of every link, button and form field. It prints each target's width and height and whether the smaller side is below 24, 44 or 48 pixels, and it can save a full-page screenshot. There is one audit program, written in Python, because it drives a browser; it audits the Java and the Python server alike.

Listing 9.10 `audit_targets.py`: measures every tappable element of the running kiosk at phone width and compares it with the three guidelines.

```
"""Measure every tappable element of a page against target-size guidelines."""
import sys
from playwright.sync_api import sync_playwright

RULES = [("WCAG 2.5.8 (AA)", 24), ("WCAG 2.5.5 / Apple", 44),
         ("Material", 48)]
JS = """() => [...document.querySelectorAll('a, button, input, select')]
  .filter(e => e.type !== 'hidden')
  .map(e => { const r = e.getBoundingClientRect();
    return [e.tagName.toLowerCase(),
            (e.getAttribute('aria-label') || e.innerText || '').trim(),
            Math.round(r.width), Math.round(r.height)]; })"""


def audit(url, shot=None, actions=()):
    with sync_playwright() as p:
        browser = p.chromium.launch()
        pg = browser.new_page(viewport={"width": 360, "height": 740},
                              device_scale_factor=2)
        pg.goto(url)
        for selector in actions:  # tap some buttons first, like a customer
            with pg.expect_navigation():
                pg.click(selector)
        rows = pg.evaluate(JS)
        if shot:
            pg.screenshot(path=shot, full_page=True)
        browser.close()
    fails = {name: 0 for name, _ in RULES}
    for tag, label, w, h in rows:
        size = min(w, h)
        bad = [name for name, px in RULES if size < px]
        for name in bad:
            fails[name] += 1
        label = label.replace("\n", " ")[:22]
        print(f"{tag:<7}{label:<23}{w:>4} x {h:<4}"
              + ("below " + bad[0] if bad else "ok"))
    print(f"{len(rows)} targets; smaller than "
          + ", ".join(f"{px} px: {fails[n]}" for n, px in RULES))


if __name__ == "__main__":
    url = sys.argv[1] if len(sys.argv) > 1 else "http://127.0.0.1:4090/"
    audit(url, sys.argv[2] if len(sys.argv) > 2 else None,
          ['button[value="esteh"]', 'button[value="esteh"]',
           'button[value="nasgor"]'])
```

9.6 Running It

Run the session through the Python front end with `python cli.py ../data/session.txt`; the Java front end, `java -cp out Cli ../data/session.txt`, prints exactly the same lines:

```
> add esteh 2
  total Rp10.000
> add nasgor
  total Rp25.000
> add kopi x
  error: quantity must be a number: x
> say tolong nasi goreng dua sama es teh satu ya
  heard: add [2 x Fried rice, 1 x Iced tea] confidence 1.0
  -> OK. Total Rp60.000
> say minta yang anget anget dong
  heard: add [nothing] confidence 0.6
  -> Sorry, which menu item? Please say its name.
> say kopi item yang kentel banget
  heard: add [1 x Coffee] confidence 0.4
  -> Did you mean 1 x Coffee? Say 'yes' to confirm.
> say ya
  -> OK. Total Rp66.000
> remove esteh 1
  total Rp61.000
> show
   2 x Iced tea        Rp10.000
   3 x Fried rice      Rp45.000
   1 x Coffee           Rp6.000
  total Rp61.000
```

Read the transcript as a conversation. The two plain commands work and show the running total after each. The command `add kopi x` is rejected with a message that names the problem — the command style's typical failure, a syntax error. The first utterance, "*tolong nasi goreng dua sama es teh satu ya*", is understood completely: two dishes, quantities after the dishes, filler words recognized, confidence 1.0, so the order goes through with only an implicit confirmation (the new total). The second, "*minta yang anget anget dong*" ("something warm, please"), contains no dish; the system does not guess but asks for a menu name. The third, "*kopi item yang kentel banget*" ("black coffee, very strong"), names coffee but has two words the parser does not know, so the confidence is $c = 2/5 = 0.4$ by Equation (9.3): below the threshold, the system asks "Did you mean 1 x Coffee?", and the customer's "*ya*" completes it. Bu Tini's "*Dua, ya?*" has become a rule in code.

Now start the first version of the kiosk (`python kiosk.py kiosk_v1.html`, or `java -cp out Kiosk kiosk_v1.html`) and run the audit in a second terminal:

```
button Iced tea Rp5.000        159 x 64  ok
button Coffee Rp6.000          159 x 64  ok
button Fried rice Rp15.000     159 x 64  ok
button Fried noodles Rp13.000  159 x 64  ok
button Fried tempeh Rp2.000    159 x 64  ok
button Remove one: Iced tea     48 x 48  ok
button Remove one: Fried rice   48 x 48  ok
button Pay                     328 x 56  ok
a      Bahasa Indonesia         99 x 13  below WCAG 2.5.8 (AA)
9 targets; smaller than 24 px: 1, 44 px: 1, 48 px: 1
```

Eight of the nine targets are comfortably large. The menu buttons are 159 by 64 pixels — two columns across a 360-pixel screen — and the minus buttons exactly 48 by 48. The language link is 99 by 13: below all three sizes. Is that a WCAG failure? Not necessarily at level AA, and this is worth being precise about. Criterion 2.5.8 has a *spacing* exception: an undersized target passes if

a circle of 24 pixels' diameter centered on it does not intersect another target or another target's circle [7]. The link sits alone at the bottom of the page, so it probably passes 2.5.8 by spacing. But it fails the 44-pixel criterion 2.5.5 and both platform guidelines, it is the one target that older customers with less precise fingers will miss, and by the Example in Section 9.3.3 it is almost two bits harder to hit than a menu button. It should be fixed.

9.7 Testing It

The core deserves tests of its own, because every front end trusts it. **Listing 9.11** checks the totals, what happens when more is removed than was ordered, numbers before and after the dish, the cancel intent, the low-confidence case and the bad quantity; **Listing 9.12** checks the same things in Java without any test library.

Listing 9.11 `test_order_core.py`: tests of the order core for pytest.

```
"""Plain checks of the order core (run: python -m pytest -q)."""
import pytest
from order_core import (CONFIRM_BELOW, Cart, load_menu, parse_command,
                        parse_utterance)

menu = load_menu()


def test_total_and_remove():
    cart = Cart(menu)
    cart.add("esteh", 2)
    cart.add("nasgor")
    assert cart.total() == 25000
    cart.remove("esteh", 5)
    assert "esteh" not in cart.qty


def test_numbers_before_and_after():
    assert parse_utterance("tolong nasi goreng dua", menu)[1] == [("nasgor", 2)]
    assert parse_utterance("tiga es teh", menu)[1] == [("esteh", 3)]


def test_intent_and_confidence():
    assert parse_utterance("batal kopi", menu)[0] == "remove"
    assert parse_utterance("kopi item yang kentel banget", menu)[2] < CONFIRM_BELOW


def test_bad_quantity():
    with pytest.raises(ValueError):
        parse_command("add kopi x")
```

Listing 9.12 `OrderCoreTest.java`: the same checks in plain Java.

```java
import java.util.*;

/** Plain checks of the order core (run: java -cp out OrderCoreTest). */
public class OrderCoreTest {
    static int failed = 0;

    static void check(boolean ok, String what) {
        System.out.println((ok ? "PASS " : "FAIL ") + what);
        if (!ok) failed++;
    }

    public static void main(String[] args) throws Exception {
        var menu = OrderCore.loadMenu();
        var cart = new OrderCore.Cart(menu);
        cart.add("esteh", 2);
        cart.add("nasgor", 1);
        check(cart.total() == 25000, "total of 2 iced tea + 1 fried rice");
        cart.remove("esteh", 5);
        check(!cart.qty.containsKey("esteh"), "removing too many deletes line");
        var h = OrderCore.parseUtterance("tolong nasi goreng dua", menu);
        check(h.items().get(0).getValue() == 2, "number after the dish");
        h = OrderCore.parseUtterance("tiga es teh", menu);
        check(h.items().get(0).getValue() == 3, "number before the dish");
        h = OrderCore.parseUtterance("batal kopi", menu);
        check(h.intent().equals("remove"), "cancel word gives remove intent");
        h = OrderCore.parseUtterance("kopi item yang kentel banget", menu);
        check(h.confidence() < OrderCore.CONFIRM_BELOW, "unclear words: confirm");
        try {
            OrderCore.parseCommand("add kopi x");
            check(false, "bad quantity rejected");
        } catch (IllegalArgumentException e) {
            check(true, "bad quantity rejected");
        }
        System.out.println(failed == 0 ? "all passed" : failed + " failed");
    }
}
```

`python -m pytest -q` reports:

```
4 passed in 0.01s
```

and `java -cp out OrderCoreTest` prints:

```
PASS total of 2 iced tea + 1 fried rice
PASS removing too many deletes line
PASS number after the dish
PASS number before the dish
PASS cancel word gives remove intent
PASS unclear words: confirm
PASS bad quantity rejected
all passed
```

Two further checks are worth doing by hand once. Fetch the page from both servers with an empty cart and compare the bytes (`curl -s http://127.0.0.1:4090/?lang=id | md5sum`, and the same for port 4091): the two hashes are equal, so the audit of one server is valid for the other. And run the audit against the Java server: it prints the same nine lines as against Flask.

9.8 Making It Better

9.8.1 A target that everyone can hit

The fix for the language link is to give it a real touch area without making the text large: display it as an inline block with padding and a minimum height of 48 pixels. The final template `kiosk.html` changes one CSS rule and one class name:

```
.lang { display: inline-block; min-height: 48px; padding: 12px 16px;
        box-sizing: border-box; }
<p><a class="lang" href="/?lang={{OTHER}}">{{SWITCH}}</a></p>
```

Start either server without an argument (it now loads `kiosk.html`) and run the audit again, this time also saving a screenshot:

```
button Iced tea Rp5.000        159 x 64  ok
button Coffee Rp6.000          159 x 64  ok
button Fried rice Rp15.000     159 x 64  ok
button Fried noodles Rp13.000  159 x 64  ok
button Fried tempeh Rp2.000    159 x 64  ok
button Remove one: Iced tea     48 x 48  ok
button Remove one: Fried rice   48 x 48  ok
button Pay                     328 x 56  ok
a      Bahasa Indonesia        173 x 48  ok
9 targets; smaller than 24 px: 0, 44 px: 0, 48 px: 0
```

Every target now meets all three guidelines. **Figure 9.2** is the screenshot the audit took of the running kiosk after the three taps.

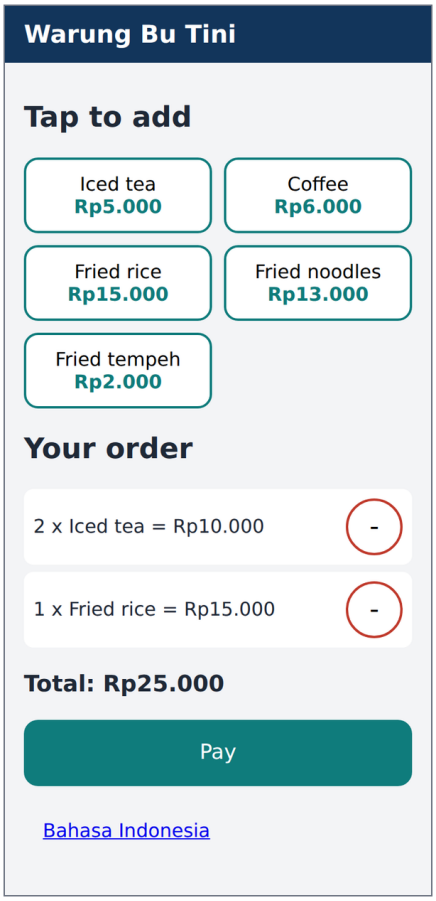


Figure 9.2 The final kiosk at a width of 360 CSS pixels after tapping “Iced tea” twice and “Fried rice” once: two columns of 64-pixel dish buttons, a 48-pixel minus button on each order line, and a language link with a 48-pixel touch area.

9.8.2 Asking at the right moment

The confirmation rule of Section 9.5.2 is deliberately simple, and running more utterances through it shows its limits. "*es teh manis dua gelas*" ("two glasses of sweet iced tea") gives one iced tea with confidence 0.6 — just at the threshold, so it is carried out, and it is *wrong*, because the word *dua* is separated from the dish by *manis*. "*berapa totalnya*" ("what is the total?") is not recognized as a question about the total, because the parser knows *total* but not *totalnya*, the same word with the Indonesian suffix *-nya*. Both failures teach something general. A confidence score measures how much was understood, not whether the understanding is correct; and a voice interface for Indonesian users must cope with affixes and with the mixing of Indonesian, Javanese and English that real customers use. Exercises 9.4 and 9.5 improve both. For costly actions — paying, in a real kiosk — the right rule is stricter: always confirm explicitly, whatever the confidence.

9.9 Using It for Real

The same pattern — one core, several front ends, measured targets — adapts to many small businesses.

- **Your own menu.** Replace `menu.csv` with your dishes, prices and, most importantly, the words your customers actually say. Listen at the counter for a week and write the aliases down; that list *is* the voice interface.
- **More front ends.** A WhatsApp or Telegram bot is a natural-language front end: feed each incoming message to `parse_utterance` and reply with the confirmation text. A real speech recognizer (on-device on Android and iOS, or a local open-source model) is another; its text output goes into the same parser, and its own confidence can be combined with Equation (9.3).
- **Older customers.** Keep the helper at the counter as a front end too. An interface that forces everyone to use a tablet has chosen a style *against* some of its users; the kiosk should shorten the queue, not replace the person.
- **Before you print the QR code.** Run the audit at 320 and 360 pixels wide, on the cheapest phone you can find, in sunlight, with a thumb. Automated measurements catch sizes; only people catch everything else.
- **Payment.** Anything that moves money needs explicit confirmation, a clear total and an undo path until the moment of payment. Chapter 12 discusses why hiding these, even by accident, destroys trust.

9.10 HCI in Practice: Profitina

Profitina, an Indonesian AI-visibility (GEO) platform that measures how ready a brand's website is to be cited by AI answer engines, uses several interaction styles in one product, and the choices follow this chapter's reasoning. The **dashboard** is designed as a cockpit: a single request gathers notifications (replies, likes, reposts, direct messages, pins, feedback status, purchases), followed channels, rank and activity, and every item is clickable straight to its section. That is direct manipulation applied to a stream of events: the objects of interest are visible, and acting on one is a single tap rather than a navigation through menus.

The **community composer** follows the 0.1-second limit of Section 9.3.2 with *optimistic posting*: when the user presses send, the composer clears at once and the post appears immediately, while the network request happens afterwards; if sending fails, the text is restored so that nothing

typed is lost. The user sees the effect within the time that feels instantaneous, and the rare failure is reversible. The **"Share to X"** action is deliberately manual and per item, with a queue and a daily quota: an action with public consequences stays under explicit human control, the same principle as the explicit confirmation of a costly voice command. And the **content engine**, which drafts GEO content with a locally run Qwen2.5 model, presents drafts for the user to edit rather than publishing them: natural language is used to *propose*, and direct manipulation to *decide*. The hands-on practice of this book uses a laundry case study instead of the product itself; Chapter 16 builds it.

9.11 Common Mistakes, and How to Avoid Them

- **One style for everyone.** A touch-only kiosk shuts out customers who cannot or will not use it; a voice-only service shuts out deaf users and noisy places. Offer a second path (Step 5 of **Table 9.3**).
- **Gestures without alternatives.** Swipe-only delete, pinch-only zoom and drag-only reordering are invisible to many users and fail WCAG 2.2 criteria 2.5.1 and 2.5.7. Add a visible button.
- **Measuring targets by eye.** A link that looks fine on a designer's large monitor is 13 pixels high on a phone. Measure in a browser at phone width, as **Listing 9.10** does.
- **Confusing the spacing exception with a pass.** An undersized target that passes 2.5.8 by spacing may still be hard to hit; aim for 44–48 pixels for anything important.
- **Guessing instead of asking.** A voice interface that acts on a low-confidence interpretation will order the wrong dish; one that confirms everything will annoy. Confirm when unsure or when the action is costly.
- **Treating confidence as correctness.** A high share of understood words does not mean the right quantity was read, as "*es teh manis dua gelas*" shows. Test the parser with real utterances from real customers.
- **Logic in the front end.** If prices or rules are computed separately in each front end, the styles will disagree. Keep one core.
- **Slow feedback.** A button that shows nothing for a second feels broken. Show the effect within 0.1 s and do the slow work afterwards.

9.12 Chapter Summary

- An interaction style is how a user expresses intentions and sees results. The five primary styles are command language, menu selection, form fill-in, direct manipulation and natural language; touch, gesture, voice and gaze are modalities that carry them (**Table 9.1**).
- Recognition (menus, visible objects) is easier than recall (commands); recall is faster for experts. Direct manipulation narrows both of Norman's gulfs by keeping objects visible and actions rapid, incremental and reversible.
- Interfaces are event-driven. Feedback within about 0.1 s feels instantaneous, within 1 s keeps the flow of thought, and beyond 10 s loses attention.
- Fitts's law (Equation (9.1)) makes small targets slow; FFitts law (Equation (9.2)) adds the finger's absolute imprecision, which makes very small touch targets error-prone. WCAG 2.2 requires 24 × 24 CSS px at level AA (with exceptions) and 44 × 44 at AAA; Apple recommends 44 pt, Material 48 dp (**Table 9.2**).
- Gestures need visible alternatives; WCAG 2.2 criteria 2.5.1 and 2.5.7 require single-pointer alternatives to path gestures and dragging.

- Voice interfaces lack visible menus, are uncertain and are fleeting. Confirm implicitly when confident, explicitly when unsure (Equation (9.3)) or when the action is costly, and always offer an example and an undo.
- The program built one order core with three front ends (**Figure 9.1**), a kiosk served identically by Flask and by the JDK's HTTP server, and a browser-based audit that found and verified the fix of an undersized link (**Figure 9.2**).

9.13 Exercises

1. Using Equation (9.1), compute the index of difficulty of targets 24, 44 and 48 pixels wide at a distance of 300 pixels. How many bits does moving from 24 to 48 pixels save, and why is the saving smaller than moving from 13 to 24?
2. Classify each step of withdrawing cash at an ATM with side buttons (insert card, choose language, enter PIN, choose "withdrawal", choose an amount, confirm) by interaction style from **Table 9.1**. Which step would you change to direct manipulation on a touch-screen ATM, and which would you keep as it is?
3. For each of the following, say whether you would use implicit or explicit confirmation, and write the sentence the system says: (a) adding one iced tea by voice with confidence 1.0; (b) paying Rp61.000 by voice; (c) cancelling a whole order by voice.
4. The utterance "*es teh manis dua gelas*" is misunderstood (Section 9.8.2). Change `parse_utterance` so that a number word within two words *after* a dish is taken as its quantity, and so that the words *manis*, *gelas* and *porsi* count as understood. Show the new result for this utterance and check that the tests still pass.
5. Make "*berapa totalnya*" work: extend the parser so that a word ending in *-nya* is also looked up without the suffix. Which other Indonesian affixes might a warung's customers use?
6. What happens to the session of **Listing 9.6** if `CONFIRM_BELOW` is changed from 0.6 to 0.5? To 0.3? Answer by reasoning from the confidences in the transcript, then check by running.
7. A map in a delivery app can be zoomed only by pinching, and saved addresses can be reordered only by dragging. Which WCAG 2.2 criteria does it fail, and what is the smallest change that makes it pass?

9.14 Solutions to the Exercises

1. $ID_{24} = \log_2(300/24 + 1) = \log_2(13.5) = 3.75$ bits; $ID_{44} = \log_2(300/44 + 1) = \log_2(7.82) = 2.97$ bits; $ID_{48} = \log_2(300/48 + 1) = \log_2(7.25) = 2.86$ bits. Moving from 24 to 48 pixels saves $3.75 - 2.86 = 0.89$ bits. Moving from 13 to 24 pixels saves $4.59 - 3.75 = 0.84$ bits although the width grows by only 11 pixels: because the index is logarithmic, each *doubling* of the width saves about the same (a little under one bit when D/W is large), so gains are largest for the smallest targets. In addition, by Equation (9.2), the 13-pixel target sits near the finger's absolute imprecision, where errors — not just time — rise.

2. Inserting the card is a physical action (direct manipulation of an object in the world). Choosing the language, choosing "withdrawal" and choosing an amount are menu selections, made with the side buttons. Entering the PIN is form fill-in (one field) on a keypad. Confirming is a menu selection with two options. On a touch-screen ATM, choosing the amount benefits most from direct manipulation: large buttons for common amounts plus an "other amount" field, with the result (the amount and the fee) visible before confirming. The PIN entry should stay on a physical or shielded keypad for privacy; a gesture or voice PIN would be overheard or overseen.

3. (a) Implicit, because the system is confident and the action is cheap and undoable: "Iced tea added — total Rp5.000. Anything else?" (b) Explicit, because paying is costly and hard to undo, whatever the confidence: "You are about to pay Rp61.000 for 2 iced tea, 3 fried rice and 1 coffee. Say 'pay' to confirm or 'cancel'." (c) Explicit, because it deletes everything: "Cancel the whole order of 6 items? Say 'yes, cancel' to confirm."

4. Two changes to `order_core.py`: add `"manis", "gelas", "porsi"` to `FILLER`, and replace the look-ahead for a number after the dish by a search over the next two words:

```
            elif any(w in NUMBERS for w in words[j:j + 2]):
                k = next(k for k in range(j, j + 2) if words[k] in NUMBERS)
                n = NUMBERS[words[k]]
                known[k] = True
                j = k + 1
```

With these changes `parse_utterance("es teh manis dua gelas", menu)` returns `('add', [('esteh', 2)], 0.8)`: the quantity is now 2. The confidence is $4/5 = 0.8$ rather than 1.0 because the scan jumps from the dish straight past *manis* to the number, so *manis* is never looked up; marking the skipped word as understood when it is a filler word would raise the confidence to 1.0. The four existing tests still pass, because in "*tolong nasi goreng dua*" the number is the very next word. The same change in `OrderCore.java` replaces the `else if` branch by a loop over `j` and `j + 1`. Note the risk this creates: in "*nasi goreng es teh dua*" the *dua* is three words after *nasi goreng*, outside the window, so the parser gives one fried rice and two iced teas, which is what the customer meant; but in "*nasi goreng teh dua*" the window reaches *dua* from the fried rice, and the parser returns two fried rice with confidence 0.75 and loses the tea altogether. Ambiguity cannot be removed by a cleverer window, only reduced — and caught by confirmation. Test such cases with real utterances.

5. Before looking a word up in `ADD`, `REMOVE`, `TOTAL` and `FILLER`, strip a final *-nya* when the word is longer than five letters: `if w.endswith("nya") and len(w) > 5: w = w[:-3]`. Then "*totalnya*" becomes *total*, the intent is `total`, and the confidence is 1.0. Other affixes a warung hears include the suffixes *-kan* and *-in* (*pesankan*, *tambahin*), the prefix *di-* (*dibungkus*), reduplication (*anget-anget*) and Javanese words (*ngunjuk*, *mbak*, *nggih*). A stemmer such as the one in the Sastrawi library handles standard Indonesian affixes; regional words have to be collected by listening.

6. The confidences in the transcript are 1.0 (first utterance), 0.6 (no dish), 0.4 (coffee). With 0.5, nothing changes: 0.4 is still below 0.5, so coffee still needs "*ya*", and the no-dish utterance is refused for lack of a dish, not for confidence. With 0.3, the coffee utterance is carried out at once without confirmation, and the following "*say ya*" is then an utterance with no dish, answered with "Sorry, which menu item?". Running the session with `CONFIRM_BELOW = 0.3` prints exactly these lines.

7. Pinch-only zoom fails success criterion 2.5.1 *Pointer Gestures* (a multipoint gesture without a single-pointer alternative); drag-only reordering fails 2.5.7 *Dragging Movements*. The smallest changes: add visible zoom-in and zoom-out buttons to the map, and add "move up" and "move down" buttons (or a "move to top" menu item) to each saved address. Both buttons must themselves be at least 24 by 24 CSS pixels to meet 2.5.8, and 44 or more to be comfortable.

References

[1] ANTARA News, "Survei APJII: Penetrasi internet di Indonesia 2026 capai 81,7 persen," May 21, 2026. [Online]. Available: https://www.antaranews.com/berita/5576225/survei-apjii-penetrasi-internet-di-indonesia-2026-capai-817-persen

[2] B. Shneiderman, C. Plaisant, M. Cohen, S. Jacobs, N. Elmqvist, and N. Diakopoulos, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 6th ed. Boston, MA, USA: Pearson, 2016.

[3] D. A. Norman, *The Design of Everyday Things*, revised and expanded ed. New York, NY, USA: Basic Books, 2013.

[4] J. Nielsen, "Response Times: The 3 Important Limits," Nielsen Norman Group, updated 2014, after J. Nielsen, *Usability Engineering*, ch. 5. San Francisco, CA, USA: Morgan Kaufmann, 1993. [Online]. Available: https://www.nngroup.com/articles/response-times-3-important-limits/

[5] P. M. Fitts, "The information capacity of the human motor system in controlling the amplitude of movement," *Journal of Experimental Psychology*, vol. 47, no. 6, pp. 381–391, 1954.

[6] X. Bi, Y. Li, and S. Zhai, "FFitts law: Modeling finger touch with Fitts' law," in *Proc. CHI '13*, Paris, France, 2013, pp. 1363–1372, doi: 10.1145/2470654.2466180.

[7] W3C, "Web Content Accessibility Guidelines (WCAG) 2.2," W3C Recommendation, Dec. 12, 2024; approved as ISO/IEC 40500:2025, Oct. 2025. [Online]. Available: https://www.w3.org/TR/WCAG22/

[8] Apple Inc., "Accessibility," *Human Interface Guidelines*, accessed Sep. 2026. [Online]. Available: https://developer.apple.com/design/human-interface-guidelines/accessibility

[9] Google, "Accessibility basics: Touch and pointer target size," *Material Design 3*, accessed Sep. 2026. [Online]. Available: https://m3.material.io/foundations/designing/structure

[10] Business Today, "Google Assistant to be replaced by Gemini starting September on Android and Wear OS," Aug. 6, 2026. [Online]. Available: https://www.businesstoday.in/technology/news/story/google-assistant-to-be-replaced-by-gemini-starting-september-on-android-and-wearos-547570-2026-08-06

[11] "I Can Do It: Exploring Voice Assistants for Adults with Intellectual Disabilities," in *Proc. 2026 CHI Conf. Human Factors in Computing Systems (CHI '26)*, 2026, doi: 10.1145/3772318.3790573.

[12] Apple Inc., "New versions of Apple's software platforms are available today," Apple Newsroom, Sep. 15, 2025. [Online]. Available: https://www.apple.com/newsroom/2025/09/new-versions-of-apples-software-platforms-are-available-today/

[13] Google Design, "Expressive Design: Google's UX Research," 2025. [Online]. Available: https://design.google/library/expressive-material-design-google-research

[14] Microsoft, "Fluent 2 Design System," accessed Sep. 2026. [Online]. Available: https://fluent2.microsoft.design/

[15] Apple Inc., "Designing for visionOS," *Human Interface Guidelines*, accessed Sep. 2026. [Online]. Available: https://developer.apple.com/design/human-interface-guidelines/designing-for-visionos

Chapter 10
Mobile and Responsive UX

Most people who will ever use your interface will hold it in one hand. In Indonesia they will often hold a budget Android phone, in the sun, on a connection that is fast in the city center and slow a few kilometers away, while doing something else. This chapter is about designing for that hand, that screen and that network. It follows the booking page of a small ojek base whose first version was designed on a desktop computer and was almost unusable on the drivers' and customers' phones, and rebuilds it mobile-first. By the end of the chapter you will be able to read Indonesia's mobile statistics correctly and explain why two respected sources seem to disagree; explain CSS pixels, device pixels and the viewport, and compute how small a desktop page becomes on a phone; build a fluid, mobile-first layout with breakpoints chosen by the content; place important actions within reach of the thumb; make phone forms fast with the right input types and autocomplete tokens; estimate load time from page weight, round-trip time and bandwidth and hold a page to a performance budget; apply the WCAG 2.2 criteria that matter most on phones; and test all of this automatically at four screen widths in a real browser, with a server in Java and in Python.

10.1 Mas Yoga's Phone

Pak Darto runs an ojek base, a *pangkalan*, on Jalan Kenari: a bench under a tree, a few motorcycles and four drivers who take turns. Most rides still begin with someone walking up to the bench, but more and more customers want to book ahead — a mother who needs to be at the puskesmas at eight, a student with a nine o'clock class. Pak Darto's nephew made a small booking web page for the base on his desktop computer. It looked tidy on the monitor: a table of drivers with their plate numbers and status, a form with four fields, and links to call or message the base.

Then Mas Yoga, one of the drivers, opened it on his phone while waiting under the tree. The page appeared as a tiny grey postcard. The text was too small to read without zooming; after zooming, he had to scroll sideways to see the status column; the phone field opened the full letter keyboard instead of the number pad; the "Call" link was so small that his thumb hit "WhatsApp" twice. A customer, Bu Ani, gave up and phoned the base instead — the page had turned a two-minute task into a frustrating one, and a phone call into the only reliable interface.

Nothing on the page was *wrong* in the desktop sense. The failure was one of design assumptions: the page assumed a wide screen, a mouse, a steady desk and a fast wired connection. Mas Yoga had a narrow screen, a thumb, a moving world and a mobile network. This chapter is about replacing those assumptions with the true ones.

10.2 The Idea in Plain Words

A **responsive** interface is one page that adapts its layout to the screen it finds, instead of one page for desktops and a separate "mobile site". The term was coined by Ethan Marcotte in 2010 for a combination of three techniques: fluid grids that are sized in proportions rather than fixed pixels, flexible images, and CSS media queries that change the layout at chosen widths [1]. **Mobile-first**, a term popularized by Luke Wroblewski, reverses the old habit of designing for the desktop and

then squeezing: design for the smallest, most constrained screen first, where only what truly matters fits, and then *add* layout for larger screens [2].

The constraints of a phone are more than size. It is used with one hand and one thumb; its user is interrupted; the sun washes out low-contrast colors; the network varies from fast to nearly absent; and the processor of a budget phone is several times slower than a laptop's. A mobile design takes all of these seriously. The good news is that designing for these constraints almost always improves the desktop version too: a page that is clear on a 360-pixel phone is clear on a 1280-pixel monitor.

> **DEFINITION.** A *responsive* design is a single interface whose layout adapts to the size and capabilities of the device, using fluid proportions, flexible media and conditional rules (media or container queries). *Mobile-first* is the practice of designing the smallest screen and the weakest conditions first and enhancing the design for larger screens and better conditions. A *breakpoint* is a width at which the layout changes.

10.3 The Theory with Evidence

10.3.1 Indonesia's mobile reality, read carefully

Three sources describe Indonesia's digital users, and they measure different things. The survey of the Indonesian Internet Service Providers Association (APJII), released in May 2026, counts *people*: 81.72% of the population, about 235 million, are internet users [3]. StatCounter counts *page views* on the websites that carry its tracking code. For August 2026 it reports that 38.98% of page views in Indonesia came from mobile devices, 60.59% from desktops and 0.43% from tablets [4]; among mobile page views, Android had 77.07% and iOS 22.88% [5]. And Ookla's Speedtest Global Index, as reported by Bloomberg Technoz, gives Indonesia's *median* mobile download speed for February 2026 as 59.18 Mbit/s, with 17.97 Mbit/s upload and a latency of 21 ms [6].

At first sight the StatCounter figure contradicts everyday experience: surely most Indonesians browse on phones? The resolution is in what is counted. StatCounter measures page views on sites that use its tracker, and a desktop user at work may view many pages of such sites in a day, while a phone user may spend the day in apps — chat, video, ride-hailing — that are not web pages at all. So the figure says something true about *web page views on those sites*, not about people. The lesson for designers is general: before quoting a number, ask *what* was counted, *where*, and *how*. For an ojek booking page, whose users are drivers and neighbors with phones, the relevant fact is the one Mas Yoga showed us, and the relevant user research is the kind described in Chapter 5.

The Ookla figure needs the same care. A median of 59 Mbit/s means that half of the measured tests were faster and half slower; it says nothing about the slowest tenth, which is where rural users, crowded cells and indoor signals live. That is why Section 10.3.6 designs against a deliberately slow profile rather than the median.

10.3.2 Pixels, the viewport, and why a desktop page shrinks

A phone's screen has many more *device pixels* than a web page's *CSS pixels*. The browser maps one CSS pixel onto several device pixels, according to the phone's **device pixel ratio** r (2, 3 or more on today's phones). If a screen is w_{dev} device pixels wide, the width available to the page in CSS pixels is

$$w_{css} = \frac{w_{dev}}{r} \tag{10.1}$$

A phone 1080 device pixels wide with $r = 3$ offers a width of 360 CSS pixels. Designers work in CSS pixels, which is why this book speaks of "a 360-pixel phone".

The **viewport** is the rectangle in which the page is laid out. Mobile browsers, to show pages that were designed only for desktops, lay a page out by default in a wide virtual viewport — typically 980 CSS pixels — and then zoom it out to fit the screen. The page can switch this off with one line in its head, `<meta name="viewport" content="width=device-width, initial-scale=1">`, which tells the browser to lay the page out at the device's own width [7]. Without that line, everything on the page is multiplied by the scale factor

$$s = \frac{w_{vp}}{w_{layout}} \tag{10.2}$$

where w_{vp} is the width of the screen in CSS pixels and w_{layout} the width at which the page was laid out. A text or a button that is h_{css} pixels high then appears to the user with an effective size of

$$h_{eff} = s \cdot h_{css} \tag{10.3}$$

> **EXAMPLE.** Pak Darto's first page has no viewport line, so a 360-pixel phone lays it out at 980 pixels. By Equation (10.2), $s = 360/980 = 0.367$. The small print under the form, at 11 CSS pixels, appears by Equation (10.3) as $0.367 \times 11 = 4.04$ pixels — about the height of this sentence's commas. The "Call" link, 12 pixels high, becomes a target of $0.367 \times 12 = 4.4$ pixels. On a 320-pixel phone it is worse: $s = 320/980 = 0.327$ and the small print shrinks to 3.6 pixels.

10.3.3 Fluid layouts and content-chosen breakpoints

A responsive layout starts as a single column that fills the width, with sizes in relative units (`%`, `rem`, `fr`) and `max-width` on the content so that lines do not become too long on a large monitor. **Media queries** such as `@media (min-width: 700px) { … }` then add layout when there is room: two columns of driver cards, a two-column form. Newer **container queries** let a component adapt to the width of its container rather than of the whole window, which suits reusable components such as a card that appears both in a narrow sidebar and in a wide main column [7].

Where should breakpoints go? Not at the widths of particular phone models, which change every year, but where the *content* needs them: widen the window slowly and add a breakpoint where the layout starts to look wrong — lines too long, cards too wide, white space wasted. For the booking page, a single column reads well up to about 700 pixels; above that, driver cards side by side waste less space. The breakpoint follows from the content, not from a device list.

10.3.4 Hands and thumbs

How do people actually hold their phones? Steven Hoober observed 1,333 people using phones in public places and found that 49% held the phone in one hand and operated it with the thumb, 36% cradled it in one hand and used the other hand's finger or thumb, and 15% held it in both hands and typed with both thumbs [8]. People also change grip often, even within one task. Two practical rules follow. First, the most frequent and most important actions should be within easy reach of a thumb, which on a large phone means the lower and middle part of the screen — hence

bottom navigation bars and floating action buttons. Second, destructive or rare actions can sit further away, where they are not hit by accident.

Mas Yoga's two most important actions are "Call" and "WhatsApp". The responsive page puts them in a bar fixed at the bottom of the screen, each button half the width of the screen and 52 pixels high: easy for the thumb in one-handed use, impossible to confuse with each other. Chapter 9's target-size rules (at least 24 CSS pixels at WCAG level AA, 44 to 48 for comfort) apply unchanged.

10.3.5 Forms on phones

Typing on glass is slow and error-prone, so a mobile form should ask for as little as possible and help with the rest. HTML offers three simple helpers [9]. The **input type** chooses the keyboard: `type="tel"` opens the number pad for a phone number, `type="email"` adds the @ key. The **`autocomplete` token** tells the browser what the field means, so that it can offer the user's saved name, telephone or address with one tap: `autocomplete="name"`, `"tel"`, `"street-address"`. The **`inputmode`** attribute chooses a keyboard without changing the kind of field, for example `inputmode="numeric"` for a numeric code. WCAG 2.2 success criterion 1.3.5 *Identify Input Purpose* (level AA) requires that fields collecting information about the user identify their purpose in this machine-readable way [10], because autocompletion helps people with motor and cognitive disabilities most of all.

Labels matter even more on phones than on desktops. A placeholder inside the field disappears as soon as the user starts typing, so the label must be a real `<label>` above the field, connected to it with `for` and `id`; then a tap on the label also focuses the field, and screen readers announce it.

10.3.6 Speed: vitals, round trips and a budget

Google's **Core Web Vitals** summarize the loading, responsiveness and visual stability of a page in three measures, each judged "good" at the 75th percentile of real visits: the **Largest Contentful Paint** (LCP), the time until the largest visible element is drawn, should be at most 2.5 s; the **Interaction to Next Paint** (INP), the delay from an interaction until the screen updates, should be at most 200 ms (INP replaced the older First Input Delay in March 2024); and the **Cumulative Layout Shift** (CLS), how much content jumps around while loading, should be at most 0.1 [11].

How long a page takes to arrive can be estimated with a simple model. Before the first byte comes back, the phone needs several **round trips** to the server: to look up the name (DNS), to open the connection (TCP), to agree on encryption (TLS) and to send the request. If each round trip takes RTT seconds and there are k of them, and the page and everything needed to draw it weigh S kilobytes arriving at a bandwidth of B kilobits per second, then the time is approximately

$$t \approx k \cdot RTT + \frac{8S}{B} \tag{10.4}$$

The factor 8 turns bytes into bits. The model ignores many details — parallel downloads, compression, caching, the time to run scripts — but it captures the two things a designer controls: how many round trips a page needs and how many bytes it sends.

To test pages under realistic stress, the Lighthouse tool that is built into Chrome simulates a mobile visit by default with a round-trip time of 150 ms, a throughput of 1,638.4 kbit/s (about 1.6 Mbit/s) and a processor four times slower than the testing computer [12]. That profile is much slower than Indonesia's median, and deliberately so: a page that is good under it is good for

nearly everyone. A **performance budget** turns this into a design rule: decide, before designing, how many kilobytes a page may weigh so that Equation (10.4) stays within 2.5 s on the chosen profile, and treat every photo, font and script as spending from that budget.

10.3.7 The WCAG criteria that bite hardest on phones

Several WCAG 2.2 criteria [10] exist largely because of small screens and zoom:

- **1.4.10 Reflow (AA).** Content must be usable at a width of 320 CSS pixels without scrolling in two directions (data tables and maps excepted). This equals a 1280-pixel desktop window zoomed to 400%, so it serves low-vision desktop users too.
- **1.4.4 Resize Text (AA).** Text must be resizable to 200% without loss of content or function.
- **1.3.4 Orientation (AA).** Content must not be locked to portrait or landscape unless essential; a driver with the phone in a dashboard holder uses landscape.
- **1.3.5 Identify Input Purpose (AA)**, as above, and **2.5.8 Target Size (Minimum) (AA)**, as in Chapter 9.

10.4 A Hands-On Method: the Mobile-First Checklist

Table 10.1 is a checklist to run on every page before it goes live. The first column is the question, the second how to check it, and the third what Pak Darto's first page scores.

Table 10.1 A mobile-first checklist, with the results for the first version of the booking page.

Check	How to check	First booking page
Viewport line present	Look in the page head	Missing: page zoomed to 37%
Reflow at 320 px	Open at 320 px; no sideways scroll	Zoomed out instead of reflowing
Text at least 16 CSS px for body text	Measure computed sizes × scale	3.6–4.0 px effective on phones
Targets at least 24 px, preferably 44–48	Measure rectangles × scale	3.9–4.4 px effective
Main actions in thumb reach	Look at the lower half of the screen	Call and WhatsApp at the top, tiny
Right keyboards and autocomplete	Check `type` and `autocomplete` of fields	Phone field is plain text
Labels above fields, connected	Tap each label; field focuses	Loose text before each field
Weight within budget	Sum the resources; apply Equation (10.4)	2,492 kB: 12.8 s on the test profile
Works in landscape and at 200% text	Rotate; zoom text	Not checked

Most rows can be automated, and Section 10.5 does so. The two that cannot — whether actions are in thumb reach and whether the page makes sense to its users — need people, the methods of Chapters 5 and 7.

10.5 Building It Step by Step

The program serves two versions of the same booking page, measures them in a real browser at four widths, and estimates their load time on three networks.

10.5.1 Two templates, one set of words

Both versions are HTML templates filled with the same words from a small table of English and Indonesian strings and the same list of drivers. The first version, **Listing 10.1**, is typical of pages designed on a desktop: a fixed body width of 960 pixels, a 13-pixel font, a table, inputs 300 pixels wide with loose text before them, a phone field of plain `text` type, and the call links in 11-pixel print. It has no viewport line.

Listing 10.1 `naive.html`: the first, desktop-only version of the booking page.

```
<!DOCTYPE html>
<html lang="{{LANG}}">
<head>
<meta charset="utf-8">
<title>{{TITLE}}</title>
<style>
body { width: 960px; margin: 0 auto; font: 13px Arial, sans-serif; }
h1 { font-size: 20px; }
table { width: 940px; border-collapse: collapse; }
td, th { border: 1px solid #999; padding: 2px 6px; }
input { width: 300px; font-size: 12px; }
.small { font-size: 11px; }
</style>
</head>
<body>
<h1>{{TITLE}} &ndash; {{TAG}}</h1>
<table><tr><th>{{DRIVERS}}</th><th>{{PLATE}}</th><th>{{STATUS}}</th></tr>
{{ROWS}}</table>
<form method="post" action="/book">
<p>{{NAME}} <input name="name"> {{PHONE}} <input name="phone"></p>
<p>{{FROM}} <input name="from"> {{TO}} <input name="to">
<input type="submit" value="{{BOOK}}"></p>
</form>
<p class="small"><a href="tel:+620000000000">{{CALL}}</a> |
<a href="https://wa.me/620000000000">{{WA}}</a></p>
</body>
</html>
```

The responsive version, **Listing 10.2**, is written mobile-first. The base styles are for a narrow screen: one column, a 16-pixel base font (`1rem`), driver cards instead of a table, labels above full-width inputs 48 pixels high, and a bar fixed at the bottom with the two main actions. One media query adds two columns from 700 pixels up. The phone field has `type="tel"` and `autocomplete="tel"`; the name and pickup fields have their own autocomplete tokens.

Listing 10.2 `responsive.html`: the mobile-first rewrite with a viewport line, fluid layout, labeled fields and a thumb-reach action bar.

```
<!DOCTYPE html>
<html lang="{{LANG}}">
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>{{TITLE}}</title>
<style>
* { box-sizing: border-box; }
body { margin: 0; font: 1rem/1.5 "DejaVu Sans", sans-serif; color: #1f2937;
       background: #f3f4f6; padding-bottom: 88px; }
header { background: #12355B; color: #fff; padding: 12px 16px; }
h1 { font-size: 1.3rem; margin: 0; }
header p { margin: 0; }
main { max-width: 960px; margin: 0 auto; padding: 0 16px; }
.drivers { list-style: none; padding: 0; display: grid; gap: 8px; }
.drivers li { background: #fff; border-radius: 8px; padding: 8px 12px; }
.ready { color: #0F7C7C; font-weight: bold; }
.trip { color: #6B7280; }
label { display: block; margin-top: 12px; font-weight: bold; }
input { width: 100%; min-height: 48px; font-size: 1rem; padding: 8px;
        border: 2px solid #6B7280; border-radius: 8px; }
button { width: 100%; min-height: 52px; margin-top: 16px; font: inherit;
         color: #fff; background: #0F7C7C; border: 0; border-radius: 12px; }
nav { position: fixed; bottom: 0; left: 0; right: 0; display: flex;
      gap: 8px; padding: 12px 16px; background: #fff;
      box-shadow: 0 -2px 6px rgba(0, 0, 0, .15); }
nav a { flex: 1; min-height: 52px; display: flex; align-items: center;
        justify-content: center; border-radius: 12px; font-weight: bold;
        text-decoration: none; color: #12355B; border: 2px solid #12355B; }
@media (min-width: 700px) {
  .drivers { grid-template-columns: repeat(2, 1fr); }
  form { display: grid; grid-template-columns: 1fr 1fr; column-gap: 16px; }
  form button { grid-column: span 2; }
}
</style>
</head>
<body>
<header><h1>{{TITLE}}</h1><p>{{TAG}}</p></header>
<main>
<h2>{{DRIVERS}}</h2>
<ul class="drivers">{{CARDS}}</ul>
<form method="post" action="/book">
<div><label for="name">{{NAME}}</label>
<input id="name" name="name" autocomplete="name"></div>
<div><label for="phone">{{PHONE}}</label>
<input id="phone" name="phone" type="tel" autocomplete="tel"></div>
```

Listing 10.2 (Continued) *responsive.html: the mobile-first rewrite with a viewport line, fluid layout, labeled fields and a thumb-reach action bar.*

```
<div><label for="from">{{FROM}}</label>
<input id="from" name="from" autocomplete="street-address"></div>
<div><label for="to">{{TO}}</label><input id="to" name="to"></div>
<button type="submit">{{BOOK}}</button>
</form>
</main>
<nav><a href="tel:+620000000000">{{CALL}}</a>
<a href="https://wa.me/620000000000">{{WA}}</a></nav>
</body>
</html>
```

The words live in a CSV table, **Listing 10.3**, so that one template serves both languages:

Listing 10.3 `strings.csv`: the page's words in English and Indonesian.

```
key,en,id
TITLE,Ojek Pak Darto,Ojek Pak Darto
TAG,Book a ride from the Jalan Kenari base,Pesan ojek dari pangkalan Jalan Kenari
DRIVERS,Drivers on duty,Pengemudi yang bertugas
NAME,Your name,Nama Anda
PHONE,Phone number,Nomor ponsel
FROM,Pick me up at,Jemput saya di
TO,Take me to,Antar saya ke
BOOK,Book the ride,Pesan ojek
PLATE,Plate,Pelat nomor
CALL,Call,Telepon
WA,WhatsApp,WhatsApp
STATUS,status,status
READY,ready,siap
TRIP,on a trip,sedang mengantar
```

10.5.2 Filling and serving the pages

`pages.py` (**Listing 10.4**) reads the strings and the drivers, builds the table rows for the first version and the cards for the second, escapes every name and replaces each `{{KEY}}` in the template. `app.py` (**Listing 10.5**) serves `/naive` and `/responsive` with Flask on port 4100, choosing the language with `?lang=id`.

Listing 10.4 `pages.py`: fills either template in either language.

```
"""Fill the two page templates (naive and responsive) in English or Indonesian."""
import csv
from html import escape
from pathlib import Path

DATA = Path(__file__).resolve().parent.parent / "data"


def rows(name):
    with open(DATA / name, encoding="utf-8") as f:
        return list(csv.DictReader(f))


def render(page, lang):
    text = {r["key"]: r[lang] for r in rows("strings.csv")}
    status = {"ready": text["READY"], "trip": text["TRIP"]}
    table, cards = "", ""
    for d in rows("drivers.csv"):
        name, plate, st = escape(d["name"]), escape(d["plate"]), d["status"]
        table += f"<tr><td>{name}</td><td>{plate}</td><td>{status[st]}</td></tr>\n"
        cards += (f'<li>{name} &middot; {plate} &middot; '
                  f'<span class="{st}">{status[st]}</span></li>')
    html = (DATA / f"{page}.html").read_text(encoding="utf-8")
    for key, val in {**text, "LANG": lang, "ROWS": table,
                     "CARDS": cards}.items():
        html = html.replace("{{" + key + "}}", val)
    return html
```

Listing 10.5 `app.py`: serves both pages with Flask on port 4100.

```
"""Serve the naive and the responsive booking page with Flask on port 4100."""
from flask import Flask, abort, request
from pages import render

app = Flask(__name__)


@app.get("/<page>")
def show(page):
    if page not in ("naive", "responsive"):
        abort(404)
    lang = "id" if request.args.get("lang") == "id" else "en"
    return render(page, lang)


@app.post("/book")
def book():
    return "<p>OK: " + str(len(request.form)) + " fields received.</p>"


if __name__ == "__main__":
    app.run(port=4100)
```

The Java version splits the same work between `Pages.java` (**Listing 10.6**) and `Server.java` (**Listing 10.7**), which uses the JDK's built-in HTTP server on port 4101. For the same page and language the two servers send identical bytes.

Listing 10.6 `Pages.java`: the same template filling 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.*;

/** Fill the two page templates (naive and responsive) in English or Indonesian. */
public class Pages {
    static final Path DATA = Path.of("..", "data");

    static List<Map<String, String>> rows(String name) throws IOException {
        List<String> lines = Files.readAllLines(DATA.resolve(name),
                                                StandardCharsets.UTF_8);
        String[] head = lines.get(0).split(",");
        List<Map<String, String>> out = new ArrayList<>();
        for (String line : lines.subList(1, lines.size())) {
            String[] f = line.split(",", -1);
            Map<String, String> r = new HashMap<>();
            for (int i = 0; i < head.length; i++) r.put(head[i], f[i]);
            out.add(r);
        }
        return out;
    }

    static String esc(String s) {
        return s.replace("&", "&amp;").replace("<", "&lt;").replace(">", "&gt;");
    }

    static String render(String page, String lang) throws IOException {
        Map<String, String> v = new HashMap<>();
        for (var r : rows("strings.csv")) v.put(r.get("key"), r.get(lang));
        Map<String, String> status = Map.of("ready", v.get("READY"),
                                            "trip", v.get("TRIP"));
        StringBuilder table = new StringBuilder(), cards = new StringBuilder();
        for (var d : rows("drivers.csv")) {
            String name = esc(d.get("name")), plate = esc(d.get("plate"));
            String st = d.get("status");
            table.append("<tr><td>" + name + "</td><td>" + plate + "</td><td>"
                         + status.get(st) + "</td></tr>\n");
            cards.append("<li>" + name + " &middot; " + plate + " &middot; "
                         + "<span class=\"" + st + "\">" + status.get(st)
                         + "</span></li>");
        }
        v.put("LANG", lang);
        v.put("ROWS", table.toString());
        v.put("CARDS", cards.toString());
```

Listing 10.6 (Continued) *Pages.java: the same template filling in Java.*

```java
        String html = Files.readString(DATA.resolve(page + ".html"));
        for (var e : v.entrySet())
            html = html.replace("{{" + e.getKey() + "}}", e.getValue());
        return html;
    }
}
```

Listing 10.7 `Server.java`: serves both pages with the JDK's HTTP server on port 4101.

```
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.charset.StandardCharsets;

/** Serve the naive and the responsive booking page on port 4101. */
public class Server {
    static void send(HttpExchange ex, int code, String html) throws IOException {
        byte[] b = html.getBytes(StandardCharsets.UTF_8);
        ex.getResponseHeaders().set("Content-Type", "text/html; charset=utf-8");
        ex.sendResponseHeaders(code, b.length);
        ex.getResponseBody().write(b);
        ex.close();
    }

    static void handle(HttpExchange ex) throws IOException {
        String path = ex.getRequestURI().getPath();
        String query = ex.getRequestURI().getQuery();
        String lang = "lang=id".equals(query) ? "id" : "en";
        if (ex.getRequestMethod().equals("POST") && path.equals("/book")) {
            String body = new String(ex.getRequestBody().readAllBytes(),
                                     StandardCharsets.UTF_8);
            send(ex, 200, "<p>OK: " + body.split("&").length
                          + " fields received.</p>");
        } else if (path.equals("/naive") || path.equals("/responsive")) {
            send(ex, 200, Pages.render(path.substring(1), lang));
        } else {
            send(ex, 404, "<p>Not found</p>");
        }
    }

    public static void main(String[] args) throws IOException {
        HttpServer server = HttpServer.create(new InetSocketAddress(4101), 0);
        server.createContext("/", Server::handle);
        server.start();
        System.out.println("Pages on http://127.0.0.1:4101/responsive");
    }
}
```

10.5.3 A browser that pretends to be a phone

`check_responsive.py` (**Listing 10.8**) uses Playwright to open each page at 320, 360, 768 and 1280 CSS pixels. Below 700 pixels it creates a *mobile* browser context (`is_mobile=True`, with touch), so that Chromium behaves as a phone browser does, including the 980-pixel default viewport for pages without a viewport line. For each width it asks the page for its layout width (`window.innerWidth`), its total width (`scrollWidth`, to detect sideways scrolling), the smallest font size of any element that holds text, the smallest side of any link, button or field, and whether the phone field is of type `tel`. It multiplies sizes by the scale factor of Equation (10.2), so that it reports what the eye and the thumb actually meet, as in Equation (10.3). It can also save a screenshot at 360 pixels.

Listing 10.8 `check_responsive.py`: measures layout width, overflow, effective text and target sizes and the phone field at four widths.

```
"""Open a page at several phone and desktop widths and report what breaks."""
import sys
from playwright.sync_api import sync_playwright

WIDTHS = [320, 360, 768, 1280]
JS = """() => {
  const vw = window.innerWidth;
  const doc = document.documentElement;
  let text = 99, target = 999;
  for (const e of document.querySelectorAll('body *')) {
    const own = [...e.childNodes]
      .some(n => n.nodeType === 3 && n.textContent.trim());
    if (own) text = Math.min(text, parseFloat(getComputedStyle(e).fontSize));
  }
  for (const e of document.querySelectorAll('a, button, input')) {
    const r = e.getBoundingClientRect();
    target = Math.min(target, r.width, r.height);
  }
  const badInputs = [...document.querySelectorAll('input[name=phone]')]
    .filter(e => e.type !== 'tel').length;
  return [vw, doc.scrollWidth, text, target, badInputs];
}"""


def check(base, page, shot_dir=None, lang="en"):
    print(f"{page}: width  layout  overflow  text px  target px  phone input")
    with sync_playwright() as p:
        browser = p.chromium.launch()
        for w in WIDTHS:
            mobile = w < 700   # phones render pages as a mobile browser does
            ctx = browser.new_context(viewport={"width": w, "height": 800},
                                      is_mobile=mobile, has_touch=mobile,
                                      device_scale_factor=2)
            pg = ctx.new_page()
            pg.goto(f"{base}/{page}?lang={lang}")
            vw, sw, text, target, bad = pg.evaluate(JS)
            scale = w / vw            # < 1 when the browser zooms the page out
            over = "yes" if sw > vw else "no"
            print(f"{'':8}{w:>5}  {vw:>6}  {over:>8}  {text * scale:>7.1f}"
                  f"  {target * scale:>9.1f}  {'type=text' if bad else 'type=tel'}")
            if shot_dir and w == 360:
                pg.screenshot(path=f"{shot_dir}/{page}_{lang}_360.png")
            ctx.close()
        browser.close()
```

Listing 10.8 (Continued) *check_responsive.py: measures layout width, overflow, effective text and target sizes and the phone field at four widths.*

```
if __name__ == "__main__":
    base = sys.argv[1] if len(sys.argv) > 1 else "http://127.0.0.1:4100"
    for page in ("naive", "responsive"):
        check(base, page, *sys.argv[2:3])
```

10.5.4 The load-time estimate

`budget.py` (**Listing 10.11**) and `Budget.java` (**Listing 10.12**) apply Equation (10.4) with $k = 4$ round trips to a list of resources and a list of networks, and compare the result with the 2.5-second LCP threshold. The resources of the first version, **Listing 10.9**, are what the nephew's page would have carried once finished — a large photo of the base, a web font, a JavaScript framework — and the second version keeps only what the page needs. The networks,

Listing 10.10, are Indonesia's median mobile connection [6], the Lighthouse test profile [12] and a weak-signal example for the edge of coverage.

Listing 10.9 `resources_v1.csv`: the resources of the first, finished design (the slim design is in `resources_v2.csv`).

```
resource,kB
page HTML,32
hero photo of the base (JPEG),1800
web font (2 files),180
JavaScript framework bundle,420
stylesheet,60
```

Listing 10.10 `networks.csv`: the three network profiles used for the estimate.

```
network,mbps,rtt_ms
Indonesia median mobile (Ookla Feb 2026),59.18,21
Lighthouse mobile test profile,1.6384,150
Weak signal (example: 0.4 Mbit/s),0.4,300
```

Listing 10.11 `budget.py`: estimates the arrival time of a page on each network and compares it with the LCP budget.

```
"""Estimate how long a page takes to arrive on different networks."""
import csv
import sys
from pathlib import Path

DATA = Path(__file__).resolve().parent.parent / "data"
ROUND_TRIPS = 4      # DNS, TCP, TLS and the request itself, before any byte
LCP_BUDGET = 2.5     # seconds: a "good" LCP


def load(name):
    with open(DATA / name, encoding="utf-8") as f:
        return list(csv.DictReader(f))


def seconds(kb, mbps, rtt_ms):
    return ROUND_TRIPS * rtt_ms / 1000 + kb * 8 / (mbps * 1000)


def report(resources):
    kb = sum(float(r["kB"]) for r in load(resources))
    print(f"{resources}: {kb:.0f} kB")
    for n in load("networks.csv"):
        t = seconds(kb, float(n["mbps"]), float(n["rtt_ms"]))
        verdict = "within" if t <= LCP_BUDGET else "OVER"
        print(f"  {n['network']:<42}{t:>6.2f} s  {verdict} {LCP_BUDGET} s")


if __name__ == "__main__":
    for name in sys.argv[1:] or ["resources_v1.csv", "resources_v2.csv"]:
        report(name)
```

Listing 10.12 `Budget.java`: the same estimate in Java.

```
import java.io.IOException;
import java.util.*;

/** Estimate how long a page takes to arrive on different networks. */
public class Budget {
    static final int ROUND_TRIPS = 4;     // DNS, TCP, TLS and the request itself
    static final double LCP_BUDGET = 2.5; // seconds: a "good" LCP

    static double seconds(double kb, double mbps, double rttMs) {
        return ROUND_TRIPS * rttMs / 1000 + kb * 8 / (mbps * 1000);
    }

    static void report(String resources) throws IOException {
        double kb = 0;
        for (var r : Pages.rows(resources)) kb += Double.parseDouble(r.get("kB"));
        System.out.printf(Locale.US, "%s: %.0f kB%n", resources, kb);
        for (var n : Pages.rows("networks.csv")) {
            double t = seconds(kb, Double.parseDouble(n.get("mbps")),
                               Double.parseDouble(n.get("rtt_ms")));
            String verdict = t <= LCP_BUDGET ? "within" : "OVER";
            System.out.printf(Locale.US, "  %-42s%6.2f s  %s %s s%n",
                              n.get("network"), t, verdict, LCP_BUDGET);
        }
    }

    public static void main(String[] args) throws IOException {
        String[] names = args.length > 0 ? args
            : new String[] {"resources_v1.csv", "resources_v2.csv"};
        for (String name : names) report(name);
    }
}
```

10.6 Running It

Start either server (`python app.py` or `java -cp out Server`) and run the check in a second terminal. Against Flask or against the Java server the output is the same:

```
naive: width  layout  overflow  text px  target px  phone input
          320     980        no      3.6        3.9  type=text
          360     980        no      4.0        4.4  type=text
          768     768       yes     11.0       12.0  type=text
         1280    1280        no     11.0       12.0  type=text
responsive: width  layout  overflow  text px  target px  phone input
          320     320        no     16.0       48.0  type=tel
          360     360        no     16.0       48.0  type=tel
          768     768        no     16.0       48.0  type=tel
         1280    1280        no     16.0       48.0  type=tel
```

Read the first block, the first version, line by line. On the two phones the layout width is 980, not 320 or 360: the missing viewport line has made the browser lay the page out as for a desktop and then shrink it. There is no sideways scrolling — the browser has zoomed out far enough — but the effective text size is 3.6 and 4.0 pixels and the smallest target 3.9 and 4.4 pixels, exactly the values of the Example in Section 10.3.2. At 768 pixels, a tablet or a small laptop window, the layout width is honest but the page's fixed 960-pixel body is wider than the window, so it scrolls sideways (overflow "yes"): a WCAG 1.4.10 failure in the making. Only at 1280 pixels does the page work as its designer saw it. The phone field is plain text at every width.

The second block, the responsive version, is uneventful in the best way: the layout width equals the screen width at every size, nothing overflows, no text is smaller than 16 pixels and no target smaller than 48. **Figure 10.1** shows both pages as a 360-pixel phone displays them.

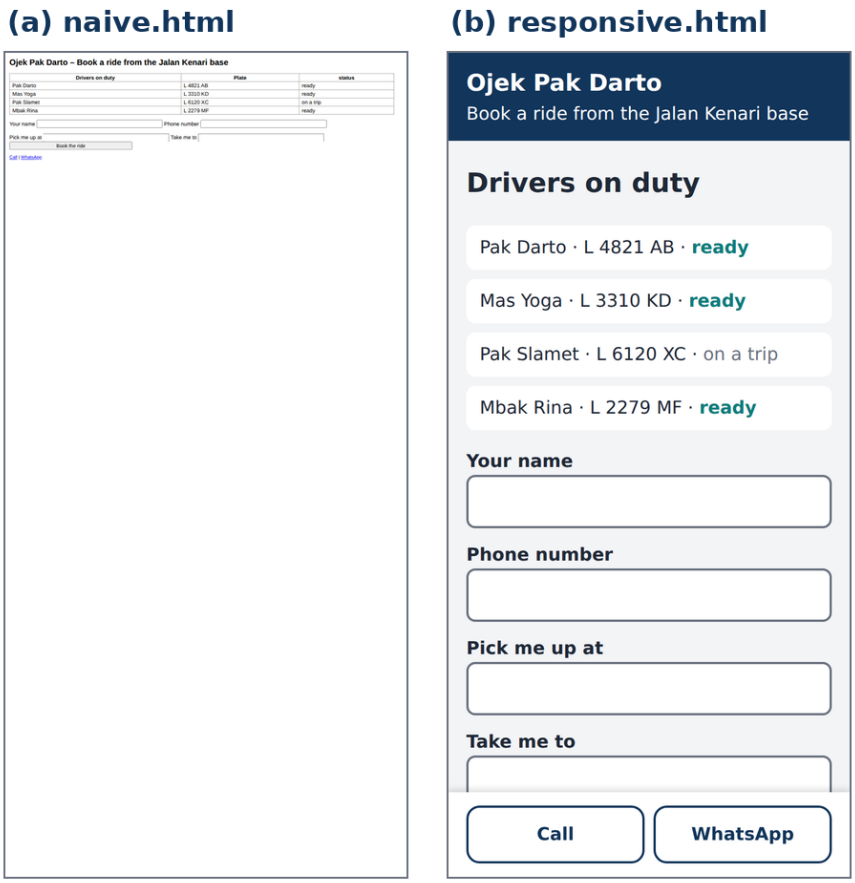


Figure 10.1 The two booking pages on a 360-pixel phone: (a) the desktop page, laid out at 980 pixels and shrunk to fit, and (b) the mobile-first page with readable text, labeled 48-pixel fields and the Call and WhatsApp bar within thumb reach.

Now the load-time estimate, `python budget.py` (the Java version prints the same):

```
resources_v1.csv: 2492 kB
  Indonesia median mobile (Ookla Feb 2026)    0.42 s  within 2.5 s
  Lighthouse mobile test profile             12.77 s  OVER 2.5 s
  Weak signal (example: 0.4 Mbit/s)          51.04 s  OVER 2.5 s
resources_v2.csv: 76 kB
  Indonesia median mobile (Ookla Feb 2026)    0.09 s  within 2.5 s
  Lighthouse mobile test profile              0.97 s  within 2.5 s
  Weak signal (example: 0.4 Mbit/s)           2.72 s  OVER 2.5 s
```

Check one line by hand with Equation (10.4). The first version weighs $S = 32 + 1800 + 180 + 420 + 60 = 2492$ kB. On the Lighthouse profile, $k \cdot RTT = 4 \times 0.150 = 0.6$ s and $8S/B = 8 \times 2492/1638.4 = 12.17$ s, so $t \approx 12.77$ s — five times the budget. On the median connection the same page arrives in 0.42 s, which is why it seemed fine in the nephew's tests in the city. The slim version, 76 kB, arrives within the budget on both the median and the test profile, and only just misses it on the weak-signal example (2.72 s). **Figure 10.2** draws the six estimates on a logarithmic scale.

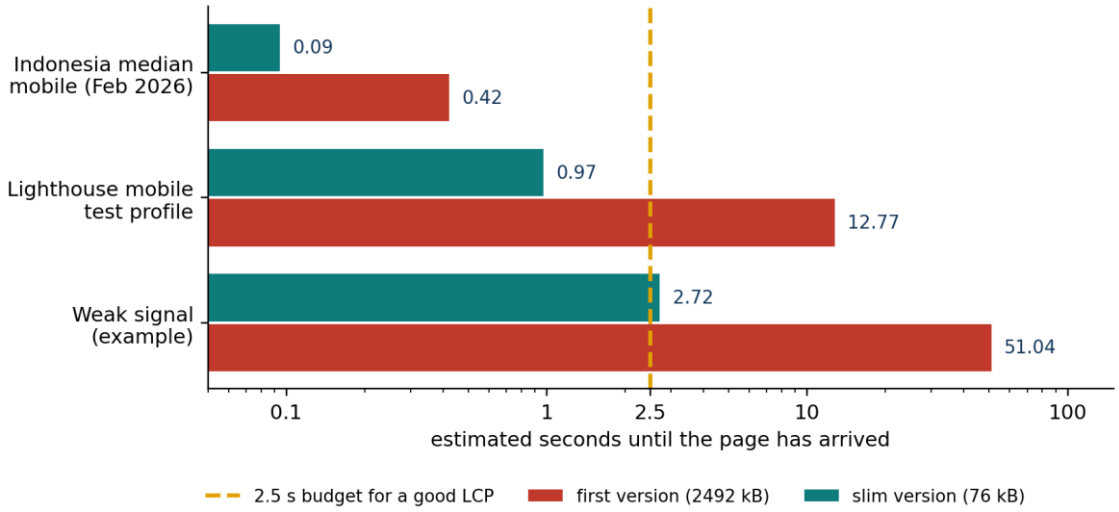


Figure 10.2 Estimated arrival time of the first (2,492 kB) and the slim (76 kB) booking page on three networks, from Equation (10.4) with four round trips; the dashed line is the 2.5-second budget for a good LCP. The time axis is logarithmic.

> **WARNING:** Testing only on a fast phone, on office Wi-Fi, in the city, hides almost every problem this chapter measures. Test on the slowest phone your users are likely to own, with a throttled or real weak connection, outdoors.

10.7 Testing It

The tests protect the parts that the browser check relies on. **Listing 10.13** checks the formula of Equation (10.4) against a value worked by hand (four round trips of 150 ms plus 200 kB at 1.6 Mbit/s is 0.6 + 1.0 = 1.6 s), that both templates render in both languages with no placeholder left, and that the responsive page — and only it — has the viewport line and the `tel` field. **Listing 10.14** makes the same seven checks in Java.

Listing 10.13 `test_ch10.py`: tests of the estimate and of the page rendering.

```
"""Checks of the page renderer and the time estimate (python -m pytest -q)."""
from budget import seconds
from pages import render


def test_estimate_formula():
    # 4 round trips of 150 ms + 200 kB at 1.6 Mbit/s = 0.6 + 1.0 seconds
    assert abs(seconds(200, 1.6, 150) - 1.6) < 1e-9


def test_both_languages_and_no_placeholders_left():
    for page in ("naive", "responsive"):
        for lang in ("en", "id"):
            html = render(page, lang)
            assert "{{" not in html and f'lang="{lang}"' in html
    assert "Pesan ojek" in render("responsive", "id")


def test_responsive_page_has_viewport_and_tel_input():
    html = render("responsive", "en")
    assert 'name="viewport"' in html and 'type="tel"' in html
    assert 'name="viewport"' not in render("naive", "en")
```

Listing 10.14 `Ch10Test.java`: the same checks in plain Java.

```java
/** Checks of the page renderer and the time estimate (java -cp out Ch10Test). */
public class Ch10Test {
    public static void main(String[] args) throws Exception {
        int failed = 0;
        // 4 round trips of 150 ms + 200 kB at 1.6 Mbit/s = 0.6 + 1.0 seconds
        if (Math.abs(Budget.seconds(200, 1.6, 150) - 1.6) > 1e-9) failed++;
        for (String page : new String[] {"naive", "responsive"})
            for (String lang : new String[] {"en", "id"}) {
                String html = Pages.render(page, lang);
                if (html.contains("{{")
                    || !html.contains("lang=\"" + lang + "\"")) failed++;
            }
        String r = Pages.render("responsive", "en");
        if (!r.contains("name=\"viewport\"") || !r.contains("type=\"tel\""))
            failed++;
        if (Pages.render("naive", "en").contains("name=\"viewport\"")) failed++;
        System.out.println(failed == 0 ? "all 7 checks passed"
                                       : failed + " failed");
    }
}
```

`python -m pytest -q` prints

```
3 passed in 0.01s
```

and `java -cp out Ch10Test` prints

```
all 7 checks passed
```

10.8 Making It Better

The responsive page passes every automated check, but the estimate shows one remaining weakness and the check hides a few others.

The weak signal. At 2.72 s the slim page misses the budget on the weak-signal example, and the photo is most of its weight (70 of 76 kB). Three changes, each cheap, bring it within: load the photo *after* the form with `loading="lazy"` so that the text and the form, the page's largest meaningful content, arrive first; serve a smaller photo to narrow screens with `srcset`; or simply drop the photo, since a customer booking a ride does not need a picture of the bench. Without the photo, $S = 6$ kB and $t \approx 4 \times 0.3 + 8 \times 6/400 = 1.32$ s.

The second visit. Round trips dominate small pages on slow links: of the 2.72 s, 1.2 s are the four round trips. A page that is cached on the phone skips them all. HTTP caching headers for the stylesheet and photo, and — for a page that must work under the tree even without signal — a small service worker that serves the last copy offline, are the next steps.

What the check cannot see. Automated checks do not test whether the order of the form fields matches how people think about a ride, whether "Pick me up at" is understood, or whether the page is usable in landscape with a phone in a dashboard holder. Take the page to the base, hand the phone to Mas Yoga and Bu Ani, and watch, as Chapter 7 describes.

10.9 Using It for Real

- **Start from the phone.** Design your page at 360 pixels first, then at 320, and only then add breakpoints for larger screens where the content looks stretched.
- **Always include the viewport line.** It is one line, and without it every other mobile effort is shrunk to a third.

- **Set a budget and write it down.** For example: "on the Lighthouse mobile profile the booking page arrives within 2.5 s", which by Equation (10.4) means at most about 389 kB. Every new photo, font or script must fit or replace something.
- **Measure at the widths your users have.** Run `check_responsive.py` against your own pages; add the widths of the phones you see in your user research.
- **Use the platform's helpers.** Input types, autocomplete tokens and `inputmode` cost nothing and save every user seconds on every visit.
- **Think about the moment of use.** A driver under a tree, a mother at the puskesmas door, a customer in a moving angkot: short tasks, interruptions, sunlight. Keep the main action visible and the page short.

10.10 HCI in Practice: Profitina

Profitina's users are brand owners and marketers who check how ready their websites are to be cited by AI answer engines, and many of them look at their scores on a phone between meetings. The **dashboard** was designed as a cockpit precisely so that the important state fits a small screen: one request gathers notifications, followed channels, rank and activity, and each item is a single tap to its section. On a phone, this avoids the chain of menus that a desktop layout might tolerate.

The **community** is embedded as a widget across the profitina.com marketing site and inside the app, which is a responsive-design problem in the container-query sense of Section 10.3.3: the same component must work in a narrow sidebar on a desktop article and full-width on a phone. The feed is delivered with **Server-Sent Events**, a single long-lived connection over which the server pushes new posts; on a mobile network this avoids the repeated round trips that polling would cost, the term $k \cdot RTT$ of Equation (10.4) paid once instead of every few seconds. **Optimistic posting** — the composer clears at once, the post appears immediately, and the text is restored if sending fails — keeps the interaction responsive in the INP sense even when the network is slow. The runnable case study of this book remains the laundry of Chapter 16; the product only shows where the same principles meet real users.

10.11 Common Mistakes, and How to Avoid Them

- **No viewport line.** The page is laid out at 980 pixels and shrunk to a third. Add `<meta name="viewport" content="width=device-width, initial-scale=1">`.
- **Fixed widths.** `width: 960px` on the body or a table breaks reflow at every smaller width. Use fluid widths and `max-width`.
- **Breakpoints by device.** Chasing the widths of last year's phones produces layouts that break on this year's. Choose breakpoints where the content needs them.
- **Placeholders as labels.** They vanish while typing and are often low in contrast. Use real labels above the fields.
- **The wrong keyboard.** A phone field of type `text` makes every user switch keyboards. Use `type="tel"`, `type="email"` and `autocomplete`.
- **Testing on the median.** A page tested only on a fast phone and a fast connection will fail the users with the least margin. Test against a slow profile.
- **Hover-only interactions.** Menus that open on hover and tooltips that need a mouse do not exist on a touch screen. Make everything work with a tap.

- **Reading statistics without asking what was counted.** Page views on some sites are not people; a median is not the tail.

10.12 Chapter Summary

- A responsive design adapts one interface to every screen with fluid proportions, flexible media and media or container queries; mobile-first design starts from the smallest screen and the weakest conditions.
- In Indonesia 81.72% of people use the internet (APJII 2026); StatCounter's page-view shares (38.98% mobile, 60.59% desktop in August 2026; Android 77.07% of mobile) and Ookla's median mobile speed (59.18 Mbit/s, February 2026) describe different things and must be read with care.
- CSS pixels are device pixels divided by the pixel ratio (Equation (10.1)). Without a viewport line, a phone lays a page out at 980 pixels and scales it by s (Equation (10.2)), shrinking text and targets to $s \cdot h_{css}$ (Equation (10.3)).
- Most people use a phone one-handed with the thumb; put frequent actions within thumb reach and keep targets large.
- Input types, autocomplete tokens and connected labels make phone forms fast and accessible (WCAG 1.3.5).
- Core Web Vitals judge a page good when LCP ≤ 2.5 s, INP ≤ 200 ms and CLS ≤ 0.1 at the 75th percentile. Arrival time is roughly $k \cdot RTT + 8S/B$ (Equation (10.4)); a performance budget keeps it within the target on a slow test profile.
- The program measured two pages at four widths (**Table 10.1**, **Figure 10.1**) and estimated their load times (**Figure 10.2**): the desktop page was shrunk to 4-pixel text and 2.5 MB, the mobile-first page kept 16-pixel text, 48-pixel targets and 76 kB.

10.13 Exercises

1. What effective size does the 13-pixel body text of the first page have on a 320-pixel phone? And on a 412-pixel phone? Use Equation (10.2) and Equation (10.3).
2. With Equation (10.4) and $k = 4$, what is the largest page that arrives within 2.5 s on the Lighthouse profile? On the weak-signal example? What does the second answer say about photos?
3. A colleague reads the StatCounter figure for August 2026 and proposes designing the booking page "for desktop first, because 60% of Indonesian traffic is desktop". Answer in a short paragraph, using what each source counts.
4. Add a row `Good 4G (example),10,50` to `networks.csv` and predict the two new lines of `budget.py` before running it.
5. Give each of the four fields of the booking form the best `type`, `autocomplete` and, where useful, `inputmode`, and explain one case where autocomplete should be switched off.
6. The responsive page switches to two columns at 700 pixels. Describe how you would test whether 700 is a good breakpoint for this content, and name one sign that it is too small and one that it is too large.

10.14 Solutions to the Exercises

1. On a 320-pixel phone, $s = 320/980 = 0.327$, so the body text appears as $0.327 \times 13 = 4.24$ pixels. On a 412-pixel phone, $s = 412/980 = 0.420$ and the text is $0.420 \times 13 = 5.47$ pixels. Even on a large phone the text is about a third of a comfortable 16 pixels; the fix is the viewport line, not a bigger phone.

2. Solve $2.5 = 4 \cdot RTT + 8S/B$ for S: $S = (2.5 - 4 \cdot RTT) \cdot B/8$. For the Lighthouse profile, $S = (2.5 - 0.6) \times 1638.4/8 = 389.1$ kB. For the weak-signal example, $S = (2.5 - 1.2) \times 400/8 = 65.0$ kB. On a weak signal, the whole page — text, styles and pictures — must weigh less than a single ordinary photo; photos must be small, lazy-loaded, or left out.

3. StatCounter counts page views on websites that carry its tracker; it does not count people, and it does not count time spent in apps. Office desktops view many web pages, and many phone users spend their time in apps, so desktop *page views* can dominate while most *people* are mobile, as APJII's survey of people suggests. The booking page's users are drivers and neighbors who will open it on phones, as the first scene of this chapter shows; our own user research, not a national page-view share, should decide. Designing mobile-first loses nothing on the desktop, because the layout expands with media queries.

4. For the first version: $4 \times 0.050 + 8 \times 2492/10000 = 0.2 + 1.99 = 2.19$ s, within the budget. For the slim version: $0.2 + 8 \times 76/10000 = 0.2 + 0.06 = 0.26$ s. Running `budget.py` prints these two values, 2.19 s and 0.26 s, both "within 2.5 s". The heavy page passes on a good connection, which is exactly why testing only on good connections misleads.

5. Name: `type="text"`, `autocomplete="name"`. Phone: `type="tel"`, `autocomplete="tel"` (the number pad appears). Pickup: `type="text"`, `autocomplete="street-address"`, because it is often the user's own address. Destination: `type="text"` with no autocomplete token, since no standard token means "where I am going"; `enterkeyhint="go"` can label the keyboard's enter key. Autocomplete should be off (`autocomplete="off"`, or better `one-time-code` for codes) for values that must not be filled from memory, such as a one-time code or a field where a stale saved value would be dangerous.

6. Open the page in a desktop browser and widen the window slowly from 320 pixels, watching the driver list and the form, or run `check_responsive.py` with a list of widths every 20 pixels and look at screenshots. If just below 700 the single-column cards are already very wide with short content and much empty space, or lines of text exceed about 75 characters, the breakpoint is too large. If just above 700 the two-column cards are so narrow that plate numbers wrap onto a second line, or the two-column form squeezes its labels, the breakpoint is too small. The right breakpoint is where the one-column layout starts to look stretched and the two-column layout still looks comfortable.

References

[1] E. Marcotte, "Responsive Web Design," *A List Apart*, no. 306, May 25, 2010. [Online]. Available: https://alistapart.com/article/responsive-web-design/

[2] L. Wroblewski, *Mobile First*. New York, NY, USA: A Book Apart, 2011.

[3] ANTARA News, "Survei APJII: Penetrasi internet di Indonesia 2026 capai 81,7 persen," May 21, 2026. [Online]. Available: https://www.antaranews.com/berita/5576225/survei-apjii-penetrasi-internet-di-indonesia-2026-capai-817-persen

[4] StatCounter Global Stats, “Desktop vs Mobile vs Tablet Market Share Indonesia,” August 2026 data, accessed Sep. 2026. [Online]. Available: https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/indonesia

[5] StatCounter Global Stats, “Mobile Operating System Market Share Indonesia,” August 2026 data, accessed Sep. 2026. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/indonesia

[6] Bloomberg Technoz, “Ranking Kecepatan Internet RI Versi Speedtest Global Index Baru,” reporting Ookla Speedtest Global Index data for February 2026. [Online]. Available: https://www.bloombergtechnoz.com/detail-news/104703/ranking-kecepatan-internet-ri-versi-speedtest-global-index-baru

[7] MDN Web Docs, “Viewport meta tag” and “CSS container queries,” Mozilla, accessed Sep. 2026. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML/Guides/Viewport_meta_element

[8] S. Hoober, “How Do Users Really Hold Mobile Devices?,” *UXmatters*, Feb. 18, 2013. [Online]. Available: https://www.uxmatters.com/mt/archives/2013/02/how-do-users-really-hold-mobile-devices.php

[9] MDN Web Docs, “HTML attribute: autocomplete” and “inputmode,” Mozilla, accessed Sep. 2026. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML/Reference/Attributes/autocomplete

[10] W3C, “Web Content Accessibility Guidelines (WCAG) 2.2,” W3C Recommendation, Dec. 12, 2024; approved as ISO/IEC 40500:2025, Oct. 2025. [Online]. Available: https://www.w3.org/TR/WCAG22/

[11] B. McQuade, “Defining the Core Web Vitals metrics thresholds,” *web.dev*, Google, accessed Sep. 2026. [Online]. Available: https://web.dev/articles/defining-core-web-vitals-thresholds

[12] Google Chrome, “Lighthouse: Network Throttling and CPU Throttling,” Lighthouse documentation, accessed Sep. 2026. [Online]. Available: https://github.com/GoogleChrome/lighthouse/blob/main/docs/throttling.md

Chapter 11

Accessibility and Inclusive Design (WCAG 2.2)

An interface is accessible when people with disabilities can perceive it, operate it, understand it and rely on it — and in practice an accessible interface is better for everyone: for the older customer whose eyes have tired, for the young father holding a baby in one arm, for anyone in bright sunlight. This chapter follows the queue kiosk of a community health center whose first web page shut out several of its patients, and repairs it with the Web Content Accessibility Guidelines (WCAG) 2.2. By the end of the chapter you will be able to explain disability as a mismatch between a person and a design, and inclusive design as the practice of fixing the design; state what Indonesian law and the European Accessibility Act require; navigate WCAG 2.2 — its four principles, three levels and nine new success criteria — and say where WCAG 3.0 stands in September 2026; compute a contrast ratio by hand and in code; explain how screen readers and other assistive technologies read a page, and why native HTML beats ARIA; interpret the WebAIM Million 2026 survey; and build, in Java and in Python, a contrast checker that suggests passing colors and an audit program that finds the six most common WCAG failures and a few more, then verify the repaired page with the keyboard.

11.1 A Morning at Puskesmas Sukamaju

At seven in the morning the waiting room of Puskesmas Sukamaju, a community health center, is already full. Patients take a queue number at a small touch kiosk by the door, choose a service — the general clinic, the dental clinic, mother and child health — and optionally type their BPJS national health insurance card number. The kiosk runs a web page on a tablet, and the same page is published on the internet so that patients can take a number from home.

Watch four patients use it. Pak Harto, seventy-one, whose eyesight has weakened with age, bends close to the screen; the grey hint text under the heading, pale on white, is invisible to him, and he cannot tell which box to type in. Mbak Laras, who is blind, opens the page at home with TalkBack, the screen reader built into Android; it announces “image, image, button, link” and nothing else, because the service boxes are not buttons and the print button and the help link are pictures without text. Mas Dimas broke his right arm last week and uses the page with his left hand and the keyboard of his laptop; the Tab key jumps past the three service boxes as if they did not exist. And Bu Wulan, holding her baby in one arm, taps with her thumb at a small help icon and misses.

None of the four is unusual. The page was built by a well-meaning volunteer who tested it with his own eyes, his own mouse and his own two hands. Its failures are exactly the most common failures on the web, and each has a simple fix. The puskesmas is also a public service, and in Indonesia public services have a legal duty to be accessible. This chapter shows how to find the failures, fix them and check the fix.

11.2 The Idea in Plain Words

For a long time disability was described only as a property of a person: a person "cannot see", "cannot walk". The **social model of disability** adds the other half of the picture: a person with low vision is *disabled by* a page with pale text, and a wheelchair user is *disabled by* a staircase, while the same people are not disabled by large text or a ramp. Disability arises in the mismatch between a person and a design. That view is empowering for designers, because designs can be changed.

Accessibility is the property of a design that people with disabilities can use it effectively. **Inclusive design** is the practice of designing from the start for the full range of human diversity. A useful tool from Microsoft's inclusive design work is the *persona spectrum*: every limitation exists in a permanent, a temporary and a situational form [1]. One arm: a person with a limb difference (permanent), Mas Dimas with his broken arm (temporary), Bu Wulan with her baby (situational). Blindness (permanent), eyes dilated after an eye examination (temporary), a driver who must not look at the screen (situational). A design that works for the permanent case therefore serves a far larger group than the permanent case alone.

> **DEFINITION.** *Accessibility* is the degree to which people with disabilities can perceive, understand, navigate and interact with a product. *Assistive technology* is hardware or software that a person uses to overcome a mismatch, such as a screen reader, a screen magnifier, a switch, voice control or a braille display. *Inclusive design* is design that considers the full range of human diversity with respect to ability, language, culture, age and other forms of human difference from the start.

11.3 The Theory with Evidence

11.3.1 Who is affected

The World Health Organization estimates that about 1.3 billion people — 16% of the world's population, one in six — experience significant disability, and that the number is growing as populations age and chronic conditions become more common [2]. Add the temporary and situational limitations of the persona spectrum and the share of people who are, at some moment of their day, poorly served by an inaccessible design approaches everyone. Age is the largest single factor for many limitations — eyesight, hearing, fine motor control and the speed of learning new interfaces all change with age — which is why the older patients of a puskesmas are the first to notice an inaccessible kiosk.

11.3.2 What the law requires

In Indonesia, **Law No. 8 of 2016 on Persons with Disabilities** (UU 8/2016) recognizes the right of persons with disabilities to accessibility and to information and communication, including in public services [3]. **Government Regulation No. 42 of 2020** (PP 42/2020) implements it for accessibility to housing, public services and protection from disasters, and obliges providers of public services to provide accessible facilities and services [4]. A puskesmas is a public service; its queue kiosk and its website are part of that service.

In the European Union, the **European Accessibility Act** (Directive (EU) 2019/882) has applied since **June 28, 2025** to a defined list of products and services, among them e-commerce, consumer banking services, e-books, electronic communications and parts of passenger transport, including their websites and mobile apps [5]. It matters to Indonesian developers who

build for European customers or app stores that sell there. Its harmonized technical standard, EN 301 549, incorporates the WCAG success criteria for web content [6].

The technical reference behind nearly all such laws is WCAG. **WCAG 2.2** became a W3C Recommendation in October 2023 (with an updated edition of December 12, 2024) [7], and in October 2025 it was approved as the international standard **ISO/IEC 40500:2025** [8], replacing the ISO version of WCAG 2.0. **WCAG 3.0** is under development: its latest Working Draft is dated September 10, 2026, and the W3C describes it as a draft that still has several years of work ahead; it is built from outcomes, requirements and assertions rather than WCAG 2's success criteria, and it is not yet something to conform to [9]. In September 2026, the standard to build and test against is WCAG 2.2 at level AA.

11.3.3 WCAG 2.2 in one page

WCAG organizes its requirements under four **principles**, remembered as **POUR**: content must be **Perceivable** (text alternatives, captions, adaptable structure, sufficient contrast), **Operable** (keyboard access, enough time, no seizure risks, navigable, input modalities beyond the keyboard), **Understandable** (readable, predictable, input assistance) and **Robust** (compatible with assistive technologies). Under the principles are guidelines, and under the guidelines the testable **success criteria**, each at one of three levels: **A** (the minimum), **AA** (the level required by most laws and policies) and **AAA** (the highest, not required as a whole for entire sites). WCAG 2.2 has 86 success criteria: it added nine to WCAG 2.1 and removed one, 4.1.1 *Parsing*, which modern browsers made obsolete [7]. **Table 11.1** lists the nine new criteria, most of which help people with cognitive, motor or low-vision disabilities.

Table 11.1 The nine success criteria added in WCAG 2.2.

Criterion	Level	In one sentence
2.4.11 Focus Not Obscured (Minimum)	AA	The focused element is not entirely hidden by sticky headers, footers or pop-ups.
2.4.12 Focus Not Obscured (Enhanced)	AAA	No part of the focused element is hidden.
2.4.13 Focus Appearance	AAA	The focus indicator is large enough and contrasts enough.
2.5.7 Dragging Movements	AA	Anything done by dragging can also be done with single taps or clicks.
2.5.8 Target Size (Minimum)	AA	Pointer targets are at least 24 × 24 CSS px, or spaced enough.
3.2.6 Consistent Help	A	Help mechanisms appear in the same relative place on every page.
3.3.7 Redundant Entry	A	Information already entered is not asked for again in the same process.
3.3.8 Accessible Authentication (Minimum)	AA	Logging in does not require a cognitive function test, such as remembering or transcribing, without an alternative.
3.3.9 Accessible Authentication (Enhanced)	AAA	As 3.3.8, without the exceptions for object and personal-content recognition.

Chapters 9 and 10 already used several criteria: 2.5.8 target size, 2.5.1 and 2.5.7 gestures and dragging, 1.4.10 reflow, 1.3.5 input purpose. This chapter adds the ones behind the most common failures: 1.1.1 *Non-text Content* (text alternatives), 1.3.1 *Info and Relationships* (structure such as labels and headings), 1.4.3 *Contrast (Minimum)* and 1.4.11 *Non-text Contrast*, 2.1.1 *Keyboard*, 2.4.4 *Link Purpose*, 2.4.7 *Focus Visible*, 3.1.1 *Language of Page* and 4.1.2 *Name, Role, Value*.

11.3.4 Contrast, computed

Text must stand out from its background, and WCAG measures this with the **contrast ratio** of two colors, computed from their **relative luminance** — the brightness of a color as the eye perceives it, from 0 for black to 1 for white [7]. The computation has three steps.

First, each of the red, green and blue channels, given as a number from 0 to 255, is divided by 255 to a value c between 0 and 1 and then *linearized*, undoing the gamma curve with which screens encode brightness:

$$c_{lin} = \begin{cases} c/12.92 & \text{if } c \le 0.04045 \\ \left(\frac{c+0.055}{1.055}\right)^{2.4} & \text{otherwise} \end{cases} \tag{11.1}$$

Second, the three linear channels are weighted by how strongly the eye responds to each — green most, blue least:

$$L = 0.2126\,R_{lin} + 0.7152\,G_{lin} + 0.0722\,B_{lin} \tag{11.2}$$

Third, with L_1 the luminance of the lighter color and L_2 that of the darker, the contrast ratio is

$$CR = \frac{L_1 + 0.05}{L_2 + 0.05} \tag{11.3}$$

The 0.05 models the glare of ambient light on the screen, and keeps the ratio finite for black. CR runs from 1 (identical colors) to 21 (black on white). Success criterion 1.4.3 at level AA requires at least **4.5:1** for normal text and **3:1** for large text (at least 18 point, or 14 point bold — about 24 and 18.66 CSS pixels); 1.4.6 at level AAA requires 7:1 and 4.5:1; and 1.4.11 requires **3:1** for the visual parts of controls, such as the border of an input field, and for meaningful graphics.

> **EXAMPLE.** The hint text of the first kiosk page is #9CA3AF on white. The red channel is 156, so $c = 156/255 = 0.6118$, and by Equation (11.1) $R_{lin} = ((0.6118 + 0.055)/1.055)^{2.4} = 0.3325$. In the same way $G_{lin} = 0.3663$ (from 163) and $B_{lin} = 0.4287$ (from 175). By Equation (11.2), $L = 0.2126 \times 0.3325 + 0.7152 \times 0.3663 + 0.0722 \times 0.4287 = 0.3636$. White has $L = 1$. By Equation (11.3), $CR = (1 + 0.05)/(0.3636 + 0.05) = 1.05/0.4136 = 2.54$. The hint needs 4.5:1 and has barely more than half of it — which is why Pak Harto could not see it.

A small warning about the formula. The linearization threshold was written as 0.03928 in older WCAG texts; WCAG 2.2 uses the correct value 0.04045. The difference changes no practical result, but code copied from old tutorials may carry the old number.

11.3.5 How assistive technologies read a page

A **screen reader** — TalkBack on Android, VoiceOver on iOS and macOS, NVDA and JAWS on Windows — does not look at the pixels. It reads the **accessibility tree**, a structure that the browser builds from the page's HTML: for each element a *role* (button, link, heading, text field), a *name* (the text a user hears, such as "Print my number") and a *state* (pressed, checked, expanded). The screen reader speaks or brailles this tree and lets the user move through it: by Tab from one control to the next, by swiping on a phone, or by jumping from heading to heading, which is how many screen-reader users skim a page.

Everything therefore depends on the HTML carrying the right roles and names. A `<button>` element has the role "button", is reachable with Tab and is activated with Enter or Space; a `<div onclick>` that merely looks like a button has no role, is skipped by Tab and cannot be activated

from the keyboard (a failure of 2.1.1). An `<img>` needs an `alt` text as its name, or `alt=""` if it is decorative; a link or button that contains only an image without `alt` has no name at all and is announced as "link" or "button" (failures of 2.4.4 and 4.1.2). An input's name comes from its `<label>`; a placeholder is not a reliable name and disappears on typing (1.3.1). Headings (`h1` to `h6`) form the outline that users jump through; skipping from `h1` to `h4` breaks it (1.3.1). And `<html lang="…">` tells the screen reader which language's pronunciation to use (3.1.1): an Indonesian page read with English rules is almost unintelligible.

ARIA (Accessible Rich Internet Applications) is a set of attributes such as `role`, `aria-label`, `aria-describedby` and `aria-live` that add or change roles, names and states when HTML alone cannot express them. Its first rule, in the W3C's own guidance, is not to use it when a native HTML element with the needed semantics exists [10]: a real `<button>` is better than `<div role="button" tabindex="0">` with keyboard handlers added by hand. ARIA used wrongly makes pages *less* accessible, as the survey in the next section shows.

Other assistive technologies depend on the same foundations. **Screen magnifiers** and browser zoom depend on reflow (1.4.10) and resizable text (1.4.4). **Switch access** and **voice control** move through the same focusable controls as the keyboard; voice control users say the visible label ("tap Print my number"), which is why 2.5.3 *Label in Name* requires the accessible name to contain the visible text.

11.3.6 What the web looks like in 2026

WebAIM analyzes the home pages of the top one million websites every year with automated tests. Its February 2026 report found detectable WCAG failures on **95.9%** of home pages, up from 94.8% in 2025, with an average of **56.1** detected errors per page [11]. Six kinds of failure account for 96% of all detected errors: low-contrast text (on 83.9% of pages), missing alternative text for images (53.1%), missing form input labels (51.0%), empty links (46.3%), empty buttons (30.6%) and a missing document language (13.5%). The report also notes that pages using ARIA had *more* detected errors on average than pages without it (59.1 against 42), a reminder of the first rule of ARIA.

Two lessons follow. The most common failures are simple, and fixing six kinds of problem would remove almost all detected errors. And automated tests detect only part of the problem: they can see that an image has no `alt`, but not whether an `alt` text is meaningful, whether the focus order makes sense, or whether a page can be understood. The method in the next section therefore combines tools with people.

11.4 A Hands-On Method: Four Layers of Testing

Accessibility testing is done in four layers, from cheapest to most revealing. Do all four before a public service goes live, and the first two on every change.

1. **Automated checks.** Run a checker on every page: the programs of this chapter, or browser tools such as the accessibility panel of Chrome DevTools, axe or WAVE. They catch the mechanical failures of **Table 11.2** in seconds.
2. **Keyboard walk-through.** Put the mouse away and use only Tab, Shift+Tab, Enter, Space and the arrow keys. Can every control be reached, in a sensible order? Is the focus always visible (2.4.7) and never hidden by a sticky bar (2.4.11)? Can every action be completed?

3. **Screen reader and zoom.** Turn on TalkBack or VoiceOver on a phone, NVDA on Windows, and listen to the page; then zoom the browser to 200% and 400%. Is every control announced with a clear name and role? Does the page reflow?
4. **People with disabilities.** Ask users with the relevant disabilities to do real tasks, with the methods of Chapter 7. Nothing else reveals whether the page is *usable*, not just technically conformant.

Table 11.2 The six most common failures of the WebAIM Million 2026, the WCAG 2.2 criteria behind them, and the checks of this chapter's programs.

Common failure (WebAIM 2026)	WCAG 2.2 criterion	Automated check in this chapter
Low-contrast text	1.4.3 (and 1.4.11 for UI parts)	`contrast.py` / `Contrast.java`
Image without alternative text	1.1.1	`a11y_audit`: img without `alt`
Form input without label	1.3.1, 4.1.2	`a11y_audit`: input without label
Empty link	2.4.4, 4.1.2	`a11y_audit`: link without readable text
Empty button	4.1.2	`a11y_audit`: button without readable text
Missing page language	3.1.1	`a11y_audit`: `html` without `lang`
(also) Clickable non-control, skipped heading level, duplicate id	2.1.1, 1.3.1, 4.1.2	`a11y_audit`

11.5 Building It Step by Step

11.5.1 The first kiosk page

Listing 11.1 is the volunteer's first page. It is short, and every line that matters contains a failure: no `lang`; a logo without `alt`; a heading that jumps from `h1` to `h4`; three service "buttons" made of clickable `div` elements; a pale hint instead of a label; a print button and a help link that contain only images; a duplicated `id`; and the current number in amber on white.

Listing 11.1 `queue_v1.html`: the first kiosk page, with the most common accessibility failures.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Queue</title>
<style>
body { font-family: sans-serif; margin: 16px; }
.hint { color: #9CA3AF; }
.svc { display: inline-block; width: 140px; height: 90px; }
</style>
</head>
<body>
<img src="logo.png">
<h1>Puskesmas Sukamaju</h1>
<h4>Choose a service</h4>
<div class="svc" onclick="take('A')">General clinic</div>
<div class="svc" onclick="take('B')">Dental clinic</div>
<div class="svc" onclick="take('C')">Mother and child</div>
<p class="hint">BPJS card number (optional)</p>
<input id="card" placeholder="0001234567890">
<button onclick="take('A')"><img src="print.png"></button>
<a href="/help"><img src="help.png"></a>
<p id="card">Now serving: <b style="color:#E0A100">A-019</b></p>
</body>
</html>
```

11.5.2 The contrast checker

The color pairs of the kiosk are listed in a CSV file, **Listing 11.2**, each with the kind of content it colors: normal `text` (4.5:1), `large` text (3:1) or a `ui` part such as a field border (3:1).

Listing 11.2 `colors.csv`: the kiosk's color pairs and the kind of content each colors.

```
use,fg,bg,kind
body text,#1F2937,#FFFFFF,text
hint text (first version),#9CA3AF,#FFFFFF,text
button label,#FFFFFF,#0F7C7C,text
amber number on white,#E0A100,#FFFFFF,large
field border,#D1D5DB,#FFFFFF,ui
field border (second version),#6B7280,#FFFFFF,ui
now serving on navy,#FFFFFF,#12355B,large
```

`contrast.py` (**Listing 11.3**) implements Equation (11.1), Equation (11.2) and Equation (11.3) in three small functions, then does something more useful than a verdict: for a failing pair it *suggests* a color that passes, by mixing the foreground with black in steps of 5% until the ratio reaches what is needed. The suggestion keeps the hue, so a brand's teal stays teal, only darker. `Contrast.java` (**Listing 11.4**) is the same program in Java.

Listing 11.3 `contrast.py`: the WCAG contrast ratio and a darker color that passes.

```
"""WCAG 2.2 contrast ratio of color pairs, with a darker color that passes."""
import csv
import sys
from pathlib import Path

DATA = Path(__file__).resolve().parent.parent / "data"
NEEDED = {"text": 4.5, "large": 3.0, "ui": 3.0}   # AA: 1.4.3 and 1.4.11


def channel(c):
    c = c / 255
    return c / 12.92 if c <= 0.04045 else ((c + 0.055) / 1.055) ** 2.4


def luminance(hexcolor):
    r, g, b = (int(hexcolor[i:i + 2], 16) for i in (1, 3, 5))
    return 0.2126 * channel(r) + 0.7152 * channel(g) + 0.0722 * channel(b)


def ratio(fg, bg):
    hi, lo = sorted((luminance(fg), luminance(bg)), reverse=True)
    return (hi + 0.05) / (lo + 0.05)


def darker(hexcolor, step):
    """Mix the color with black: step 0.05 removes 5% of each channel."""
    return "#" + "".join(f"{round(int(hexcolor[i:i + 2], 16) * (1 - step)):02X}"
                         for i in (1, 3, 5))


def fix(fg, bg, needed):
    step = 0.0
    while ratio(darker(fg, step), bg) < needed and step < 1:
        step = round(step + 0.05, 2)
    return darker(fg, step)
```

Listing 11.3 (Continued) *contrast.py: the WCAG contrast ratio and a darker color that passes.*

```
def main(path):
    with open(path, encoding="utf-8") as f:
        rows = list(csv.DictReader(f))
    print(f"{'use':<30}{'fg':>8}{'bg':>8}{'ratio':>7}  need  result")
    for r in rows:
        need = NEEDED[r["kind"]]
        k = ratio(r["fg"], r["bg"])
        result = "pass" if k >= need else "FAIL -> try " + fix(r["fg"], r["bg"],
                                                              need)
        print(f"{r['use']:<30}{r['fg']:>8}{r['bg']:>8}{k:>7.2f}  {need:>4}  "
              + result)


if __name__ == "__main__":
    main(sys.argv[1] if len(sys.argv) > 1 else DATA / "colors.csv")
```

Listing 11.4 `Contrast.java`: the same contrast checker 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.*;

/** WCAG 2.2 contrast ratio of color pairs, with a darker color that passes. */
public class Contrast {
    static final Map<String, Double> NEEDED = Map.of("text", 4.5, "large", 3.0,
                                                     "ui", 3.0); // 1.4.3, 1.4.11

    static double channel(int c8) {
        double c = c8 / 255.0;
        return c <= 0.04045 ? c / 12.92 : Math.pow((c + 0.055) / 1.055, 2.4);
    }

    static int part(String hex, int i) {
        return Integer.parseInt(hex.substring(i, i + 2), 16);
    }

    static double luminance(String hex) {
        return 0.2126 * channel(part(hex, 1)) + 0.7152 * channel(part(hex, 3))
             + 0.0722 * channel(part(hex, 5));
    }

    static double ratio(String fg, String bg) {
        double a = luminance(fg), b = luminance(bg);
        return (Math.max(a, b) + 0.05) / (Math.min(a, b) + 0.05);
    }

    /** Mix the color with black: step 0.05 removes 5% of each channel. */
    static String darker(String hex, double step) {
        StringBuilder s = new StringBuilder("#");
        for (int i : new int[] {1, 3, 5})
            s.append(String.format("%02X", Math.round(part(hex, i) * (1 - step))));
        return s.toString();
    }

    static String fix(String fg, String bg, double needed) {
        double step = 0;
        while (ratio(darker(fg, step), bg) < needed && step < 1)
            step = Math.round((step + 0.05) * 100) / 100.0;
        return darker(fg, step);
    }
```

Listing 11.4 (Continued) *Contrast.java: the same contrast checker in Java.*

```java
    public static void main(String[] args) throws IOException {
        Path p = Path.of(args.length > 0 ? args[0] : "../data/colors.csv");
        List<String> lines = Files.readAllLines(p, StandardCharsets.UTF_8);
        System.out.printf("%-30s%8s%8s%7s  need  result%n", "use", "fg", "bg",
                          "ratio");
        for (String line : lines.subList(1, lines.size())) {
            String[] f = line.split(",");
            double need = NEEDED.get(f[3]), k = ratio(f[1], f[2]);
            String result = k >= need ? "pass"
                                      : "FAIL -> try " + fix(f[1], f[2], need);
            System.out.printf(Locale.US, "%-30s%8s%8s%7.2f  %4s  %s%n", f[0], f[1],
                              f[2], k, need, result);
        }
    }
}
```

11.5.3 The audit

`a11y_audit.py` (**Listing 11.5**) reads an HTML file and reports, with line numbers, the failures of **Table 11.2**. It uses two regular expressions rather than a full HTML parser, so that the Java version, which has no HTML parser in its standard library, can be identical: `TAG` finds every start and end tag with its attributes, and `PAIR` finds each `a`, `button` and `label` element together with its content. The rules are those of Section 11.3.5. An input counts as labeled if a `<label for>` points at its `id`, if it sits inside a `<label>`, or if it has `aria-label` or `aria-labelledby`. A link or button counts as named if its content has text, or an image with non-empty `alt`, or if it has `aria-label`. A tag with `onclick` that is not a link or button is reported as unreachable by keyboard. `A11yAudit.java` (**Listing 11.6**) applies the same two patterns with `java.util.regex` and prints the same report. A regular-expression audit has limits — it does not see styles, scripts or the rendered page — and Section 11.8 says what to do about them.

Listing 11.5 `a11y_audit.py`: finds missing language, alternative texts, labels and names, keyboard traps of clickable elements, heading jumps and duplicate ids.

```
"""A small accessibility audit of an HTML file: the most common WCAG failures."""
import re
import sys

TAG = re.compile(r"<(/?)([a-zA-Z][a-zA-Z0-9]*)([^>]*)>")
ATTR = re.compile(r'([a-zA-Z-]+)\s*=\s*"([^"]*)"')
PAIR = re.compile(r"<(a|button|label)\b([^>]*)>(.*?)</\1>", re.S | re.I)


def attrs(text):
    return {k.lower(): v for k, v in ATTR.findall(text)}


def visible_text(inner):
    alts = " ".join(attrs(m.group(3)).get("alt", "")
                    for m in TAG.finditer(inner) if m.group(2).lower() == "img")
    return (re.sub(r"<[^>]+>", " ", inner) + " " + alts).strip()


def audit(html):
    found = []

    def report(pos, check, msg):
        found.append((html.count("\n", 0, pos) + 1, check, msg))

    labelled = {attrs(m.group(3)).get("for") for m in TAG.finditer(html)
                if m.group(2).lower() == "label"}
    wrapped = [m.span() for m in PAIR.finditer(html)
               if m.group(1).lower() == "label"]
    ids, last_h = {}, 0
    for m in TAG.finditer(html):
        close, name, a = m.group(1), m.group(2).lower(), attrs(m.group(3))
        if close:
            continue
        if name == "html" and not a.get("lang"):
            report(m.start(), "3.1.1", "page has no lang attribute")
        if name == "img" and "alt" not in a:
            report(m.start(), "1.1.1", f"img {a.get('src', '')} has no alt")
        if name == "input" and a.get("type") not in ("hidden", "submit", "button"):
            inside = any(s <= m.start() < e for s, e in wrapped)
            if not (a.get("id") in labelled or inside or "aria-label" in a
                    or "aria-labelledby" in a):
                report(m.start(), "1.3.1", f"input {a.get('id', '')} has no label")
        if re.fullmatch(r"h[1-6]", name):
            level = int(name[1])
            if last_h and level > last_h + 1:
```

Listing 11.5 (Continued) *a11y_audit.py: finds missing language, alternative texts, labels and names, keyboard traps of clickable elements, heading jumps and duplicate ids.*

```
                report(m.start(), "1.3.1",
                       f"heading jumps from h{last_h} to {name}")
            last_h = level
        if "onclick" in a and name not in ("a", "button"):
            report(m.start(), "2.1.1",
                   f"clickable <{name}> is not reachable by keyboard")
        if "id" in a:
            if a["id"] in ids:
                report(m.start(), "4.1.2", f"id \"{a['id']}\" is used twice")
            ids[a["id"]] = True
    for m in PAIR.finditer(html):
        name = m.group(1).lower()
        if name in ("a", "button") and not (visible_text(m.group(3))
                                            or "aria-label" in attrs(m.group(2))):
            report(m.start(), "2.4.4" if name == "a" else "4.1.2",
                   f"{name} has no text a screen reader can read")
    return sorted(found)


if __name__ == "__main__":
    for path in sys.argv[1:]:
        problems = audit(open(path, encoding="utf-8").read())
        print(f"{path}: {len(problems)} problem(s)")
        for line, check, msg in problems:
            print(f"  line {line:>2}  SC {check:<6} {msg}")
```

Listing 11.6 `A11yAudit.java`: the same audit in Java with `java.util.regex`.

```java
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.*;
import java.util.regex.*;

/** A small accessibility audit of an HTML file: the most common WCAG failures. */
public class A11yAudit {
    static final Pattern TAG =
        Pattern.compile("<(/?)([a-zA-Z][a-zA-Z0-9]*)([^>]*)>");
    static final Pattern ATTR =
        Pattern.compile("([a-zA-Z-]+)\\s*=\\s*\"([^\"]*)\"");
    static final Pattern PAIR = Pattern.compile(
        "<(a|button|label)\\b([^>]*)>(.*?)</\\1>",
        Pattern.DOTALL | Pattern.CASE_INSENSITIVE);

    record Problem(int line, String check, String msg) {}

    static Map<String, String> attrs(String text) {
        Map<String, String> a = new HashMap<>();
        Matcher m = ATTR.matcher(text);
        while (m.find()) a.put(m.group(1).toLowerCase(), m.group(2));
        return a;
    }

    static String visibleText(String inner) {
        StringBuilder alts = new StringBuilder();
        Matcher m = TAG.matcher(inner);
        while (m.find())
            if (m.group(2).equalsIgnoreCase("img"))
                alts.append(" ").append(attrs(m.group(3)).getOrDefault("alt", ""));
        return (inner.replaceAll("<[^>]+>", " ") + " " + alts).strip();
    }

    static List<Problem> audit(String html) {
        List<Problem> found = new ArrayList<>();
        java.util.function.BiConsumer<Integer, String[]> report = (pos, cm) ->
            found.add(new Problem((int) html.substring(0, pos).chars()
                                      .filter(c -> c == '\n').count() + 1,
                                  cm[0], cm[1]));
        Set<String> labelled = new HashSet<>();
        List<int[]> wrapped = new ArrayList<>();
        Matcher m = TAG.matcher(html);
        while (m.find())
            if (m.group(2).equalsIgnoreCase("label"))
                labelled.add(attrs(m.group(3)).get("for"));
```

Listing 11.6 (Continued) *A11yAudit.java: the same audit in Java with java.util.regex.*

```
        m = PAIR.matcher(html);
        while (m.find())
            if (m.group(1).equalsIgnoreCase("label"))
                wrapped.add(new int[] {m.start(), m.end()});
        Set<String> ids = new HashSet<>();
        int lastH = 0;
        m = TAG.matcher(html);
        while (m.find()) {
            String name = m.group(2).toLowerCase();
            var a = attrs(m.group(3));
            int pos = m.start();
            if (!m.group(1).isEmpty()) continue;
            if (name.equals("html") && a.getOrDefault("lang", "").isEmpty())
                report.accept(pos, new String[] {"3.1.1",
                    "page has no lang attribute"});
            if (name.equals("img") && !a.containsKey("alt"))
                report.accept(pos, new String[] {"1.1.1",
                    "img " + a.getOrDefault("src", "") + " has no alt"});
            if (name.equals("input") && !List.of("hidden", "submit", "button")
                    .contains(a.getOrDefault("type", ""))) {
                boolean inside = wrapped.stream()
                    .anyMatch(s -> s[0] <= pos && pos < s[1]);
                if (!(labelled.contains(a.get("id")) || inside
                      || a.containsKey("aria-label")
                      || a.containsKey("aria-labelledby")))
                    report.accept(pos, new String[] {"1.3.1",
                        "input " + a.getOrDefault("id", "") + " has no label"});
            }
            if (name.matches("h[1-6]")) {
                int level = name.charAt(1) - '0';
                if (lastH > 0 && level > lastH + 1)
                    report.accept(pos, new String[] {"1.3.1",
                        "heading jumps from h" + lastH + " to " + name});
                lastH = level;
            }
            if (a.containsKey("onclick") && !name.equals("a")
                    && !name.equals("button"))
                report.accept(pos, new String[] {"2.1.1",
                    "clickable <" + name + "> is not reachable by keyboard"});
            if (a.containsKey("id") && !ids.add(a.get("id")))
                report.accept(pos, new String[] {"4.1.2",
                    "id \"" + a.get("id") + "\" is used twice"});
        }
        m = PAIR.matcher(html);
        while (m.find()) {
            String name = m.group(1).toLowerCase();
```

Listing 11.6 (Continued) *A11yAudit.java: the same audit in Java with java.util.regex.*

```
            if ((name.equals("a") || name.equals("button"))
                    && visibleText(m.group(3)).isEmpty()
                    && !attrs(m.group(2)).containsKey("aria-label"))
                report.accept(m.start(), new String[] {
                    name.equals("a") ? "2.4.4" : "4.1.2",
                    name + " has no text a screen reader can read"});
        }
        found.sort(Comparator.comparingInt(Problem::line)
                .thenComparing(Problem::check).thenComparing(Problem::msg));
        return found;
    }

    public static void main(String[] args) throws IOException {
        for (String path : args) {
            var problems = audit(Files.readString(Path.of(path)));
            System.out.println(path + ": " + problems.size() + " problem(s)");
            for (var p : problems)
                System.out.printf("  line %2d  SC %-6s %s%n", p.line(), p.check(),
                                  p.msg());
        }
    }
}
```

11.5.4 The repaired page

Listing 11.7 repairs every failure with plain HTML. The page has `lang="en"` (the Indonesian copy has `lang="id"`). The logo is decorative next to the visible name, so it has `alt=""` and is skipped by screen readers. The headings go `h1`, `h2`. The services are real `<button>` elements, 64 pixels high, with the queue letter in their visible text. The card field has a real label, a hint connected with `aria-describedby` so that it is read after the label, `inputmode="numeric"` for the number pad and a 3:1 border. The print button and the help link have visible text. The current number sits in a region with `aria-live="polite"`, so that a screen reader announces “Now serving: A-020” when it changes. And a thick amber focus ring, drawn only for keyboard focus with `:focus-visible`, shows where the keyboard is.

Listing 11.7 `queue_v2.html`: the repaired kiosk page, using native HTML elements, labels, text alternatives, a live region and a visible focus ring.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>Take a queue number - Puskesmas Sukamaju</title>
<style>
body { font: 1.125rem/1.5 "DejaVu Sans", sans-serif; margin: 0; color: #1F2937; }
header { background: #12355B; color: #fff; padding: 12px 16px; }
header img { vertical-align: middle; }
h1 { display: inline; font-size: 1.4rem; margin-left: 8px; }
main { padding: 8px 16px 16px; }
.now { font-size: 1.5rem; }
.svc { display: grid; gap: 10px; }
.svc button, .print { width: 100%; min-height: 64px; font: inherit;
  text-align: left; padding: 8px 16px; border: 2px solid #0F7C7C;
  border-radius: 12px; background: #fff; color: #1F2937; }
.print { background: #0F7C7C; color: #fff; text-align: center; }
label { display: block; margin-top: 16px; font-weight: bold; }
.hint { color: #4B5563; font-size: 1rem; }
input { width: 100%; min-height: 48px; font: inherit; padding: 8px;
  border: 2px solid #6B7280; border-radius: 8px; box-sizing: border-box; }
button:focus-visible, input:focus-visible, a:focus-visible {
  outline: 4px solid #E0A100; outline-offset: 2px; }
a { display: inline-block; min-height: 48px; padding: 12px 0; color: #12355B; }
</style>
</head>
<body>
<header><img src="logo.png" alt="" width="32" height="32">
<h1>Puskesmas Sukamaju</h1></header>
<main>
<p class="now" aria-live="polite">Now serving: <strong>A-019</strong></p>
<h2>Choose a service</h2>
<div class="svc">
<button type="button">A &middot; General clinic</button>
<button type="button">B &middot; Dental clinic</button>
<button type="button">C &middot; Mother and child</button>
</div>
<label for="card">BPJS card number (optional)</label>
<p class="hint" id="card-hint">13 digits, on the front of the card</p>
<input id="card" inputmode="numeric" autocomplete="off"
       aria-describedby="card-hint">
<p><button class="print" type="button">Print my number</button></p>
<p><a href="/help">Help and other languages</a></p>
</main>
</body>
```

Listing 11.7 (Continued) *queue_v2.html: the repaired kiosk page, using native HTML elements, labels, text alternatives, a live region and a visible focus ring.*

```
</html>
```

11.6 Running It

Run the contrast checker, `python contrast.py` or `java -cp out Contrast`:

```
use                            fg       bg   ratio  need  result
body text                   #1F2937 #FFFFFF  14.68   4.5  pass
hint text (first version)   #9CA3AF #FFFFFF   2.54   4.5  FAIL -> try #6D727A
button label                #FFFFFF #0F7C7C   5.01   4.5  pass
amber number on white       #E0A100 #FFFFFF   2.27   3.0  FAIL -> try #BE8900
field border                #D1D5DB #FFFFFF   1.47   3.0  FAIL -> try #929599
field border (second version) #6B7280 #FFFFFF  4.83   3.0  pass
now serving on navy         #FFFFFF #12355B  12.46   3.0  pass
```

Three pairs fail. The hint text has 2.54:1, as computed by hand in the Example of Section 11.3.4; the checker suggests #6D727A, which reaches 4.84:1 — the repaired page uses the similar grey #4B5563 for its hint. The amber number on white has 2.27:1, below even the 3:1 for large text; darkening the amber to #BE8900 would just pass at 3.10:1, but the repaired page solves it differently, by showing the number in dark text on white and keeping amber for the focus ring, where it sits on white next to a dark border. The light field border #D1D5DB has 1.47:1 and fails 1.4.11; the repaired page uses #6B7280 with 4.83:1. **Figure 11.1** shows all seven pairs as the eye sees them.

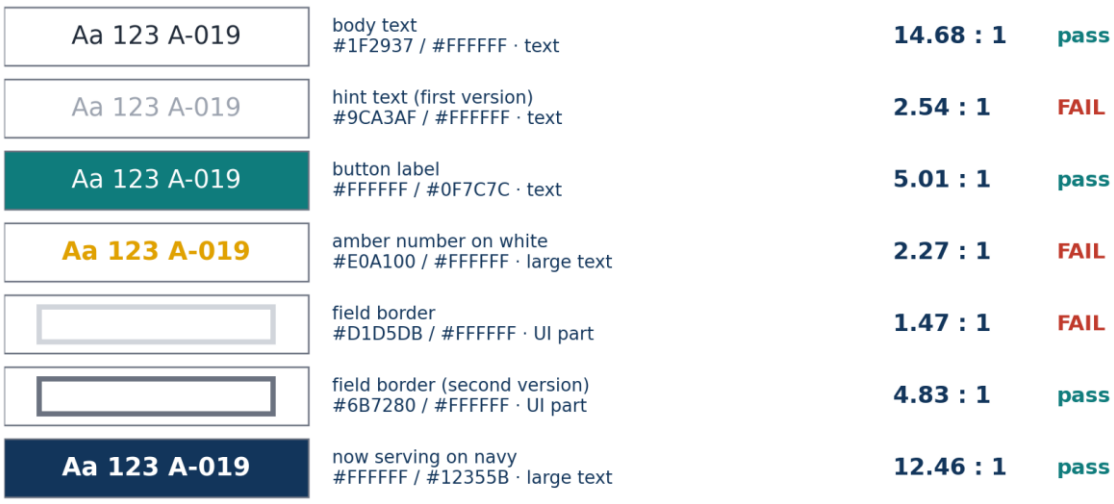


Figure 11.1 The kiosk's seven color pairs with their contrast ratios from Equation (11.3): three fail the WCAG 2.2 AA thresholds (4.5:1 for text, 3:1 for large text and for the parts of controls).

Now the audit of both pages, `python a11y_audit.py ../data/queue_v1.html ../data/queue_v2.html` (the Java version prints the same):

```
../data/queue_v1.html: 12 problem(s)
  line  2  SC 3.1.1  page has no lang attribute
  line 13  SC 1.1.1  img logo.png has no alt
  line 15  SC 1.3.1  heading jumps from h1 to h4
  line 16  SC 2.1.1  clickable <div> is not reachable by keyboard
  line 17  SC 2.1.1  clickable <div> is not reachable by keyboard
  line 18  SC 2.1.1  clickable <div> is not reachable by keyboard
  line 20  SC 1.3.1  input card has no label
  line 21  SC 1.1.1  img print.png has no alt
  line 21  SC 4.1.2  button has no text a screen reader can read
  line 22  SC 1.1.1  img help.png has no alt
  line 22  SC 2.4.4  a has no text a screen reader can read
  line 23  SC 4.1.2  id "card" is used twice
../data/queue_v2.html: 0 problem(s)
```

The twelve problems of the first page map onto the four patients of Section 11.1. Pak Harto suffers from the missing label (line 20) as much as from the pale hint: with a real label, the text is larger, darker and tapping it focuses the field. Mbak Laras hears "image" for the logo (line 13), cannot operate the three service `div`s (lines 16–18), hears only "button" and "link" for the print button and help link (lines 21–22), and her phone reads the English page with whatever pronunciation it guesses (line 2). Mas Dimas cannot reach the services with Tab (lines 16–18). The duplicated `id` "card" (line 23) means that a label pointing at `card` could land on the paragraph instead of the field. The repaired page has no problems that the audit can find. **Figure 11.2** shows it on a phone after pressing Tab three times: the focus ring is on the third service button.

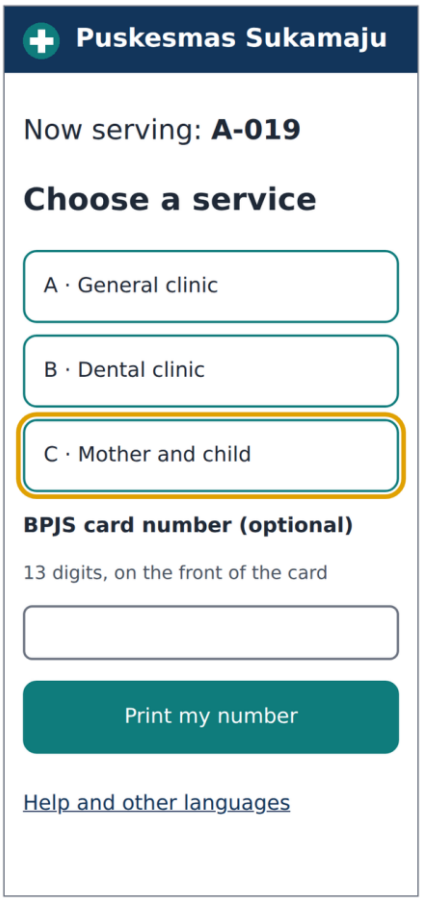


Figure 11.2 The repaired kiosk page at 360 CSS pixels after three presses of Tab: real buttons with visible text, a labeled card field with a hint and a 3:1 border, and a 4-pixel amber focus ring on the focused button.

11.7 Testing It

Listing 11.8 checks the contrast formula at its two extremes (21:1 for black on white, 1:1 for a color on itself) and at a well-known value — #767676 on white, the lightest grey that passes 4.5:1 for normal text, gives 4.54 — and checks that the suggested fix really passes, that the audit finds exactly twelve problems in the first page and none in the second, and that an input wrapped in its label counts as labeled. **Listing 11.9** makes the same six checks in Java.

Listing 11.8 `test_ch11.py`: tests of the contrast checker and the audit.

```python
"""Checks of the contrast formula and the audit (python -m pytest -q)."""
from pathlib import Path
from a11y_audit import audit
from contrast import fix, ratio

DATA = Path(__file__).resolve().parent.parent / "data"


def test_contrast_extremes_and_a_known_pair():
    assert round(ratio("#000000", "#FFFFFF"), 2) == 21.0
    assert ratio("#777777", "#777777") == 1.0
    assert round(ratio("#767676", "#FFFFFF"), 2) == 4.54   # the classic grey


def test_fix_reaches_the_target():
    assert ratio(fix("#9CA3AF", "#FFFFFF", 4.5), "#FFFFFF") >= 4.5


def test_audit_counts():
    v1 = (DATA / "queue_v1.html").read_text(encoding="utf-8")
    v2 = (DATA / "queue_v2.html").read_text(encoding="utf-8")
    assert len(audit(v1)) == 12 and audit(v2) == []


def test_wrapped_label_counts_as_label():
    html = '<html lang="id"><label>Nama <input name="n"></label></html>'
    assert audit(html) == []
```

Listing 11.9 `Ch11Test.java`: the same checks in plain Java.

```java
import java.nio.file.Files;
import java.nio.file.Path;

/** Checks of the contrast formula and the audit (java -cp out Ch11Test). */
public class Ch11Test {
    static int failed = 0;

    static void check(boolean ok) { if (!ok) failed++; }

    public static void main(String[] args) throws Exception {
        check(Math.round(Contrast.ratio("#000000", "#FFFFFF") * 100) == 2100);
        check(Contrast.ratio("#777777", "#777777") == 1.0);
        check(Math.round(Contrast.ratio("#767676", "#FFFFFF") * 100) == 454);
        String fixed = Contrast.fix("#9CA3AF", "#FFFFFF", 4.5);
        check(Contrast.ratio(fixed, "#FFFFFF") >= 4.5);
        String v1 = Files.readString(Path.of("../data/queue_v1.html"));
        String v2 = Files.readString(Path.of("../data/queue_v2.html"));
        check(A11yAudit.audit(v1).size() == 12 && A11yAudit.audit(v2).isEmpty());
        String html = "<html lang=\"id\"><label>Nama <input name=\"n\"></label>";
        check(A11yAudit.audit(html + "</html>").isEmpty());
        System.out.println(failed == 0 ? "all 6 checks passed"
                                       : failed + " failed");
    }
}
```

`python -m pytest -q` prints

```
4 passed in 0.01s
```

and `java -cp out Ch11Test` prints

```
all 6 checks passed
```

11.8 Making It Better

A clean automated audit is the beginning of accessibility, not the end. Walk through the repaired page with the next two layers of Section 11.4.

Keyboard. Open `queue_v2.html` and press Tab repeatedly. The focus visits, in order, the three service buttons, the card field, the print button and the help link — six stops, in the order a sighted user reads the page — and the amber ring is always visible. The fixed header does not cover anything, so 2.4.11 holds. One improvement remains: after a service is chosen, the focus should move to (or a live message should announce) the result, "Your number is B-024", so that a screen-reader user knows the action worked.

Screen reader. With TalkBack, the page is read as "Puskesmas Sukamaju, heading level 1; Now serving: A-019; Choose a service, heading level 2; A, General clinic, button; …; BPJS card number (optional), edit box, 13 digits, on the front of the card; Print my number, button; Help and other languages, link". The dot between the queue letter and the service name is read as a pause. The live region announces changes of the current number without moving the focus.

What remains for people. Is "Mother and child" the name that patients use, or do they say "KIA"? Can Pak Harto, with his eyes, find the right button without help? Is the printed ticket readable? These questions need the patients themselves, and Chapter 7's methods. A puskesmas can also offer what no page can: a staff member who takes a number for a patient who asks. Inclusive design includes the people around the interface.

11.9 Using It for Real

- **Put accessibility into the requirements.** When a puskesmas, a school or a village office orders a website or kiosk, the order should say "conforms to WCAG 2.2 level AA", as UU 8/2016 and PP 42/2020 imply for public services. Then the four layers of testing become part of acceptance.
- **Automate the cheap part.** Run `a11y_audit.py` and `contrast.py` (or a mainstream tool) on every page in every release, as a test that must pass, just as unit tests must.
- **Fix with HTML first.** Most failures disappear when the right native element is used: `button`, `a`, `label`, `h1`–`h6`, `alt`, `lang`. Reach for ARIA only when no element fits.
- **Choose colors once, well.** Test the palette of a design system before it is used anywhere; a failing brand color costs one decision to fix at the start and a thousand fixes later.
- **Publish an accessibility statement.** Say which standard the service aims at, what is known not to work yet, and how people can report a problem or get help another way.
- **Remember the persona spectrum.** Designing for a blind user, an older user or a user with one hand free improves the service for everyone who is tired, busy or outdoors.

11.10 HCI in Practice: Profitina

Profitina's product is a score — how ready a brand's website is to be cited by AI answer engines — and a score is only useful if every user can read it. That makes success criterion 1.4.1 *Use of Color* central: a readiness level must never be conveyed by a red, amber or green dot alone, because many people, most of them men, have some form of color-vision deficiency, and screen readers do not announce colors. The number, a word for the level and the explanation of each signal must be text. The same applies to the weekly trend chart of the historical dashboard, which needs a text or table alternative (1.1.1) and a CSV export for users who prefer their own tools.

The platform's self-service compliance with Indonesia's personal-data law — explicit consent, data export, email and password change, account purge, and legal pages published as plain HTML — is also an accessibility matter. Consent that a screen-reader user cannot perceive, or a data-export button that cannot be reached by keyboard, is not meaningful consent or a usable right. Publishing the legal pages as ordinary HTML rather than as images or PDF scans gives them headings, real text that can be enlarged, and a language attribute. And the community's Link Safety screening, which warns about suspicious links, must present its warning as text announced to assistive technologies, not only as a colored icon. The worked, runnable case study of this book remains the laundry of Chapter 16.

11.11 Common Mistakes, and How to Avoid Them

- **Clickable `div`s and `span`s.** They look like buttons but have no role and no keyboard access. Use `<button>` or `<a href>`.
- **Images without `alt` — or with bad `alt`.** Informative images need a short description of their purpose; decorative ones need `alt=""`. File names ("print.png") and "image of" are not descriptions.
- **Placeholder instead of label.** Use a `<label>` connected with `for`/`id`; put hints in a separate element connected with `aria-describedby`.
- **Pale grey text and borders.** Check every pair against 4.5:1 (text) and 3:1 (large text, control parts). Glossy translucent styles need the same check.
- **Removing the focus outline.** `outline: none` without a replacement makes the keyboard invisible. Style `:focus-visible` instead.
- **Color as the only signal.** Add text or an icon with a text alternative to every colored status.
- **ARIA as decoration.** Wrong roles and labels are worse than none. Follow the first rule of ARIA.
- **Trusting the tool.** A clean automated report is necessary, not sufficient. Do the keyboard, screen-reader and user layers.
- **Treating accessibility as a final step.** Retrofitting costs far more than designing it in; put it in the requirements and the design system.

11.12 Chapter Summary

- Disability arises in the mismatch between a person and a design; inclusive design fixes the design, and the persona spectrum (permanent, temporary, situational) shows that it serves far more people than those with permanent disabilities. About 1.3 billion people, one in six, experience significant disability (WHO).
- UU 8/2016 and PP 42/2020 require accessible public services in Indonesia; the European Accessibility Act has applied since June 28, 2025; WCAG 2.2 is ISO/IEC 40500:2025; WCAG 3.0 is a Working Draft (September 10, 2026) with years of work ahead.
- WCAG 2.2 has four principles (POUR), three levels and 86 success criteria, nine of them new (**Table 11.1**); AA is the usual legal target.
- Contrast is computed from linearized channels (Equation (11.1)), relative luminance (Equation (11.2)) and the ratio (Equation (11.3)); AA needs 4.5:1 for text and 3:1 for large text and control parts (**Figure 11.1**).
- Screen readers read the accessibility tree of roles, names and states; native HTML provides them, and ARIA should be used only when HTML cannot.

- The WebAIM Million 2026 found failures on 95.9% of home pages; six kinds of failure make up 96% of detected errors (**Table 11.2**).
- Test in four layers: automated, keyboard, screen reader and zoom, people with disabilities. The programs of this chapter automate the first layer, and found twelve problems in the first kiosk page and none in the repaired one (**Figure 11.2**).

11.13 Exercises

1. Compute by hand, with Equation (11.1), Equation (11.2) and Equation (11.3), the contrast ratio of #767676 on white. Does it pass for normal text at level AA? At AAA?
2. The amber #E0A100 fails on white. Compute (or run) its ratio on the navy #12355B, and white's ratio on amber. Which of the three combinations would you use for the "now serving" number, and why?
3. For each of the twelve problems in the audit of the first page, name which of the four patients of Section 11.1 it affects most, and the one-line fix.
4. Extend `a11y_audit` so that it also reports an image whose `alt` text is just a file name, such as `alt="print.png"`. Show the new rule and its result on `<html lang="en"><img src="a.png" alt="a.png"></html>`.
5. List the keyboard focus order of `queue_v2.html`. What would change if the print button were placed before the service buttons in the HTML but moved below them with CSS?
6. A village office's login page asks users to type the distorted letters of a picture (a CAPTCHA) and then a password. Which WCAG 2.2 criterion does it touch, what alternatives satisfy it, and why are passkeys a good answer?

11.14 Solutions to the Exercises

1. Each channel is 118, so $c = 118/255 = 0.4627$, which is above 0.04045, so by Equation (11.1) $c_{lin} = ((0.4627 + 0.055)/1.055)^{2.4} = 0.1812$. All three channels are equal, so by Equation (11.2) $L = (0.2126 + 0.7152 + 0.0722) \times 0.1812 = 0.1812$. By Equation (11.3), $CR = (1 + 0.05)/(0.1812 + 0.05) = 1.05/0.2312 = 4.54$. It passes AA for normal text (4.5:1) — just — and fails AAA (7:1). The test in **Listing 11.8** checks the same value.

2. Amber on navy gives 5.50:1 and white on amber gives 2.27:1 (the same as amber on white, because the ratio is symmetric). Amber on navy passes AA for normal text and comfortably for large text, so a large amber number on a navy band would work; white on amber fails. On the white page, the simplest choice is dark text on white (14.68:1), which is why the repaired page uses it and keeps amber for the focus ring. Any choice must also avoid using amber *alone* to signal meaning (1.4.1).

3. Line 2, no `lang`: Mbak Laras (wrong pronunciation); add `lang="en"` or `lang="id"`. Line 13, logo without `alt`: Mbak Laras hears "image"; add `alt=""`, since the name is in the heading. Line 15, `h1` to `h4`: Mbak Laras's heading navigation; use `h2`. Lines 16–18, clickable `div`s: Mas Dimas (keyboard) and Mbak Laras (no role); use `<button>`. Line 20, input without label: Pak Harto (cannot see the pale hint) and Mbak Laras (unnamed field); add a `<label for="card">`. Line 21, image without `alt` in the button, and the button without a name: Mbak Laras; give the button visible text. Line 22, the same for the help link: Mbak Laras, and Bu Wulan, for whom a text link can also be made larger; give it visible text and a 48-pixel area. Line 23, duplicated `id`: everyone who relies on the label; make ids unique.

4. Add the rule before the existing `img` check:

```
        if name == "img" and re.search(r"\.(png|jpe?g|gif|svg|webp)$",
                                       a.get("alt", ""), re.I):
            report(m.start(), "1.1.1", f"img alt is a file name: {a['alt']}")
```

On the given snippet the audit now reports `[(1, '1.1.1', 'img alt is a file name: a.png')]`, and the first kiosk page still has twelve problems, because none of its images has an `alt` at all. In Java the same test is `a.getOrDefault("alt", "").matches("(?i).*\\.(png|jpe?g|gif|svg|webp)")`.

5. The order is: General clinic, Dental clinic, Mother and child, the card field, Print my number, Help and other languages — six stops, following the HTML source. If the print button came first in the HTML but were moved down with CSS (for example with `order` in a flex layout), keyboard and screen-reader users would meet "Print my number" first, before choosing a service, while sighted mouse users would see it last: the focus order would no longer match the visual order, failing 2.4.3 *Focus Order*. Keep the source order and the visual order the same.

6. It touches 3.3.8 *Accessible Authentication (Minimum)*, level AA: a login step must not rely on a cognitive function test — such as transcribing distorted characters or remembering a password — unless an alternative method, a mechanism to help (such as allowing password managers and pasting), or an object-recognition test is offered. A text CAPTCHA fails for many users with dyslexia, low vision or cognitive disabilities. Alternatives include a login link or code sent to the user's phone or email, allowing paste and password managers, and **passkeys**, which replace the password with the device's own unlock (fingerprint, face or PIN). Passkeys need no memory and no transcription, and they are now widely known: the FIDO Alliance's 2026 report found that 90% of consumers surveyed were aware of passkeys and 75% had enabled one on at least one account [12].

References

[1] Microsoft, *Inclusive Design Toolkit*, Microsoft Design, 2016. [Online]. Available: https://inclusive.microsoft.design/

[2] World Health Organization, "Disability," fact sheet, Mar. 7, 2023. [Online]. Available: https://www.who.int/news-room/fact-sheets/detail/disability-and-health

[3] Republik Indonesia, *Undang-Undang Nomor 8 Tahun 2016 tentang Penyandang Disabilitas*, 2016. [Online]. Available: https://peraturan.bpk.go.id/Details/37251/uu-no-8-tahun-2016

[4] Republik Indonesia, *Peraturan Pemerintah Nomor 42 Tahun 2020 tentang Aksesibilitas terhadap Permukiman, Pelayanan Publik, dan Pelindungan dari Bencana bagi Penyandang Disabilitas*, 2020. [Online]. Available: https://peraturan.bpk.go.id/Details/142301/pp-no-42-tahun-2020

[5] European Parliament and Council, "Directive (EU) 2019/882 on the accessibility requirements for products and services (European Accessibility Act)," *Official Journal of the European Union*, L 151, Jun. 7, 2019; applicable from Jun. 28, 2025. [Online]. Available: https://eur-lex.europa.eu/eli/dir/2019/882/oj

[6] ETSI, "EN 301 549 V3.2.1: Accessibility requirements for ICT products and services," Mar. 2021. [Online]. Available: https://www.etsi.org/deliver/etsi_en/301500_301599/301549/03.02.01_60/en_301549v030201p.pdf

[7] W3C, "Web Content Accessibility Guidelines (WCAG) 2.2," W3C Recommendation, Dec. 12, 2024. [Online]. Available: https://www.w3.org/TR/WCAG22/

[8] W3C, "Web Content Accessibility Guidelines (WCAG) 2.2 Approved as ISO/IEC International Standard," W3C News, Oct. 21, 2025. [Online]. Available: https://www.w3.org/news/2025/web-content-accessibility-guidelines-wcag-2-2-approved-as-iso-iec-international-standard

[9] W3C, "W3C Accessibility Guidelines (WCAG) 3.0," W3C Working Draft, Sep. 10, 2026. [Online]. Available: https://www.w3.org/TR/wcag-3.0/

[10] W3C, "Using ARIA," W3C Working Group Note, accessed Sep. 2026. [Online]. Available: https://www.w3.org/TR/using-aria/

[11] WebAIM, "The WebAIM Million: The 2026 report on the accessibility of the top 1,000,000 home pages," Feb. 2026. [Online]. Available: https://webaim.org/projects/million/

[12] FIDO Alliance, "The State of Passkeys: Global Consumer and Workforce Report," May 2026. [Online]. Available: https://fidoalliance.org/wp-content/uploads/2026/05/The-State-of-Passkeys-Global-Consumer-and-Workforce-Report-1.pdf

Chapter 12

Emotional Design, Trust and Ethical UX

People do not only *use* interfaces; they feel something about them, and those feelings decide whether they come back, recommend a service to a neighbor, or never trust it again. This chapter is about designing for those feelings honestly. It follows a small laundry that moved its ordering from WhatsApp chats to a web app with a paid membership, copied the persuasion tricks of large apps, and learned why tricks cost more than they earn. By the end of the chapter you will be able to explain Norman's three levels of emotional design and the aesthetic–usability effect; describe how trust is built and lost in an interface; recognize the common *dark patterns* — deceptive or manipulative designs — by name, and explain how the EU Digital Services Act, the United States Federal Trade Commission and Indonesia's personal-data law (UU PDP 27/2022, with its implementing regulation PP 33/2026) treat them; design a consent screen that people can understand and refuse; and build, in Java and in Python, a linter that finds dark patterns in a described sign-up flow and a ledger that turns consent events into evidence of what a user has and has not allowed.

12.1 Bu Sri's Membership

Bu Sri runs Laundry Wangi, a *kiloan* laundry in a boarding-house neighborhood near a campus. For years her customers ordered by WhatsApp: a message with a photo of the bag, a reply with the price per kilogram, another message when the clothes were ready. It worked, and customers liked her: she answered quickly, she told them honestly when the rainy season would delay drying, and she once returned a forgotten wallet with its money counted.

This year her nephew built her a web app, with order tracking and a membership, *Member Hemat*: Rp25,000 a month for a lower price per kilogram. To "increase conversion", he copied what he had seen in large apps. A red banner shouted "50% off laundry!", a countdown said "Offer ends in 09:59" — and started again at 09:59 on every visit. "Only 3 places left today!" appeared every day. A single pre-ticked box accepted the terms, the privacy policy, WhatsApp promotions and sharing the customer's number with "partner shops", all at once. The big button said "I agree to everything"; refusing meant finding a small grey link that said "No thanks, I like paying more". Joining took two taps; cancelling required a WhatsApp message to customer service and six steps.

For a month the sign-ups rose. Then the complaints began. A student found her number in the promotional broadcast of a motorcycle-repair shop. Several members who wanted to cancel discovered that they could not do it in the app and posted about it in the boarding-house group chat. A regular customer of five years said, in a sentence Bu Sri remembered, "*Kok sekarang Ibu kayak aplikasi pinjol?*" — "Why are you like a loan-shark app now?" The trust she had built in years of honest replies had been spent in weeks.

This chapter explains what went wrong, in the language of emotional design and trust; names each trick; shows what the law now says about them; and rebuilds the sign-up flow so that it is persuasive in the only way that lasts — by being clear and fair.

12.2 The Idea in Plain Words

Every interface makes people feel something: calm or rushed, respected or tricked, capable or stupid. **Emotional design** is the deliberate design of those feelings. At its best it produces *delight*: a laundry app that says "Your clothes are being ironed — ready at 16:00" makes a customer feel looked after at almost no cost. **Trust** is the user's willingness to rely on a system when they cannot check everything themselves: to give it their phone number, their money, their data. Trust grows slowly from many small experiences of competence and honesty, and it can be destroyed by a single discovered trick.

A **dark pattern** — the term Harry Brignull introduced in 2010, now often called a *deceptive pattern* [1] — is a design that makes users do something they did not mean to do, or would not do if they understood, for the benefit of the business. The fake countdown, the pre-ticked box, the tiny "no" link, the shaming refusal text and the cancellation maze of Laundry Wangi are all classic examples. **Ethical UX** is design that persuades only with true information and fair choices, and that treats the user's attention, money, time and data as theirs.

> **DEFINITION.** A *dark pattern* (deceptive pattern) is a user-interface design choice that benefits a service by deceiving, coercing or manipulating users into decisions they would not otherwise make. *Consent* in the sense of personal-data law is a freely given, specific, informed and unambiguous indication of a person's agreement to the processing of their data for a stated purpose, which can be withdrawn.

12.3 The Theory with Evidence

12.3.1 Three levels of emotional design

Norman distinguishes three levels at which a product affects people [2]. The **visceral** level is the immediate, pre-conscious reaction to appearance: attractive or ugly, calm or alarming. It is fast — people form a first aesthetic impression of a web page within about 50 milliseconds [3]. The **behavioral** level is the experience of use: does it work, is it easy, do I feel in control? The **reflective** level is conscious thought and memory: what does this product say about me, do I trust the people behind it, would I recommend it? **Figure 12.1** applies the three levels to the laundry.

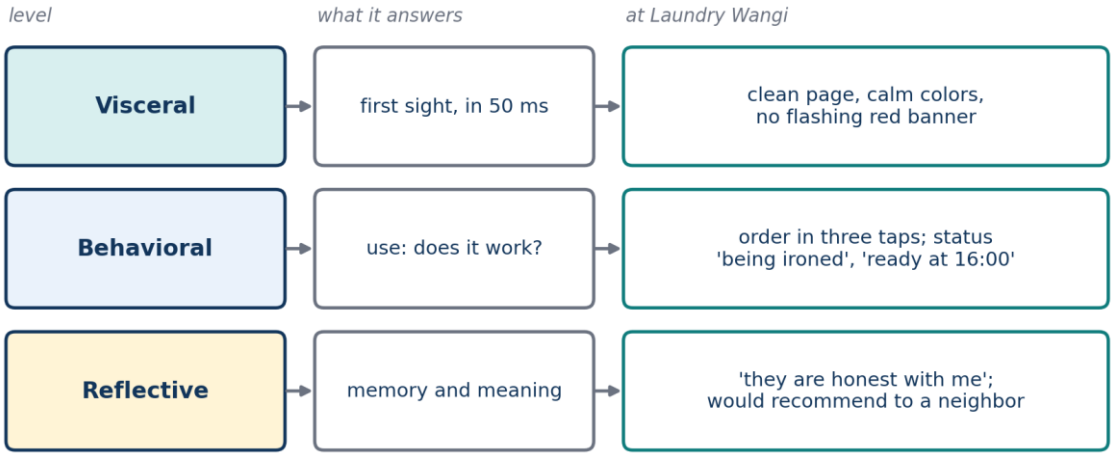


Figure 12.1 Norman's three levels of emotional design applied to Laundry Wangi: the visceral first impression, the behavioral experience of ordering and tracking, and the reflective judgment of honesty that decides whether a customer recommends the laundry.

The levels interact. The **aesthetic–usability effect**, first reported by Kurosu and Kashimura in 1995, is the finding that people rate attractive interfaces as easier to use, even when they are not [4]: the visceral level colors the behavioral judgment. It is a reason to care about appearance, and a warning, because a beautiful interface can hide real usability problems from a test that only asks for opinions. Dark patterns work on the visceral and behavioral levels — urgency, colorful buttons, friction — and damage the reflective level, where trust lives. Bu Sri's customer was speaking from the reflective level when she compared the laundry to a loan-shark app.

12.3.2 How trust is built and lost

Trust in an interface rests on the same three qualities as trust in a person: **competence** (it works, reliably), **benevolence** (it acts in my interest, not only its own) and **honesty** (it tells the truth and keeps its promises). Interfaces signal these in concrete ways: accurate status ("ready at 16:00" and then really ready), clear prices with no surprises at the end, visible ways to undo and to leave, admitting errors plainly, and asking only for the data they need. They lose trust in equally concrete ways: a discovered lie (a countdown that restarts), a hidden cost, data used for an unexpected purpose, and a way in that is much easier than the way out.

Security is part of trust too, and the way it is presented matters. People abandon purchases and sign-ins when they forget a password; the FIDO Alliance's 2026 survey reports that 47% of consumers are likely to abandon a purchase or sign-in because of a forgotten password, while 90% are now aware of **passkeys** and 75% have enabled one on at least one account [5]. A passkey lets the user sign in with the phone's own unlock, which is both safer and less stressful — an example of a design that improves the behavioral and the reflective level at once.

12.3.3 A catalog of dark patterns

Researchers have catalogued dark patterns systematically. Mathur and colleagues crawled about 11,000 shopping websites in 2019 and found 1,818 instances of dark patterns on 1,254 of them, about 11%, including fake countdown timers, fake low-stock messages and confirmshaming [6]. Gray and colleagues brought the many overlapping taxonomies from academia and regulators together in an ontology of dark-pattern knowledge presented at CHI 2024, with high-level strategies such as *sneaking*, *obstruction*, *interface interference*, *forced action* and *social engineering* [7]. **Table 12.1** lists the patterns that appeared at Laundry Wangi, with the ethical alternative for each.

Table 12.1 The dark patterns of the first Laundry Wangi flow, why each is unfair, and the fair alternative.

Pattern	What Laundry Wangi did	Why it is unfair	Fair alternative
False urgency	Countdown that restarts at 09:59	A false claim that pressures a decision	Show a real deadline, or none
False scarcity	"Only 3 places left today!" every day	A false claim of scarcity	State real limits only
Pre-ticked box	Optional purposes already on	Silence is treated as agreement	Optional purposes off by default
Bundled consent	One tick for terms, privacy, promotions and sharing	Refusing one purpose means refusing the service	One choice per purpose
Interface interference	Big filled "agree" button, tiny grey "no" link	The refusal is hard to see and to hit	Equal buttons for equal choices
Confirmshaming	"No thanks, I like paying more"	Guilt instead of information	Neutral wording: "Not now"
Hard to cancel (roach motel)	Join in 2 steps, leave in 6 via WhatsApp	Obstruction traps paying users	Cancel where you joined, as easily

Two quantities make the last two rows measurable. The **friction asymmetry** of a subscription is the number of steps to leave divided by the number of steps to join,

$$A = \frac{s_{out}}{s_{in}} \tag{12.1}$$

and a fair flow has $A \leq 1$: it is never harder to leave than to join. The **prominence ratio** of a two-button choice is the area of the button that the business prefers divided by the area of the other,

$$P = \frac{w_a h_a}{w_r h_r} \tag{12.2}$$

where w and h are the widths and heights of the accept and reject buttons. Equal choices deserve equal weight; this book uses $P \leq 2$ together with the same visual style for both buttons as a practical rule of thumb (it is not a legal threshold). For Laundry Wangi's first screen, $A = 6/2 = 3$ by Equation (12.1), and by Equation (12.2) $P = (320 \times 56)/(90 \times 14) = 17920/1260 = 14.2$: the "agree" button is fourteen times larger than the way to refuse.

12.3.4 What the law says

Dark patterns have moved from design critique into law.

The European Union. Article 25 of the **Digital Services Act** (Regulation (EU) 2022/2065), applicable to all online platforms since February 17, 2024, forbids providers of online platforms to design, organize or operate their interfaces in a way that deceives or manipulates users or otherwise materially distorts or impairs their ability to make free and informed decisions [8]. A CHI 2026 study by Ahuja, Gunawan, Bielova and Santos mapped 59 dark patterns against the three kinds of autonomy violation named in the DSA and identified eight design factors that help decide when a pattern crosses the line [9] — a sign that HCI research and regulation are now working on the same questions. The European Commission has announced a **Digital Fairness Act** to address dark patterns, addictive design and unfair personalization more broadly; as of September 2026 it is expected to be presented in the fourth quarter of 2026 [10].

The United States. The Federal Trade Commission documented the practices in its 2022 staff report *Bringing Dark Patterns to Light* and has acted against them under its general power against

unfair and deceptive practices. Its 2024 "click-to-cancel" rule, which would have required cancellation to be as easy as sign-up, was vacated on procedural grounds by the Eighth Circuit Court of Appeals in July 2025, days before it took effect; on January 30, 2026 the FTC sent a new advance notice of proposed rulemaking to the Office of Management and Budget to restart the process [11]. The principle behind Equation (12.1) therefore remains policy in intent, if not yet a rule, in the United States.

Indonesia. Law No. 27 of 2022 on Personal Data Protection (UU PDP) makes explicit, valid consent one of the lawful bases for processing personal data (Article 20); requires that consent be given in writing or recorded, electronically or otherwise, that a request for consent be clearly distinguishable from other matters and use clear, easily understood language (Article 22); gives the data subject the right to withdraw consent (Article 9); requires the controller to be able to show evidence of the consent it relies on (Article 24); and requires the consent of a parent or guardian for children's data (Article 25) [12]. **Government Regulation No. 33 of 2026** (PP 33/2026), the implementing regulation of UU PDP, was enacted on July 16, 2026 and takes effect on January 16, 2027; among much else it restates that consent must be explicit and valid and given for one or more specific purposes that the controller has communicated [13]. Laundry Wangi's pre-ticked, bundled box and its WhatsApp-only withdrawal sit badly with every one of these provisions.

> **WARNING:** A consent that the user did not actively give, could not understand, or cannot withdraw as easily as it was given is weak evidence at best. Designing the consent screen is not decoration around the law; it *is* how the law's requirements reach the user.

12.3.5 Measuring feelings

Emotional design can be evaluated, not only argued about. Questionnaires such as the **User Experience Questionnaire** (UEQ), with scales for attractiveness, perspicuity, efficiency, dependability, stimulation and novelty, and its eight-item short form UEQ-S, measure how an interface is experienced beyond task success; Chapter 7's SUS measures perceived usability. Behavioral signals matter more than stated feelings when trust is the question: the rate of cancellations, of consent withdrawals, of complaints and of recommendations to others. In Bu Sri's case the reflective level was measured in the most direct way possible — by a sentence in a group chat. Chapter 14 turns such signals into metrics; the ethical rule for this chapter is that a design which raises sign-ups while raising withdrawals and complaints is not a success.

12.3.6 Persuasion that respects the user

Not all persuasion is manipulation. A laundry may legitimately want members, and an interface may legitimately encourage a choice: by explaining a real benefit, by making the recommended option the default when it is truly in the user's interest (a stronger password, a reminder before renewal), or by showing social proof that is true ("chosen by 120 customers in this neighborhood"). Three questions separate persuasion from manipulation. **Transparency:** would the design still work if the user fully understood how it works? A true savings figure still works when understood; a fake countdown stops working the moment it is recognized. **Reversibility:** can the user easily undo the decision the design encourages? A membership that can be cancelled in two steps is a reversible encouragement; one that needs six steps and a chat is a trap. **Alignment:** does the design serve the user's goal as well as the business's? A reminder that the clothes are ready serves both; sharing a customer's number with partner shops serves only one. A design that passes all three may be as persuasive as its makers like.

12.4 A Hands-On Method: the Fair-Flow Review

Before a sign-up, subscription or consent flow is released, review it with **Table 12.2**. Every "no" is a defect to fix before release.

Table 12.2 The fair-flow review: questions to answer "yes" to before releasing a sign-up, subscription or consent flow.

Question	Check
Is every claim true?	Countdowns, stock counts, "most popular" labels, prices
Is every optional purpose off by default?	Promotions, sharing, profiling, notifications
Is there one choice per purpose, in plain language?	Each purpose named with what it means for the user
Do equal choices look equal?	Same style for accept and reject; prominence ratio $P \leq 2$ by Equation (12.2)
Is the refusal worded neutrally?	No guilt, no jokes at the user's expense
Is leaving as easy as joining?	Friction asymmetry $A \leq 1$ by Equation (12.1); same channel
Can consent be withdrawn where it was given?	A visible page in the app, not only a chat with staff
Is consent recorded as evidence?	Who, which purpose, when, which text version, give or withdraw
Is the total cost clear before paying?	Renewal date, price, how to stop auto-renewal, reminder before charging

Most of the questions can be checked by reading a description of the flow; Section 12.5 turns them into a program. The last question and the truthfulness of claims need a person who knows the business.

12.5 Building It Step by Step

12.5.1 Describing a flow

To check a flow automatically, it must be described in a form a program can read. **Listing 12.1** describes Laundry Wangi's first flow in a simple line format: one line per purpose (its id, its text, whether it is required, and whether it is on by default), whether the purposes are bundled into one tick, the two buttons with their labels, sizes and styles, the countdown's behavior, the steps to subscribe and to cancel, and where consent can be withdrawn.

Listing 12.1 `flow_v1.txt`: the first sign-up and membership flow of Laundry Wangi, described line by line.

```
# Laundry Wangi sign-up and membership, first version
purpose|orders|Process my laundry orders|required|on
purpose|promo|Send me promotions by WhatsApp|optional|on
purpose|partners|Share my contact with partner shops|optional|on
bundled|yes
button|accept|I agree to everything|320|56|filled
button|reject|No thanks, I like paying more|90|14|text
countdown|resets
steps|subscribe|2
steps|cancel|6
withdraw|WhatsApp to customer service only
```

12.5.2 The linter

`consent_lint.py` (**Listing 12.2**) reads such a description and applies one rule per row of **Table 12.1**: an optional purpose that is on by default is a pre-ticked box; `bundled|yes` is bundled consent; a prominence ratio above 2 by Equation (12.2), or a reject button that is not styled like the accept button, is interface interference; a refusal label containing a known shaming phrase is confirmshaming; a countdown that resets is false urgency; a friction asymmetry above 1 by Equation (12.1) is a hard-to-cancel flow; and a withdrawal that is possible "only" through some other channel is flagged as well. The list of shaming phrases is short and includes Indonesian phrases, because that is where it will be used; extending it is Exercise 12.4. `ConsentLint.java` (**Listing 12.3**) is the same linter in Java.

Listing 12.2 `consent_lint.py`: checks a described flow for pre-ticked and bundled consent, interface interference, confirmshaming, false urgency and hard exits.

```
"""Check a consent and subscription flow for common dark patterns."""
import sys

SHAMING = ("i like paying more", "i don't want to save", "no thanks, i",
           "saya suka bayar mahal", "saya tidak mau hemat")
MAX_PROMINENCE = 2.0   # accept may be at most twice the area of reject


def load(path):
    flow = {"purpose": [], "button": {}, "steps": {}}
    for line in open(path, encoding="utf-8"):
        line = line.strip()
        if not line or line.startswith("#"):
            continue
        f = line.split("|")
        if f[0] == "purpose":
            flow["purpose"].append(f[1:])
        elif f[0] == "button":
            flow["button"][f[1]] = (f[2], int(f[3]), int(f[4]), f[5])
        elif f[0] == "steps":
            flow["steps"][f[1]] = int(f[2])
        else:
            flow[f[0]] = f[1]
    return flow


def lint(flow):
    found = []
    for pid, text, need, default in flow["purpose"]:
        if not text.strip():
            found.append(("unclear purpose", f"purpose '{pid}' has no text"))
        if need == "optional" and default == "on":
            found.append(("pre-ticked box", f"optional purpose '{pid}' is on"))
    if flow.get("bundled") == "yes":
        found.append(("bundled consent", "one tick covers all purposes"))
    (_, aw, ah, _), (rlabel, rw, rh, rstyle) = (flow["button"]["accept"],
                                                flow["button"]["reject"])
    prominence = (aw * ah) / (rw * rh)
    if prominence > MAX_PROMINENCE or rstyle != "filled":
        found.append(("interface interference",
                      f"accept has {prominence:.1f}x the area of reject, "
                      f"reject is '{rstyle}'"))
    if any(p in rlabel.lower() for p in SHAMING):
        found.append(("confirmshaming", f"reject label: '{rlabel}'"))
    if flow.get("countdown") == "resets":
        found.append(("false urgency", "countdown restarts on every visit"))
```

Listing 12.2 (Continued) *consent_lint.py: checks a described flow for pre-ticked and bundled consent, interface interference, confirmshaming, false urgency and hard exits.*

```
    asym = flow["steps"]["cancel"] / flow["steps"]["subscribe"]
    if asym > 1:
        found.append(("hard to cancel",
                      f"cancel takes {asym:.1f}x the steps of subscribing"))
    if "only" in flow.get("withdraw", "").lower():
        found.append(("hard to withdraw", "withdraw: " + flow["withdraw"]))
    return found, prominence, asym


if __name__ == "__main__":
    for path in sys.argv[1:]:
        found, prominence, asym = lint(load(path))
        print(f"{path}: prominence {prominence:.1f}, cancel/subscribe "
              f"{asym:.1f}, {len(found)} finding(s)")
        for kind, detail in found:
            print(f"  - {kind:<23}{detail}")
```

Listing 12.3 `ConsentLint.java`: the same linter in Java.

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.*;

/** Check a consent and subscription flow for common dark patterns. */
public class ConsentLint {
    static final List<String> SHAMING = List.of("i like paying more",
        "i don't want to save", "no thanks, i", "saya suka bayar mahal",
        "saya tidak mau hemat");
    static final double MAX_PROMINENCE = 2.0; // accept at most 2x reject's area

    record Result(List<String[]> found, double prominence, double asym) {}

    static Map<String, Object> load(String path) throws IOException {
        Map<String, Object> flow = new HashMap<>();
        List<String[]> purposes = new ArrayList<>();
        Map<String, String[]> buttons = new HashMap<>();
        Map<String, Integer> steps = new HashMap<>();
        var lines = Files.readAllLines(Path.of(path), StandardCharsets.UTF_8);
        for (String raw : lines) {
            String line = raw.strip();
            if (line.isEmpty() || line.startsWith("#")) continue;
            String[] f = line.split("\\|");
            switch (f[0]) {
                case "purpose" -> purposes.add(Arrays.copyOfRange(f, 1, f.length));
                case "button" -> buttons.put(f[1], Arrays.copyOfRange(f, 2, 6));
                case "steps" -> steps.put(f[1], Integer.parseInt(f[2]));
                default -> flow.put(f[0], f[1]);
            }
        }
        flow.put("purpose", purposes);
        flow.put("button", buttons);
        flow.put("steps", steps);
        return flow;
    }
```

Listing 12.3 (Continued) *ConsentLint.java: the same linter in Java.*

```
    @SuppressWarnings("unchecked")
    static Result lint(Map<String, Object> flow) {
        List<String[]> found = new ArrayList<>();
        for (String[] p : (List<String[]>) flow.get("purpose")) {
            if (p[1].isBlank())
                found.add(new String[] {"unclear purpose",
                    "purpose '" + p[0] + "' has no text"});
            if (p[2].equals("optional") && p[3].equals("on"))
                found.add(new String[] {"pre-ticked box",
                    "optional purpose '" + p[0] + "' is on"});
        }
        if ("yes".equals(flow.get("bundled")))
            found.add(new String[] {"bundled consent",
                                    "one tick covers all purposes"});
        var b = (Map<String, String[]>) flow.get("button");
        String[] acc = b.get("accept"), rej = b.get("reject");
        double prominence =
            (Double.parseDouble(acc[1]) * Double.parseDouble(acc[2]))
            / (Double.parseDouble(rej[1]) * Double.parseDouble(rej[2]));
        if (prominence > MAX_PROMINENCE || !rej[3].equals("filled"))
            found.add(new String[] {"interface interference", String.format(
                Locale.US, "accept has %.1fx the area of reject, reject is '%s'",
                prominence, rej[3])});
        String label = rej[0].toLowerCase();
        if (SHAMING.stream().anyMatch(label::contains))
            found.add(new String[] {"confirmshaming",
                                    "reject label: '" + rej[0] + "'"});
        if ("resets".equals(flow.get("countdown")))
            found.add(new String[] {"false urgency",
                                    "countdown restarts on every visit"});
        var s = (Map<String, Integer>) flow.get("steps");
        double asym = (double) s.get("cancel") / s.get("subscribe");
        if (asym > 1)
            found.add(new String[] {"hard to cancel", String.format(Locale.US,
                "cancel takes %.1fx the steps of subscribing", asym)});
        String w = (String) flow.getOrDefault("withdraw", "");
        if (w.toLowerCase().contains("only"))
            found.add(new String[] {"hard to withdraw", "withdraw: " + w});
        return new Result(found, prominence, asym);
    }
```

Listing 12.3 (Continued) *ConsentLint.java: the same linter in Java.*

```
    public static void main(String[] args) throws IOException {
        for (String path : args) {
            Result r = lint(load(path));
            System.out.printf(Locale.US,
                "%s: prominence %.1f, cancel/subscribe %.1f, %d finding(s)%n",
                path, r.prominence(), r.asym(), r.found().size());
            for (String[] f : r.found())
                System.out.printf("  - %-23s%s%n", f[0], f[1]);
        }
    }
}
```

12.5.3 The second flow

Listing 12.4 is the repaired flow. The required purpose says why it is required. The optional purposes are off. Each purpose has its own tick. The two buttons have the same size and the same filled style, and their labels describe the two outcomes — “Save my choices” and “Only what is

needed" — without guilt. There is no countdown. Cancelling takes two steps, like joining, and consent is withdrawn on the Account page, on the same kind of screen where it was given.

Listing 12.4 `flow_v2.txt`: the repaired flow, with purposes off by default, equal buttons, no countdown and an exit as easy as the entrance.

```
# Laundry Wangi sign-up and membership, second version
purpose|orders|Process my laundry orders (needed to serve you)|required|on
purpose|promo|Send me promotions by WhatsApp|optional|off
purpose|partners|Share my contact with partner shops|optional|off
bundled|no
button|accept|Save my choices|150|48|filled
button|reject|Only what is needed|150|48|filled
countdown|none
steps|subscribe|2
steps|cancel|2
withdraw|Account page, same screen as sign-up
```

12.5.4 The consent ledger

Consent is not a single moment but a history: a customer gives it, withdraws it, gives it again to a newer text. UU PDP's requirement that the controller can show evidence of consent (Article 24) is met by keeping every event — who, which purpose, what action, when, and which version of the consent text — and deriving the current state from the log rather than overwriting a single flag. **Listing 12.5** is a small log.

Listing 12.5 `consent_log.csv`: consent events with time, user, purpose, action and the version of the consent text shown.

```
time,user,purpose,action,text_version
2026-09-01T08:02,U17,orders,give,v2
2026-09-01T08:02,U17,promo,give,v2
2026-09-03T19:40,U17,promo,withdraw,v2
2026-09-04T10:15,U23,orders,give,v2
2026-09-04T10:15,U23,partners,give,v2
2026-09-20T07:30,U23,promo,give,v3
```

`consent_ledger.py` (**Listing 12.6**) replays the events in time order, keeps the latest action for each pair of user and purpose, and answers the only question the rest of the system should ask: *may we use this user's data for this purpose now?* A pair that never appears in the log has no consent. `ConsentLedger.java` (**Listing 12.7**) does the same.

Listing 12.6 `consent_ledger.py`: replays consent events into the current state and answers whether a purpose is allowed.

```
"""Replay consent events into the current state, as evidence of consent."""
import csv
import sys


def replay(path):
    state = {}   # (user, purpose) -> (action, time, text version)
    with open(path, encoding="utf-8") as f:
        for e in sorted(csv.DictReader(f), key=lambda e: e["time"]):
            state[(e["user"], e["purpose"])] = (e["action"], e["time"],
                                                e["text_version"])
    return state


def may_use(state, user, purpose):
    return state.get((user, purpose), ("none",))[0] == "give"


if __name__ == "__main__":
    state = replay(sys.argv[1] if len(sys.argv) > 1 else "../data/consent_log.csv")
    for (user, purpose), (action, time, ver) in sorted(state.items()):
        now = "may use" if action == "give" else "must NOT use"
        print(f"{user} {purpose:<9}{action:<9}{time}  text {ver}  -> {now}")
    print("U17 promo allowed?", may_use(state, "U17", "promo"))
    print("U23 promo allowed?", may_use(state, "U23", "promo"))
```

Listing 12.7 `ConsentLedger.java`: the same ledger 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.*;

/** Replay consent events into the current state, as evidence of consent. */
public class ConsentLedger {
    // (user, purpose) -> {action, time, text version}
    static TreeMap<String, String[]> replay(String path) throws IOException {
        List<String> lines = Files.readAllLines(Path.of(path),
                                                StandardCharsets.UTF_8);
        List<String[]> events = new ArrayList<>();
        for (String line : lines.subList(1, lines.size()))
            events.add(line.split(","));
        events.sort(Comparator.comparing(e -> e[0]));  // by time, stable
        TreeMap<String, String[]> state = new TreeMap<>();
        for (String[] e : events)
            state.put(e[1] + " " + e[2], new String[] {e[3], e[0], e[4]});
        return state;
    }

    static boolean mayUse(Map<String, String[]> state, String user,
                          String purpose) {
        String[] s = state.get(user + " " + purpose);
        return s != null && s[0].equals("give");
    }

    public static void main(String[] args) throws IOException {
        var state = replay(args.length > 0 ? args[0] : "../data/consent_log.csv");
        for (var e : state.entrySet()) {
            String[] k = e.getKey().split(" "), v = e.getValue();
            String now = v[0].equals("give") ? "may use" : "must NOT use";
            System.out.printf("%s %-9s%-9s%s  text %s  -> %s%n", k[0], k[1], v[0],
                              v[1], v[2], now);
        }
        System.out.println("U17 promo allowed? " + (mayUse(state, "U17", "promo")
                                                    ? "True" : "False"));
        System.out.println("U23 promo allowed? " + (mayUse(state, "U23", "promo")
                                                    ? "True" : "False"));
    }
}
```

12.6 Running It

Lint both flows with `python consent_lint.py ../data/flow_v1.txt ../data/flow_v2.txt`; the Java linter prints the same:

```
../data/flow_v1.txt: prominence 14.2, cancel/subscribe 3.0, 8 finding(s)
  - pre-ticked box         optional purpose 'promo' is on
  - pre-ticked box         optional purpose 'partners' is on
  - bundled consent        one tick covers all purposes
  - interface interference accept has 14.2x the area of reject, reject is 'text'
  - confirmshaming         reject label: 'No thanks, I like paying more'
  - false urgency          countdown restarts on every visit
  - hard to cancel         cancel takes 3.0x the steps of subscribing
  - hard to withdraw       withdraw: WhatsApp to customer service only
../data/flow_v2.txt: prominence 1.0, cancel/subscribe 1.0, 0 finding(s)
```

The first flow has eight findings, one for each problem Bu Sri's customers met, and the two numbers of Section 12.3.3: a prominence ratio of 14.2 and a friction asymmetry of 3.0. The second flow has none, with both ratios at 1.0. **Figure 12.2** shows the two consent screens as a customer sees them on a phone.

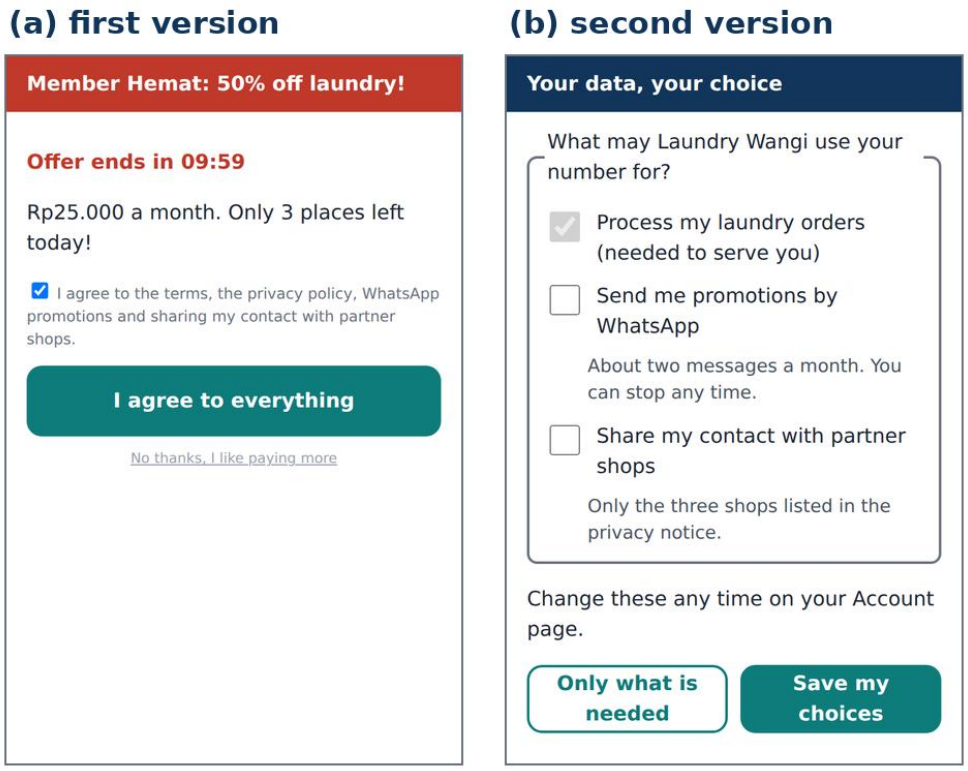


Figure 12.2 The two consent screens on a phone: (a) the first version, with a red banner, a restarting countdown, one pre-ticked box for four purposes, a large “agree” button and a small shaming link; (b) the second version, with one choice per purpose, optional purposes off, plain explanations and two equal buttons.

Now the ledger, `python consent_ledger.py`:

```
U17 orders   give     2026-09-01T08:02  text v2  -> may use
U17 promo    withdraw 2026-09-03T19:40  text v2  -> must NOT use
U23 orders   give     2026-09-04T10:15  text v2  -> may use
U23 partners give     2026-09-04T10:15  text v2  -> may use
U23 promo    give     2026-09-20T07:30  text v3  -> may use
U17 promo allowed? False
U23 promo allowed? True
```

Customer U17 gave consent for promotions on September 1 and withdrew it two days later; the withdrawal is the latest event, so the answer for promotions is “must NOT use”, while orders may still be processed. Customer U23 gave consent for promotions on September 20 to version v3 of the text, which the log records. If Laundry Wangi later changes what promotions mean — more messages, or messages from partners — the text version shows which customers agreed to which wording, and those who agreed to an older version can be asked again rather than assumed to agree.

12.7 Testing It

Listing 12.8 checks that the first flow has all eight findings with the ratios computed by hand ($320 \times 56/(90 \times 14) = 14.2$ and $6/2 = 3.0$), that the second flow is clean, and that in the ledger a later withdrawal wins over an earlier consent and a never-asked purpose is not allowed. **Listing 12.9** makes the same six checks in Java.

Listing 12.8 `test_ch12.py`: tests of the linter and the ledger.

```
"""Checks of the consent linter and ledger (python -m pytest -q)."""
from pathlib import Path
from consent_ledger import may_use, replay
from consent_lint import lint, load

DATA = Path(__file__).resolve().parent.parent / "data"


def test_first_flow_has_all_eight_findings():
    found, prominence, asym = lint(load(DATA / "flow_v1.txt"))
    assert len(found) == 8 and asym == 3.0
    assert round(prominence, 1) == 14.2          # 320*56 / (90*14)


def test_second_flow_is_clean():
    assert lint(load(DATA / "flow_v2.txt"))[0] == []


def test_withdrawal_wins_over_earlier_consent():
    state = replay(DATA / "consent_log.csv")
    assert not may_use(state, "U17", "promo")
    assert may_use(state, "U17", "orders")
    assert not may_use(state, "U99", "orders")   # never asked: no consent
```

Listing 12.9 `Ch12Test.java`: the same checks in plain Java.

```
/** Checks of the consent linter and ledger (java -cp out Ch12Test). */
public class Ch12Test {
    static int failed = 0;

    static void check(boolean ok) { if (!ok) failed++; }

    public static void main(String[] args) throws Exception {
        var r1 = ConsentLint.lint(ConsentLint.load("../data/flow_v1.txt"));
        check(r1.found().size() == 8 && r1.asym() == 3.0);
        check(Math.round(r1.prominence() * 10) == 142);   // 320*56 / (90*14)
        check(ConsentLint.lint(ConsentLint.load("../data/flow_v2.txt"))
              .found().isEmpty());
        var state = ConsentLedger.replay("../data/consent_log.csv");
        check(!ConsentLedger.mayUse(state, "U17", "promo"));
        check(ConsentLedger.mayUse(state, "U17", "orders"));
        check(!ConsentLedger.mayUse(state, "U99", "orders")); // never asked
        System.out.println(failed == 0 ? "all 6 checks passed"
                                       : failed + " failed");
    }
}
```

`python -m pytest -q` prints

```
3 passed in 0.01s
```

and `java -cp out Ch12Test` prints

```
all 6 checks passed
```

12.8 Making It Better

Delight that is true. With the tricks gone, the laundry needs honest reasons to join and to stay, and emotional design supplies them. A status line for every order (“received”, “washing”, “drying — the rain may add half a day”, “ironing”, “ready at 16:00”) serves the behavioral level and, because it is truthful even when the news is bad, the reflective level too. A short thank-you with

the member's savings this month ("You saved Rp18,000 in September") turns an abstract discount into a visible benefit. Neither needs a countdown.

Reminders before charging. Auto-renewal is fair when it is visible: show the next renewal date on the Account page, send a reminder a day before charging, and let the member stop renewal in one step from the reminder itself.

Beyond what a linter sees. The linter checks a *description*; a flow can pass it and still mislead, for example if the text of a purpose is technically true but vague ("to improve our services"). Read every purpose aloud to someone outside the business and ask what they think it allows; if their answer differs from the truth, rewrite it. The linter also cannot see truthfulness: whether "ready at 16:00" is usually right is a question for the laundry's data, and Chapter 14 shows how to measure it.

12.9 Using It for Real

- **Run the fair-flow review (Table 12.2) on every flow that asks for money, data or attention**, and describe the flow in the format of **Listing 12.1** so that the linter can run with every release.
- **Keep a consent ledger, not a consent flag.** Store events with text versions; derive the current state; show it to the user on their Account page with a withdraw control for each purpose.
- **Write purposes in the user's language.** For an Indonesian audience that means clear Bahasa Indonesia first, with concrete consequences: "about two messages a month", "only the three shops listed".
- **Make the exit as visible as the entrance.** The cancel button belongs on the same page as the subscription status.
- **Count the costs of tricks.** Track withdrawals, cancellations, complaints and refunds next to sign-ups. A pattern that raises one while raising the others is borrowing trust at a high interest rate.
- **When in doubt, ask whether you would explain the design to the user's face.** Bu Sri would never have said "*Tinggal 3 kuota*" every day at her counter; her app should not say it either.

12.10 HCI in Practice: Profitina

Profitina, the Indonesian AI-visibility platform, treats consent and honesty as product features. Its self-service compliance with UU PDP 27/2022 includes explicit consent, password hashing with Argon2id, a login rate limit, data export, email and password change, a full account purge that users start themselves, and legal pages published as HTML. In the terms of this chapter, the way out (export, purge) is as available as the way in, and the evidence of consent is kept rather than assumed.

Several product decisions follow the fair-flow review. The **Pro subscription** (Rp50,000 for 30 days, paid via QRIS or by redeeming PROFIT) can renew automatically from the user's balance, and the user is reminded within 24 hours before renewal — a renewal that announces itself. Creating a **user channel** requires holding a balance of 50 PROFIT, which is checked but *not deducted*, and the rule says so; limits (at most two new channels a day, five active) are stated rather than discovered. The **content engine**, which drafts content with a locally run Qwen2.5 model, only serves an AI draft when it is genuinely better than a built-in template, and when no model is available the app *says so* instead of silently serving the template: honesty about

competence builds the reflective trust that a hidden fallback would erode. And the principle written into the product's rules — *the app advises; the user decides* — is the positive form of everything Section 12.3.3 warns against. The runnable case study of this book remains the laundry of Chapter 16.

12.11 Common Mistakes, and How to Avoid Them

- **Copying "growth" tricks from large apps.** What a large app can survive, a small business built on personal trust cannot. Review every copied pattern with **Table 12.1**.
- **Fake urgency and scarcity.** A countdown or stock count that is not true is a false claim, whatever the design intent.
- **Pre-ticked optional boxes and bundled consent.** Silence is not agreement; one tick for many purposes is not a choice.
- **Unequal buttons for equal choices.** Measure the prominence ratio of Equation (12.2); give the refusal the same style.
- **Guilt-trip wording.** Replace confirmshaming with neutral labels that describe the outcome.
- **Exits harder than entrances.** Keep the friction asymmetry of Equation (12.1) at or below 1, in the same channel.
- **A single consent flag overwritten on each change.** It destroys the evidence the law asks for. Keep events.
- **Vague purposes.** "To improve our services" explains nothing. Name the use and its consequence.
- **Mistaking a sign-up spike for success.** Watch withdrawals, cancellations and complaints alongside.

12.12 Chapter Summary

- Norman's three levels — visceral, behavioral, reflective — describe how an interface is felt at first sight, in use and in memory (**Figure 12.1**); the aesthetic–usability effect makes attractive interfaces seem easier, which helps and can mislead.
- Trust rests on competence, benevolence and honesty; it is built by true status, clear costs and easy exits, and lost by discovered tricks. Passkeys show that security can also reduce stress.
- Dark patterns deceive or manipulate users for the service's benefit; common ones include false urgency and scarcity, pre-ticked and bundled consent, interface interference, confirmshaming and hard cancellation (**Table 12.1**). Research found them on about 11% of 11,000 shopping sites.
- Friction asymmetry $A = s_{out}/s_{in}$ (Equation (12.1)) and prominence ratio P (Equation (12.2)) make two of them measurable; fair flows have $A \leq 1$ and equal buttons.
- The EU DSA (Article 25) prohibits deceptive and manipulative interfaces on online platforms; a Digital Fairness Act is expected in late 2026; the FTC's click-to-cancel rule was vacated in July 2025 and a new rulemaking began in 2026; UU PDP requires explicit, clear, withdrawable and evidenced consent, and PP 33/2026 (in force from January 16, 2027) implements it.
- The fair-flow review (**Table 12.2**), the linter and the consent ledger turn these principles into checks: eight findings in the first Laundry Wangi flow, none in the second (**Figure 12.2**), and a ledger that answers "may we use this?" from evidence.

12.13 Exercises

1. A food-delivery app lets users subscribe in one tap from the home screen, but cancelling requires opening Settings, then Account, then Membership, then "Manage", then answering a survey, then confirming, then typing a reason. Compute A with Equation (12.1) and propose a flow with $A \leq 1$.
2. A cookie banner has an "Accept all" button 300 × 48 pixels and a "Settings" link 70 × 16 pixels, with no "Reject all". Compute P with Equation (12.2), name the patterns, and sketch a fair banner.
3. Classify each of these by Norman's level and say whether it builds or risks trust: (a) a pleasant animation when an order is placed; (b) an order status that says "ready" although the clothes are still drying; (c) a monthly message showing the member's actual savings.
4. Add the phrases "*tidak, terima kasih, saya mau rugi*" and "no, i prefer to miss out" to the linter's shaming list, and write a flow file whose only problem is a reject button labeled "No, I prefer to miss out". What does the linter print?
5. In the consent log, customer U23 agreed to promotions to text version v3. Suppose Laundry Wangi introduces version v4, which adds partner offers to the promotions. Explain, with reference to UU PDP, why the ledger should not treat U23's v3 consent as consent to v4, and change `may_use` so that it takes the required text version as a parameter.
6. The DSA's Article 25 speaks of designs that "materially distort or impair" users' ability to make free and informed decisions. Take the "Only 3 places left today!" message and argue both sides: when might a scarcity message be legitimate information, and what makes Laundry Wangi's version a dark pattern?

12.14 Solutions to the Exercises

1. Joining takes $s_{in} = 1$ step; leaving takes $s_{out} = 7$ (Settings, Account, Membership, Manage, survey, confirm, reason), so by Equation (12.1) $A = 7/1 = 7$. A fair design changes both sides. Subscribing becomes two steps — tap "Subscribe", then confirm a screen that states the price, the renewal date and how to cancel — which protects users from accidental one-tap charges. Cancelling becomes two steps as well: a "Cancel membership" button on the same membership card, then one confirmation that states the end date ("Your membership ends on October 31; you will not be charged again"), with an optional, skippable question about the reason. Now $s_{in} = s_{out} = 2$ and $A = 1$.

2. $P = (300 \times 48)/(70 \times 16) = 14400/1120 = 12.9$. The patterns are interface interference (a large button against a small link), and obstruction or forced action by the missing "Reject all": refusing needs more steps than accepting ($A > 1$). A fair banner shows three buttons of the same size and style — "Accept all", "Reject all", "Choose" — with each non-essential category off by default in the "Choose" view.

3. (a) Visceral and behavioral: a short, pleasant confirmation reassures the user that the tap worked; it builds trust as long as it is quick and not in the way. (b) Behavioral and above all reflective: a false status is discovered the moment the customer arrives, and it damages the laundry's honesty in the customer's eyes; it risks trust badly. Better: "drying — rain has delayed us, expected tomorrow at 10:00". (c) Reflective: a true, personal benefit reminds the member why they joined; it builds trust, provided the number is accurate.

4. Add the two phrases in lower case to `SHAMING` in both languages' linter (the list is compared with the lower-cased label). A flow file with the required purpose, no optional purposes on,

`bundled|no`, two equal filled buttons, no countdown, `steps|subscribe|2`, `steps|cancel|2`, a withdraw line without "only", and `button|reject|No, I prefer to miss out|150|48|filled` produces one finding:

```
    - confirmshaming          reject label: 'No, I prefer to miss out'
```

with prominence 1.0 and cancel/subscribe 1.0 in the header line.

5. Consent under UU PDP is given for specific purposes that were communicated to the data subject (Articles 20 and 22, restated by PP 33/2026), and the controller must be able to show evidence of it (Article 24). U23 agreed to the purpose as described in v3; v4 describes a different purpose (partner offers), to which U23 has never agreed. Treating the old consent as covering the new text would process data for a purpose the person was not told about. The fix is to require the version:

```
def may_use(state, user, purpose, version=None):
    action, _, ver = state.get((user, purpose), ("none", "", ""))
    return action == "give" and (version is None or ver == version)
```

Then `may_use(state, "U23", "promo", "v4")` is `False`, and the app shows U23 the new text and asks again.

6. A scarcity message is legitimate information when it is true, specific and relevant: a laundry with capacity for exactly twenty members at a special price may say "20 member places this month, 3 left", if the count is real and changes as places are taken. That helps a customer decide. Laundry Wangi's message was shown every day regardless of any real limit; it was false, and its only function was to create pressure that shortens deliberation. That is the "material distortion" of the DSA's wording: the design changes the decision not by informing but by misinforming. The test is simple — would the message still appear if it were not true? For Laundry Wangi it would, so it is a dark pattern.

References

[1] H. Brignull, "Deceptive Patterns," accessed Sep. 2026. [Online]. Available: https://www.deceptive.design/

[2] D. A. Norman, *Emotional Design: Why We Love (or Hate) Everyday Things*. New York, NY, USA: Basic Books, 2004.

[3] G. Lindgaard, G. Fernandes, C. Dudek, and J. Brown, "Attention web designers: You have 50 milliseconds to make a good first impression!," *Behaviour & Information Technology*, vol. 25, no. 2, pp. 115–126, 2006.

[4] M. Kurosu and K. Kashimura, "Apparent usability vs. inherent usability: Experimental analysis on the determinants of the apparent usability," in *Conf. Companion CHI '95*, Denver, CO, USA, 1995, pp. 292–293.

[5] FIDO Alliance, "The State of Passkeys: Global Consumer and Workforce Report," May 2026. [Online]. Available: https://fidoalliance.org/wp-content/uploads/2026/05/The-State-of-Passkeys-Global-Consumer-and-Workforce-Report-1.pdf

[6] A. Mathur, G. Acar, M. J. Friedman, E. Lucherini, J. Mayer, M. Chetty, and A. Narayanan, "Dark patterns at scale: Findings from a crawl of 11K shopping websites," *Proc. ACM Hum.-Comput. Interact.*, vol. 3, no. CSCW, art. 81, 2019.

[7] C. M. Gray, C. T. Santos, N. Bielova, and T. Mildner, "An ontology of dark patterns knowledge: Foundations, definitions, and a pathway for shared knowledge-building," in *Proc. CHI '24*, Honolulu, HI, USA, 2024, doi: 10.1145/3613904.3642436.

[8] European Parliament and Council, "Regulation (EU) 2022/2065 on a Single Market for Digital Services (Digital Services Act)," Art. 25, *Official Journal of the European Union*, L 277, Oct. 27, 2022. [Online]. Available: https://eur-lex.europa.eu/eli/reg/2022/2065/oj

[9] S. Ahuja, J. Gunawan, N. Bielova, and C. T. Santos, "Dark patterns and the EU Digital Services Act: Mapping autonomy violations and design factors," in *Proc. CHI '26*, 2026, doi: 10.1145/3772318.3791479.

[10] European Parliament, "Digital Fairness Act," Legislative Train Schedule, accessed Sep. 2026. [Online]. Available: https://www.europarl.europa.eu/legislative-train/theme-protecting-our-democracy-upholding-our-values/file-digital-fairness-act

[11] Crowell & Moring LLP, "Clicking All the Right Boxes: FTC Moves to Revive 'Click-to-Cancel' Rule Following Eighth Circuit Vacatur," Feb. 11, 2026. [Online]. Available: https://www.crowell.com/en/insights/client-alerts/clicking-all-the-right-boxes-ftc-moves-to-revive-click-to-cancel-rule-following-eighth-circuit-vacatur

[12] Republik Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*, 2022. [Online]. Available: https://peraturan.bpk.go.id/Details/229798/uu-no-27-tahun-2022

[13] Republik Indonesia, *Peraturan Pemerintah Nomor 33 Tahun 2026 tentang Peraturan Pelaksanaan Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*, enacted Jul. 16, 2026, in force Jan. 16, 2027; summarized in Veritask, "Pengaturan Teknis Pelindungan Data Pribadi … Akhirnya Terbit Lewat PP 33/2026," 2026. [Online]. Available: https://veritask.ai/id/artikel/pengaturan-teknis-pelindungan-data-pribadi-dan-kewajiban-pengendali-serta-prosesor-akhirnya-terbit-lewat-pp-33-2026

Chapter 13

Designing AI Interfaces: Chat, Copilots and Agents

A laundry owner wants a chat assistant that takes orders at midnight, while she sleeps. The assistant will read messages written in a hurry, guess what the customer means, quote a price and book a pickup, and every one of those steps can go wrong in a way that costs money or trust. This chapter is about designing the conversation between people and a system that guesses. By the end of it you will be able to tell a chat interface, a copilot and an agent apart and say what each owes its user; apply the published guidelines for human-AI interaction; place every action of an AI system on a ladder of autonomy according to its risk; show uncertainty in words people understand and ask instead of guessing; give people a way to confirm, undo and reach a human; measure over-reliance, under-reliance and calibration from a log of decisions; build a small order assistant in Java and Python that follows these rules; and meet the transparency duties that apply in 2026 under the EU AI Act and Indonesian law.

13.1 Bu Wati's Midnight Messages

Bu Wati runs Profitina Laundry, the small shop that has served as the running example of this book: three washing machines, two dryers, an ironing table and a WhatsApp number printed on every plastic bag. Most of her customers are students from the boarding houses nearby and young workers who leave for the office before seven. They send their orders when they remember them, and they remember them late: after an assignment is finished, after a night shift, after the last episode of a series. Every morning Bu Wati opens her phone to twenty or thirty messages that arrived between ten at night and two in the morning.

The messages are short and written in a hurry. "Can you wash 4 kg and iron, pick up tomorrow at 9?" is a good one. "I have a bag of laundry, pick up at 16 please" says nothing about the weight or the service. "express wash 6 kilo, today at 13:30" is clear about the speed but not about ironing. Bu Wati answers each one, asks what is missing, quotes the price, and books the pickup for her cousin who drives the motorbike. By the time she has answered, some customers have gone to another shop that answered at midnight.

Her nephew Dimas, who studies informatics, suggests an AI assistant: a program that answers the WhatsApp messages at once, day and night, understands what the customer wants, quotes the price and books the pickup. Bu Wati likes the idea and fears it in equal measure. She has seen screenshots of chatbots that promised refunds nobody had approved, that invented opening hours, and that could not be escaped when a customer wanted a real person. She asks three questions that are the subject of this chapter: *How will my customers know they are talking to a machine? What happens when it misunderstands? And who is to blame when it books the wrong thing?*

Those questions are not about the language model inside the assistant. They are about the interface around it: what the assistant shows, what it asks, what it does on its own and what it leaves to people. Two assistants built on the same model can be pleasant and trustworthy, or

infuriating and dangerous, depending entirely on those design decisions. In 2025 and 2026 the question moved from chat windows to *agents* that act: OpenAI's ChatGPT agent, launched on July 17, 2025, browses websites, fills forms and can make purchases, and is "trained to explicitly ask for your permission before taking actions with real-world consequences" [1]. The design problem Bu Wati faces with one WhatsApp number is the same problem the largest companies face with millions of users.

13.2 The Idea in Plain Words

13.2.1 Chat, copilot and agent

Three kinds of AI interface are common in 2026, and it helps to name them before designing one.

> **DEFINITION.** A *chat interface* answers in words: the person asks, the system replies, and the person decides what to do with the reply. A *copilot* works inside another tool and proposes changes the person can accept or reject, such as a suggested reply in an email program or a suggested line in a code editor. An *agent* takes actions in the world on the person's behalf, such as booking, paying, sending or deleting, often in several steps without a new instruction for each one.

The line between the three is the line of *consequence*. A chat reply that is wrong wastes a minute. A copilot suggestion that is wrong and accepted without reading becomes the person's own mistake. An agent action that is wrong has already happened: the motorbike has driven to the wrong boarding house, or the money has left the account. The more an AI system can do, the more its interface must help people see what it is about to do and stop it.

Bu Wati's assistant is a small agent. It chats (it asks what is missing), it proposes (it shows the order it understood), and it acts (it books the pickup). Each part needs a different kind of care.

13.2.2 Four questions every AI interface must answer

Classic interfaces are *deterministic*: the same button does the same thing every time, and Chapter 3 built much of good design on that consistency. An AI system guesses, and a guess can be wrong in a new way each time. The person using it therefore needs answers to four questions that a light switch never raises, collected in **Table 13.1**.

Table 13.1 The four questions a person has about any AI system, and how the order assistant answers each one.

Question the person has	What the interface must show	In Bu Wati's assistant
What can it do, and how well?	Its scope and its limits, before the first use	"Automated assistant" line; it takes orders, it does not handle complaints
How sure is it about this answer?	Uncertainty in a form people understand	"sure", "likely – please check", "assumed – please check" next to each part of the order
What will it do next?	The action it proposes, before it happens	"Book this pickup? (yes / no)" with the full order and price
How do I correct, stop or undo it?	A way back, and a way to a person	"no" to change, UNDO for one message, STAFF to reach Mbak Dewi

These four questions summarize a large body of research, which Section 13.3 presents with its evidence. They also give a quick test for any AI product you meet: if you cannot answer one of the four from the screen alone, the design has a gap.

13.2.3 A worked example by hand

Take the message “I have a bag of laundry, pick up at 16 please”. Read it as the assistant must.

1. *Service.* No word for washing, ironing or folding appears. The shop’s most common service is wash and fold, so the assistant may *assume* it, but it must say that it assumed.
2. *Weight.* No number with “kg” or “kilo” appears. “A bag” could be two kilograms or eight. The price depends on it, so guessing is not acceptable; the assistant must ask. A good question also offers a way out for people who do not know: “we weigh it at pickup and send the exact price before washing”.
3. *Speed.* No word such as “express” or “today”. Assume regular, and say so.
4. *Pickup.* “at 16” gives an hour but no day. Tomorrow is the likely meaning at night, but it is only likely.

The customer answers “about 2.5 kg”. Now the assistant can propose: wash and fold (assumed), 2.5 kg (sure), regular (assumed), tomorrow 16:00 (likely). The shop charges a minimum of 3 kg, so the price is 3 kg times Rp8,000, which is Rp24,000, and the assistant must say *why* it charges for 3 kg when the customer said 2.5. Only after the customer answers “yes” does it book.

The price rule, written once so that every part of the program uses the same formula, is

$$P = r_s \cdot \max(w,\, w_{\min}) \cdot e \tag{13.1}$$

where r_s is the rate per kilogram of service s, w the weight in kilograms, $w_{\min} = 3$ the minimum charge, and e equal to 1.5 for express and 1 otherwise. For this order, Equation (13.1) gives $P = 8000 \cdot \max(2.5,3) \cdot 1 = 24000$ rupiah.

13.3 The Theory, with Evidence

13.3.1 Guidelines for human-AI interaction

In 2019 a team at Microsoft collected more than 150 design recommendations for AI products from industry and research, reduced them to 18 guidelines, and tested them with 49 designers on 20 real AI products; the result, published at CHI 2019, is still the most cited checklist of the field and is maintained as the HAX Toolkit [2]. The guidelines are grouped by *when* they apply: initially, during interaction, when the system is wrong, and over time. **Table 13.2** lists the ones that matter most for a small assistant.

Table 13.2 A selection of the 18 guidelines for human-AI interaction and how each shapes the order assistant.

Phase	Guideline (paraphrased from [2])	What it means for the order assistant
Initially	G1 Make clear what the system can do	Say "automated assistant" and what it handles
Initially	G2 Make clear how well it can do it	Show confidence per part of the order
During	G4 Show contextually relevant information	Show the price and the minimum-charge reason with the order
During	G5 Match relevant social norms	Polite, short, in the customer's register
When wrong	G8 Support efficient dismissal	"no" discards the proposal at once
When wrong	G9 Support efficient correction	Change one part without retyping the whole order
When wrong	G10 Scope services when in doubt	Ask for the weight instead of guessing it
When wrong	G11 Make clear why the system did what it did	Say "assumed" when a default was used
Over time	G13 Learn from user behavior	Remember what the customer already said in this chat
Over time	G17 Provide global controls	STAFF to reach a person at any time

Google's People + AI Guidebook, from its People + AI Research (PAIR) group, covers the same ground from a product team's point of view, with chapters on user needs, mental models, explainability and trust, feedback and control, and "errors and graceful failure" [3]. Its central advice for trust is that explanations should help people calibrate how much to rely on the system, not persuade them to rely on it more. Both documents agree on the point that matters most for Bu Wati: an AI feature should be designed around what happens when it is wrong, because it will be.

13.3.2 Levels of automation and the autonomy ladder

Long before language models, engineers of aircraft and factories asked how much a machine should decide on its own. Parasuraman, Sheridan and Wickens proposed in 2000 that automation be judged separately for four functions (acquiring information, analyzing it, deciding, and acting) and that each function can be automated to a different level, from "the computer offers no assistance" to "the computer acts and ignores the human" [4]. Their key message was that the right level is chosen from the consequences: higher automation of *actions* is acceptable only when errors are cheap or reversible.

Horvitz's principles of mixed-initiative interfaces, written in 1999 for early assistants, add the economic argument: an assistant should act on its own only when the expected benefit of acting exceeds the expected cost of being wrong, and it should ask when it is unsure, because asking is cheaper than repairing [5].

Figure 13.1 turns these ideas into a four-rung ladder for everyday design. At the bottom the AI only suggests and the person acts. On the second rung the AI proposes a complete action and nothing happens until the person confirms. On the third it acts at once but the action is easy to undo. On the top rung it acts and then reports. The rung is chosen *per action*, not per product: Bu Wati's assistant suggests the service, proposes and waits for a pickup booking, and could send the daily "your laundry is ready" reminder on its own.

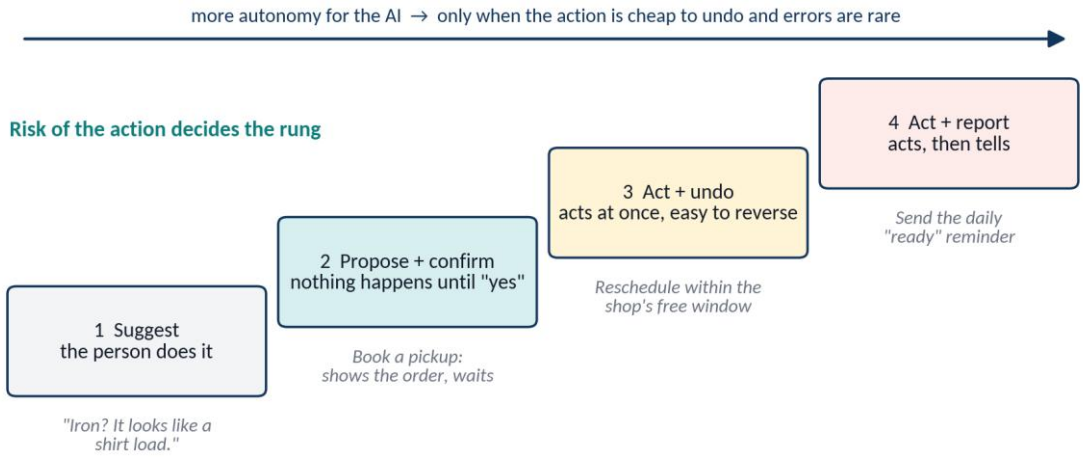


Figure 13.1 The autonomy ladder: the risk and reversibility of each action decide how much the AI may do without asking, with an example from the laundry for every rung.

> **KEY INSIGHT:** Autonomy is not a property of the AI; it is a property of each action. The same assistant should suggest one thing, ask before another, and simply do a third. A payment, a deletion or a message sent in the customer's name sits low on the ladder however good the model is.

13.3.3 Trust, reliance and calibration

People do not simply trust or distrust a system; they *rely* on it or not, one decision at a time. Lee and See's review of trust in automation defines the goal as *appropriate reliance*: relying on the automation when it is right and not relying on it when it is wrong [6]. Two errors are possible. *Over-reliance* (also called automation bias) is accepting a suggestion that is wrong; *under-reliance* is rejecting one that is right. Given a log in which each case records whether the AI was right and whether the person accepted it, the two rates are

$$\text{over-reliance} = \frac{\text{accepted and AI wrong}}{\text{AI wrong}} \tag{13.2}$$

$$\text{under-reliance} = \frac{\text{rejected and AI right}}{\text{AI right}} \tag{13.3}$$

and the share of final decisions that are right, which is what the shop actually cares about, is

$$\text{team right} = \frac{\text{accepted and AI right} + \text{rejected and AI wrong}}{\text{all cases}} \tag{13.4}$$

A CHI 2026 analytical review of human-AI decision-making studies by Raees, Khan, Lykourentzou and Papangelis finds that, across the literature, "people over-rely on AI advice without analytically engaging with it", and that the interventions which make people engage, such as asking them to decide first or showing why the AI suggests what it suggests, help reliance but cost time and effort [7]. There is no free lunch: every design that reduces over-reliance adds a little friction, and the designer must decide where that friction belongs.

Reliance can only be appropriate if the confidence the system shows is honest. A system is *calibrated* when, among all the cases in which it says it is 80% sure, it is right about 80% of the time. Modern neural networks are often overconfident [8]. The usual measure is the *expected*

calibration error: sort the cases into B bins by stated confidence, and in each bin b compare the mean confidence $\text{conf}(b)$ with the share actually right $\text{acc}(b)$, weighting by the number of cases n_b out of N:

$$\text{ECE} = \sum_{b=1}^{B} \frac{n_b}{N} |\text{acc}(b) - \text{conf}(b)| \tag{13.5}$$

An ECE of 0 means perfectly honest confidence; an ECE of 0.10 means the stated confidence is off by ten percentage points on average. When the ECE is high, showing a percentage misleads, and words such as “sure” and “please check”, chosen from measured thresholds, serve people better.

13.3.4 Agents in 2025 and 2026

Two developments changed AI interfaces between 2025 and 2026. The first is agents that operate software as a person does. ChatGPT agent, for example, narrates what it is doing on screen, asks permission before consequential actions such as a purchase, requires the person’s active oversight for tasks such as sending email, and lets the person interrupt, take over the browser or stop the task at any point [1]. These are exactly rungs two and three of the ladder, together with the “global controls” of guideline G17.

The second is a common way for agents to reach tools and data. The Model Context Protocol (MCP), introduced by Anthropic in late 2024, lets an AI application connect to “servers” that expose tools such as a calendar, a database or a shop’s order book. In December 2025 MCP was contributed to the new Agentic AI Foundation under the Linux Foundation, together with other agent projects [9]. For interface design this matters because an agent that can reach many tools needs an interface that shows *which* tool it is about to use and with *what* data, and lets the person approve that use.

13.3.5 The rules in 2026: disclosure and consent

Designing an AI interface in 2026 also means meeting legal duties that did not exist a few years earlier.

In the European Union, Article 50 of the AI Act requires that people be informed when they interact with an AI system such as a chatbot, unless it is obvious, and that AI-generated or manipulated content be marked or disclosed [10]. These transparency duties apply from August 2, 2026. A “digital omnibus” agreed in 2026 postponed the high-risk obligations of the Act to December 2027 and August 2028, but left the Article 50 duties in place, with only a short grace period to December 2, 2026 for the technical marking of content by systems already on the market [11]. An Indonesian shop that serves customers in Europe, or a product sold there, must follow these rules.

In Indonesia, the Personal Data Protection Law (UU PDP 27/2022) governs any assistant that handles names, addresses and phone numbers. Its implementing regulation, PP 33/2026, requires consent that is given freely, specifically and unambiguously, not bundled with terms and conditions, preceded by short and accurate information, withdrawable at any time, and it forbids deceptive consent requests and degrading a service merely because consent is withheld [12]. Two presidential regulations on artificial intelligence, one on a national AI roadmap for 2026–2029 and one on AI ethics that sorts uses into prohibited, strictly supervised and lower-risk, were awaiting the President’s signature as of September 2026 [13]. The practical rule for a designer is

simple and holds in every jurisdiction: say that it is a machine, say what data it uses, and let the person reach a human.

13.4 A Method: The AI Interaction Canvas

Principles become useful when they become a routine. **Table 13.3** is a one-page canvas to fill in for every AI feature before any code is written. Each row forces one decision that the four questions of **Table 13.1** require.

Table 13.3 The AI interaction canvas, filled in for the laundry's order assistant.

Row	Decide and write down	Order assistant
1. Job	What the AI does, in one sentence, and what it does not do	Takes pickup orders; not complaints, not refunds
2. Disclosure	How the person learns it is AI, and when	First line of every chat
3. Inputs and data	What it reads and stores; consent needed	Message text, pickup address; consent at first order
4. Uncertainty	How confidence is shown; thresholds	Words: sure ≥ 0.9, likely ≥ 0.6, else assumed
5. Ask or assume	Which missing parts it asks for, which it may assume	Asks weight and pickup time; may assume service and speed
6. Actions and rungs	Every action with its rung on the ladder	Quote: rung 1; booking: rung 2; reminder: rung 4
7. Undo and stop	How each action is reversed or stopped	"no" before booking; UNDO for one message after
8. Human handoff	How to reach a person, and what the person sees	STAFF; the whole chat is passed on
9. Measures	What is logged to check reliance and calibration	Proposal, confidence, accepted or changed, outcome
10. Failure	What happens when the model is unavailable	Say so and pass to a person; never a silent template

Fill the canvas with the people who own the risk. For Bu Wati that is herself and her cousin who drives the motorbike: he knows that a pickup booked for "at 16" without a day is the one that goes wrong. A canvas that only the programmer filled in usually leaves rows 7, 8 and 10 empty, because a programmer imagines the system working.

A short checklist completes the method. Before release, a person who did not build the assistant should be able to answer "yes" to each of these from the screen alone: *Do I know it is a machine? Can I see what it understood? Do I know which parts it guessed? Can I say no? Can I undo? Can I reach a person? Does it ask before it spends my money or acts in my name?*

13.5 Building the Order Assistant

The assistant in this chapter does not use a language model. It reads the message with a few keyword rules, because the subject here is the interface around the guess, not the guess. The rules produce the same kind of output a model would: a value for each part of the order, called a *slot*, and a confidence for each slot. Replacing the rules by a model later changes only the function `parse`, and everything this chapter builds around it still applies.

13.5.1 The program in Python

Listing 13.1 is the whole assistant. The function `parse` turns one message into four slots with confidences: the service (ironing is recognized with 0.95, a bare “wash” with 0.8, nothing at all falls back to wash and fold with 0.4), the weight (a number followed by “kg” or “kilo”), the speed and the pickup time. The function `label` turns a confidence into the words of **Table 13.3**. The class `Assistant` keeps the conversation state: the slots it has already heard, a pending proposal waiting for yes or no, and the last booking, which can be undone during one more message.

Listing 13.1 `laundry_assistant.py`: the order assistant, with confidence words, questions for missing slots, confirmation, undo and a way to a person.

```
"""Profitina Laundry order assistant: propose, show confidence, ask, then act."""
import re
import sys

RATES = {"wash + fold": 8000, "wash + iron": 10000, "iron only": 6000}
MIN_KG = 3.0
DISCLOSURE = "[Automated assistant. Type STAFF at any time to reach a person.]"


def label(conf):
    # Words, not percentages: customers read words faster than numbers.
    if conf >= 0.9:
        return "sure"
    if conf >= 0.6:
        return "likely - please check"
    return "assumed - please check"


def parse(text):
    """Turn one message into slots, each a (value, confidence) pair."""
    t = text.lower()
    slots = {}
    if "iron only" in t or "setrika saja" in t:
        slots["service"] = ("iron only", 0.95)
    elif "iron" in t or "setrika" in t:
        slots["service"] = ("wash + iron", 0.95)
    elif "wash" in t or "cuci" in t:
        slots["service"] = ("wash + fold", 0.8)
    else:
        slots["service"] = ("wash + fold", 0.4)
    m = re.search(r"(\d+(?:[.,]\d)?)\s*(kg|kilo)", t)
    if m:
        slots["kg"] = (float(m.group(1).replace(",", ".")), 0.95)
    else:
        slots["kg"] = (None, 0.0)
    fast = any(w in t for w in ("express", "kilat", "urgent", "today"))
    slots["speed"] = ("express", 0.9) if fast else ("regular", 0.5)
    m = re.search(r"(tomorrow|today)?\s*at\s*(\d{1,2})(?::(\d\d))?", t)
    if m:
        day = m.group(1) or "tomorrow"
        conf = 0.95 if m.group(1) else 0.7
        slots["pickup"] = ("%s %02d:%s" % (day, int(m.group(2)),
                                           m.group(3) or "00"), conf)
    else:
        slots["pickup"] = (None, 0.0)
    return slots
```

Listing 13.1 (Continued) *laundry_assistant.py: the order assistant, with confidence words, questions for missing slots, confirmation, undo and a way to a person.*

```
def price(slots):
    kg = max(slots["kg"][0], MIN_KG)
    tenths = round(kg * 10)
    total = RATES[slots["service"][0]] * tenths // 10
    if slots["speed"][0] == "express":
        total = total * 3 // 2
    return kg, total


class Assistant:
    def __init__(self):
        self.known = {}          # slots remembered across messages
        self.pending = None      # a proposal waiting for yes/no
        self.booked = None       # the last booking, undoable for one turn
        self.undo_turns = 0

    def reply(self, text):
        t = text.strip().lower()
        self.undo_turns -= 1
        if t in ("staff", "human", "person"):
            return ["Passing you to Mbak Dewi (07:00-21:00).",
                    "She sees this whole chat; no need to repeat it."]
        if t == "undo":
            if self.booked and self.undo_turns >= 0:
                self.booked = None
                return ["Cancelled. Nothing will be picked up."]
            return ["There is nothing left to undo."]
        if self.pending and t in ("yes", "ok", "confirm", "ya"):
            self.booked, self.pending = self.pending, None
            self.undo_turns = 1
            return ["Booked: pickup %s." % self.booked["pickup"][0],
                    "Type UNDO in your next message to cancel."]
        if self.pending and t in ("no", "change", "tidak"):
            self.pending = None
            return ["No problem, nothing was booked.",
                    "Tell me what to change."]
        for name, (value, conf) in parse(text).items():
            # Keep what the customer said before unless the new message
            # says it more clearly.
            if name not in self.known or conf > self.known[name][1]:
                self.known[name] = (value, conf)
        slots = self.known
        if slots["kg"][0] is None:
            return ["How many kilograms, roughly? If you are not sure,",
```

Listing 13.1 (Continued) *laundry_assistant.py: the order assistant, with confidence words, questions for missing slots, confirmation, undo and a way to a person.*

```
                    "we weigh it at pickup and send the exact price",
                    "before washing."]
        if slots["pickup"][0] is None:
            return ["When should we pick it up? Example: tomorrow at 9."]
        kg, total = price(slots)
        lines = ["Here is what I understood:"]
        for name in ("service", "kg", "speed", "pickup"):
            value, conf = slots[name]
            shown = "%g kg" % value if name == "kg" else value
            lines.append("  %-8s %-16s (%s)" % (name, shown, label(conf)))
        lines.append("  price    %g kg x Rp%s%s = Rp%s" % (
            kg, format(RATES[slots["service"][0]], ","),
            " x 1.5" if slots["speed"][0] == "express" else "",
            format(total, ",")))
        if kg > slots["kg"][0]:
            lines.append("  (minimum charge is %g kg)" % MIN_KG)
        lines.append("Book this pickup? (yes / no)")
        self.pending = dict(slots)
        return lines


def run(path):
    number = 0
    bot = None
    for raw in open(path, encoding="utf-8"):
        line = raw.rstrip("\n")
        if line.startswith("#"):
            continue
        if not line.strip():
            bot = None
            continue
        if bot is None:
            number += 1
            bot = Assistant()
            print("=== Conversation %d ===" % number)
            print("Bot : " + DISCLOSURE)
        print("You : " + line)
        for i, out in enumerate(bot.reply(line)):
            print(("Bot : " if i == 0 else "      ") + out)


if __name__ == "__main__":
    run(sys.argv[1] if len(sys.argv) > 1 else "../data/conversations.txt")
```

Four design decisions are visible in the code. The disclosure is printed before anything else. When the weight or the pickup time is missing the assistant asks instead of guessing (guideline G10), and its question about the weight offers a way out for a customer who does not know. Every slot the assistant proposes carries its label, so “assumed” appears exactly where a default was used (guideline G11). And the merging loop keeps what the customer said in an earlier message unless a new message says it more clearly, so that the customer never has to repeat the whole order (guideline G13).

13.5.2 The program in Java

Listing 13.2 is the same assistant in Java. A `record Slot` holds a value and its confidence; the maps are `LinkedHashMap`s so that the slots are shown in a fixed order, as in Python.

Listing 13.2 `LaundryAssistant.java`: the same assistant in Java.

```java
import java.nio.file.*;
import java.util.*;
import java.util.regex.*;

/** Profitina Laundry order assistant: propose, show confidence, ask, act. */
public class LaundryAssistant {
    static final Map<String, Integer> RATES = Map.of(
        "wash + fold", 8000, "wash + iron", 10000, "iron only", 6000);
    static final double MIN_KG = 3.0;
    static final String DISCLOSURE =
        "[Automated assistant. Type STAFF at any time to reach a person.]";

    record Slot(Object value, double conf) {}

    static String label(double conf) {
        // Words, not percentages: customers read words faster than numbers.
        if (conf >= 0.9) return "sure";
        if (conf >= 0.6) return "likely - please check";
        return "assumed - please check";
    }

    static String num(double x) {           // 4.0 -> "4", 2.5 -> "2.5"
        return x == Math.rint(x) ? String.valueOf((long) x) : String.valueOf(x);
    }

    static boolean any(String t, String... words) {
        for (String w : words) if (t.contains(w)) return true;
        return false;
    }

    static Map<String, Slot> parse(String text) {
        String t = text.toLowerCase();
        Map<String, Slot> s = new LinkedHashMap<>();
        if (any(t, "iron only", "setrika saja"))
            s.put("service", new Slot("iron only", 0.95));
        else if (any(t, "iron", "setrika"))
            s.put("service", new Slot("wash + iron", 0.95));
        else if (any(t, "wash", "cuci"))
            s.put("service", new Slot("wash + fold", 0.8));
        else s.put("service", new Slot("wash + fold", 0.4));
        Matcher m = Pattern.compile("(\\d+(?:[.,]\\d)?)\\s*(kg|kilo)").matcher(t);
        s.put("kg", m.find()
            ? new Slot(Double.parseDouble(m.group(1).replace(",", ".")), 0.95)
            : new Slot(null, 0.0));
        s.put("speed", any(t, "express", "kilat", "urgent", "today")
            ? new Slot("express", 0.9) : new Slot("regular", 0.5));
```

Listing 13.2 (Continued) *LaundryAssistant.java: the same assistant in Java.*

```
        m = Pattern.compile("(tomorrow|today)?\\s*at\\s*(\\d{1,2})(?::(\\d\\d))?")
            .matcher(t);
        if (m.find()) {
            String day = m.group(1) != null ? m.group(1) : "tomorrow";
            double conf = m.group(1) != null ? 0.95 : 0.7;
            String min = m.group(3) != null ? m.group(3) : "00";
            s.put("pickup", new Slot(String.format("%s %02d:%s", day,
                Integer.parseInt(m.group(2)), min), conf));
        } else s.put("pickup", new Slot(null, 0.0));
        return s;
    }

    static long[] price(Map<String, Slot> s) {   // {kg in tenths, rupiah}
        double kg = Math.max((Double) s.get("kg").value(), MIN_KG);
        long tenths = Math.round(kg * 10);
        long total = RATES.get((String) s.get("service").value()) * tenths / 10;
        if (s.get("speed").value().equals("express")) total = total * 3 / 2;
        return new long[] {tenths, total};
    }

    Map<String, Slot> known = new LinkedHashMap<>();  // remembered slots
    Map<String, Slot> pending = null;   // a proposal waiting for yes/no
    Map<String, Slot> booked = null;    // the last booking, undoable
    int undoTurns = 0;

    List<String> reply(String text) {
        String t = text.strip().toLowerCase();
        undoTurns--;
        if (Set.of("staff", "human", "person").contains(t))
            return List.of("Passing you to Mbak Dewi (07:00-21:00).",
                "She sees this whole chat; no need to repeat it.");
        if (t.equals("undo")) {
            if (booked != null && undoTurns >= 0) {
                booked = null;
                return List.of("Cancelled. Nothing will be picked up.");
            }
            return List.of("There is nothing left to undo.");
        }
        if (pending != null && Set.of("yes", "ok", "confirm", "ya").contains(t)) {
            booked = pending;
            pending = null;
            undoTurns = 1;
            return List.of("Booked: pickup " + booked.get("pickup").value() + ".",
                "Type UNDO in your next message to cancel.");
        }
        if (pending != null && Set.of("no", "change", "tidak").contains(t)) {
```

Listing 13.2 (Continued) *LaundryAssistant.java: the same assistant in Java.*

```
            pending = null;
            return List.of("No problem, nothing was booked.",
                "Tell me what to change.");
        }
        for (var e : parse(text).entrySet()) {
            // Keep what the customer said before unless the new message
            // says it more clearly.
            Slot old = known.get(e.getKey());
            if (old == null || e.getValue().conf() > old.conf())
                known.put(e.getKey(), e.getValue());
        }
        var s = known;
        if (s.get("kg").value() == null)
            return List.of("How many kilograms, roughly? If you are not sure,",
                "we weigh it at pickup and send the exact price",
                "before washing.");
        if (s.get("pickup").value() == null)
            return List.of("When should we pick it up? Example: tomorrow at 9.");
        long[] p = price(s);
        double kg = p[0] / 10.0;
        List<String> lines = new ArrayList<>();
        lines.add("Here is what I understood:");
        for (String name : List.of("service", "kg", "speed", "pickup")) {
            Slot sl = s.get(name);
            String shown = name.equals("kg")
                ? num((Double) sl.value()) + " kg" : (String) sl.value();
            lines.add(String.format("  %-8s %-16s (%s)", name, shown,
                label(sl.conf())));
        }
        String service = (String) s.get("service").value();
        lines.add(String.format(Locale.US, "  price    %s kg x Rp%,d%s = Rp%,d",
            num(kg), RATES.get(service),
            s.get("speed").value().equals("express") ? " x 1.5" : "", p[1]));
        if (kg > (Double) s.get("kg").value())
            lines.add("  (minimum charge is " + num(MIN_KG) + " kg)");
        lines.add("Book this pickup? (yes / no)");
        pending = new LinkedHashMap<>(s);
        return lines;
    }
```

Listing 13.2 (Continued) *LaundryAssistant.java: the same assistant in Java.*

```
    public static void main(String[] args) throws Exception {
        String path = args.length > 0 ? args[0] : "../data/conversations.txt";
        int number = 0;
        LaundryAssistant bot = null;
        for (String line : Files.readAllLines(Path.of(path))) {
            if (line.startsWith("#")) continue;
            if (line.isBlank()) { bot = null; continue; }
            if (bot == null) {
                number++;
                bot = new LaundryAssistant();
                System.out.println("=== Conversation " + number + " ===");
                System.out.println("Bot : " + DISCLOSURE);
            }
            System.out.println("You : " + line);
            List<String> out = bot.reply(line);
            for (int i = 0; i < out.size(); i++)
                System.out.println((i == 0 ? "Bot : " : "      ") + out.get(i));
        }
    }
}
```

Two details keep the Java and Python programs printing exactly the same text. Python's `%g` prints 4.0 as "4" and 2.5 as "2.5"; the helper `num` does the same in Java. And the price is computed with the weight in tenths of a kilogram and integer arithmetic, so that Equation (13.1) gives the same rupiah in both languages without rounding surprises.

13.5.3 Running it

The file `data/conversations.txt` holds three test conversations, separated by blank lines. Run the assistant from the `python` folder with `python laundry_assistant.py`, or from the `java` folder with `javac -d out *.java` and `java -cp out LaundryAssistant`. Both print the same transcript:

```
=== Conversation 1 ===
Bot : [Automated assistant. Type STAFF at any time to reach a person.]
You : Hi, can you wash 4 kg of clothes and iron them? Pick up tomorrow at 9.
Bot : Here is what I understood:
        service  wash + iron      (sure)
        kg       4 kg             (sure)
        speed    regular          (assumed - please check)
        pickup   tomorrow 09:00   (sure)
        price    4 kg x Rp10,000 = Rp40,000
      Book this pickup? (yes / no)
You : yes
Bot : Booked: pickup tomorrow 09:00.
      Type UNDO in your next message to cancel.
You : undo
Bot : Cancelled. Nothing will be picked up.
=== Conversation 2 ===
Bot : [Automated assistant. Type STAFF at any time to reach a person.]
You : I have a bag of laundry, pick up at 16 please
Bot : How many kilograms, roughly? If you are not sure,
      we weigh it at pickup and send the exact price
      before washing.
You : about 2.5 kg
Bot : Here is what I understood:
        service  wash + fold      (assumed - please check)
        kg       2.5 kg           (sure)
        speed    regular          (assumed - please check)
        pickup   tomorrow 16:00   (likely - please check)
        price    3 kg x Rp8,000 = Rp24,000
        (minimum charge is 3 kg)
      Book this pickup? (yes / no)
You : yes
Bot : Booked: pickup tomorrow 16:00.
      Type UNDO in your next message to cancel.
You : undo
Bot : Cancelled. Nothing will be picked up.
You : undo
Bot : There is nothing left to undo.
=== Conversation 3 ===
Bot : [Automated assistant. Type STAFF at any time to reach a person.]
You : express wash 6 kilo, today at 13:30
Bot : Here is what I understood:
        service  wash + fold      (likely - please check)
        kg       6 kg             (sure)
        speed    express          (sure)
        pickup   today 13:30      (sure)
        price    6 kg x Rp8,000 x 1.5 = Rp72,000
      Book this pickup? (yes / no)
You : no
Bot : No problem, nothing was booked.
      Tell me what to change.
You : STAFF
Bot : Passing you to Mbak Dewi (07:00-21:00).
```

```
      She sees this whole chat; no need to repeat it.
```

Read the transcript against **Table 13.1**. In the first conversation the customer gave everything except the speed; the assistant proposes, marks “regular” as assumed, books on “yes”, and cancels on “undo”. In the second, the weight is missing, so the assistant asks; when the customer answers “about 2.5 kg” it remembers the pickup hour from the first message, marks the day as only likely, and explains the minimum charge. The second “undo” arrives too late and is refused politely: the undo window is one message long, which is a design choice Bu Wati can change. In the third, the customer declines and then asks for a person, who receives the whole chat.

13.6 Measuring Reliance and Calibration

An assistant that shows confidence must be checked: are its confidences honest, and do people rely on it appropriately? Dimas plans a small study for the next feature, a suggestion to staff about whether a garment is safe for machine washing, based on a photo of its care label. Staff see the AI’s suggestion and accept or override it. Two versions of the screen are compared: version A shows the suggestion only, and version B shows the suggestion with its confidence in words and a one-line reason.

Before running the study with real staff, Dimas writes a *simulated* log with the program in **Listing 13.3** to test the analysis. The simulation builds in two assumptions that the analysis should find: the model is overconfident by eight percentage points, and staff in version B accept less often when the confidence is low or the reason looks wrong.

Listing 13.3 `make_log.py`: writes a seeded, simulated log of 200 reviewed cases for each version of the screen.

```
"""Write the simulated review log of Chapter 13 (seeded, repeatable)."""
import random

rng = random.Random(13)
with open("../data/reliance_log.csv", "w", encoding="utf-8") as f:
    f.write("version,case,confidence,ai_correct,accepted\n")
    for version in ("A", "B"):
        for case in range(1, 201):
            conf = round(rng.uniform(0.5, 1.0), 2)
            correct = rng.random() < conf - 0.08   # the model is overconfident
            if version == "A":                     # suggestion only
                p = 0.90
            else:                                  # confidence + reason shown
                p = 0.93 if conf >= 0.8 else 0.50
                if not correct:
                    p -= 0.25                      # the reason gives it away
            accepted = rng.random() < p
            f.write("%s,%d,%.2f,%d,%d\n" % (version, case, conf,
                                            correct, accepted))
```

Listing 13.4 computes the three rates of Equation (13.2), Equation (13.3) and Equation (13.4) for each version, and the calibration table and ECE of Equation (13.5) with five bins of width 0.1 between 0.5 and 1.0.

Listing 13.4 `reliance.py`: over-reliance, under-reliance, right final decisions and the expected calibration error.

```
"""Measure how staff rely on an AI suggestion, and how honest its confidence is."""
import csv
import sys


def load(path):
    with open(path, encoding="utf-8") as f:
        return [(r["version"], float(r["confidence"]), r["ai_correct"] == "1",
                 r["accepted"] == "1") for r in csv.DictReader(f)]


def reliance(rows):
    right = [r for r in rows if r[2]]
    wrong = [r for r in rows if not r[2]]
    over = sum(r[3] for r in wrong) / len(wrong)          # accepted but wrong
    under = sum(not r[3] for r in right) / len(right)     # rejected but right
    # The final decision is right when staff accept a right suggestion
    # or reject a wrong one.
    team = (sum(r[3] for r in right) + sum(not r[3] for r in wrong)) / len(rows)
    return len(rows), len(right) / len(rows), over, under, team


def calibration(rows, bins=5):
    """Expected calibration error over equal-width confidence bins."""
    table, ece = [], 0.0
    for b in range(bins):
        lo, hi = 0.5 + b * 0.5 / bins, 0.5 + (b + 1) * 0.5 / bins
        inside = [r for r in rows
                  if lo <= r[1] < hi or (b == bins - 1 and r[1] == hi)]
        if not inside:
            continue
        conf = sum(r[1] for r in inside) / len(inside)
        acc = sum(r[2] for r in inside) / len(inside)
        ece += len(inside) / len(rows) * abs(acc - conf)
        table.append((lo, hi, len(inside), conf, acc))
    return table, ece
```

Listing 13.4 (Continued) *reliance.py: over-reliance, under-reliance, right final decisions and the expected calibration error.*

```
def main(path):
    rows = load(path)
    print("version  cases  AI right  over-reliance  under-reliance  team right")
    for v in ("A", "B"):
        n, acc, over, under, team = reliance([r for r in rows if r[0] == v])
        print("%-7s  %5d  %7.1f%%  %12.1f%%  %13.1f%%  %9.1f%%" % (
            v, n, 100 * acc, 100 * over, 100 * under, 100 * team))
    table, ece = calibration(rows)
    print("\nconfidence bin   cases  mean confidence  actually right")
    for lo, hi, n, conf, acc in table:
        print("%.1f - %.1f      %5d  %14.3f  %14.3f" % (lo, hi, n, conf, acc))
    print("expected calibration error (ECE) = %.3f" % ece)


if __name__ == "__main__":
    main(sys.argv[1] if len(sys.argv) > 1 else "../data/reliance_log.csv")
```

Listing 13.5 is the same analysis in Java.

Listing 13.5 `Reliance.java`: the same analysis in Java.

```java
import java.nio.file.*;
import java.util.*;

/** Measure how staff rely on an AI suggestion, and how honest its confidence is. */
public class Reliance {
    record Row(String version, double conf, boolean right, boolean accepted) {}

    static List<Row> load(String path) throws Exception {
        List<Row> rows = new ArrayList<>();
        List<String> lines = Files.readAllLines(Path.of(path));
        for (String line : lines.subList(1, lines.size())) {
            String[] f = line.split(",");
            rows.add(new Row(f[0], Double.parseDouble(f[2]), f[3].equals("1"),
                f[4].equals("1")));
        }
        return rows;
    }

    static double[] reliance(List<Row> rows) {
        int right = 0, wrong = 0, overN = 0, underN = 0;
        for (Row r : rows) {
            if (r.right()) { right++; if (!r.accepted()) underN++; }
            else { wrong++; if (r.accepted()) overN++; }
        }
        // The final decision is right when staff accept a right suggestion
        // or reject a wrong one.
        double team = (double) ((right - underN) + (wrong - overN))
            / rows.size();
        return new double[] {rows.size(), (double) right / rows.size(),
            (double) overN / wrong, (double) underN / right, team};
    }

    public static void main(String[] args) throws Exception {
        List<Row> rows = load(args.length > 0 ? args[0]
            : "../data/reliance_log.csv");
        System.out.println(
            "version  cases  AI right  over-reliance  under-reliance  team right");
        for (String v : List.of("A", "B")) {
            List<Row> part = rows.stream()
                .filter(r -> r.version().equals(v)).toList();
            double[] x = reliance(part);
            System.out.printf("%-7s  %5d  %7.1f%%  %12.1f%%  %13.1f%%  %9.1f%%%n",
                v, (int) x[0], 100 * x[1], 100 * x[2], 100 * x[3], 100 * x[4]);
        }
        int bins = 5;
        double ece = 0;
```

Listing 13.5 (Continued) *Reliance.java: the same analysis in Java.*

```
        System.out.println(
            "\nconfidence bin   cases  mean confidence  actually right");
        for (int b = 0; b < bins; b++) {
            double lo = 0.5 + b * 0.5 / bins, hi = 0.5 + (b + 1) * 0.5 / bins;
            int n = 0, ok = 0;
            double sum = 0;
            for (Row r : rows)
                if ((lo <= r.conf() && r.conf() < hi)
                        || (b == bins - 1 && r.conf() == hi)) {
                    n++; sum += r.conf(); if (r.right()) ok++;
                }
            if (n == 0) continue;
            double conf = sum / n, acc = (double) ok / n;
            ece += (double) n / rows.size() * Math.abs(acc - conf);
            System.out.printf("%.1f - %.1f      %5d  %14.3f  %14.3f%n",
                lo, hi, n, conf, acc);
        }
        System.out.printf("expected calibration error (ECE) = %.3f%n", ece);
    }
}
```

Both print:

```
version  cases  AI right  over-reliance  under-reliance  team right
A          200     66.0%          91.2%           6.1%       65.0%
B          200     69.0%          48.4%          26.1%       67.0%

confidence bin   cases  mean confidence  actually right
0.5 - 0.6         79           0.552           0.443
0.6 - 0.7         74           0.637           0.635
0.7 - 0.8         94           0.744           0.660
0.8 - 0.9         69           0.849           0.812
0.9 - 1.0         84           0.948           0.833
expected calibration error (ECE) = 0.072
```

Three findings stand out. With version A, staff accepted 91.2% of the *wrong* suggestions: the classic over-reliance that the CHI 2026 review describes [7]. Version B cut over-reliance to 48.4%, but under-reliance rose from 6.1% to 26.1%, because staff now also rejected many right suggestions whose confidence was low. The share of right final decisions barely moved, from 65.0% to 67.0%. Showing confidence changed *how* people were wrong more than *how often*.

The second half of the output explains why. **Figure 13.2** draws the calibration table: in every bin except one the model is right less often than it claims, and in the top bin it claims 0.948 but is right 0.833 of the time. The ECE is 0.072. Staff were asked to trust numbers that were not honest.

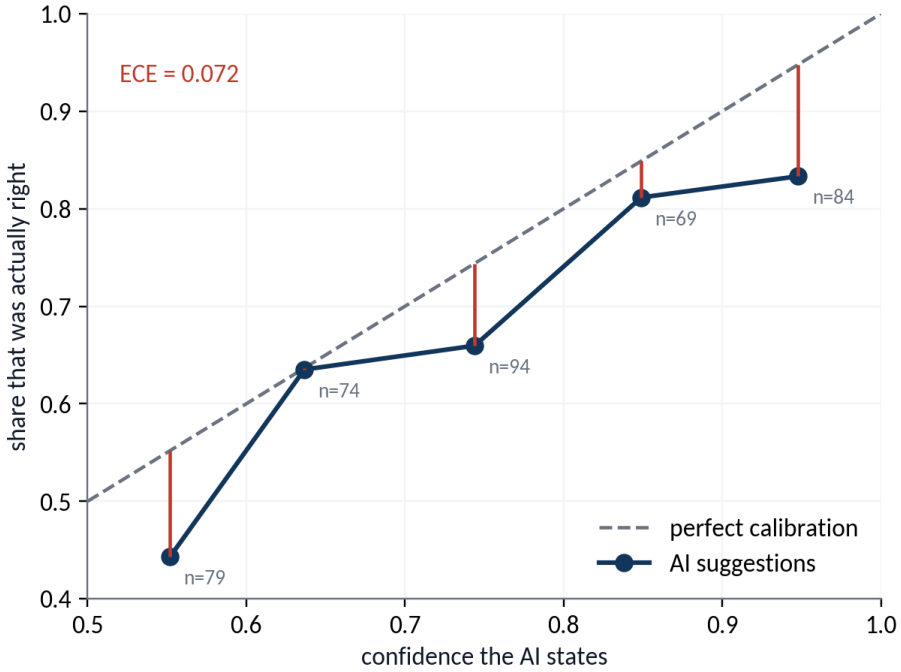


Figure 13.2 Calibration of the simulated care-label model: points below the diagonal mean the model claims more confidence than it earns; the red lines are the gaps that the ECE averages.

The lesson for design is concrete. First calibrate, then show confidence: set the word thresholds from what the model actually achieves in each bin, not from what it claims. Then measure reliance again, because the right interface is the one that raises the share of right final decisions, not the one that merely lowers over-reliance.

13.7 Testing It

Rules that decide what an assistant may do on its own must be tested like any other code. **Listing 13.6** checks the parser, the price with its minimum and express factor, and, most importantly, the confirmation and the undo window.

Listing 13.6 `test_assistant.py`: tests of parsing, price and the undo window.

```
"""Checks of the order assistant; run: python -m unittest test_assistant"""
import unittest

from laundry_assistant import Assistant, parse, price


class AssistantTest(unittest.TestCase):
    def test_parse_and_price(self):
        s = parse("wash 4 kg and iron, tomorrow at 9")
        self.assertEqual(s["service"][0], "wash + iron")
        self.assertEqual(s["kg"][0], 4.0)
        self.assertEqual(s["pickup"][0], "tomorrow 09:00")
        self.assertEqual(price(s)[1], 40000)

    def test_minimum_and_express(self):
        s = parse("express wash 2 kg today at 8")
        self.assertEqual(price(s)[1], 36000)

    def test_confirm_and_undo_window(self):
        bot = Assistant()
        bot.reply("2 kg at 10")
        self.assertTrue(bot.reply("yes")[0].startswith("Booked"))
        bot.reply("hello")
        self.assertTrue(bot.reply("undo")[0].startswith("There is nothing"))


if __name__ == "__main__":
    unittest.main()
```

Listing 13.7 makes the same checks in Java without a test framework, so that it runs with the JDK alone.

Listing 13.7 `AssistantTest.java`: the same checks in Java.

```
import java.util.*;

/** Checks of the order assistant; run: java AssistantTest */
public class AssistantTest {
    static int failed = 0;

    static void check(boolean ok, String what) {
        System.out.println((ok ? "ok   " : "FAIL ") + what);
        if (!ok) failed++;
    }

    public static void main(String[] args) {
        var s = LaundryAssistant.parse("wash 4 kg and iron, tomorrow at 9");
        check(s.get("service").value().equals("wash + iron"), "service");
        check((Double) s.get("kg").value() == 4.0, "weight");
        check(s.get("pickup").value().equals("tomorrow 09:00"), "pickup");
        check(LaundryAssistant.price(s)[1] == 40000, "price 40,000");
        s = LaundryAssistant.parse("express wash 2 kg today at 8");
        check(LaundryAssistant.price(s)[1] == 36000, "minimum 3 kg, express");
        var bot = new LaundryAssistant();
        bot.reply("2 kg at 10");
        check(bot.reply("yes").get(0).startsWith("Booked"), "confirm books");
        bot.reply("hello");
        check(bot.reply("undo").get(0).startsWith("There is nothing"),
            "undo expires after one message");
        System.out.println(failed == 0 ? "all checks passed" : failed + " failed");
        System.exit(failed == 0 ? 0 : 1);
    }
}
```

`python -m unittest test_assistant` reports three passing tests; the Java version prints:

```
ok   service
ok   weight
ok   pickup
ok   price 40,000
ok   minimum 3 kg, express
ok   confirm books
ok   undo expires after one message
all checks passed
```

Notice what the last test protects. “Undo expires after one message” is not a detail of the implementation; it is a promise made in the interface (“Type UNDO in your next message”). A test that pins the promise stops a later change from breaking it silently.

13.8 Making It Better

The assistant of Section 13.5 follows the guidelines, but a week of real use would reveal more. Four improvements follow directly from the theory of Section 13.3.

Explain the guess, briefly. “tomorrow 16:00 (likely – please check)” tells the customer *that* the day was guessed but not *why*. Adding “because you wrote at night and gave no day” costs one line and lets the customer correct the reasoning, not only the result. Keep explanations short: the PAIR guidebook warns that long explanations are skipped [3].

Set the thresholds from data. The words “sure” and “likely” should mean something measurable. Once real orders are logged, compute a calibration table like **Figure 13.2** for each slot, and move the thresholds so that “sure” is right at least nine times in ten.

Ask one question at a time, in the order of cost. The assistant asks for the weight before the pickup time because a wrong weight changes the price. When two slots are missing, ask for the one whose error costs most.

Keep a record the customer can see. Every booking should end with a short summary that the customer can forward or screenshot: service, weight, price, pickup time and the shop’s phone number. A written record turns a disagreement (“I said 9, not 16”) into a quick check.

A fifth improvement is tempting and wrong: letting the assistant book without confirmation when all slots are “sure”. A booking sends a person on a motorbike across town. Even at high confidence it belongs on rung two of the ladder, where a single “yes” costs the customer one second.

13.9 Using It for Real

The canvas of **Table 13.3** and the code of this chapter adapt to any small business that takes orders by chat. Replace the table `RATES` and the words in `parse` with your own services, keep the four questions, and fill in all ten rows of the canvas before you connect the assistant to a real WhatsApp Business account.

Consider Pak Joko’s catering service, which takes orders for rice boxes for office meetings. Its order has slots for the menu, the number of boxes, the delivery time and the address, and one slot that is new: dietary needs. That slot must *never* be assumed. A missing “no peanuts” is not a small error; for a customer with a peanut allergy it is dangerous, so the assistant must ask about allergies every time and pass any mention of an allergy to a person. The general rule: for any slot where a wrong default could harm someone, there is no default.

Three practical points apply to every real deployment. First, log every proposal, its confidences, and what the customer did with it, because that log is what Section 13.6 needs. Ask for consent for this logging as PP 33/2026 requires, in a separate, plain sentence [12]. Second, decide in advance what the assistant does when the model or the network is down: in the spirit of row 10 of the canvas, it says so and passes to a person. Third, read a sample of real chats every week. The keyword rules and the thresholds will need adjusting, and only real messages show how.

13.10 HCI in Practice: Profitina

The same design questions appear in Profitina, the AI Visibility (GEO) platform at app.profitina.com, which measures how ready a brand's website is to be cited by AI answer engines and helps fix what it finds. Its Content Engine drafts GEO content and schema.org and robots.txt help with Qwen2.5 running locally on Profitina's own server, so that user data never leaves the server for AI processing.

Three decisions in Profitina follow this chapter. First, *honest failure*: when no model is available the app says so instead of silently serving a template. That is row 10 of the canvas. Second, a *quality gate*: an AI draft is scored against a built-in template and served only when it is genuinely better; after the September 2026 revision AI drafts win in 5 of 6 benchmark cases, which means the gate still chooses the template once in six and says so. The gate is a calibration decision made on the user's behalf: the system shows AI output only where it has earned it. Third, *the app advises and the user decides*: every score and suggestion comes with the signal that caused it, and nothing is published in the user's name without an explicit action, as the manual "Share to X" with its queue and daily quota shows. The feedback pipeline (submit, curate, dataset, train and measure, release) is the "learn over time" guideline made visible, and users can see where their feedback went.

13.11 Common Mistakes, and How to Avoid Them

- **Hiding that it is a machine.** A human name and a human photo on a chatbot break trust the day customers find out, and in the EU break Article 50 [10]. Say "automated assistant" in the first message.
- **Guessing what must be asked.** Defaulting the weight, the address or an allergy saves one message and costs a wrong order. Only assume slots whose wrong value is cheap and visible.
- **Showing percentages that are not calibrated.** "92% sure" invites people to believe it. Measure calibration first (Equation (13.5)); if it is poor, use words with measured thresholds.
- **Measuring only over-reliance.** An interface that makes people reject everything has no over-reliance and is useless. Report Equation (13.2), Equation (13.3) and Equation (13.4) together.
- **One autonomy level for the whole product.** Placing every action on the same rung either nags people about trivial things or lets the system spend money unasked. Decide per action (**Figure 13.1**).
- **No way out.** A chatbot that cannot pass to a person traps exactly the customers with the hardest problems. Offer the handoff at any time and pass the whole conversation.
- **Forgetting the conversation.** An assistant that asks again for what the customer said two messages ago feels stupid. Keep the slots across messages, as the merging loop does.
- **An undo that is not really one.** "Cancel" that only hides the booking from the customer while the motorbike still leaves is worse than no undo. Test the promise, as **Listing 13.6** does.

13.12 Chapter Summary

- A chat interface answers, a copilot proposes inside another tool, and an agent acts. The more an AI system can do, the more its interface must show what it is about to do and let people stop it.
- Every AI interface must answer four questions (**Table 13.1**): what it can do, how sure it is, what it will do next, and how to correct, stop or undo it.
- The guidelines for human-AI interaction [2] and the People + AI Guidebook [3] turn those questions into concrete rules; **Table 13.2** shows the ones a small assistant needs.
- Autonomy is chosen per action from risk and reversibility, on a ladder from suggest to act-and-report (**Figure 13.1**) [4], [5].
- Appropriate reliance means accepting right suggestions and rejecting wrong ones; over-reliance, under-reliance and right final decisions are measured with Equation (13.2), Equation (13.3) and Equation (13.4). Confidence must be calibrated before it is shown (Equation (13.5)).
- In the simulated study, showing confidence cut over-reliance from 91.2% to 48.4% but raised under-reliance to 26.1%, and right final decisions rose only from 65.0% to 67.0%, because the model was overconfident (ECE 0.072).
- The order assistant, in Java and Python, discloses itself, asks for missing slots, labels assumptions, proposes with a price from Equation (13.1), books only on “yes”, allows undo for one message and passes to a person on request.
- In 2026 the EU AI Act’s Article 50 requires disclosure of AI interaction [10], [11]; in Indonesia, UU PDP 27/2022 and PP 33/2026 govern consent for the personal data an assistant handles [12].

13.13 Exercises

1. By hand, process the message “setrika saja 5 kg, tomorrow at 7” with the rules of `parse`. Give every slot with its confidence and label, and the price from Equation (13.1). Then check your answer by adding the message to a new conversation in `data/conversations.txt`.
2. For each of these actions of a pharmacy’s chat assistant, choose a rung on the ladder of **Figure 13.1** and give one sentence of reasoning: (a) show opening hours; (b) reserve a medicine for pickup; (c) reorder a regular prescription; (d) cancel a reservation made yesterday; (e) send a reminder to take a medicine.
3. A staff study logs 300 cases. The AI was right in 240; staff accepted 228 of those and 36 of the 60 wrong ones. Compute over-reliance, under-reliance and the share of right final decisions with Equation (13.2), Equation (13.3) and Equation (13.4).
4. A model’s cases fall into two bins: 100 cases with mean confidence 0.62 of which 58 are right, and 300 cases with mean confidence 0.91 of which 246 are right. Compute the ECE with Equation (13.5) and say whether you would show this model’s confidence as a percentage.
5. Change the assistant so that the undo window lasts two messages instead of one. Which line changes in each language, which message text must change with it, and which test must be updated?
6. Fill in the ten rows of the canvas of **Table 13.3** for a puskesmas chat assistant that answers questions about opening hours and takes appointment requests. Pay special attention to rows 5, 6 and 8.

13.14 Solutions to the Exercises

1. “setrika saja” matches the first rule, so service = iron only (0.95, sure). “5 kg” gives weight = 5 (0.95, sure). No speed word: regular (0.5, assumed – please check). “tomorrow at 7” gives pickup = tomorrow 07:00 (0.95, sure). The price is $P = 6000 \cdot \max(5,3) \cdot 1 = 30000$ rupiah, so the assistant prints `price    5 kg x Rp6,000 = Rp30,000` and asks “Book this pickup? (yes / no)”. Note that “iron only” must be tested before “iron”, which is why the `if` for “iron only” comes first in `parse`.

2. (a) Rung 4 or simply a chat answer: showing information has no side effect. (b) Rung 2: a reservation holds stock and makes a promise; propose it with the medicine name and the pickup time and wait for “yes”. (c) Rung 2, and never higher: a prescription medicine is a health matter, and the pharmacist must be able to check it; the assistant proposes, the person confirms, and the pharmacist approves. (d) Rung 2 with a clear statement of what will be lost; after cancellation, offer to reserve again, because a cancel cannot always be undone once the stock is sold. (e) Rung 4 only if the person asked for reminders in advance and can stop them with one word; the content must remind, never advise a dose change, and should say “ask the pharmacist or your doctor if anything feels wrong”.

3. AI wrong in $300 - 240 = 60$ cases. Over-reliance $= 36/60 = 0.60$. Under-reliance: rejected and right $= 240 - 228 = 12$, so $12/240 = 0.05$. Right final decisions $= (228 + (60 - 36))/300 = 252/300 = 0.84$. The staff rarely reject a right suggestion but accept most wrong ones: the design should make wrong suggestions easier to spot.

4. Bin 1: acc $= 58/100 = 0.58$, gap $|0.58 - 0.62| = 0.04$, weight $100/400 = 0.25$. Bin 2: acc $= 246/300 = 0.82$, gap $|0.82 - 0.91| = 0.09$, weight 0.75. ECE $= 0.25 \cdot 0.04 + 0.75 \cdot 0.09 = 0.01 + 0.0675 = 0.0775$. The high-confidence bin overstates by nine points; a displayed “91%” would mislead. Show words, and put “sure” only above a threshold where the measured accuracy reaches the level the shop needs.

5. In Python change `self.undo_turns = 1` to `self.undo_turns = 2`; in Java change `undoTurns = 1;` to `undoTurns = 2;`. The promise must change with it: “Type UNDO in your next message to cancel.” becomes, for example, “Type UNDO within your next two messages to cancel.” The test “undo expires after one message” must now send two ordinary messages before the late undo, and a new test should check that an undo in the second message still works. Changing the code without changing the text would make the interface lie.

6. One possible canvas. (1) Job: answers opening hours and service questions, takes appointment *requests*; never gives medical advice. (2) Disclosure: first line, “automated assistant of Puskesmas X”. (3) Data: name, phone, preferred day, service; consent in a separate sentence under PP 33/2026. (4) Uncertainty: words with thresholds. (5) Ask or assume: always ask the service and the day; never assume anything about symptoms. (6) Actions: answering hours, rung 4; an appointment request, rung 2, confirmed by staff before it becomes an appointment; reminders, rung 4 only after opt-in. (7) Undo: cancel a request by one word until staff confirm it. (8) Handoff: “STAFF” at any time; and any message that mentions chest pain, difficulty breathing, heavy bleeding or a baby with high fever is answered at once with “please go to the puskesmas or the nearest emergency service now” and passed to staff. (9) Measures: questions answered, requests confirmed, handoffs. (10) Failure: when the assistant is down, an automatic reply gives the phone number of the registration desk.

References

[1] OpenAI, "Introducing ChatGPT agent: bridging research and action," Jul. 17, 2025. [Online]. Available: https://openai.com/index/introducing-chatgpt-agent/

[2] S. Amershi *et al.*, "Guidelines for human-AI interaction," in *Proc. CHI Conf. Human Factors in Computing Systems (CHI '19)*, Glasgow, UK, 2019, pp. 1–13, doi: 10.1145/3290605.3300233. Maintained as the Microsoft HAX Toolkit: https://www.microsoft.com/en-us/haxtoolkit/ai-guidelines/

[3] Google People + AI Research (PAIR), *People + AI Guidebook*. [Online]. Available: https://pair.withgoogle.com/guidebook-v2/ (accessed Sep. 2026).

[4] R. Parasuraman, T. B. Sheridan, and C. D. Wickens, "A model for types and levels of human interaction with automation," *IEEE Trans. Systems, Man, and Cybernetics – Part A*, vol. 30, no. 3, pp. 286–297, 2000.

[5] E. Horvitz, "Principles of mixed-initiative user interfaces," in *Proc. CHI '99*, Pittsburgh, PA, USA, 1999, pp. 159–166.

[6] J. D. Lee and K. A. See, "Trust in automation: Designing for appropriate reliance," *Human Factors*, vol. 46, no. 1, pp. 50–80, 2004.

[7] M. Raees, V. J. Khan, I. Lykourentzou, and K. Papangelis, "Do people appropriately rely on AI-advice? An analytical review of HCI research on human-AI decision-making," in *Proc. CHI '26*, Barcelona, Spain, Apr. 2026, doi: 10.1145/3772318.3791467.

[8] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, "On calibration of modern neural networks," in *Proc. 34th Int. Conf. Machine Learning (ICML)*, 2017, pp. 1321–1330.

[9] Model Context Protocol, "MCP joins the Agentic AI Foundation," Dec. 9, 2025. [Online]. Available: https://blog.modelcontextprotocol.io/posts/2025-12-09-mcp-joins-agentic-ai-foundation/

[10] European Commission, "Transparency obligations under Article 50 of the AI Act," FAQ, *Shaping Europe's Digital Future*. [Online]. Available: https://digital-strategy.ec.europa.eu/en/faqs/transparency-obligations-under-article-50-ai-act (accessed Sep. 2026). Regulation (EU) 2024/1689.

[11] J. M. Loring, M. Ramsden, and A. R. Lee, "Yes, August 2 still matters: The EU approved a high-risk AI delay, but most transparency obligations remain," Jones Walker LLP, Jul. 16, 2026. [Online]. Available: https://www.joneswalker.com/en/insights/blogs/ai-law-blog/yes-august-2-still-matters-the-eu-approved-a-high-risk-ai-delay-but-most-trans.html

[12] Bisnis.com, "Aturan Turunan UU PDP: Persetujuan Data Konsumen Lebih Ketat," Aug. 29, 2026, on Peraturan Pemerintah No. 33 Tahun 2026 implementing UU No. 27 Tahun 2022. [Online]. Available: https://teknologi.bisnis.com/read/20260829/84/1999753/aturan-turunan-uu-pdp-persetujuan-data-konsumen-lebih-ketat

[13] Selular.id, "Dua Perpres AI Tunggu Tanda Tangan Presiden Prabowo," Sep. 4, 2026. [Online]. Available: https://selular.id/2026/09/dua-perpres-ai-tunggu-tanda-tangan-presiden-prabowo/

Chapter 14

Data-Informed UX: Analytics, Experiments and Metrics

A warung's ordering website gets plenty of visitors, yet the orders do not come. The owner has opinions about why, her son has other opinions, and the web designer has a third set. This chapter is about settling such arguments with what people actually do, measured honestly and with their consent. By the end of it you will be able to choose metrics that follow from goals with the HEART framework; describe an interface as a stream of events and turn the stream into a funnel; read current usage statistics for Indonesia with a critical eye; plan an A/B experiment with a hypothesis, a primary metric, guardrails and a sample size computed in advance; check an experiment for sample ratio mismatch before believing it; compute a difference with its confidence interval and a two-proportion z-test; build all of this in Java and Python; and collect the data in a way that respects UU PDP 27/2022, its implementing regulation PP 33/2026 and current analytics consent rules.

14.1 Bu Tini's Quiet Checkout

Bu Tini sells home-cooked rice meals from a warung in a housing estate: nasi campur, soto, a changing vegetable dish and iced tea. For years her customers ordered by phone or walked in. Last year her son Raka built a small ordering website, linked from her Instagram and her WhatsApp status: pick dishes from the menu, put them in a cart, type the delivery address, pay, done. Hundreds of people open the menu every week. Far fewer orders arrive than Bu Tini expected.

Everybody has an explanation. Bu Tini thinks the photos are not appetizing. Raka thinks the menu is too long. A friend who designs websites says the colors are old-fashioned. Each explanation leads to a different week of work, and each is a guess. What none of them knows is *where* people stop: on the menu, in the cart, at the address form, or at the payment step.

The website can answer that question if it records a small number of well-chosen *events*, one each time a visitor reaches a step. Counting how many visitors reach each step gives a *funnel*, and the funnel shows the step where most people give up. In Bu Tini's funnel the steepest drop turns out to be between the checkout, where people must type their full address on a phone keyboard, and the payment. Raka builds a second version in which a returning customer's address is remembered and payment is one tap with QRIS, Indonesia's standard QR payment. Now there are two versions, and a new question: *is the new one really better, or did it just happen to have a good week?* Answering that question honestly is what a controlled experiment, an A/B test, is for.

14.2 The Idea in Plain Words

14.2.1 What people say and what people do

Chapters 5 and 7 taught methods that ask people and watch them: interviews, surveys, think-aloud tests. Those methods explain *why*. They work with a handful of people. Analytics works the

other way around: it records *what* thousands of people do, without asking, and it cannot say why. A funnel can show that 44% of Bu Tini's visitors abandon the checkout; only a conversation with a few of them reveals that typing "Jl. Kenanga Blok C-12 No. 7, RT 03/RW 05" on a small phone is the reason. The two kinds of evidence need each other. The phrase *data-informed*, rather than *data-driven*, says that numbers inform decisions made by people who also understand the context.

14.2.2 Events, funnels and metrics

> **DEFINITION.** An *event* is a record that something happened in the interface: who (an anonymous user identifier), what (a name such as `add_to_cart`), when, and sometimes a few properties. A *funnel* is an ordered list of steps with the number of users who reached each step. A *metric* is a number computed from events that tracks a goal, such as the share of visitors who complete an order. A *conversion rate* is a metric of the form "users who did X divided by users who could have done X".

For a funnel with steps $1, 2, \dots, K$ and n_k users reaching step k, two rates matter. The *step conversion* says how many of the users at one step reach the next:

$$s_k = \frac{n_k}{n_{k-1}}, \quad k = 2, \dots, K \tag{14.1}$$

and the *overall conversion* says how many of those who started reached the end:

$$c = \frac{n_K}{n_1} = s_2 \cdot s_3 \cdots s_K \tag{14.2}$$

The product form of Equation (14.2) explains why funnels are useful: every step multiplies the loss of the steps before it, so improving the weakest step raises the overall conversion most.

14.2.3 A worked example by hand

Suppose that in one week 1,000 people open Bu Tini's menu, 610 add something to the cart, 430 start the checkout, 245 reach the payment step and 228 complete the order. By Equation (14.1) the step conversions are $610/1000 = 0.61$, $430/610 = 0.705$, $245/430 = 0.570$ and $228/245 = 0.931$. By Equation (14.2) the overall conversion is $228/1000 = 0.228$, and indeed $0.61 \cdot 0.705 \cdot 0.570 \cdot 0.931 \approx 0.228$.

Where should Raka work? The checkout-to-payment step loses 43% of the people who reach it, more than any other. If a better checkout raised that step from 0.570 to 0.66 and nothing else changed, the overall conversion would become $0.61 \cdot 0.705 \cdot 0.66 \cdot 0.931 \approx 0.264$, about 36 more orders per thousand visitors. The same effort spent on the payment step, already at 0.931, could win at most $0.228 \cdot (1/0.931 - 1) \approx 0.017$, or 17 orders. The funnel does not say how to fix the checkout; it says where fixing pays.

14.3 The Theory, with Evidence

14.3.1 Choosing metrics: the HEART framework

The most common mistake with analytics is to measure what is easy (page views, clicks, time on site) instead of what matters. A team at Google proposed the HEART framework in 2010 to fix this for user experience at scale [1]. HEART names five kinds of user-centered metric: *Happiness*

(attitudes such as satisfaction), *Engagement* (depth of use), *Adoption* (new users of a product or feature), *Retention* (users who come back) and *Task success* (effectiveness, efficiency, errors). Not every product needs all five. The discipline lies in the second half of the framework, the *Goals–Signals–Metrics* process: first state the goal in words, then name the signal in user behavior that would show progress toward it, and only then define the metric that counts the signal.

Table 14.1 applies the process to Bu Tini's warung.

Table 14.1 The HEART Goals–Signals–Metrics process applied to the warung's ordering website; engagement is left out on purpose, because more time on an ordering page is not a goal.

Category	Goal	Signal	Metric
Happiness	Customers find ordering easy	Answer to one question after ordering	Share answering 6 or 7 on "ordering was easy" (1–7)
Adoption	Regular walk-in customers try online ordering	First online order by a phone number seen before	New online customers per week
Retention	Online customers order again	Second order within 30 days	30-day repeat rate
Task success	People who want to order can finish	Reaching `order_done` after `view_menu`	Overall conversion, Equation (14.2); time to order
Task success	Orders arrive correct	Complaints about wrong address	Wrong-address orders per 100

Leaving out Engagement is a deliberate choice. For a social network, more time spent may be the goal. For an ordering site, a customer who spends ten minutes has probably struggled. A metric that rises when the experience gets worse is worse than no metric.

14.3.2 Where the users are: Indonesia in 2026

Metrics are always read against the population they come from. According to the survey of the Indonesian Internet Service Providers Association (APJII) published in May 2026, 81.7% of Indonesians use the internet, about 235 million people, up from 80.66% in 2025; penetration is 85.95% in Java but 69.74% in Maluku and Papua, 84.75% in cities and 78.18% in villages, and above 89% for Generation Z and Millennials [2]. A warung in a housing estate in Java therefore serves a population that is almost entirely online, while a similar shop in a rural district of Papua serves one where three in ten people are not.

Device statistics need more care. StatCounter, which counts page views on the websites that carry its tracking code, reported for Indonesia in August 2026 that 60.59% of page views came from desktop computers, 38.98% from mobile devices and 0.43% from tablets [3]. That seems to contradict everyday experience, in which almost everyone orders food on a phone, and it is a useful lesson in reading metrics: page-view shares count *page views on particular websites*, not *people*, and are shifted by where the tracking code is installed, by office and school computers that load many pages, and by automated traffic. The only device mix that matters to Bu Tini is the one in her own logs, which for a food-ordering site linked from Instagram will be overwhelmingly mobile. Always ask what a number counts before deciding what it means.

14.3.3 Controlled experiments

Comparing this week's orders with last week's cannot show that a new checkout works, because many things change from week to week: payday, the weather, a holiday, a competitor's discount. A *controlled experiment* removes those differences by running both versions at the same time and assigning each visitor to one of them at random. Because the assignment is random, the two

groups are alike in everything except the version they see, and a difference in outcome can be attributed to the version. Kohavi, Tang and Xu's book, written from experience with experiments at Microsoft, Google and LinkedIn, is the standard guide [4].

Let group A (the current checkout, the *control*) have n_A users of whom x_A order, and group B (the new checkout, the *treatment*) n_B and x_B. The conversion rates are $p_A = x_A/n_A$ and $p_B = x_B/n_B$. The question is whether the difference $d = p_B - p_A$ is larger than chance alone would produce. Under the null hypothesis that both versions convert equally, the best estimate of the common rate is the pooled rate $\bar{p} = (x_A + x_B)/(n_A + n_B)$, and the test statistic

$$z = \frac{p_B - p_A}{\sqrt{\bar{p}\,(1-\bar{p})\left(\frac{1}{n_A} + \frac{1}{n_B}\right)}} \tag{14.3}$$

follows approximately the standard normal distribution when the groups are large. A $|z|$ above 1.96 corresponds to a two-sided p-value below 0.05. The p-value answers a narrow question: if the two versions were truly the same, how often would a difference at least this large appear by chance?

A p-value alone says nothing about the size of the effect, which is what Bu Tini cares about. The 95% confidence interval for the difference gives both size and uncertainty:

$$d \pm 1.96\sqrt{\frac{p_A(1-p_A)}{n_A} + \frac{p_B(1-p_B)}{n_B}} \tag{14.4}$$

If the whole interval lies above zero, the new version is better; how much better is somewhere within the interval.

Finally, an experiment must be large enough to detect the smallest difference worth acting on, the *minimum detectable effect*. For a two-sided test at the 5% level with 80% power (an 80% chance of detecting a true effect of that size), the number of users needed in *each* group to detect a change from p_1 to p_2 is approximately

$$n = \frac{(z_{\alpha/2} + z_{\beta})^2\,[p_1(1-p_1) + p_2(1-p_2)]}{(p_2 - p_1)^2} \tag{14.5}$$

with $z_{\alpha/2} = 1.96$ and $z_{\beta} = 0.8416$. The squared difference in the denominator is the important part: halving the effect you want to detect multiplies the required sample by four. Computing n *before* the experiment decides how long it must run.

14.3.4 Trustworthy experiments: the traps

Experiments fail in ways that produce confident, wrong numbers. Four traps are common enough to check every time.

Sample ratio mismatch. If visitors are assigned 50/50, the two groups should be about equal in size. A clear imbalance means that something removed users from one group: a bug that crashes the new version on some phones, a redirect that loses tracking, or a bot filter that treats the groups differently. Fabijan and colleagues at Microsoft found such mismatches in a noticeable share of real experiments and argued that a result with a sample ratio mismatch should not be trusted until the cause is found [5]. The check is a chi-square test of the observed group sizes

against the planned split. With $n_A + n_B$ users and a planned 50/50 split, the expected size of each group is $e = (n_A + n_B)/2$ and

$$\chi^2 = \frac{(n_A - e)^2}{e} + \frac{(n_B - e)^2}{e} \tag{14.6}$$

has one degree of freedom. Because the check is run on every experiment, a strict threshold is used, commonly $p < 0.001$.

Peeking. Looking at the p-value every day and stopping as soon as it drops below 0.05 raises the real rate of false positives far above 5%, because chance fluctuations are given many opportunities to cross the line. Decide the sample size with Equation (14.5) in advance and read the result once, at the end, or use a sequential method designed for repeated looks [4].

Novelty and learning effects. A new design may win in its first days only because it is new, or lose only because regular customers must relearn it. Run the experiment for at least one full weekly cycle, and preferably two, so that weekdays, weekends and paydays are all included.

Goodhart's law. "When a measure becomes a target, it ceases to be a good measure" is the anthropologist Marilyn Strathern's generalization of an observation by the economist Charles Goodhart [6]. If Raka were rewarded for conversion alone, he could raise it by hiding the delivery fee until after payment, which would be a dark pattern of exactly the kind Chapter 12 condemned. Every experiment therefore needs *guardrail metrics* that must not get worse, such as complaints, refunds and wrong-address orders.

14.3.5 Data with consent

Analytics is personal-data processing whenever events can be linked to a person, and a phone number or a delivery address can always be linked. In Indonesia, UU PDP 27/2022 applies, and its implementing regulation PP 33/2026 requires consent that is free, specific and unambiguous, not bundled with the terms and conditions, preceded by short and accurate information, and withdrawable at any time, with processing stopped within 3 × 24 hours of a verified withdrawal [7]. A service may not be degraded merely because a person declines consent unless the processing is needed for the service itself. For Bu Tini this separates two kinds of data: the address needed to deliver the order, which the service requires, and the analytics events used to improve the website, which need their own consent.

Analytics tools moved in the same direction. Google Analytics 4 uses *consent mode*, in which the page tells the tool whether the visitor granted analytics storage (`analytics_storage`) and advertising storage (`ad_storage`); from June 15, 2026 the two are governed separately, so that analytics consent no longer implies anything about advertising use of the data [8]. A small site can avoid most of this complexity by collecting only anonymous, first-party events on its own server, as the programs of this chapter assume, and by never sending the address or phone number into the analytics at all.

14.3.6 Metrics for AI features

Chapter 13 added a new kind of feature to many products: one that guesses. Its metrics need extra care, because the usual ones reward the wrong thing. A *suggestion acceptance rate*, the share of AI suggestions that people accept, rises when the suggestions get better, but it also rises when people stop reading them, which is over-reliance. Measured alone, it cannot tell the two apart. Three further metrics make the picture honest. The *edit rate* counts how often an accepted

suggestion is changed before it is used; many edits mean the suggestion was only a starting point. The *override-and-outcome* pair records, for every case in which a person rejected the AI, whether the person or the AI turned out to be right, which is exactly the log that Section 13.6 analyzed. And a *harm guardrail* counts the errors that matter most, such as a wrong pickup booked by the laundry's assistant.

For an experiment that compares two versions of an AI feature, the primary metric should therefore be the outcome the business cares about (orders completed correctly, questions answered without a handoff), with the acceptance rate reported only as context. A version that raises acceptance while the share of right final decisions stays flat has made people more trusting, not the system more useful.

14.4 A Method: The Experiment Plan

An experiment is only as good as the plan written *before* it starts. **Table 14.2** is a one-page template, filled in for Bu Tini's checkout.

Table 14.2 The experiment plan, filled in for the warung's checkout; every row is decided before the first visitor is assigned.

Item	Write down	Warung checkout experiment
Hypothesis	"Because we saw X, we believe that change Y will cause Z"	Because 43% drop at the address step, remembering the address and one-tap QRIS will raise completed orders
Variants	Control and treatment, precisely	A: current checkout; B: address remembered + QRIS
Unit and split	Who is randomized, and in what ratio	Each visitor (anonymous id), 50/50
Primary metric	One metric that decides	Overall conversion, `order_done` / `view_menu`
Guardrails	Metrics that must not get worse	Wrong-address orders; complaints; payment failures
Minimum detectable effect	The smallest change worth acting on	+2 percentage points
Sample size and duration	From Equation (14.5), rounded up to whole weeks	About 7,000 per variant; at least two full weeks
Checks before reading	What must pass first	Sample ratio (Equation (14.6)), $p \geq 0.001$; logging tested on both variants
Decision rule	What result leads to what action	Ship B if the 95% interval of Equation (14.4) lies above 0 and guardrails hold
Consent and data	What is collected, under which consent	Anonymous events only; separate analytics consent; no address in events

The plan has two uses. It prevents the team from choosing, after the fact, the metric that happens to look good. And it makes disagreements productive: if Bu Tini believes the photos matter more, her hypothesis gets its own plan and its own experiment.

14.5 Building the Funnel Analysis

14.5.1 The event log

The programs of this chapter read an event log in the simplest possible form: a CSV file with one row per event, `user_id,variant,step`. Before Raka's website had enough real traffic, he wrote **Listing 14.1** to create *simulated* logs with known properties, so that the analysis could be tested against a truth he had set himself. Each of 5,000 simulated visitors is assigned to A or B by a coin

flip and then walks down the funnel, continuing at each step with the probability listed in `GO_ON`; the only difference between the variants is the checkout-to-payment step, 0.58 for A and 0.66 for B. A second log, `events_srm.csv`, is identical except that 15% of variant B's visitors are silently lost, as if a logging bug on older phones dropped them.

Listing 14.1 `make_events.py`: writes two seeded, simulated event logs, one healthy and one with a logging bug in variant B.

```
"""Write the simulated checkout experiment logs of Chapter 14 (seeded)."""
import random

STEPS = ["view_menu", "add_to_cart", "checkout", "pay", "order_done"]
GO_ON = {"A": [0.62, 0.70, 0.58, 0.93],     # old checkout: type the address
         "B": [0.62, 0.70, 0.66, 0.93]}     # new: address remembered, QRIS


def write(path, lose_b=0.0, seed=14):
    rng = random.Random(seed)
    with open(path, "w", encoding="utf-8") as f:
        f.write("user_id,variant,step\n")
        for user in range(1, 5001):
            variant = "A" if rng.random() < 0.5 else "B"
            if variant == "B" and rng.random() < lose_b:
                continue            # a logging bug: these users vanish
            f.write("u%04d,%s,%s\n" % (user, variant, STEPS[0]))
            for step, p in zip(STEPS[1:], GO_ON[variant]):
                if rng.random() >= p:
                    break
                f.write("u%04d,%s,%s\n" % (user, variant, step))


write("../data/events.csv")
write("../data/events_srm.csv", lose_b=0.15)
```

14.5.2 The funnel in Python and Java

Listing 14.2 reads the log into a set of user identifiers for each variant and step. Counting *distinct users* rather than events matters: a visitor who opens the menu five times is one user who viewed the menu, not five. For each step the program prints the users, the step conversion of Equation (14.1) and the share of the starting users, which is Equation (14.2) at the last step.

Listing 14.2 `funnel.py`: users per checkout step and variant, with step and overall conversion.

```
"""Count users at each checkout step, per variant, from an event log."""
import csv
import sys

STEPS = ["view_menu", "add_to_cart", "checkout", "pay", "order_done"]


def load(path):
    """Return {variant: {step: set of user ids}}."""
    users = {}
    with open(path, encoding="utf-8") as f:
        for r in csv.DictReader(f):
            steps = users.setdefault(r["variant"], {s: set() for s in STEPS})
            steps[r["step"]].add(r["user_id"])
    return users


def main(path):
    users = load(path)
    for variant in sorted(users):
        counts = [len(users[variant][s]) for s in STEPS]
        print("Variant %s" % variant)
        print("  step          users  from previous  from start")
        for i, (step, n) in enumerate(zip(STEPS, counts)):
            prev = n / counts[i - 1] if i else 1.0
            print("  %-12s %6d  %13.1f%%  %9.1f%%" % (
                step, n, 100 * prev, 100 * n / counts[0]))


if __name__ == "__main__":
    main(sys.argv[1] if len(sys.argv) > 1 else "../data/events.csv")
```

Listing 14.3 is the same program in Java. A `TreeMap` keeps the variants in alphabetical order and a `LinkedHashMap` keeps the steps in funnel order, so the output matches Python's line for line.

Listing 14.3 `Funnel.java`: the same funnel in Java.

```java
import java.nio.file.*;
import java.util.*;

/** Count users at each checkout step, per variant, from an event log. */
public class Funnel {
    static final List<String> STEPS = List.of(
        "view_menu", "add_to_cart", "checkout", "pay", "order_done");

    /** Return {variant: {step: set of user ids}}. */
    static Map<String, Map<String, Set<String>>> load(String path)
            throws Exception {
        Map<String, Map<String, Set<String>>> users = new TreeMap<>();
        List<String> lines = Files.readAllLines(Path.of(path));
        for (String line : lines.subList(1, lines.size())) {
            String[] f = line.split(",");
            var steps = users.computeIfAbsent(f[1], v -> {
                Map<String, Set<String>> m = new LinkedHashMap<>();
                for (String s : STEPS) m.put(s, new HashSet<>());
                return m;
            });
            steps.get(f[2]).add(f[0]);
        }
        return users;
    }

    public static void main(String[] args) throws Exception {
        var users = load(args.length > 0 ? args[0] : "../data/events.csv");
        for (String variant : users.keySet()) {
            int[] counts = new int[STEPS.size()];
            for (int i = 0; i < counts.length; i++)
                counts[i] = users.get(variant).get(STEPS.get(i)).size();
            System.out.println("Variant " + variant);
            System.out.println("  step          users  from previous  from start");
            for (int i = 0; i < counts.length; i++) {
                double prev = i > 0 ? (double) counts[i] / counts[i - 1] : 1.0;
                System.out.printf("  %-12s %6d  %13.1f%%  %9.1f%%%n",
                    STEPS.get(i), counts[i], 100 * prev,
                    100.0 * counts[i] / counts[0]);
            }
        }
    }
}
```

14.5.3 Running it

From the `python` folder, `python make_events.py` writes the two logs and `python funnel.py` prints the funnel; from the `java` folder, `java -cp out Funnel` prints the same:

```
Variant A
  step          users  from previous  from start
  view_menu      2481          100.0%      100.0%
  add_to_cart    1503           60.6%       60.6%
  checkout       1043           69.4%       42.0%
  pay             588           56.4%       23.7%
  order_done      545           92.7%       22.0%
Variant B
  step          users  from previous  from start
  view_menu      2519          100.0%      100.0%
  add_to_cart    1556           61.8%       61.8%
  checkout       1109           71.3%       44.0%
  pay             753           67.9%       29.9%
  order_done      698           92.7%       27.7%
```

Figure 14.1 draws the same numbers. The two variants are nearly identical until the checkout, as the simulation intended, and then B keeps 67.9% of its users at the payment step where A keeps 56.4%. Because the later steps multiply, the gap widens to 27.7% against 22.0% at the end.

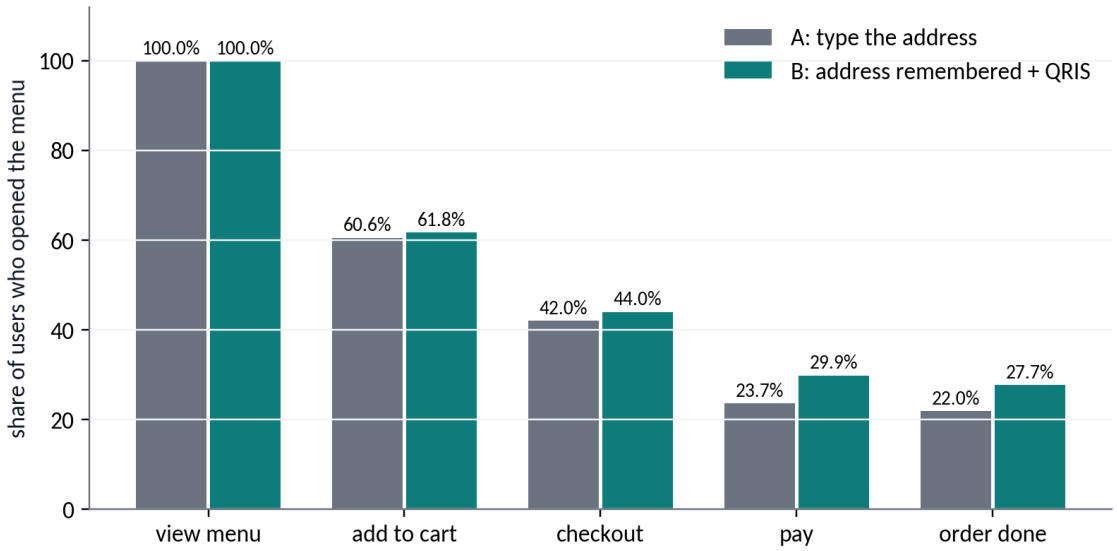


Figure 14.1 The checkout funnel of the simulated experiment: share of the users who opened the menu that reached each step, for the old checkout (A) and the new one (B).

A funnel like this is descriptive. It shows that B's users paid more often, but on its own it cannot rule out chance or a broken split. That is the job of the next program.

14.6 Building the A/B Check

14.6.1 The program in Python

Listing 14.4 runs the checks of Section 14.3 in the order of the plan in **Table 14.2**: first the sample ratio with Equation (14.6), and only if it passes, the difference, its confidence interval with Equation (14.4), the relative lift, the z-statistic of Equation (14.3) with its p-value, and finally the sample size of Equation (14.5) needed to detect a two-point change from the observed baseline.

Listing 14.4 `abtest.py`: sample ratio check, difference with 95% confidence interval, z-test and sample size.

```
"""Check an A/B test: sample ratio first, then the difference, then power."""
import math
import sys

from funnel import load


def erfc(x):
    """Complementary error function (Numerical Recipes, error < 1.2e-7)."""
    z = abs(x)
    t = 1 / (1 + 0.5 * z)
    r = t * math.exp(-z * z - 1.26551223 + t * (1.00002368 + t * (
        0.37409196 + t * (0.09678418 + t * (-0.18628806 + t * (
            0.27886807 + t * (-1.13520398 + t * (1.48851587 + t * (
                -0.82215223 + t * 0.17087277)))))))))
    return r if x >= 0 else 2 - r


def srm(n_a, n_b):
    """Chi-square test of a planned 50/50 split (1 degree of freedom)."""
    e = (n_a + n_b) / 2
    chi2 = (n_a - e) ** 2 / e + (n_b - e) ** 2 / e
    return chi2, erfc(math.sqrt(chi2 / 2))


def compare(x_a, n_a, x_b, n_b):
    p_a, p_b = x_a / n_a, x_b / n_b
    pooled = (x_a + x_b) / (n_a + n_b)
    z = (p_b - p_a) / math.sqrt(pooled * (1 - pooled) * (1 / n_a + 1 / n_b))
    se = math.sqrt(p_a * (1 - p_a) / n_a + p_b * (1 - p_b) / n_b)
    d = p_b - p_a
    return p_a, p_b, z, erfc(abs(z) / math.sqrt(2)), d - 1.96 * se, d + 1.96 * se


def sample_size(p1, p2, z_alpha=1.96, z_beta=0.8416):
    """Users per variant to detect p1 -> p2 (5% two-sided, 80% power)."""
    var = p1 * (1 - p1) + p2 * (1 - p2)
    return math.ceil((z_alpha + z_beta) ** 2 * var / (p2 - p1) ** 2)


def show_p(p):
    return "p < 0.0001" if p < 0.0001 else "p = %.4f" % p
```

Listing 14.4 (Continued) *abtest.py: sample ratio check, difference with 95% confidence interval, z-test and sample size.*

```
def main(path):
    users = load(path)
    n_a = len(users["A"]["view_menu"])
    n_b = len(users["B"]["view_menu"])
    chi2, p = srm(n_a, n_b)
    print("users: A = %d, B = %d" % (n_a, n_b))
    print("sample ratio check: chi2 = %.2f, %s" % (chi2, show_p(p)))
    if p < 0.001:
        # A broken split makes every later number untrustworthy.
        print("STOP: sample ratio mismatch - find the bug before reading on")
        return
    x_a = len(users["A"]["order_done"])
    x_b = len(users["B"]["order_done"])
    p_a, p_b, z, pv, lo, hi = compare(x_a, n_a, x_b, n_b)
    print("orders:  A = %d (%.2f%%), B = %d (%.2f%%)" % (
        x_a, 100 * p_a, x_b, 100 * p_b))
    print("difference = %+.2f points, 95%% CI [%+.2f, %+.2f]" % (
        100 * (p_b - p_a), 100 * lo, 100 * hi))
    print("relative lift = %+.1f%%, z = %.2f, %s" % (
        100 * (p_b / p_a - 1), z, show_p(pv)))
    print("users per variant needed to detect +2 points from %.1f%%: %d" % (
        100 * p_a, sample_size(p_a, p_a + 0.02)))


if __name__ == "__main__":
    main(sys.argv[1] if len(sys.argv) > 1 else "../data/events.csv")
```

The function `erfc` computes the complementary error function, from which both p-values follow: for a standard normal z, the two-sided p-value is $\operatorname{erfc}\left(|z|/\sqrt{2}\right)$, and for a chi-square with one degree of freedom it is $\operatorname{erfc}\left(\sqrt{\chi^2/2}\right)$. Python has `math.erfc`, but Java's standard library does not, so both programs use the same published approximation, accurate to better than 1.2×10^{-7}, and therefore print the same digits.

14.6.2 The program in Java

Listing 14.5 is the same check in Java; it reuses `Funnel.load`.

Listing 14.5 `AbTest.java`: the same check in Java.

```
/** Check an A/B test: sample ratio first, then the difference, then power. */
public class AbTest {
    /** Complementary error function (Numerical Recipes, error < 1.2e-7). */
    static double erfc(double x) {
        double z = Math.abs(x), t = 1 / (1 + 0.5 * z);
        double r = t * Math.exp(-z * z - 1.26551223 + t * (1.00002368 + t * (
            0.37409196 + t * (0.09678418 + t * (-0.18628806 + t * (
            0.27886807 + t * (-1.13520398 + t * (1.48851587 + t * (
            -0.82215223 + t * 0.17087277)))))))));
        return x >= 0 ? r : 2 - r;
    }

    /** Chi-square test of a planned 50/50 split (1 degree of freedom). */
    static double[] srm(int nA, int nB) {
        double e = (nA + nB) / 2.0;
        double chi2 = (nA - e) * (nA - e) / e + (nB - e) * (nB - e) / e;
        return new double[] {chi2, erfc(Math.sqrt(chi2 / 2))};
    }

    static double[] compare(int xA, int nA, int xB, int nB) {
        double pA = (double) xA / nA, pB = (double) xB / nB;
        double pooled = (double) (xA + xB) / (nA + nB);
        double z = (pB - pA)
            / Math.sqrt(pooled * (1 - pooled) * (1.0 / nA + 1.0 / nB));
        double se = Math.sqrt(pA * (1 - pA) / nA + pB * (1 - pB) / nB);
        double d = pB - pA;
        return new double[] {pA, pB, z, erfc(Math.abs(z) / Math.sqrt(2)),
            d - 1.96 * se, d + 1.96 * se};
    }

    /** Users per variant to detect p1 -> p2 (5% two-sided, 80% power). */
    static long sampleSize(double p1, double p2) {
        double za = 1.96, zb = 0.8416;
        double var = p1 * (1 - p1) + p2 * (1 - p2);
        double gap = p2 - p1;
        return (long) Math.ceil((za + zb) * (za + zb) * var / (gap * gap));
    }

    static String showP(double p) {
        return p < 0.0001 ? "p < 0.0001" : String.format("p = %.4f", p);
    }
```

Listing 14.5 (Continued) *AbTest.java: the same check in Java.*

```
    public static void main(String[] args) throws Exception {
        var users = Funnel.load(args.length > 0 ? args[0] : "../data/events.csv");
        int nA = users.get("A").get("view_menu").size();
        int nB = users.get("B").get("view_menu").size();
        double[] s = srm(nA, nB);
        System.out.printf("users: A = %d, B = %d%n", nA, nB);
        System.out.printf("sample ratio check: chi2 = %.2f, %s%n",
            s[0], showP(s[1]));
        if (s[1] < 0.001) {
            // A broken split makes every later number untrustworthy.
            System.out.println(
                "STOP: sample ratio mismatch - find the bug before reading on");
            return;
        }
        int xA = users.get("A").get("order_done").size();
        int xB = users.get("B").get("order_done").size();
        double[] c = compare(xA, nA, xB, nB);
        System.out.printf("orders:  A = %d (%.2f%%), B = %d (%.2f%%)%n",
            xA, 100 * c[0], xB, 100 * c[1]);
        System.out.printf("difference = %+.2f points, 95%% CI [%+.2f, %+.2f]%n",
            100 * (c[1] - c[0]), 100 * c[4], 100 * c[5]);
        System.out.printf("relative lift = %+.1f%%, z = %.2f, %s%n",
            100 * (c[1] / c[0] - 1), c[2], showP(c[3]));
        System.out.printf(
            "users per variant needed to detect +2 points from %.1f%%: %d%n",
            100 * c[0], sampleSize(c[0], c[0] + 0.02));
    }
}
```

14.6.3 Running it on the healthy log

`python abtest.py` (or `java -cp out AbTest`) prints:

```
users: A = 2481, B = 2519
sample ratio check: chi2 = 0.29, p = 0.5910
orders:  A = 545 (21.97%), B = 698 (27.71%)
difference = +5.74 points, 95% CI [+3.35, +8.13]
relative lift = +26.1%, z = 4.70, p < 0.0001
users per variant needed to detect +2 points from 22.0%: 6940
```

Read the output from the top. The split is 2,481 to 2,519, and the sample ratio check gives $\chi^2 = 0.29$ with $p = 0.59$: an imbalance this size is entirely ordinary, so the experiment passes the first gate. The new checkout converted 27.71% against 21.97%, a difference of 5.74 percentage points with a 95% interval from +3.35 to +8.13 points, a relative lift of 26.1%, and $z = 4.70$. The whole interval lies above zero and far above the two-point minimum set in the plan. The last line shows that detecting a *two*-point change would need 6,940 users per variant; this experiment was big enough only because the true effect was larger. If the effect had been two points, 2,500 users per variant would have been too few, and a "no significant difference" would have meant "we could not tell", not "no difference".

14.6.4 Running it on the broken log

Now run the same program on the log with the logging bug: `python abtest.py ../data/events_srm.csv`.

```
users: A = 2472, B = 2155
sample ratio check: chi2 = 21.72, p < 0.0001
STOP: sample ratio mismatch - find the bug before reading on
```

The program stops. With 2,472 users in A and 2,155 in B, $\chi^2 = 21.72$ and the p-value is below 0.0001: a split this uneven would almost never happen by chance under a true 50/50 assignment. Had the program continued, it would have reported a lift for B, and nobody could say whether the lost 15% of B's users were the ones who would have ordered or the ones who would have left. That is why the check comes first and why the program refuses to go on: in the words of Fabijan and colleagues, a result with a sample ratio mismatch has to be treated as untrustworthy until the cause is found [5].

14.7 Testing It

Statistical code can be wrong in ways that still print plausible numbers, so it must be tested against values known independently. **Listing 14.6** checks the error function at two points, a perfect split, equal rates, and a sample size computed by hand in the same way as Exercise 4 of Section 14.13.

Listing 14.6 `test_abtest.py`: tests of the error function, the ratio check, the z-test and the sample size.

```
"""Checks of the A/B arithmetic; run: python -m unittest test_abtest"""
import math
import unittest

from abtest import compare, erfc, sample_size, srm


class AbTestTest(unittest.TestCase):
    def test_erfc(self):
        self.assertAlmostEqual(erfc(0), 1, places=6)
        self.assertAlmostEqual(erfc(1.96 / math.sqrt(2)), 0.05, places=4)

    def test_srm_and_equal_rates(self):
        self.assertEqual(srm(500, 500)[0], 0)
        self.assertEqual(compare(100, 1000, 100, 1000)[2], 0)

    def test_sample_size(self):
        self.assertEqual(sample_size(0.20, 0.22), 6507)


if __name__ == "__main__":
    unittest.main()
```

Listing 14.7 makes the same checks in Java.

Listing 14.7 `AbTestTest.java`: the same checks in Java.

```java
/** Checks of the A/B arithmetic; run: java AbTestTest */
public class AbTestTest {
    static int failed = 0;

    static void check(boolean ok, String what) {
        System.out.println((ok ? "ok   " : "FAIL ") + what);
        if (!ok) failed++;
    }

    public static void main(String[] args) {
        check(Math.abs(AbTest.erfc(0) - 1) < 1e-6, "erfc(0) = 1");
        check(Math.abs(AbTest.erfc(1.96 / Math.sqrt(2)) - 0.05) < 1e-4,
            "|z| = 1.96 gives p = 0.05");
        check(AbTest.srm(500, 500)[0] == 0, "a perfect split has chi2 = 0");
        double[] c = AbTest.compare(100, 1000, 100, 1000);
        check(c[2] == 0, "equal rates give z = 0");
        check(AbTest.sampleSize(0.20, 0.22) == 6507, "sample size 20% -> 22%");
        System.out.println(failed == 0 ? "all checks passed" : failed + " failed");
        System.exit(failed == 0 ? 0 : 1);
    }
}
```

`python -m unittest test_abtest` reports three passing tests, and the Java version prints:

```
ok   erfc(0) = 1
ok   |z| = 1.96 gives p = 0.05
ok   a perfect split has chi2 = 0
ok   equal rates give z = 0
ok   sample size 20% -> 22%
all checks passed
```

The check that $|z| = 1.96$ gives $p = 0.05$ is the most valuable one. It ties the code to a number every statistics table agrees on, so a mistake such as forgetting the factor $\sqrt{2}$ would be caught at once.

14.8 Making It Better

Segment, but plan the segments. The overall lift may hide differences: perhaps B helps returning customers, whose address is remembered, far more than first-time customers. Add a column such as `returning` to the events and report the funnel for each segment. Decide the segments in the plan, though; searching through twenty segments after the fact will find a "significant" one by chance.

Reduce variance with data from before the experiment. A method called CUPED, developed at Microsoft, adjusts each user's outcome using the same user's behavior before the experiment, which can shrink confidence intervals considerably and therefore shorten experiments [9]. It needs users who were seen before, so it helps Bu Tini's returning customers more than new visitors.

Report guardrails next to the primary metric. Extend `abtest.py` to read complaints and wrong-address orders per variant and print them in the same format. A lift in orders with a rise in wrong addresses is not a win.

Look at where people tap. A funnel says at which step people leave, not what they tried to do there. Logging the position of each tap on the checkout page, relative to the page's width and height, and drawing the taps as a density map shows where attention and frustration gather: a cluster of taps on the grey delivery-fee text, for example, means that customers expected it to be a button that explains the fee. Such a *tap heatmap* is a cheap cousin of eye tracking. It shows only

where fingers landed, never where eyes looked, and it records nothing about the person, so the positions can be stored without an address or phone number.

Watch the metric over time after shipping. An experiment measures two or three weeks. After B becomes the only checkout, keep the funnel on a weekly chart. A slow drift back toward the old conversion suggests a novelty effect; a sudden drop points to a bug or a change elsewhere, such as a new payment app version. The chart turns the experiment's single number into a trend that the team keeps looking at.

Pair the numbers with people. Once B has won, ask five customers who used it to order while thinking aloud, as in Chapter 7. The experiment says B is better; the sessions say what is still hard, and they supply the hypothesis for the next experiment.

14.9 Using It for Real

The programs of this chapter work on any log in the `user_id,variant,step` form. To use them for a real website, three pieces are needed.

Collect the events on your own server. In the page, send a small request such as `POST /event` with the step name each time a step is reached; on the server, append a line to a file or a database table with an anonymous identifier kept in a first-party cookie or in local storage. Do this only after the visitor has given analytics consent, in a separate, plain question, as PP 33/2026 requires [7]. Never put names, phone numbers or addresses in the events.

Assign variants deterministically. Instead of a random choice on every page load, compute the variant from the anonymous identifier, for example from a hash of it modulo 2, so that the same visitor always sees the same version. Store the variant in each event.

Keep a log of decisions. For each experiment, keep the filled-in plan of **Table 14.2** together with the output of `abtest.py` and the decision taken. After a year the log shows which ideas worked, and it protects the team from repeating experiments that failed.

A different business uses the same pieces. Pak Made's laundry on Bali, for example, could test whether showing the total price before the pickup time is chosen, rather than after, raises completed bookings. The funnel steps change (`open_form`, `choose_service`, `choose_time`, `confirm`) but the programs, the plan and the checks do not.

14.10 HCI in Practice: Profitina

Profitina, the AI Visibility (GEO) platform at app.profitina.com, is itself an exercise in metrics that must be legible to people who are not analysts. Its historical analytics dashboard shows each brand's weekly AI-visibility score as a trend, exports it as CSV, and re-scans automatically at a cadence set by the subscription tier, with alerts when a score changes. The design question is the one of Section 14.3.1: a brand owner's *goal* is to be cited by AI answer engines, the *signals* are the measured properties of the website, and the *metric* is the score. Because the score is only a metric, the dashboard explains each signal behind it, so that owners do not chase the number instead of the goal, which would be Goodhart's law at work.

The same care appears in small places. The "N new" counters on followed community channels count only top-level posts, not replies, because a counter that jumps with every reply in a busy thread would overstate what is new for the reader. The dashboard gathers notifications, followed channels, rank and activity with a single request, so the first screen appears quickly and each item

leads straight to its section. And in line with UU PDP 27/2022, users can export their data, change their email and password, and purge their account entirely: the data behind the metrics remains the user's.

14.11 Common Mistakes, and How to Avoid Them

- **Measuring what is easy instead of what matters.** Page views and time on site rise when people are lost. Start from goals with Goals–Signals–Metrics (**Table 14.1**).
- **Counting events instead of users.** One visitor who reloads the menu five times is not five visitors. Count distinct user identifiers per step, as `funnel.py` does.
- **Reading a page-view share as a share of people.** StatCounter's 60.59% desktop share for Indonesia counts page views on tracked sites [3]. Use your own logs for your own users.
- **Comparing before and after.** A change in orders from one week to the next mixes the new design with payday, weather and holidays. Randomize and run both versions together.
- **Skipping the sample ratio check.** An uneven split means lost users, and lost users bias everything. Run Equation (14.6) first, and stop if $p < 0.001$.
- **Peeking and stopping early.** Stopping when p first drops below 0.05 inflates false positives. Fix the sample size with Equation (14.5) in advance.
- **Reporting a p-value without an effect size.** "Significant" says nothing about how much. Report the difference with its interval from Equation (14.4).
- **Treating "not significant" as "no effect".** With too few users, a real effect is invisible. Compare the sample with Equation (14.5) before concluding anything.
- **No guardrails.** A conversion lift bought with hidden fees or confusing consent is a dark pattern, not an improvement.
- **Collecting personal data in analytics.** Addresses and phone numbers do not belong in events, and analytics needs its own consent under PP 33/2026 [7].

14.12 Chapter Summary

- Analytics records what many people do; research with a few people explains why. Data-informed design uses both.
- An event records one thing that happened; a funnel counts the distinct users reaching each step; step and overall conversion follow from Equation (14.1) and Equation (14.2), and the weakest step is where improvement pays most.
- The HEART framework and its Goals–Signals–Metrics process derive metrics from goals (**Table 14.1**) [1]; a metric that rises when the experience worsens must be dropped.
- In 2026, 81.7% of Indonesians use the internet, with large regional differences [2]; device shares from StatCounter count page views, not people [3].
- A controlled experiment randomizes users between versions; the difference is tested with Equation (14.3), sized with the interval of Equation (14.4), and planned with the sample size of Equation (14.5) [4].
- Trustworthy experiments check the sample ratio first (Equation (14.6)) [5], do not peek, run through full weekly cycles, and protect guardrail metrics against Goodhart's law [6].
- On the simulated warung data, the new checkout raised orders from 21.97% to 27.71% (95% interval +3.35 to +8.13 points); on the log with a logging bug, the sample ratio check ($\chi^2 = 21.72$) stopped the analysis before it could mislead.

- Analytics that can be linked to a person needs its own, unbundled consent under UU PDP 27/2022 and PP 33/2026 [7]; GA4 separates analytics and advertising consent from June 15, 2026 [8].

14.13 Exercises

1. In one week 1,200 people open a laundry's booking page, 780 choose a service, 520 choose a time, 290 reach the confirmation and 270 confirm. Compute every step conversion and the overall conversion with Equation (14.1) and Equation (14.2), and say which step you would work on first.
2. An experiment ends with A: 120 orders out of 1,000 users and B: 150 out of 1,000. Compute the pooled rate, z with Equation (14.3), the 95% interval of the difference with Equation (14.4), and state your decision.
3. An experiment planned 50/50 has 5,210 users in A and 4,790 in B. Compute χ^2 with Equation (14.6). What do you do next, and why?
4. The current conversion is 10%. How many users per variant are needed to detect an increase to 11% at the 5% level with 80% power? Use Equation (14.5), then check with `sample_size(0.10, 0.11)`. If the site has 1,500 visitors a day, how long must the experiment run?
5. For a puskesmas queue kiosk, apply the Goals–Signals–Metrics process to two HEART categories of your choice. Which category would you deliberately leave out, and why?
6. Extend `funnel.py` and `Funnel.java` so that, after each variant's table, they print the step with the lowest step conversion. Which lines do you add, and what does the program print for the healthy log?

14.14 Solutions to the Exercises

1. Step conversions: $780/1200 = 0.650$; $520/780 = 0.667$; $290/520 = 0.558$; $270/290 = 0.931$. Overall: $270/1200 = 0.225$. The step from choosing a time to reaching the confirmation loses 44% and is the weakest; work there first, and find out *why* with a few think-aloud sessions (perhaps the time slots are shown in a way that hides free slots).

2. Pooled rate $\bar{p} = 270/2000 = 0.135$. Standard error under the null: $\sqrt{0.135 \cdot 0.865 \cdot (1/1000 + 1/1000)} = \sqrt{0.00023355} = 0.015282$. $z = 0.03/0.015282 = 1.963$, so $p \approx 0.0496$, just below 0.05. Interval: $\sqrt{0.12 \cdot 0.88/1000 + 0.15 \cdot 0.85/1000} = 0.015268$, and $0.03 \pm 1.96 \cdot 0.015268 = 0.03 \pm 0.0299$, from $+0.0001$ to $+0.0599$, that is +0.01 to +5.99 points. The result is barely significant and the interval stretches from almost nothing to six points. A careful decision is to ship B only if it costs little and has no guardrail problems, or better, to run the experiment to a size computed with Equation (14.5) for the smallest effect that matters.

3. $e = 10000/2 = 5000$; $\chi^2 = 210^2/5000 + 210^2/5000 = 8.82 + 8.82 = 17.64$; $p = \operatorname{erfc}\left(\sqrt{8.82}\right) \approx 0.000027$, far below 0.001. Stop: do not read the conversion rates. Look for what removes users from B: errors on particular browsers or phones, redirects, a bot filter, or a difference in when logging starts. Fix it and rerun; a result from a mismatched split cannot be repaired afterwards.

4. $(1.96 + 0.8416)^2 = 7.849$; $p_1(1-p_1) + p_2(1-p_2) = 0.09 + 0.0979 = 0.1879$; $(p_2 - p_1)^2 = 0.0001$. So $n = 7.849 \cdot 0.1879/0.0001 \approx 14748.3$, rounded up to 14,749 per variant, as

`sample_size(0.10, 0.11)` confirms. Both variants together need 29,498 users; at 1,500 a day that is about 20 days, so run for three full weeks.

5. One possible answer. *Task success* — goal: a patient gets the right queue number quickly; signal: a number printed for the service the patient then attends; metric: share of numbers used at the matching counter, and median seconds from first touch to number. *Happiness* — goal: older patients feel confident using the kiosk; signal: answer to one spoken question at the counter; metric: share of patients aged 60+ answering "easy". *Engagement* is left out: a patient who spends longer at a queue kiosk is struggling, not engaged.

6. In Python, inside the loop over variants and after the table, add three lines: compute `rates = [counts[i] / counts[i - 1] for i in range(1, len(STEPS))]`, find `worst = rates.index(min(rates)) + 1`, and print `"  weakest step: %s (%.1f%%)" % (STEPS[worst], 100 * rates[worst - 1])`. In Java, loop over `i` from 1, keep the index with the smallest `(double) counts[i] / counts[i - 1]`, and print the same text with `printf`. For the healthy log the program prints `weakest step: pay (56.4%)` for A and `weakest step: add_to_cart (61.8%)` for B: once the checkout is fixed, the menu becomes the weakest step, and Bu Tini's idea about the photos becomes the natural next experiment.

References

[1] K. Rodden, H. Hutchinson, and X. Fu, "Measuring the user experience on a large scale: User-centered metrics for web applications," in *Proc. CHI '10*, Atlanta, GA, USA, 2010, pp. 2395–2398.

[2] ANTARA News, "Survei APJII: Penetrasi internet di Indonesia 2026 capai 81,7 persen," May 21, 2026. [Online]. Available: https://www.antaranews.com/berita/5576225/survei-apjii-penetrasi-internet-di-indonesia-2026-capai-817-persen

[3] StatCounter Global Stats, "Desktop vs Mobile vs Tablet Market Share Indonesia," August 2026. [Online]. Available: https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/indonesia/

[4] R. Kohavi, D. Tang, and Y. Xu, *Trustworthy Online Controlled Experiments: A Practical Guide to A/B Testing*. Cambridge, UK: Cambridge University Press, 2020.

[5] A. Fabijan *et al.*, "Diagnosing sample ratio mismatch in online controlled experiments: A taxonomy and rules of thumb for practitioners," in *Proc. 25th ACM SIGKDD Int. Conf. Knowledge Discovery & Data Mining (KDD '19)*, Anchorage, AK, USA, 2019, pp. 2156–2164, doi: 10.1145/3292500.3330722.

[6] M. Strathern, "'Improving ratings': Audit in the British university system," *European Review*, vol. 5, no. 3, pp. 305–321, 1997.

[7] Bisnis.com, "Aturan Turunan UU PDP: Persetujuan Data Konsumen Lebih Ketat," Aug. 29, 2026, on Peraturan Pemerintah No. 33 Tahun 2026 implementing UU No. 27 Tahun 2022. [Online]. Available: https://teknologi.bisnis.com/read/20260829/84/1999753/aturan-turunan-uu-pdp-persetujuan-data-konsumen-lebih-ketat

[8] ALM Corp, "Google Analytics 4 and Google Ads consent controls split on June 15, 2026: What ad_storage now controls and what teams must fix," 2026. [Online]. Available: https://almcorp.com/blog/ga4-google-ads-consent-controls-split-june-2026/

[9] A. Deng, Y. Xu, R. Kohavi, and T. Walker, "Improving the sensitivity of online controlled experiments by utilizing pre-experiment data," in *Proc. 6th ACM Int. Conf. Web Search and Data Mining (WSDM '13)*, Rome, Italy, 2013, pp. 123–132.

Chapter 15

Emerging Interfaces: Spatial, Wearable and Ambient Computing

Interfaces are leaving the rectangle of the phone and the monitor. They float in the air in front of a headset wearer, sit in the corner of a pair of glasses, tap on a wrist, glow from a lamp on the wall, or speak from a speaker in the ceiling. This chapter is about designing for these new places without forgetting the people who will never own the devices. By the end of it you will be able to describe spatial, wearable and ambient interfaces and the devices that carry them in 2026; express the size of any target as a visual angle and convert it to centimeters at any distance; explain why gaze input needs a separate confirmation and how to choose a dwell time when it does not have one; design glanceable, haptic and calm notifications; check comfort, safety, privacy and accessibility for interfaces worn on the body or placed in a room; build a target-size calculator and a dwell-time analysis in Java and Python; and apply a context-of-use checklist before choosing any emerging interface.

15.1 Mbah Sumi and the Queue Screen

Every second Tuesday Mbah Sumi, who is seventy-two, takes a becak to the puskesmas for her blood-pressure check. The registration desk gives her a paper slip with a number, L-017, and she sits down in the waiting room on a long bench. On the far wall, about five meters away, a television shows the number being served; a recorded voice calls each number over a loudspeaker. Mbah Sumi’s eyes are no longer sharp and her hearing is worse. The voice echoes, the television is small from where she sits, and twice she has missed her turn and waited another hour.

The puskesmas nurse, Mbak Ratna, sees the same thing happen to other older patients every week. She reads that the newest glasses can show text in the corner of the lens, that a watch can tap its wearer’s wrist, that a headset can place a screen anywhere in a room, and she wonders whether any of this could help. Her nephew, who studies informatics, is enthusiastic. Mbak Ratna is cautious: most of her patients do not own a smartwatch, none owns smart glasses, and the waiting room must work for all of them.

Both are right, and the tension between them is the subject of this chapter. Emerging interfaces open possibilities that phones cannot offer: information that appears where the person is looking, a tap on the wrist that nobody else hears, a lamp that changes color when the turn is near. They also bring new ways to fail: text too small at a distance, buttons that activate when merely looked at, notifications that interrupt at the wrong moment, cameras that bystanders did not agree to. And the most important lesson comes before any device is chosen: the first improvement to Mbah Sumi’s waiting room is a larger number on the television, sized from the distance of the bench, which costs nothing and helps everyone. The same geometry that sizes a floating button in a headset sizes that number.

15.2 The Idea in Plain Words

15.2.1 Beyond the rectangle

> **DEFINITION.** A *spatial* interface places digital content in the three-dimensional space around the person, in a headset or glasses, so that windows, buttons and objects have a position and a distance. A *wearable* interface is carried on the body, such as a watch, a ring, a wristband or glasses, and is used in short glances while the person does something else. An *ambient* interface is built into the environment, such as a lamp, a sound or a display on a wall, and conveys information at the edge of attention without asking for it.

The three share one property that desktop and phone interfaces do not: the person's attention is usually somewhere else. A headset wearer is also moving in a room; a watch wearer is walking, cooking or talking; the patients in a waiting room are chatting, dozing or looking after a child. The interface must therefore be *easy to perceive at a glance* and *hard to trigger by accident*. Most of this chapter follows from those two demands.

15.2.2 Size is an angle

On a phone we speak of button sizes in millimeters or points. For anything seen from a varying distance, the size that matters to the eye is the *visual angle*: the angle that the object covers at the eye. **Figure 15.1** shows the geometry. An object of size s at distance d covers the angle

$$\theta = 2\arctan\left(\frac{s}{2d}\right) \tag{15.1}$$

and, turned around, an object must have the size

$$s = 2d\tan\left(\frac{\theta}{2}\right) \tag{15.2}$$

to cover a given angle θ at distance d.

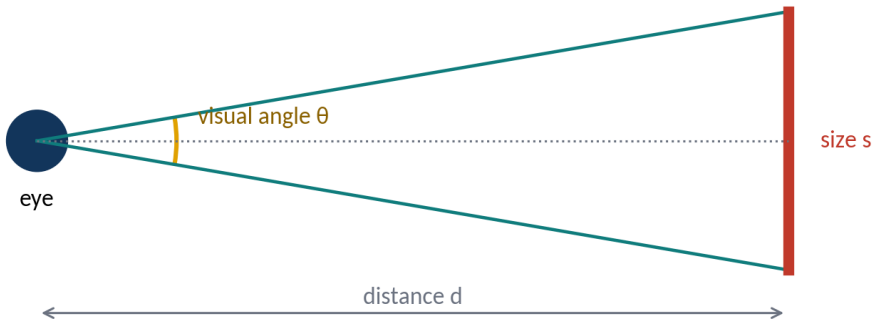


Figure 15.1 The visual angle θ of an object of size s at distance d: the same angle can be produced by a small object nearby or a large object far away.

For the small angles of everyday interfaces, $\tan x \approx x$ with x in radians, and one radian is about 57.3 degrees, so a handy approximation in degrees is

$$\theta_{\text{deg}} \approx 57.3\,\frac{s}{d} \tag{15.3}$$

which says that an object one hundredth of its distance across covers a little more than half a degree. A thumbnail at arm's length covers roughly one to two degrees, which is a convenient ruler to carry around.

15.2.3 A worked example by hand

The queue number on the puskesmas television is 4 cm tall, and Mbah Sumi's bench is 5 m from the wall. By Equation (15.1), $\theta = 2\arctan(4/1000) = 2 \cdot 0.0040$ radians $= 0.0080$ radians, or about 0.46 degrees; Equation (15.3) gives $57.3 \cdot 4/500 = 0.46$ degrees as well. That is about 27.5 minutes of arc.

Is that enough? The letters on the 20/20 line of a Snellen eye chart cover 5 minutes of arc. A person who reads only the 20/100 line needs letters five times larger, 25 minutes, so for a patient with that eyesight the number on the wall is at the very edge of legibility, which is not a comfortable place to read from while dozing on a bench. If the puskesmas wants its number to cover 1.5 degrees (90 minutes of arc, several times the threshold even for weak eyes) from 5 m, Equation (15.2) gives $s = 2 \cdot 500 \cdot \tan(0.75°) = 1000 \cdot 0.01309 = 13.1$ cm. The number must be about three times taller than it is. On a 43-inch television, whose screen is about 53 cm high, a 13 cm number still leaves room for the service name and the next two numbers.

15.3 The Theory, with Evidence

15.3.1 The devices of 2025 and 2026

The devices change every year; the principles do not. **Table 15.1** lists the ones that set the terms of design in 2026, described in words rather than by their logos.

Table 15.1 Devices that shape spatial and wearable interface design in 2026, with the inputs and outputs a designer must plan for; prices are at launch in the United States.

Device (announced)	Kind	Main inputs	Main outputs	Launch price
Apple Vision Pro with M5 (Oct. 2025) [1]	Headset, spatial	Eyes, hands, voice; optional controllers and pencil	Micro-OLED displays up to 120 Hz, spatial audio	from US$3,499
Samsung Galaxy XR on Android XR (Oct. 2025) [2]	Headset, spatial	Eyes, hands, voice with Gemini	Displays with passthrough, audio	US$1,799.99
Meta Ray-Ban Display with Neural Band (Sep. 2025) [3]	Glasses, wearable	EMG wristband gestures, voice, touch	Small full-color display off to one side, audio	US$799
Ray-Ban Meta Audio (Sep. 2026) [4]	Glasses, wearable	Voice, touch	Audio only, no camera	US$349
Android XR audio glasses (announced May 2026, fall 2026) [5]	Glasses, wearable	Voice with Gemini, tap on the frame	Audio	not announced
Apple Watch on watchOS 26 [6]	Watch, wearable	Touch, crown, double tap, wrist flick	Screen, haptics, audio	various

Three trends are visible in the table. Inputs are moving away from touching a surface, toward the eyes, small hand and wrist movements, and voice. Outputs are shrinking to glances and sounds: a display "off to the side" that is "not always on" [3], or no display at all. And an AI assistant is built into most of them, so that the voice interface of Chapter 9 and the AI interface of Chapter 13 meet in a device with no keyboard. Apple's software for Vision Pro also moved on: visionOS 26 arrived with the M5 model, and visionOS 27, announced in June 2026, adds eye-based expansion of notifications and a Siri that can answer questions about what the wearer is looking at [1], [7].

Not everything is growing. In April 2026 MacRumors reported, from unnamed sources, that the M5 refresh had not revived Vision Pro sales, that price, weight and discomfort in long sessions were the main complaints, and that Apple was shifting its effort toward lighter smart glasses [8]. Expensive headsets remain a small market. For a designer in Indonesia in 2026 the practical consequences are clear: the phone remains the main interface, wearables and voice reach more people than headsets, and ideas from spatial computing are most useful when they improve ordinary screens and rooms.

15.3.2 Input in space: looking is not choosing

In a headset the most natural pointer is the eyes: people look at what they want before they reach for it. But the eyes are also how people *read*, and they move three or four times a second whether or not the person wants anything. In 1990 Robert Jacob named the resulting problem the *Midas touch*: if everything you look at is activated, like everything King Midas touched turning to gold, you cannot look at anything without triggering it [9]. The two classic solutions are a second signal that means "yes, this one" and a *dwell time*, a period for which the gaze must rest on a target before it activates.

Current headsets choose the first solution. On Vision Pro and Galaxy XR the person looks at a target and confirms with a small pinch of thumb and finger; the eyes point, the hand commits [1], [2]. Meta's Neural Band does the same for glasses by reading the electrical activity of the wrist muscles (electromyography, EMG), so that a pinch, a thumb swipe or a wrist rotation can be recognized with the hand resting at the side [3]. Dwell remains important where the hands are not available: for people with motor impairments, for sterile or dirty hands, and for public kiosks with eye tracking. Choosing its length is a trade-off that Section 15.6 measures.

Gaze pointing is also imprecise. Eye trackers estimate where a person is looking with errors that vary widely between people, between their two eyes, and with lighting and glasses; Feit and colleagues measured two trackers with 80 participants in everyday conditions and showed how to derive target sizes from such data rather than from the manufacturer's best case [10]. That is why interactive elements in spatial interfaces are larger than on phones. In its 2023 guidance for spatial interfaces, Apple asked for interactive elements of at least 60 points, compared with the 44-point minimum long used on iOS [11], and it keeps visionOS content at comfortable distances so that those points cover a generous visual angle.

15.3.3 Wearables: glance, tap, go

A watch or a pair of glasses is used in glances that last a second or two. Three rules follow. *Show one thing:* the queue number and how many people are ahead, not a menu. *Prefer haptics and short sounds to text:* a tap on the wrist is private and needs no reading. *Offer a gesture that needs no aiming:* Apple Watch's double tap (pinching index finger and thumb twice), introduced in 2023, and the wrist flick added in watchOS 26 (turning the wrist away and back quickly) dismiss or silence notifications, calls, timers and alarms without touching the screen [6]. These gestures are an accessibility feature as much as a convenience: they help anyone whose other hand is busy holding a child, a cane or a shopping bag.

Haptic patterns can carry a small vocabulary if they are few and distinct. For Mbah Sumi, two patterns would suffice: a gentle double tap when three people are ahead ("get ready") and a long, strong buzz when her number is called ("go now"). More patterns than that must be learned and will be confused.

Voice is the other input that needs no aiming, and audio glasses rely on it almost entirely [4], [5]. It brings its own design rules. Spoken output is linear and fleeting, so keep it short and put the most important word first: "L-017, your turn, door three" rather than a sentence that ends with the number. Confirm actions by repeating what was understood before acting, as the laundry assistant of Chapter 13 repeated the order. Remember that speaking is public: a patient may not want to say a symptom aloud in a waiting room, so a voice interface must offer a silent alternative. And plan for the languages people actually speak. In Indonesia a single waiting room may hear Indonesian, Javanese, Madurese and mixtures of them, and speech recognizers trained mainly on standard Indonesian recognize regional speech less reliably; test with the voices of the real users before trusting a voice command.

15.3.4 Ambient and calm technology

In 1991 Mark Weiser described a future in which computers would "disappear" into everyday objects, and in 1996, with John Seely Brown, he argued that such technology should be *calm*: it should inform without demanding attention, moving information between the center and the *periphery* of attention as needed [12], [13]. A good ambient display is noticed when it matters and ignored when it does not, like the sound of a car engine that a driver hears only when it changes.

For the waiting room this suggests a lamp above each service door that glows softly and brightens as the queue moves, or a colored band on the television that shows how close the next group of numbers is. Nobody has to read anything, and a person who glances up learns the state of the queue at once. Calm design has a condition, though: an ambient signal carries only a little information, so it must be paired with a precise channel (the number on the screen, the voice, or the tap on the wrist) for the moment of action.

15.3.5 Comfort, safety, privacy and accessibility

Emerging interfaces bring risks that screens do not.

Comfort and sickness. Headsets can cause *cybersickness*, a discomfort similar to motion sickness when what the eyes see and what the balance organs feel disagree; it is worst when the virtual view moves without the body moving. Keep the virtual world stable, let the person move the view, avoid long sessions, and never require a headset for an essential task [14].

Safety. A person wearing a headset or reading glasses while walking notices less of the world. Glanceable content must stay short, and a spatial interface must keep the real room visible and warn about obstacles.

Privacy. Glasses with cameras and always-listening microphones record people who did not agree to it. That is why some manufacturers now offer audio-only glasses without a camera [4], and why a visible light when recording matters. In Indonesia, images and voices of identifiable people are personal data under UU PDP 27/2022, and a puskesmas waiting room is exactly the kind of place where patients do not expect to be filmed.

Accessibility. Gaze, hand and voice input each exclude someone: people with eye conditions, people without the use of a hand, people who cannot speak or are hard to understand for a speech recognizer. The W3C's XR Accessibility User Requirements list needs such as alternatives for every input mode, captions and audio description for immersive content, and the ability to customize size, color and motion [15]. The rule for every emerging interface is the same as for the queue: offer at least two ways in and two ways out, one of which works without the new device.

15.3.6 Spatial ideas on flat screens

Ideas from spatial computing are already reaching ordinary phones. In June 2025 Apple introduced *Liquid Glass*, a translucent material for controls and navigation that refracts and reflects the content behind it, inspired by the depth of visionOS and applied across iOS 26 and Apple's other systems [16]. Layered, see-through surfaces give a sense of depth and keep content visible behind controls, and they raise an old question from Chapter 8 and Chapter 11 in a new form: text on a translucent surface has no fixed background, so its contrast changes with whatever is scrolled behind it. A label that passes the 4.5:1 contrast ratio of WCAG 2.2 over a white page may fail over a photograph. Designers who adopt such materials must test legibility over the worst background the content can produce, and respect the system settings that people with low vision use, such as reducing transparency and increasing contrast, which replace the glass with solid colors.

The same caution applies to every trend that moves from headsets to phones: depth, motion and translucency can guide attention, but they can also cost legibility and cause discomfort for people sensitive to motion. Offer the calm, solid version as a setting, and make sure that everything essential works in it.

15.4 A Method: The Context-of-Use Checklist

Before choosing an emerging interface, describe the context in which it will be used. **Table 15.2** is a checklist with the answers for the puskesmas waiting room.

Table 15.2 The context-of-use checklist for emerging interfaces, answered for the puskesmas waiting room.

Question	Why it matters	Puskesmas waiting room
How far is the person from the information?	Sizes from Equation (15.2)	Bench to wall: 3–5 m
Are the hands free?	Touch, pinch or voice	Often not: children, bags, canes
Where is the attention?	Glance, ambient or focused	On companions, a phone, dozing
How noisy and how bright is it?	Sound and screen legibility	Loud voices, echo; bright windows
Who else is present?	Privacy of what is shown and said	Other patients; names must not be shown
What devices do people have?	Reach and fairness	Most have a basic phone; few have watches; none have glasses
What abilities vary?	Accessibility	Low vision, hearing loss, low literacy, many languages
What happens if it fails?	Fallback	The staff call the number aloud at the desk
How often is it used, by the same people?	Learnability	Monthly visits; nothing may need training

The checklist usually shows that the new device is the last step, not the first. In the waiting room it points to three improvements in order of reach: a larger, higher-contrast number on the television, sized with Equation (15.2); a calm colored lamp above each door; and, for patients who choose it, an SMS or WhatsApp message and an optional wrist tap when their turn is near. Glasses come last, if at all.

15.5 Building the Target-Size Calculator

The first program turns Equation (15.1) and Equation (15.2) into a table that designers can read. It prints the physical size needed for chosen visual angles at four typical distances (a phone in the hand, a kiosk touchscreen, a virtual panel in a headset, and a queue television on a wall), and the visual angle of a one-centimeter button at each distance.

Listing 15.1 `visual_angle.py`: physical size for a visual angle at four distances, and the angle of a 1 cm button.

```
"""How big must a target be? Visual angle <-> physical size at a distance."""
import math
import sys


def size_for(angle_deg, distance_cm):
    """Physical size (cm) that covers angle_deg at distance_cm."""
    return 2 * distance_cm * math.tan(math.radians(angle_deg) / 2)


def angle_of(size_cm, distance_cm):
    """Visual angle (degrees) of an object size_cm wide at distance_cm."""
    return math.degrees(2 * math.atan(size_cm / (2 * distance_cm)))


PLACES = [("phone in the hand", 30), ("kiosk touchscreen", 50),
          ("virtual panel (headset)", 150), ("queue TV on the wall", 300)]


def main(angles):
    print("%-24s %8s" % ("place", "distance") +
          "".join("%9s" % ("%g deg" % a) for a in angles))
    for name, d in PLACES:
        cells = "".join("%6.2f cm" % size_for(a, d) for a in angles)
        print("%-24s %5d cm %s" % (name, d, cells))
    print("\nA 1.0 cm button seen from each place:")
    for name, d in PLACES:
        print("  %-24s %5.2f deg" % (name, angle_of(1.0, d)))


if __name__ == "__main__":
    main([float(a) for a in sys.argv[1:]] or [1.0, 2.0, 3.0])
```

Listing 15.2 is the same calculator in Java. The helper `g` prints 1.0 as "1", as Python's `%g` does, so that the column headings match.

Listing 15.2 `VisualAngle.java`: the same calculator in Java.

```java
/** How big must a target be? Visual angle <-> physical size at a distance. */
public class VisualAngle {
    /** Physical size (cm) that covers angleDeg at distanceCm. */
    static double sizeFor(double angleDeg, double distanceCm) {
        return 2 * distanceCm * Math.tan(Math.toRadians(angleDeg) / 2);
    }

    /** Visual angle (degrees) of an object sizeCm wide at distanceCm. */
    static double angleOf(double sizeCm, double distanceCm) {
        return Math.toDegrees(2 * Math.atan(sizeCm / (2 * distanceCm)));
    }

    static final String[] NAMES = {"phone in the hand", "kiosk touchscreen",
        "virtual panel (headset)", "queue TV on the wall"};
    static final int[] DIST = {30, 50, 150, 300};

    static String g(double x) {             // like Python's %g for 1.0 -> "1"
        return x == Math.rint(x) ? String.valueOf((long) x) : String.valueOf(x);
    }

    public static void main(String[] args) {
        double[] angles = args.length == 0 ? new double[] {1.0, 2.0, 3.0}
            : new double[args.length];
        for (int i = 0; i < args.length; i++)
            angles[i] = Double.parseDouble(args[i]);
        StringBuilder head = new StringBuilder(
            String.format("%-24s %8s", "place", "distance"));
        for (double a : angles) head.append(String.format("%9s", g(a) + " deg"));
        System.out.println(head);
        for (int p = 0; p < NAMES.length; p++) {
            StringBuilder cells = new StringBuilder();
            for (double a : angles)
                cells.append(String.format("%6.2f cm", sizeFor(a, DIST[p])));
            System.out.printf("%-24s %5d cm %s%n", NAMES[p], DIST[p], cells);
        }
        System.out.println("\nA 1.0 cm button seen from each place:");
        for (int p = 0; p < NAMES.length; p++)
            System.out.printf("  %-24s %5.2f deg%n", NAMES[p],
                angleOf(1.0, DIST[p]));
    }
}
```

Run without arguments, both print the sizes for 1, 2 and 3 degrees:

```
place                   distance    1 deg    2 deg    3 deg
phone in the hand          30 cm   0.52 cm  1.05 cm  1.57 cm
kiosk touchscreen          50 cm   0.87 cm  1.75 cm  2.62 cm
virtual panel (headset)   150 cm   2.62 cm  5.24 cm  7.86 cm
queue TV on the wall      300 cm   5.24 cm 10.47 cm 15.71 cm

A 1.0 cm button seen from each place:
  phone in the hand         1.91 deg
  kiosk touchscreen         1.15 deg
  virtual panel (headset)   0.38 deg
  queue TV on the wall      0.19 deg
```

The table makes the scale of the problem visible. A target that covers 2 degrees is about 1 cm on a phone, 1.75 cm on a kiosk, over 5 cm on a virtual panel and over 10 cm on the wall. Read the second half the other way round: a one-centimeter button, perfectly adequate in the hand at 1.91 degrees, shrinks to 0.19 degrees on a television across the room. Angles can also be given on the command line; `python visual_angle.py 0.5 1.5` prints:

```
place                    distance  0.5 deg  1.5 deg
phone in the hand           30 cm   0.26 cm  0.79 cm
kiosk touchscreen           50 cm   0.44 cm  1.31 cm
virtual panel (headset)    150 cm   1.31 cm  3.93 cm
queue TV on the wall       300 cm   2.62 cm  7.85 cm

A 1.0 cm button seen from each place:
  phone in the hand         1.91 deg
  kiosk touchscreen         1.15 deg
  virtual panel (headset)   0.38 deg
  queue TV on the wall      0.19 deg
```

The 1.5-degree column confirms the worked example of Section 15.2.3: at 3 m a number needs 7.85 cm, and at the 5 m of Mbah Sumi's bench it would need 13.1 cm.

15.6 Choosing a Dwell Time

15.6.1 The simulated gaze log

Suppose the puskesmas tries an eye-controlled kiosk for patients who cannot use their hands, where a button activates when looked at long enough. How long is long enough? Too short, and ordinary looking triggers buttons (Midas touch). Too long, and people who really want a button give up before it activates, or feel the kiosk is slow. The answer depends on how long people look at things when they are only reading, and how long they are willing to stare when they want something.

To study the trade-off before a real test, **Listing 15.3** writes a simulated log of 1,000 glances. About one in five is an *intended* selection, in which the person keeps looking until the button fires or their patience runs out; patience is drawn from a log-normal distribution with a median of 1,200 ms. The rest are ordinary glances while reading labels, with a median of 260 ms, close to the mean fixation durations of about 225 to 250 ms reported for silent reading [17]. Both distributions are assumptions to be replaced by data from a real pilot.

Listing 15.3 `make_gaze.py`: writes a seeded, simulated log of 1,000 glances, marking which ones were intended selections.

```
"""Write the simulated gaze log of Chapter 15 (seeded, repeatable)."""
import math
import random

rng = random.Random(15)
with open("../data/gaze_log.csv", "w", encoding="utf-8") as f:
    f.write("glance,duration_ms,intended\n")
    for g in range(1, 1001):
        if rng.random() < 0.2:
            # The user wants this button and keeps looking until it fires
            # or until patience runs out.
            ms = math.exp(rng.gauss(math.log(1200), 0.35))
            f.write("%d,%d,1\n" % (g, ms))
        else:
            # Ordinary looking: reading labels, checking the queue number.
            ms = math.exp(rng.gauss(math.log(260), 0.45))
            f.write("%d,%d,0\n" % (g, ms))
```

15.6.2 The analysis in Python and Java

For a dwell time T, an ordinary glance longer than T is a *false activation*, and an intended selection whose patience is shorter than T is *missed*. Let $F(T)$ be the false activations per 100 glances and

$M(T)$ the percentage of intended selections missed. A single number to minimize needs a judgment about which error is worse; for a kiosk that issues queue numbers, activating the wrong service is more harmful than having to look again, so the program uses the cost

$$J(T) = 3\,F(T) + M(T) \tag{15.4}$$

in which one false activation counts as much as three misses. The weight 3 is a design decision, not a law of nature, and Exercise 5 asks what happens when it changes. **Listing 15.4** evaluates dwell times from 300 to 1,200 ms and reports the one with the lowest cost.

Listing 15.4 `dwell.py`: false activations and missed selections for each dwell time, and the dwell with the lowest cost.

```
"""Choose a dwell time for gaze selection: Midas touch versus waiting."""
import csv
import sys


def load(path):
    with open(path, encoding="utf-8") as f:
        return [(int(r["duration_ms"]), r["intended"] == "1")
                for r in csv.DictReader(f)]


def evaluate(glances, dwell_ms):
    """Return (false per 100 glances, % intended missed) for one dwell."""
    false = sum(1 for ms, want in glances if not want and ms >= dwell_ms)
    wanted = [ms for ms, want in glances if want]
    missed = sum(1 for ms in wanted if ms < dwell_ms)
    return 100 * false / len(glances), 100 * missed / len(wanted)


def main(path):
    glances = load(path)
    print("glances: %d, of which intended selections: %d" % (
        len(glances), sum(w for _, w in glances)))
    print("dwell (ms)  false activations /100  intended missed")
    best = None
    for dwell in range(300, 1300, 100):
        false, missed = evaluate(glances, dwell)
        print("%10d  %22.1f  %14.1f%%" % (dwell, false, missed))
        # A simple cost: a false activation hurts three times a miss.
        cost = 3 * false + missed
        if best is None or cost < best[0]:
            best = (cost, dwell)
    print("lowest cost (3 x false + missed) at dwell = %d ms" % best[1])


if __name__ == "__main__":
    main(sys.argv[1] if len(sys.argv) > 1 else "../data/gaze_log.csv")
```

Listing 15.5 is the same analysis in Java.

Listing 15.5 `Dwell.java`: the same analysis in Java.

```
import java.nio.file.*;
import java.util.*;

/** Choose a dwell time for gaze selection: Midas touch versus waiting. */
public class Dwell {
    record Glance(int ms, boolean intended) {}

    static List<Glance> load(String path) throws Exception {
        List<Glance> out = new ArrayList<>();
        List<String> lines = Files.readAllLines(Path.of(path));
        for (String line : lines.subList(1, lines.size())) {
            String[] f = line.split(",");
            out.add(new Glance(Integer.parseInt(f[1]), f[2].equals("1")));
        }
        return out;
    }

    /** Return {false per 100 glances, % intended missed} for one dwell. */
    static double[] evaluate(List<Glance> glances, int dwellMs) {
        int falseN = 0, wanted = 0, missed = 0;
        for (Glance g : glances) {
            if (!g.intended() && g.ms() >= dwellMs) falseN++;
            if (g.intended()) { wanted++; if (g.ms() < dwellMs) missed++; }
        }
        return new double[] {100.0 * falseN / glances.size(),
            100.0 * missed / wanted};
    }

    public static void main(String[] args) throws Exception {
        var glances = load(args.length > 0 ? args[0] : "../data/gaze_log.csv");
        long wanted = glances.stream().filter(Glance::intended).count();
        System.out.printf("glances: %d, of which intended selections: %d%n",
            glances.size(), wanted);
        System.out.println("dwell (ms)  false activations /100  intended missed");
        double bestCost = Double.MAX_VALUE;
        int bestDwell = 0;
        for (int dwell = 300; dwell < 1300; dwell += 100) {
            double[] r = evaluate(glances, dwell);
            System.out.printf("%10d  %22.1f  %14.1f%%%n", dwell, r[0], r[1]);
            // A simple cost: a false activation hurts three times a miss.
            double cost = 3 * r[0] + r[1];
            if (cost < bestCost) { bestCost = cost; bestDwell = dwell; }
        }
        System.out.printf("lowest cost (3 x false + missed) at dwell = %d ms%n",
            bestDwell);
    }
```

Listing 15.5 (Continued) *Dwell.java: the same analysis in Java.*

```
}
```

15.6.3 Reading the result

Both programs print:

```
glances: 1000, of which intended selections: 166
dwell (ms)  false activations /100  intended missed
       300                   31.5             0.0%
       400                   15.4             0.0%
       500                    7.9             0.0%
       600                    3.4             2.4%
       700                    1.8             6.0%
       800                    0.7            12.0%
       900                    0.1            16.9%
      1000                    0.0            22.9%
      1100                    0.0            33.7%
      1200                    0.0            37.3%
lowest cost (3 x false + missed) at dwell = 700 ms
```

Figure 15.2 draws the two columns. At 300 ms almost a third of all glances would trigger a button: the kiosk would be unusable. False activations fall quickly as the dwell grows, below 2 per 100 at 700 ms and to zero by 1,000 ms, while missed selections begin to climb from 600 ms and pass 20% at 1,000 ms. The cost of Equation (15.4) is lowest at 700 ms, where 1.8 glances per 100 still trigger a button by mistake and 6% of intended selections are missed.

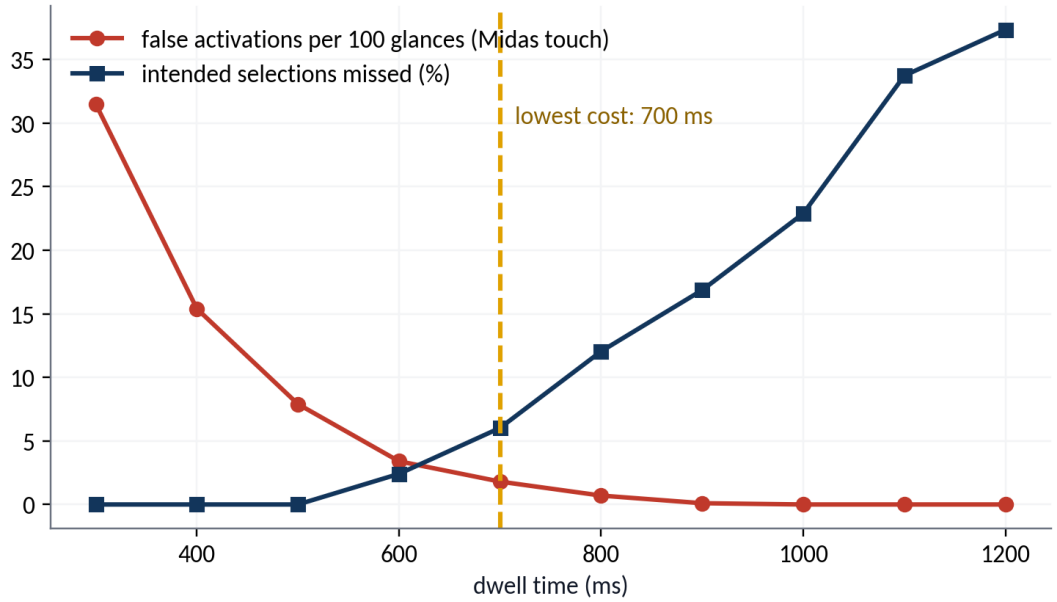


Figure 15.2 The dwell-time trade-off on the simulated gaze log: false activations fall and missed selections rise as the dwell grows; the lowest cost of Equation (15.4) is at 700 ms.

Neither error disappears at any dwell time, and that is the real lesson of the curve. A dwell-only interface always trades one error for the other; the only way out is a second signal. That is why headsets pair gaze with a pinch, and why a public eye-controlled kiosk should add a confirmation step for anything that cannot be undone: “You chose Older adults. Look at YES for one second to print your number.”

15.7 Testing It

Listing 15.6 checks that Equation (15.1) and Equation (15.2) are inverses of each other, that one degree at 3 m is about 5.24 cm, and that the dwell analysis behaves as the trade-off demands: a longer dwell never gives more false activations or fewer misses.

Listing 15.6 `test_spatial.py`: tests of the geometry and of the direction of the dwell trade-off.

```
"""Checks of the spatial helpers; run: python -m unittest test_spatial"""
import unittest

from dwell import evaluate, load
from visual_angle import angle_of, size_for


class SpatialTest(unittest.TestCase):
    def test_round_trip(self):
        self.assertAlmostEqual(angle_of(size_for(2.0, 50), 50), 2.0)
        self.assertAlmostEqual(size_for(1.0, 300), 5.236, places=3)

    def test_dwell_trade_off(self):
        glances = load("../data/gaze_log.csv")
        low, high = evaluate(glances, 300), evaluate(glances, 1000)
        self.assertGreater(low[0], high[0])
        self.assertLessEqual(low[1], high[1])


if __name__ == "__main__":
    unittest.main()
```

Listing 15.7 is the Java version.

Listing 15.7 `SpatialTest.java`: the same checks in Java.

```
/** Checks of the spatial helpers; run: java SpatialTest */
public class SpatialTest {
    static int failed = 0;

    static void check(boolean ok, String what) {
        System.out.println((ok ? "ok   " : "FAIL ") + what);
        if (!ok) failed++;
    }

    public static void main(String[] args) throws Exception {
        double s = VisualAngle.sizeFor(2.0, 50);
        check(Math.abs(VisualAngle.angleOf(s, 50) - 2.0) < 1e-9, "round trip");
        check(Math.abs(VisualAngle.sizeFor(1.0, 300) - 5.236) < 0.001,
            "1 degree at 3 m is about 5.24 cm");
        var glances = Dwell.load("../data/gaze_log.csv");
        double[] low = Dwell.evaluate(glances, 300);
        double[] high = Dwell.evaluate(glances, 1000);
        check(low[0] > high[0], "longer dwell, fewer false activations");
        check(low[1] <= high[1], "longer dwell, more missed selections");
        System.out.println(failed == 0 ? "all checks passed" : failed + " failed");
        System.exit(failed == 0 ? 0 : 1);
    }
}
```

`python -m unittest test_spatial` reports two passing tests; the Java version prints:

```
ok   round trip
ok   1 degree at 3 m is about 5.24 cm
ok   longer dwell, fewer false activations
ok   longer dwell, more missed selections
all checks passed
```

The second group of tests checks a *property* rather than a number: whatever the data, raising the dwell cannot increase false activations. Property tests of this kind stay valid when the simulated log is replaced by real data, which is exactly when they are most needed.

15.8 Making It Better

Confirm, do not only dwell. For the eye-controlled kiosk, use the 700 ms dwell to *select* and a second, deliberate step to *commit*: a separate YES target, a blink pattern chosen by the user, or a switch for people who can press one. The error that remains then requires two mistakes in a row.

Adapt the dwell to the person. Reading speed and patience vary. A kiosk can start at 700 ms and shorten the dwell for a person who selects quickly and never cancels, or lengthen it after a false activation that the person undoes.

Give feedback while dwelling. A ring that fills around the target during the dwell shows that the kiosk has seen the gaze and lets the person look away in time. It turns an invisible timer into a visible promise.

Size targets from the worst reasonable case. Use the angles of **Listing 15.1** with the distance of the farthest seat, not the nearest, and with the eyesight of the patients, not the designers. For gaze, add the tracker's measured error for the people who will use it, as Feit and colleagues recommend [10].

Design the ambient channel with the precise one. A lamp that brightens as the queue approaches tells patients to pay attention; the number on the screen, the voice and the optional message on the phone tell them precisely when to go. Test that the two agree: a lamp that shows "soon" while the screen shows ten people ahead teaches people to ignore the lamp.

15.9 Using It for Real

Mbak Ratna uses this chapter's tools in three steps. First, she measures the distance from the farthest bench to the television (6 m) and runs `python visual_angle.py 1.5`; the result, about 15.7 cm, becomes the specification for the number on the screen, displayed in a heavy typeface with high contrast as Chapter 11 required. Second, she proposes a colored strip above each service door, driven by the same queue data, that turns from grey to amber when five numbers are left and to teal when the number is called. Third, patients who register a phone number and agree to it in a separate, plain sentence receive an SMS when three numbers are ahead of them; those with a watch that shows phone notifications feel it on the wrist.

Only then does she consider eye-controlled or glasses-based options, and only for patients who ask for them. For a patient who cannot use their hands, the gaze kiosk with a 700 ms dwell and a confirmation step would be an alternative way to take a number, next to the touch kiosk of Chapter 16 and the registration desk.

The costs are worth writing down, because they decide what a puskesmas can actually do. A larger number on the existing television costs a change to the queue software's font setting. A strip of colored light above each door costs little more than the light itself and a small controller connected to the queue system. SMS reminders cost a few hundred rupiah per message and some programming. A gaze kiosk costs an eye tracker, a sturdy screen and careful testing with the patients who need it. The order of the list, from cheapest to most expensive, is also the order of how many patients each step helps, which is rarely a coincidence in accessibility work.

The same method serves other places. In a bus terminal, the departure board is sized from the far end of the hall; in a factory, a warning light is calm until it matters and then impossible to miss; in a market, a stall's price sign is sized for the aisle, not for the counter.

15.10 HCI in Practice: Profitina

Profitina is a web application, not a headset app, but the move toward glasses and voice changes its purpose directly. Profitina measures how ready a brand's website is to be cited by AI answer engines. On a phone or a desktop, an answer engine can show several sources and links. Through audio glasses or a voice assistant, the kind announced by Meta and Google for 2026 [4], [5], the person hears *one* spoken answer, perhaps naming one source. Being the source that the answer quotes becomes more valuable when there is no page of links to scroll. That is the argument for Generative Engine Optimization in a sentence, and it is why content that is extractable, structured with schema.org and technically credible matters even to a warung.

Profitina's own interface follows the ambient lessons of this chapter in small ways. The real-time community uses Server-Sent Events to keep "N new" counters current without the user reloading, a quiet peripheral signal rather than an interruption, and the dashboard cockpit gathers notifications in one place, each leading to its section, instead of pushing each one as an alert. And because the AI inside Profitina runs locally on Profitina's own server, a future voice or glasses front end would not need to send users' data to an outside AI service.

15.11 Common Mistakes, and How to Avoid Them

- **Sizing in pixels or centimeters without a distance.** A size means nothing until the viewing distance is known. Specify targets as visual angles and convert with Equation (15.2) for the farthest realistic distance.
- **Designing from the nearest seat.** The designer stands close to the screen; the patient sits at the back. Measure from where the users are.
- **Letting a look activate.** Gaze without a second signal suffers from the Midas touch [9]. Confirm with a pinch, a switch, a separate target or a well-chosen dwell (**Figure 15.2**).
- **Too many haptic patterns.** Two or three distinct patterns can be learned; eight cannot. Pair each pattern with a single meaning.
- **Ambient signals that carry precise information.** A lamp cannot say "your number is L-017". Use ambient signals for "pay attention" and a precise channel for "go now".
- **Requiring the new device.** If the waiting room only works with a smartwatch, most patients are excluded. Every emerging interface needs a path that works without it.
- **Forgetting bystanders.** Cameras and microphones in glasses record people who did not agree. Prefer designs that do not need them and make recording visible.
- **Ignoring comfort.** Long headset sessions and virtual views that move on their own cause sickness [14]. Keep sessions short and optional.

15.12 Chapter Summary

- Spatial interfaces place content in three-dimensional space, wearables are used in glances on the body, and ambient interfaces inform from the environment; all three must be easy to perceive at a glance and hard to trigger by accident.
- The size that matters to the eye is the visual angle (Equation (15.1)); the physical size needed for an angle at a distance follows from Equation (15.2), and Equation (15.3) is a quick approximation.

- In 2026 headsets use eyes, hands and voice [1], [2]; glasses move toward wristband gestures, voice and audio [3], [4], [5]; watches add hands-free gestures such as double tap and wrist flick [6] (**Table 15.1**).
- Looking is not choosing: gaze input needs a second signal or a dwell time to avoid the Midas touch [9], and targets must allow for tracker error that varies between people [10]; spatial interfaces use larger targets, such as Apple's 60-point minimum [11].
- Calm technology moves information between the periphery and the center of attention [12], [13]; ambient signals should say "pay attention" and leave precise information to another channel.
- Comfort, safety, bystander privacy and accessibility need explicit design [14], [15]; every emerging interface needs an alternative that works without it.
- The target-size calculator showed that a 1 cm button covers 1.91 degrees in the hand and 0.19 degrees on a wall at 3 m; the dwell analysis of a simulated gaze log found the lowest cost (Equation (15.4)) at 700 ms, with neither error vanishing at any dwell.
- The context-of-use checklist (**Table 15.2**) usually shows that the cheapest improvement for everyone comes first, and the new device last.

15.13 Exercises

1. A pharmacy shows the number of the next customer on a screen 4 m from the chairs. The number is 6 cm tall. Compute its visual angle with Equation (15.1) and Equation (15.3), in degrees and minutes of arc, and the height needed for 1.5 degrees with Equation (15.2).
2. Run `python visual_angle.py 0.5 1.5` and use the output to answer: at which of the four distances is a 1 cm button smaller than 0.5 degrees? What would you change in each such place?
3. Using the output of `dwell.py`, find the dwell time with the lowest cost if a false activation counts only as much as one miss, $J(T) = F(T) + M(T)$. Explain in one sentence why the answer moves in that direction.
4. For each of these situations, choose spatial, wearable, ambient or none, and justify it with rows of **Table 15.2**: (a) telling a motorbike driver of a laundry that a new pickup is waiting; (b) showing a cook in a warung kitchen which order is next; (c) helping a technician read a machine's wiring diagram while both hands hold tools.
5. Change `dwell.py` and `Dwell.java` so that the weight of a false activation is read from the command line (default 3). Which lines change? What does the program report with weight 10, and why?
6. Design two haptic patterns and one ambient signal for Mbah Sumi's queue, and write the text of the SMS she receives. Check each against the Common Mistakes of Section 15.11.

15.14 Solutions to the Exercises

1. $\theta = 2\arctan(6/800) = 2 \cdot 0.0075 = 0.0150$ radians $= 0.86$ degrees, about 51.6 minutes of arc; Equation (15.3) gives $57.3 \cdot 6/400 = 0.86$ degrees. For 1.5 degrees, $s = 2 \cdot 400 \cdot \tan(0.75^\circ) = 800 \cdot 0.01309 = 10.5$ cm. The number should be almost twice as tall.

2. A 1 cm button covers 1.91 degrees at 30 cm, 1.15 at 50 cm, 0.38 at 150 cm and 0.19 at 300 cm, so it is below 0.5 degrees on the virtual panel and on the wall. On the virtual panel, enlarge interactive targets (for 1.5 degrees at 150 cm the table gives 3.93 cm) or bring the panel closer;

on the wall, nothing should be a *button* at all, and text should be sized from the 1.5-degree column, 7.85 cm at 3 m.

3. With $J = F + M$: 300 ms gives $31.5 + 0 = 31.5$; 400 gives 15.4; 500 gives 7.9; 600 gives $3.4 + 2.4 = 5.8$; 700 gives $1.8 + 6.0 = 7.8$; 800 gives $0.7 + 12.0 = 12.7$. The lowest cost is at 600 ms. When false activations are judged less harmful, the best dwell moves shorter, accepting more accidental activations in exchange for fewer missed selections.

4. (a) Wearable or phone: the driver's hands and eyes are on the road, so no screen; a short haptic pattern on a watch or a sound in a helmet headset says "new pickup", and details are read only when stopped (rows on hands, attention and safety). (b) Ambient plus a screen: a large display on the kitchen wall sized from the stove with Equation (15.2), with a colored light for "an order is waiting" (distance, hands busy, noise). (c) Spatial, if the budget allows: glasses or a headset that show the diagram next to the machine, controlled by voice because the hands are busy; the paper diagram remains the fallback (hands, failure).

5. In Python, read `weight = float(sys.argv[2]) if len(sys.argv) > 2 else 3.0`, pass it to `main`, and replace `3 * false` by `weight * false` and the text "3 x false" by "%g x false" % weight. In Java, parse `args[1]` in the same way and use it in `cost` and in the last `printf`. With weight 10, the costs are $10F + M$: 700 ms gives $18 + 6 = 24$, 800 ms gives $7 + 12 = 19$, 900 ms gives $1 + 16.9 = 17.9$, 1,000 ms gives $0 + 22.9 = 22.9$, so the lowest cost moves to 900 ms: the more a false activation is feared, the longer the dwell.

6. One possible design. Haptics: a gentle double tap when three numbers are ahead ("get ready"); one long, strong vibration when the number is called ("go now"). Ambient: a strip of light above the older-adults door that turns amber when five numbers are left and teal when the number is called. SMS: "Puskesmas Sehat Selalu: number L-017, 3 people ahead of you. Please stay near the Older adults door." Checks: only two haptic patterns; the ambient signal says "pay attention", and the precise number comes by SMS and on the screen; the SMS needs no new device; it contains no name or diagnosis, so it reveals nothing to someone else reading the phone; and patients who do not register still have the screen, the voice and the staff.

References

[1] Apple Inc., "Apple Vision Pro upgraded with the M5 chip and Dual Knit Band," press release, Oct. 15, 2025. [Online]. Available: https://www.apple.com/newsroom/2025/10/apple-vision-pro-upgraded-with-the-m5-chip-and-dual-knit-band/

[2] Samsung Electronics, "Introducing Galaxy XR: Opening new worlds," Oct. 2025. [Online]. Available: https://news.samsung.com/us/introducing-galaxy-xr-opening-new-worlds/

[3] Meta, "Meta Ray-Ban Display: AI glasses with an EMG wristband," Sep. 17, 2025. [Online]. Available: https://about.fb.com/news/2025/09/meta-ray-ban-display-ai-glasses-emg-wristband/

[4] Engadget, "Everything announced at Meta Connect 2026," Sep. 2026. [Online]. Available: https://www.engadget.com/2267230/everything-announced-at-meta-connect-2026/

[5] Google, "Intelligent eyewear with Gemini is coming this fall," *The Keyword*, May 19, 2026. [Online]. Available: https://blog.google/products-and-platforms/platforms/android/android-xr-io-2026/

[6] 9to5Mac, "watchOS 26 added a new Apple Watch trick that I've been loving," Feb. 4, 2026. [Online]. Available: https://9to5mac.com/2026/02/04/watchos-26-added-a-new-apple-watch-trick-that-ive-been-loving/

[7] 9to5Mac, "visionOS 27 announced with new features for Vision Pro," Jun. 8, 2026. [Online]. Available: https://9to5mac.com/2026/06/08/visionos-27-announced-with-new-features-for-vision-pro/

[8] J. Clover, “Apple has given up on the Vision Pro after M5 refresh flop,” MacRumors, Apr. 29, 2026. [Online]. Available: https://www.macrumors.com/2026/04/29/apple-vision-pro-m5-flop/

[9] R. J. K. Jacob, “What you look at is what you get: Eye movement-based interaction techniques,” in *Proc. CHI ’90*, Seattle, WA, USA, 1990, pp. 11–18.

[10] A. M. Feit *et al.*, “Toward everyday gaze input: Accuracy and precision of eye tracking and implications for design,” in *Proc. CHI ’17*, Denver, CO, USA, 2017, pp. 1118–1130, doi: 10.1145/3025453.3025599.

[11] Apple Inc., “Design for spatial user interfaces,” WWDC23 session 10076, Jun. 2023. [Online]. Available: https://developer.apple.com/videos/play/wwdc2023/10076/

[12] M. Weiser, “The computer for the 21st century,” *Scientific American*, vol. 265, no. 3, pp. 94–104, 1991.

[13] M. Weiser and J. S. Brown, “Designing calm technology,” *PowerGrid Journal*, vol. 1, no. 1, 1996.

[14] J. J. LaViola Jr., E. Kruijff, R. P. McMahan, D. A. Bowman, and I. Poupyrev, *3D User Interfaces: Theory and Practice*, 2nd ed. Boston, MA, USA: Addison-Wesley, 2017.

[15] W3C, “XR Accessibility User Requirements,” W3C Group Note, Aug. 25, 2021. [Online]. Available: https://www.w3.org/TR/xaur/

[16] Apple Inc., “Apple introduces a delightful and elegant new software design,” press release, Jun. 9, 2025. [Online]. Available: https://www.apple.com/newsroom/2025/06/apple-introduces-a-delightful-and-elegant-new-software-design/

[17] K. Rayner, “Eye movements in reading and information processing: 20 years of research,” *Psychological Bulletin*, vol. 124, no. 3, pp. 372–422, 1998.

Chapter 16

Capstone: Designing and Evaluating a Complete Interactive System

Fifteen chapters have each taught one part of human-computer interaction. This chapter puts them together on one real problem, from the first visit to the people who have it to the evidence that a design helps them. The problem is small enough to finish and serious enough to matter: a self-service kiosk at which patients of a community health center take a queue number. By the end of the chapter you will be able to run a complete, iterative design project: study the context and the people; turn what you learn into requirements, personas and scenarios; apply human-factors laws, design heuristics, accessibility success criteria and ethical rules as design constraints; prototype on paper and then as a working web application in Python and Java; test it with users, including older adults, and measure it with task completion, the System Usability Scale, UMUX-Lite and color-contrast checks; turn the findings into a prioritized list of changes; and present design decisions so that others can check them. Every step names the chapter of this book whose method it applies.

16.1 A Kiosk for Puskesmas Sehat Selalu

Puskesmas Sehat Selalu, a community health center on the edge of a city in East Java, serves about two hundred patients on a busy morning. Registration happens at one desk. Patients arrive from seven, wait in a line that curls out of the door, give their name and complaint, and receive a paper slip with a number for one of four services: the general clinic, maternal and child health (KIA), the dental clinic, or the older-adults clinic. Chapter 15 met one of them, Mbah Sumi, who has missed her turn twice because she could not read the number on the television.

The head of the puskesmas wants to shorten the line at the desk. Her idea is a touchscreen kiosk at the entrance, at which patients who already know which service they need take a number themselves; the desk then serves only new patients and questions. She asks a team of three informatics students to design it, with Mbak Ratna, the nurse, as their guide. The students have a semester, no budget beyond a donated tablet, and one rule from the head of the puskesmas: *nobody may be worse off than today.*

That rule is the capstone's real subject. A kiosk that is fast for young people with phones and baffling for Mbah Sumi would shorten the line and still make the puskesmas worse for the people who need it most. The students must therefore do what this book has taught: find out who the users are and what they need, design for the least advantaged of them, build something that works, and prove with evidence that it helps.

16.2 The Idea in Plain Words

16.2.1 A complete interactive system

A *complete* interactive system is more than a screen that works. It is a service that fits the place, the people and the rules around it: the kiosk, the paper slip it replaces, the desk it relieves, the

nurse who calls the numbers, the patients who cannot or will not use it, and the law that protects their data and their access. ISO 9241-210, the international standard for human-centered design of interactive systems, describes the work as a cycle of four activities repeated until the design meets the requirements: understand and specify the context of use, specify the user requirements, produce design solutions, and evaluate the designs against the requirements [1]. Chapter 4 taught that cycle; this chapter runs it once, completely, and starts a second round.

Figure 16.1 shows the path of the capstone in the four phases of the Double Diamond, the design process published by the UK Design Council, with the chapters whose methods each phase uses [2]. The arrow at the bottom is the most important part of the picture: evidence from the evaluation sends the team back to an earlier phase.

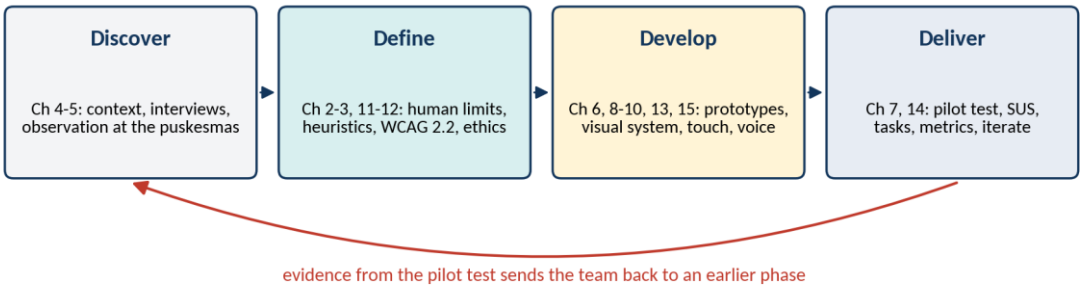


Figure 16.1 The capstone path: four phases, the chapters whose methods each phase applies, and the loop back that turns one design into an iterative process.

16.2.2 Every chapter as a decision

The simplest way to see how the book fits together is to list, for each chapter, the decision it makes in the kiosk project. **Table 16.1** does that for Chapters 2 to 15. Read it now as a map; the sections that follow walk through it.

Table 16.1 The methods of Chapters 2–15 and the decision each one contributed to the queue kiosk.

Ch	Method from the chapter	What it decided for the kiosk
2	Perception, memory, Hick's and Fitts's laws	Four choices only; large targets near the hand; number shown, not to be remembered
3	Design principles and heuristics	Visibility of status (the number and people ahead), match with the real world (service names used at the desk), error prevention
4	User-centered process	One cycle of research, design, test and a second iteration planned
5	User research	Observation at the desk on two mornings; interviews with six patients and two staff
6	Prototyping	Paper prototype, then a working HTML prototype served by Python and Java
7	Usability evaluation	Pilot test with eight participants: tasks, SUS, UMUX-Lite
8	Visual design and design systems	One typeface, one navy for actions, one red only for urgent help; consistent spacing
9	Interaction styles	Touch for choosing; no gestures to learn; the desk and voice remain alternatives
10	Mobile and responsive UX	The same page works on the tablet, a phone and a desktop browser
11	Accessibility (WCAG 2.2)	Contrast, target size, skip link, language of parts, status messages, larger-text mode
12	Ethics and trust	No personal data collected; no pressure; an urgent-help note that sends people to a person
13	AI interfaces	No AI in the kiosk: nothing here needs a guess, and a guess could harm
14	Data-informed UX	HEART metrics chosen; anonymous counts only; the status endpoint for monitoring
15	Emerging interfaces	Sizes from viewing distance; queue television sized from the far bench

The row for Chapter 13 is worth a second look. Choosing *not* to use a technology is a design decision too. A kiosk that asked patients to describe their complaint to an AI, which then chose the service, would add a guess where the patient already knows the answer, and a wrong guess sends a sick person to the wrong room. The capstone uses each method where it earns its place and nowhere else.

16.3 Discover: Research at the Registration Desk

Following Chapter 5, the team began by watching. On two mornings, with the permission of the head of the puskesmas and after explaining to patients what they were doing, two students sat near the registration desk from seven to ten and noted what happened, without recording names or faces. They interviewed six patients while they waited, chosen to include two older adults, a mother with a baby and a young worker, and they interviewed Mbak Ratna and the registration clerk.

Four findings shaped everything that followed. First, about two thirds of the patients on those mornings were *returning* patients who already knew their service; they queued only to get a slip. Second, most older patients came with a companion, often a grandchild, but some came alone, and several said that they "do not dare" to touch screens in public places for fear of breaking something or looking foolish. Third, the clerk regularly moved patients forward in the line when they looked seriously ill, and she worried that a kiosk would hide such patients among the others. Fourth, patients spoke Indonesian, Javanese and Madurese, and one interviewee, a factory worker from another province, asked for the forms in "simple Indonesian, not officials' Indonesian".

From these findings the team wrote two personas and a scenario, as Chapter 4 recommends. *Bu Lestari*, 34, brings her baby to the KIA clinic every month for immunization, knows exactly where she is going, and wants the number in ten seconds while holding the baby with one arm. *Mbah Sumi*, 72, comes alone every second Tuesday for her blood-pressure check, reads large print only, has never used a touchscreen kiosk, and will not use one if it makes her feel slow. The scenario followed Mbah Sumi from the becak to the bench, and it contained the sentence that became the first requirement: *she must be able to take a number with one touch, or ignore the kiosk entirely and go to the desk as before.*

16.4 Define: Requirements and Constraints

16.4.1 Requirements

The team wrote the requirements as testable statements, each traceable to a finding.

1. A returning patient takes a number for one of four services in one touch after reading the screen (finding 1).
2. The kiosk works without any personal data: no name, no national identity number, no phone number (Chapter 12; finding 2).
3. Anyone with urgent symptoms is sent to a person at once (finding 3).
4. The screen is usable in Indonesian and English, with the words used at the desk, and can switch to larger text (finding 4; Chapter 11).
5. Nobody is forced to use the kiosk: the desk stays open, and the paper slip and the kiosk number are equivalent (the head's rule).
6. The number and the people ahead are shown in large type, and the patient does not need to remember anything (Chapter 2).

16.4.2 Human-factors budget

Chapter 2 gives two laws that turn requirement 1 into numbers. Hick's law says that the time to choose among n equally likely options grows with the logarithm of n [3]:

$$T_{\text{choose}} = a + b \log_2(n + 1) \tag{16.1}$$

With four services the choice involves $\log_2 5 = 2.32$ bits; with the eight services that a first draft listed (four clinics plus pharmacy, laboratory, registration questions and "other"), it would involve $\log_2 9 = 3.17$ bits, more than a third more. The team kept four, and left the other purposes to the desk.

Fitts's law says that the time to reach a target grows with the index of difficulty, which depends on the distance D to the target and its width W along the direction of movement [4]:

$$ID = \log_2\left(\frac{D}{W} + 1\right) \tag{16.2}$$

On the tablet, mounted at waist height, a hand resting at the bottom edge travels at most about 300 pixels to the highest button. With buttons 64 pixels tall, $ID = \log_2(300/64 + 1) = \log_2 5.69 = 2.51$ bits; with the 32-pixel text links of a typical website it would be $\log_2(300/32 + 1) = 3.37$ bits. Making the buttons full width and tall is cheap and helps everyone, and it helps most those whose hands are unsteady.

16.4.3 Accessibility, law and ethics as constraints

Accessibility is not a feature added at the end; it is a set of constraints known from the start. WCAG 2.2, which became the international standard ISO/IEC 40500:2025 in October 2025, provides the testable success criteria [5]. **Table 16.2** lists those the kiosk must meet and how it meets them.

Table 16.2 The WCAG 2.2 success criteria checked for the queue kiosk and how each is met.

WCAG 2.2 success criterion	Requirement	How the kiosk meets it
1.4.3 Contrast (Minimum), AA	Text contrast at least 4.5:1	Checked with `evaluate.py`; lowest pair 7.56:1
1.4.11 Non-text Contrast, AA	Focus indicators and controls at least 3:1	Focus ring changed from amber (2.27:1) to navy (12.46:1)
1.4.4 Resize Text, AA	Text can be enlarged without loss	"Larger text" link; layout reflows
2.4.1 Bypass Blocks, A	A way to skip repeated content	"Skip to the services" link, visible on focus
2.4.7 Focus Visible, AA	Keyboard focus is visible	Thick outline with offset on every link and button
2.5.8 Target Size (Minimum), AA	Targets at least 24 by 24 CSS pixels	Buttons 64 pixels tall, full width; links about 34 pixels tall
3.1.1 and 3.1.2 Language of Page and of Parts, A and AA	Language declared	`lang` on the page and on the link to the other language
4.1.3 Status Messages, AA	Changes announced to assistive technology	Number shown in a `role="status"` region with `aria-live`

The law sets further constraints. Indonesia's Law 8/2016 on persons with disabilities makes accessible public services a right [6], and Government Regulation PP 42/2020 sets out the accessibility of public services for persons with disabilities [7]; a puskesmas is a public service. Outside Indonesia, the European Accessibility Act has applied since June 28, 2025 to many products and services, including self-service terminals such as ticketing and check-in machines [8], a sign of where the rest of the world is going. And because the kiosk collects no personal data at all, it needs no consent screen under UU PDP 27/2022 and PP 33/2026 [9]; the cheapest way to comply with a data-protection law is to collect nothing that it protects.

Chapter 12's ethical rules complete the constraints: no countdown timers, no "are you sure you want to leave?", no advertising on a health kiosk, and an urgent-help note that points to a person rather than trying to diagnose.

16.5 Develop: From Paper to a Working Kiosk

16.5.1 The paper prototype

Chapter 6 recommends starting on paper, and the team did. Three sketches on A4 sheets, held in front of the tablet stand, were enough to test the flow with four patients in the waiting room. Two lessons came out at once. The first sketch opened with a welcome screen and a "Start" button; all four patients touched the logo instead, and one asked "is it broken?". The second sketch put the four services on the first screen, and all four patients chose correctly without hesitation. Mbah Sumi, who agreed to try, said the words were "too small for old eyes", which is how the larger-text mode began. A prototype that costs ten minutes removed a whole screen from the design.

16.5.2 The design decisions

The decisions that followed are visible in the running kiosk of **Figure 16.2**. The language and text-size links sit at the top (marker 1). An urgent-help note, the only red element, tells anyone with chest pain, difficulty breathing, heavy bleeding or a baby with a high fever to go to the registration counter at once (marker 2): the kiosk does not triage, it hands off to a person, exactly as the clerk's practice required. The four services are full-width buttons in one navy color, named as the desk names them (marker 3). After a touch, the second screen shows the service, the number in very large type and the number of people ahead, inside a status region that screen readers announce (marker 4), with a single "Done" button.



Figure 16.2 The running kiosk, captured with Playwright: the home screen and the number screen. Markers: 1 language and text size, 2 urgent-help note, 3 the four services, 4 the number and the people ahead in a status region.

Two decisions were deliberately left out. There is no estimated waiting time, because the puskesmas cannot predict it honestly; "people ahead of you" is a fact, a minute estimate would be a guess. And there is no JavaScript: every page is plain HTML and a form, so the kiosk works on an old tablet, with the browser's own accessibility features, and cannot break on a script error.

16.6 Building the Kiosk

16.6.1 The kiosk in Python

Listing 16.1 is the whole kiosk as a Flask application. The dictionary `TEXT` holds every string in both languages, so that switching language changes words and nothing else. The function `page` wraps a body in the common layout: `lang` on the page, the skip link, the tool links (with `lang` on the link to the other language), and the style sheet. The route `/` shows the services as buttons in one form; `/take` issues the next number for the chosen service and shows it with the people ahead; and `/status.json` returns the counts that the desk or a dashboard can read.

Listing 16.1 `kiosk_app.py`: the queue kiosk in Python with Flask, in two languages and with a larger-text mode.

```
"""Puskesmas queue kiosk: large targets, two languages, no JavaScript needed."""
from html import escape

from flask import Flask, redirect, request

TEXT = {
    "id": {"title": "Ambil Nomor Antrean", "skip": "Langsung ke pilihan layanan",
           "pick": "Pilih layanan yang Anda tuju:",
           "urgent": "Nyeri dada, sesak napas, perdarahan hebat, atau bayi "
                     "demam tinggi? Jangan menunggu: datangi loket "
                     "pendaftaran sekarang dan beri tahu petugas.",
           "your": "Nomor Anda", "ahead": "Orang di depan Anda",
           "show": "Tunjukkan nomor ini di loket saat dipanggil.",
           "done": "Selesai", "bigger": "Perbesar huruf",
           "smaller": "Huruf normal", "other": "English",
           "services": {"U": "Poli Umum", "K": "KIA (Ibu dan Anak)",
                        "G": "Poli Gigi", "L": "Lansia (60 tahun ke atas)"}},
    "en": {"title": "Take a Queue Number", "skip": "Skip to the services",
           "pick": "Choose the service you need:",
           "urgent": "Chest pain, hard to breathe, heavy bleeding, or a baby "
                     "with a high fever? Do not wait: go to the registration "
                     "counter now and tell the staff.",
           "your": "Your number", "ahead": "People ahead of you",
           "show": "Show this number at the counter when it is called.",
           "done": "Done", "bigger": "Larger text",
           "smaller": "Normal text", "other": "Bahasa Indonesia",
           "services": {"U": "General clinic", "K": "Mother and child",
                        "G": "Dental clinic", "L": "Older adults (60+)"}},
}
CSS = """body{font:1.4rem/1.5 system-ui,sans-serif;margin:0;color:#111827;
background:#fff}body.big{font-size:1.9rem}
main{max-width:40rem;margin:auto;padding:1rem}
h1{color:#12355B;margin:.5rem 0}
.skip{position:absolute;left:-99em}.skip:focus{left:1rem}
.urgent{background:#FDECEA;border-left:.5rem solid #C0392B;
padding:.75rem}
button,.btn{display:block;width:100%;min-height:4rem;margin:.6rem 0;
font:inherit;color:#fff;background:#12355B;border:0;
border-radius:.5rem;text-align:center;text-decoration:none;
padding:.8rem}.btn{display:flex;align-items:center;
justify-content:center}.tools a{color:#0B5E5E;margin-right:1.5rem}
a:focus,button:focus{outline:.25rem solid #12355B;outline-offset:.2rem}
.number{font-size:4rem;color:#12355B;font-weight:bold;margin:0}"""
queues = {code: 0 for code in "UKGL"}     # numbers given today
served = {code: 0 for code in "UKGL"}     # numbers already called
```

Listing 16.1 (Continued) *kiosk_app.py: the queue kiosk in Python with Flask, in two languages and with a larger-text mode.*

```
def page(lang, big, body):
    t = TEXT[lang]
    other = "en" if lang == "id" else "id"
    size = "0" if big else "1"
    return ("<!DOCTYPE html>\n<html lang=\"%s\"><head><meta charset=\"utf-8\">"
            "<meta name=\"viewport\" content=\"width=device-width\">"
            "<title>%s</title><style>%s</style></head>\n<body%s>"
            "<a class=\"skip\" href=\"#services\">%s</a><main>"
            "<p class=\"tools\"><a href=\"/?lang=%s&amp;big=%s\" lang=\"%s\">%s</a>"
            "<a href=\"/?lang=%s&amp;big=%s\">%s</a></p>\n%s</main></body></html>"
            % (lang, t["title"], CSS, " class=\"big\"" if big else "",
               t["skip"], other, "1" if big else "0", other, t["other"],
               lang, size, t["smaller"] if big else t["bigger"], body))


app = Flask(__name__)


def options():
    lang = request.values.get("lang", "id")
    return (lang if lang in TEXT else "id"), request.values.get("big") == "1"


@app.get("/")
def home():
    lang, big = options()
    t = TEXT[lang]
    buttons = "".join(
        "<button name=\"service\" value=\"%s\">%s</button>" % (c, escape(n))
        for c, n in t["services"].items())
    body = ("<h1>%s</h1><p class=\"urgent\" role=\"note\">%s</p>"
            "<form method=\"post\" action=\"/take\" id=\"services\">"
            "<input type=\"hidden\" name=\"lang\" value=\"%s\">"
            "<input type=\"hidden\" name=\"big\" value=\"%s\">"
            "<p>%s</p>%s</form>" % (t["title"], t["urgent"], lang,
                                    "1" if big else "0", t["pick"], buttons))
    return page(lang, big, body)
```

Listing 16.1 (Continued) *kiosk_app.py: the queue kiosk in Python with Flask, in two languages and with a larger-text mode.*

```
@app.post("/take")
def take():
    lang, big = options()
    code = request.form.get("service", "")
    if code not in queues:
        return redirect("/?lang=" + lang)
    queues[code] += 1
    t = TEXT[lang]
    body = ("<h1>%s</h1><div role=\"status\" aria-live=\"polite\">"
            "<p>%s:</p><p class=\"number\">%s-%03d</p>"
            "<p>%s: <strong>%d</strong></p></div><p>%s</p>"
            "<a class=\"btn\" href=\"/?lang=%s&amp;big=%s\">%s</a>"
            % (escape(t["services"][code]), t["your"], code, queues[code],
               t["ahead"], queues[code] - served[code] - 1, t["show"],
               lang, "1" if big else "0", t["done"]))
    return page(lang, big, body)


@app.get("/status.json")
def status():
    return {c: {"given": queues[c], "served": served[c]} for c in queues}


if __name__ == "__main__":
    app.run(host="127.0.0.1", port=4160)
```

Four details carry the design decisions into the code. The text is escaped with `html.escape` where it could contain special characters. The form posts to `/take` and the result is a new page, so the browser's Back button and the "Done" button both return to a clean start. An unknown service code simply redirects to the home page, so a tampered form cannot create a strange queue. And the focus outline is navy, the color the contrast check of Section 16.7 required.

16.6.2 The kiosk in Java

Listing 16.2 is the same kiosk with the JDK's built-in `com.sun.net.httpserver`, so that it runs with nothing but Java installed. It serves exactly the same HTML as the Python version; the team checked this by fetching both pages with `curl` and comparing them byte for byte.

Listing 16.2 `KioskServer.java`: the same kiosk with the JDK's HTTP server.

```
import com.sun.net.httpserver.*;
import java.io.*;
import java.net.*;
import java.nio.charset.StandardCharsets;
import java.util.*;

/** Puskesmas queue kiosk: large targets, two languages, no JavaScript needed. */
public class KioskServer {
    static final Map<String, Map<String, String>> TEXT = Map.of(
        "id", Map.ofEntries(Map.entry("title", "Ambil Nomor Antrean"),
            Map.entry("skip", "Langsung ke pilihan layanan"),
            Map.entry("pick", "Pilih layanan yang Anda tuju:"),
            Map.entry("urgent", "Nyeri dada, sesak napas, perdarahan hebat, "
                + "atau bayi demam tinggi? Jangan menunggu: datangi loket "
                + "pendaftaran sekarang dan beri tahu petugas."),
            Map.entry("your", "Nomor Anda"),
            Map.entry("ahead", "Orang di depan Anda"),
            Map.entry("show", "Tunjukkan nomor ini di loket saat dipanggil."),
            Map.entry("done", "Selesai"), Map.entry("bigger", "Perbesar huruf"),
            Map.entry("smaller", "Huruf normal"), Map.entry("other", "English"),
            Map.entry("U", "Poli Umum"), Map.entry("K", "KIA (Ibu dan Anak)"),
            Map.entry("G", "Poli Gigi"),
            Map.entry("L", "Lansia (60 tahun ke atas)")),
        "en", Map.ofEntries(Map.entry("title", "Take a Queue Number"),
            Map.entry("skip", "Skip to the services"),
            Map.entry("pick", "Choose the service you need:"),
            Map.entry("urgent", "Chest pain, hard to breathe, heavy bleeding, "
                + "or a baby with a high fever? Do not wait: go to the "
                + "registration counter now and tell the staff."),
            Map.entry("your", "Your number"),
            Map.entry("ahead", "People ahead of you"),
            Map.entry("show",
                "Show this number at the counter when it is called."),
            Map.entry("done", "Done"), Map.entry("bigger", "Larger text"),
            Map.entry("smaller", "Normal text"),
            Map.entry("other", "Bahasa Indonesia"),
            Map.entry("U", "General clinic"), Map.entry("K", "Mother and child"),
            Map.entry("G", "Dental clinic"), Map.entry("L", "Older adults (60+)")));
    static final String CSS = """
        body{font:1.4rem/1.5 system-ui,sans-serif;margin:0;color:#111827;
        background:#fff}body.big{font-size:1.9rem}
        main{max-width:40rem;margin:auto;padding:1rem}
        h1{color:#12355B;margin:.5rem 0}
        .skip{position:absolute;left:-99em}.skip:focus{left:1rem}
        .urgent{background:#FDECEA;border-left:.5rem solid #C0392B;
        padding:.75rem}
```

Listing 16.2 (Continued) *KioskServer.java: the same kiosk with the JDK's HTTP server.*

```
        button,.btn{display:block;width:100%;min-height:4rem;margin:.6rem 0;
        font:inherit;color:#fff;background:#12355B;border:0;
        border-radius:.5rem;text-align:center;text-decoration:none;
        padding:.8rem}.btn{display:flex;align-items:center;
        justify-content:center}.tools a{color:#0B5E5E;margin-right:1.5rem}
        a:focus,button:focus{outline:.25rem solid #12355B;outline-offset:.2rem}
        .number{font-size:4rem;color:#12355B;font-weight:bold;margin:0}""";
    static final String CODES = "UKGL";
    static final Map<String, Integer> queues = new LinkedHashMap<>();  // given
    static final Map<String, Integer> served = new LinkedHashMap<>();  // called

    static String esc(String s) {
        return s.replace("&", "&amp;").replace("<", "&lt;").replace(">", "&gt;")
            .replace("\"", "&quot;").replace("'", "&#x27;");
    }

    static String page(String lang, boolean big, String body) {
        var t = TEXT.get(lang);
        String other = lang.equals("id") ? "en" : "id";
        return String.format("<!DOCTYPE html>\n<html lang=\"%s\"><head>"
            + "<meta charset=\"utf-8\"><meta name=\"viewport\" "
            + "content=\"width=device-width\"><title>%s</title><style>%s</style>"
            + "</head>\n<body%s><a class=\"skip\" href=\"#services\">%s</a><main>"
            + "<p class=\"tools\"><a href=\"/?lang=%s&amp;big=%s\" "
            + "lang=\"%s\">%s</a><a href=\"/?lang=%s&amp;big=%s\">%s</a></p>"
            + "\n%s</main></body></html>",
            lang, t.get("title"), CSS, big ? " class=\"big\"" : "", t.get("skip"),
            other, big ? "1" : "0", other, t.get("other"), lang, big ? "0" : "1",
            big ? t.get("smaller") : t.get("bigger"), body);
    }

    static Map<String, String> form(HttpExchange ex) throws IOException {
        String q = ex.getRequestURI().getRawQuery();
        String body = new String(ex.getRequestBody().readAllBytes(),
            StandardCharsets.UTF_8);
        Map<String, String> m = new HashMap<>();
        for (String part : ((q == null ? "" : q) + "&" + body).split("&")) {
            String[] kv = part.split("=", 2);
            if (kv.length == 2)
                m.put(URLDecoder.decode(kv[0], StandardCharsets.UTF_8),
                    URLDecoder.decode(kv[1], StandardCharsets.UTF_8));
        }
        return m;
    }
```

Listing 16.2 (Continued) *KioskServer.java: the same kiosk with the JDK's HTTP server.*

```
    static void send(HttpExchange ex, int code, String type, String text)
            throws IOException {
        byte[] b = text.getBytes(StandardCharsets.UTF_8);
        ex.getResponseHeaders().set("Content-Type", type + "; charset=utf-8");
        ex.sendResponseHeaders(code, b.length);
        try (OutputStream os = ex.getResponseBody()) { os.write(b); }
    }

    static void handle(HttpExchange ex) throws IOException {
        var f = form(ex);
        String lang = f.getOrDefault("lang", "id");
        if (!TEXT.containsKey(lang)) lang = "id";   // Map.of rejects null keys
        boolean big = "1".equals(f.get("big"));
        var t = TEXT.get(lang);
        String path = ex.getRequestURI().getPath();
        if (path.equals("/status.json")) {
            StringBuilder j = new StringBuilder("{");
            for (String c : queues.keySet())
                j.append(String.format("%s\"%s\": {\"given\": %d, \"served\": %d}",
                    j.length() > 1 ? ", " : "", c, queues.get(c), served.get(c)));
            send(ex, 200, "application/json", j + "}");
        } else if (path.equals("/take") && ex.getRequestMethod().equals("POST")) {
            String code = f.getOrDefault("service", "");
            if (code.length() != 1 || !CODES.contains(code)) {
                ex.getResponseHeaders().set("Location", "/?lang=" + lang);
                send(ex, 303, "text/plain", "");
                return;
            }
            int n = queues.merge(code, 1, Integer::sum);
            String body = String.format("<h1>%s</h1><div role=\"status\" "
                + "aria-live=\"polite\"><p>%s:</p><p class=\"number\">%s-%03d</p>"
                + "<p>%s: <strong>%d</strong></p></div><p>%s</p>"
                + "<a class=\"btn\" href=\"/?lang=%s&amp;big=%s\">%s</a>",
                esc(t.get(code)), t.get("your"), code, n, t.get("ahead"),
                n - served.get(code) - 1, t.get("show"), lang, big ? "1" : "0",
                t.get("done"));
            send(ex, 200, "text/html", page(lang, big, body));
        } else if (path.equals("/")) {
            StringBuilder buttons = new StringBuilder();
            for (char c : CODES.toCharArray())
                buttons.append(String.format(
                    "<button name=\"service\" value=\"%c\">%s</button>",
                    c, esc(t.get(String.valueOf(c)))));
            String body = String.format("<h1>%s</h1><p class=\"urgent\" "
                + "role=\"note\">%s</p><form method=\"post\" action=\"/take\" "
                + "id=\"services\"><input type=\"hidden\" name=\"lang\" "
```

Listing 16.2 (Continued) *KioskServer.java: the same kiosk with the JDK's HTTP server.*

```
                + "value=\"%s\"><input type=\"hidden\" name=\"big\" value=\"%s\">"
                + "<p>%s</p>%s</form>", t.get("title"), t.get("urgent"), lang,
                big ? "1" : "0", t.get("pick"), buttons);
            send(ex, 200, "text/html", page(lang, big, body));
        } else send(ex, 404, "text/plain", "not found");
    }

    public static void main(String[] args) throws IOException {
        for (char c : CODES.toCharArray()) {
            queues.put(String.valueOf(c), 0);
            served.put(String.valueOf(c), 0);
        }
        HttpServer server = HttpServer.create(
            new InetSocketAddress("127.0.0.1", 4161), 0);
        server.createContext("/", ex -> {
            try {
                handle(ex);
            } catch (Exception e) {   // show the error, keep serving
                e.printStackTrace();
                send(ex, 500, "text/plain", "internal error");
            }
        });
        server.start();
        System.out.println("Kiosk running at http://127.0.0.1:4161/");
    }
}
```

Two small differences between the languages taught the team something. `Map.of` in Java rejects `null` keys with an exception, so the first version crashed on every request without a `lang` parameter; the fix reads the parameter with a default before looking it up, and the handler now catches exceptions, logs them and answers with an error page instead of closing the connection. A kiosk must never hang silently: a patient in front of a frozen screen does not know whether to wait or leave.

16.6.3 Running it

Start the Python kiosk with `python kiosk_app.py` in the `python` folder and open http://127.0.0.1:4160/, or the Java kiosk with `javac -d out *.java` and `java -cp out KioskServer` in the `java` folder and open http://127.0.0.1:4161/. The kiosk can also be driven from a terminal, which is how the team tested it before any user saw it. Taking two numbers for the older-adults clinic

```
$ curl -s -X POST -d "lang=en&service=L" http://127.0.0.1:4160/take
$ curl -s -X POST -d "lang=en&service=L" http://127.0.0.1:4160/take
```

returns two pages whose key lines are

```
L-001
ahead of you: 0
L-002
ahead of you: 1
```

and the status endpoint, pretty-printed with `curl -s http://127.0.0.1:4160/status.json | python -m json.tool`, then reports

```
{
    "G": {
        "given": 0,
        "served": 0
    },
    "K": {
        "given": 0,
        "served": 0
    },
    "L": {
        "given": 2,
        "served": 0
    },
    "U": {
        "given": 0,
        "served": 0
    }
}
```

The screenshots in this chapter come from the running kiosk itself, captured by the Playwright script of **Listing 16.3** [10]. It opens the home page in both languages, presses the fourth button, opens the larger-text mode and presses Tab once so that the skip link appears with its focus ring. **Figure 16.3** shows the result: the larger text reflows without being cut off, and the navy focus ring is clearly visible around the skip link.

Listing 16.3 `shoot.py`: Playwright screenshots of our own running kiosk, in both languages.

```
"""Screenshots of our own running kiosk (start kiosk_app.py first)."""
import sys

from playwright.sync_api import sync_playwright

BASE = "http://127.0.0.1:4160"


def main(out_dir):
    with sync_playwright() as p:
        browser = p.chromium.launch()
        page = browser.new_page(viewport={"width": 720, "height": 900})
        for lang in ("en", "id"):
            page.goto(BASE + "/?lang=" + lang)
            page.screenshot(path="%s/kiosk_home_%s.png" % (out_dir, lang))
            page.get_by_role("button").nth(3).click()     # older adults
            page.screenshot(path="%s/kiosk_number_%s.png" % (out_dir, lang))
            page.goto(BASE + "/?lang=%s&big=1" % lang)
            page.keyboard.press("Tab")                     # the skip link
            page.screenshot(path="%s/kiosk_big_%s.png" % (out_dir, lang))
        browser.close()


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

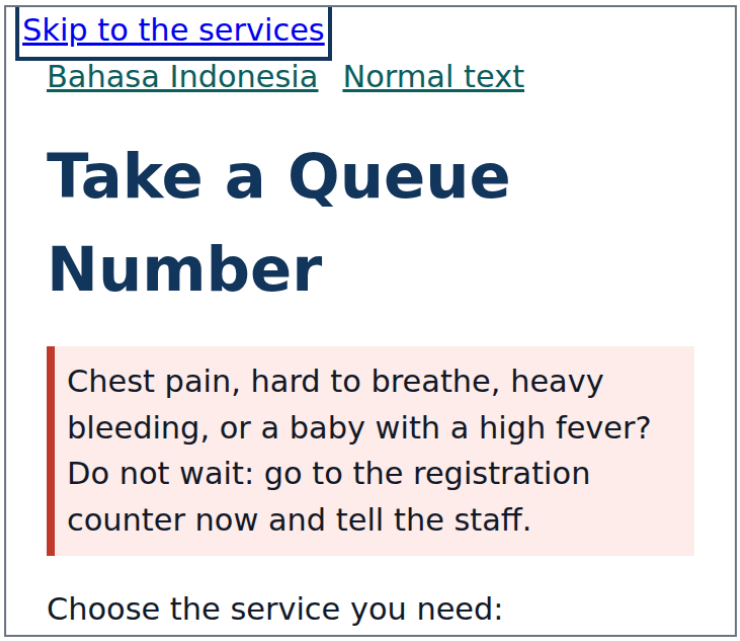


Figure 16.3 The larger-text mode after one press of the Tab key: the skip link appears with a navy focus ring, and the text reflows.

16.7 Evaluating the Kiosk

16.7.1 The pilot test

Chapter 7 describes the usability test; the team ran a small one in the puskesmas meeting room with eight participants: three aged 18–39, two aged 40–59 and three aged 60 or over. Eight is enough to find most of the serious problems, following Nielsen and Landauer's finding that a small number of users uncovers most problems when tests are repeated after each redesign [11], but it is far too few for precise averages, and the analysis must say so. Each participant did the three tasks of **Table 16.3**, thinking aloud, and then answered the ten SUS statements and the two UMUX-Lite statements.

Table 16.3 The three pilot-test tasks and their success criteria.

Task	Instruction read to the participant	Success means
T1	"You are here for the general clinic. Take your number."	A U-number is shown
T2	"Your friend reads English better than Indonesian. Switch the screen to English and take a number for mother and child."	The screen is in English and a K-number is shown
T3	"The letters are too small for you. Make them bigger and take a number for the older-adults clinic."	Larger text is on and an L-number is shown

The results are in three CSV files in `data/`: `sus.csv` with each participant's ten answers from 1 to 5, `umux_lite.csv` with the two UMUX-Lite answers from 1 to 7, and `tasks.csv` with success and seconds per task. They are a practice data set with the shape of a real pilot, so that the analysis can be learned before real participants are recruited.

16.7.2 The measures

The System Usability Scale has ten statements, alternately positive (odd items) and negative (even items), each answered from 1 (strongly disagree) to 5 (strongly agree) [12]. Each odd item

contributes its answer minus 1, each even item 5 minus its answer, and the sum is multiplied by 2.5:

$$\text{SUS} = 2.5\left[\sum_{i\in\{1,3,5,7,9\}}(a_i - 1) + \sum_{i\in\{2,4,6,8,10\}}(5 - a_i)\right] \tag{16.3}$$

which gives a score from 0 to 100. It is not a percentage. Across many studies the average SUS score is about 68, so scores above 68 are better than average [13].

UMUX-Lite asks only two statements, “This system’s capabilities meet my requirements” and “This system is easy to use”, each from 1 to 7 [14]. The raw score, and the regression-adjusted score that Lewis, Utesch and Maher proposed to make it comparable with SUS, are

$$\begin{aligned}\text{UMUX-Lite} &= \frac{(u_1 - 1) + (u_2 - 1)}{12}\cdot 100, \qquad \text{SUS-comparable} \\ &= 0.65\cdot\text{UMUX-Lite} + 22.9\end{aligned} \tag{16.4}$$

For task completion with few participants, the usual interval formula fails at 8 out of 8 (it would claim certainty). The adjusted Wald interval, recommended by Sauro and Lewis for small samples [13], adds $z^2/2$ successes and z^2 trials before computing the interval, with $z = 1.96$ for 95%:

$$\tilde{p} = \frac{x + z^2/2}{n + z^2}, \qquad \tilde{p} \pm z\sqrt{\frac{\tilde{p}(1-\tilde{p})}{n + z^2}} \tag{16.5}$$

Task times are skewed, a few slow participants pulling the arithmetic mean up, so the geometric mean of the successful times is reported instead.

Finally, the color contrast of WCAG 2.2 is computed from the *relative luminance* L of each color, in which each sRGB channel c from 0 to 1 is first linearized, $c_{\text{lin}} = c/12.92$ for $c \le 0.04045$ and $((c + 0.055)/1.055)^{2.4}$ otherwise, and then weighted [5]:

$$L = 0.2126\,R_{\text{lin}} + 0.7152\,G_{\text{lin}} + 0.0722\,B_{\text{lin}}, \qquad \text{ratio} = \frac{L_1 + 0.05}{L_2 + 0.05} \tag{16.6}$$

with L_1 the lighter and L_2 the darker color. Text needs at least 4.5:1, and focus indicators and other non-text elements at least 3:1.

16.7.3 The evaluation program

Listing 16.4 computes all of these: SUS per participant with Equation (16.3) and its mean with a t-based 95% interval, UMUX-Lite with Equation (16.4), completion per task with the interval of Equation (16.5) and the geometric mean time, and the contrast of every color pair of the kiosk’s style sheet with Equation (16.6), including the draft amber focus ring and the navy one that replaced it.

Listing 16.4 `evaluate.py`: SUS, UMUX-Lite, task completion with adjusted Wald intervals, and WCAG color contrast.

```
"""Evaluate the kiosk pilot test: SUS, UMUX-Lite, tasks and color contrast."""
import csv
import math
import sys

T95 = [12.706, 4.303, 3.182, 2.776, 2.571, 2.447, 2.365, 2.306, 2.262,
       2.228, 2.201, 2.179]              # t for 95%, df = 1 .. 12


def rows(path):
    with open(path, encoding="utf-8") as f:
        return list(csv.DictReader(f))


def sus(answers):
    """System Usability Scale: 10 answers 1..5 -> 0..100."""
    odd = sum(answers[i] - 1 for i in range(0, 10, 2))
    even = sum(5 - answers[i] for i in range(1, 10, 2))
    return 2.5 * (odd + even)


def mean_ci(xs):
    n = len(xs)
    m = sum(xs) / n
    sd = math.sqrt(sum((x - m) ** 2 for x in xs) / (n - 1))
    t = T95[n - 2] if n - 1 <= len(T95) else 1.96
    return m, sd, m - t * sd / math.sqrt(n), m + t * sd / math.sqrt(n)


def adjusted_wald(successes, n, z=1.96):
    """95% interval for a completion rate, good even for small n."""
    p = (successes + z * z / 2) / (n + z * z)
    half = z * math.sqrt(p * (1 - p) / (n + z * z))
    return max(0.0, p - half), min(1.0, p + half)


def luminance(hex_color):
    """WCAG relative luminance of a color such as '#12355B'."""
    def channel(c):
        c = int(hex_color[c:c + 2], 16) / 255
        return c / 12.92 if c <= 0.04045 else ((c + 0.055) / 1.055) ** 2.4
    return 0.2126 * channel(1) + 0.7152 * channel(3) + 0.0722 * channel(5)
```

Listing 16.4 (Continued) *evaluate.py: SUS, UMUX-Lite, task completion with adjusted Wald intervals, and WCAG color contrast.*

```
def contrast(a, b):
    hi, lo = sorted((luminance(a), luminance(b)), reverse=True)
    return (hi + 0.05) / (lo + 0.05)


PAIRS = [("button text", "#FFFFFF", "#12355B", 4.5),
         ("body text", "#111827", "#FFFFFF", 4.5),
         ("tool links", "#0B5E5E", "#FFFFFF", 4.5),
         ("urgent note", "#111827", "#FDECEA", 4.5),
         ("focus, draft", "#E0A100", "#FFFFFF", 3.0),
         ("focus, fixed", "#12355B", "#FFFFFF", 3.0)]


def main(data):
    print("SUS per participant")
    scores = {}
    for r in rows(data + "/sus.csv"):
        s = sus([int(r["q%d" % i]) for i in range(1, 11)])
        scores[r["participant"]] = s
        print("  %s  %-6s %5.1f" % (r["participant"], r["age_group"], s))
    m, sd, lo, hi = mean_ci(list(scores.values()))
    print("SUS mean %.1f (SD %.1f), 95%% CI [%.1f, %.1f]" % (m, sd, lo, hi))
    lite = [(int(r["capabilities"]) - 1 + int(r["easy"]) - 1) / 12 * 100
            for r in rows(data + "/umux_lite.csv")]
    m2 = sum(lite) / len(lite)
    print("UMUX-Lite mean %.1f, SUS-comparable %.1f" % (m2, 0.65 * m2 + 22.9))
    print("\ntask  done  completion  95% CI (adj. Wald)  geo. mean time")
    tasks = {}
    for r in rows(data + "/tasks.csv"):
        tasks.setdefault(r["task"], []).append(
            (r["success"] == "1", int(r["seconds"])))
    for task, results in sorted(tasks.items()):
        ok = [sec for success, sec in results if success]
        lo, hi = adjusted_wald(len(ok), len(results))
        geo = math.exp(sum(math.log(s) for s in ok) / len(ok))
        print("%-4s  %d/%d  %9.0f%%  [%3.0f%%, %3.0f%%]  %14.1f s" % (
            task, len(ok), len(results), 100 * len(ok) / len(results),
            100 * lo, 100 * hi, geo))
    print("\ncolor pair      ratio  needed  verdict")
    for name, fg, bg, need in PAIRS:
        ratio = contrast(fg, bg)
        print("%-14s %6.2f  %6.1f  %s" % (name, ratio, need,
                                          "pass" if ratio >= need else "FAIL"))
```

Listing 16.4 (Continued) *evaluate.py: SUS, UMUX-Lite, task completion with adjusted Wald intervals, and WCAG color contrast.*

```
if __name__ == "__main__":
    main(sys.argv[1] if len(sys.argv) > 1 else "../data")
```

Listing 16.5 is the same evaluation in Java.

Listing 16.5 `Evaluate.java`: the same evaluation in Java.

```java
import java.nio.file.*;
import java.util.*;

/** Evaluate the kiosk pilot test: SUS, UMUX-Lite, tasks and color contrast. */
public class Evaluate {
    static final double[] T95 = {12.706, 4.303, 3.182, 2.776, 2.571, 2.447,
        2.365, 2.306, 2.262, 2.228, 2.201, 2.179};   // t for 95%, df 1..12

    static List<String[]> rows(String path) throws Exception {
        List<String> lines = Files.readAllLines(Path.of(path));
        List<String[]> out = new ArrayList<>();
        for (String l : lines.subList(1, lines.size())) out.add(l.split(","));
        return out;
    }

    /** System Usability Scale: 10 answers 1..5 -> 0..100. */
    static double sus(int[] a) {
        int sum = 0;
        for (int i = 0; i < 10; i += 2) sum += a[i] - 1;
        for (int i = 1; i < 10; i += 2) sum += 5 - a[i];
        return 2.5 * sum;
    }

    static double[] meanCi(List<Double> xs) {
        int n = xs.size();
        double m = 0, ss = 0;
        for (double x : xs) m += x;
        m /= n;
        for (double x : xs) ss += (x - m) * (x - m);
        double sd = Math.sqrt(ss / (n - 1));
        double t = n - 1 <= T95.length ? T95[n - 2] : 1.96;
        return new double[] {m, sd, m - t * sd / Math.sqrt(n),
            m + t * sd / Math.sqrt(n)};
    }

    /** 95% interval for a completion rate, good even for small n. */
    static double[] adjustedWald(int successes, int n) {
        double z = 1.96, p = (successes + z * z / 2) / (n + z * z);
        double half = z * Math.sqrt(p * (1 - p) / (n + z * z));
        return new double[] {Math.max(0, p - half), Math.min(1, p + half)};
    }
```

Listing 16.5 (Continued) *Evaluate.java: the same evaluation in Java.*

```
    /** WCAG relative luminance of a color such as "#12355B". */
    static double luminance(String hex) {
        double[] w = {0.2126, 0.7152, 0.0722};
        double y = 0;
        for (int k = 0; k < 3; k++) {
            double c = Integer.parseInt(hex.substring(1 + 2 * k, 3 + 2 * k), 16)
                / 255.0;
            c = c <= 0.04045 ? c / 12.92 : Math.pow((c + 0.055) / 1.055, 2.4);
            y += w[k] * c;
        }
        return y;
    }

    static double contrast(String a, String b) {
        double la = luminance(a), lb = luminance(b);
        return (Math.max(la, lb) + 0.05) / (Math.min(la, lb) + 0.05);
    }

    static final String[][] PAIRS = {
        {"button text", "#FFFFFF", "#12355B", "4.5"},
        {"body text", "#111827", "#FFFFFF", "4.5"},
        {"tool links", "#0B5E5E", "#FFFFFF", "4.5"},
        {"urgent note", "#111827", "#FDECEA", "4.5"},
        {"focus, draft", "#E0A100", "#FFFFFF", "3.0"},
        {"focus, fixed", "#12355B", "#FFFFFF", "3.0"}};

    public static void main(String[] args) throws Exception {
        String data = args.length > 0 ? args[0] : "../data";
        System.out.println("SUS per participant");
        List<Double> scores = new ArrayList<>();
        for (String[] r : rows(data + "/sus.csv")) {
            int[] a = new int[10];
            for (int i = 0; i < 10; i++) a[i] = Integer.parseInt(r[i + 2]);
            double s = sus(a);
            scores.add(s);
            System.out.printf("  %s  %-6s %5.1f%n", r[0], r[1], s);
        }
        double[] c = meanCi(scores);
        System.out.printf("SUS mean %.1f (SD %.1f), 95%% CI [%.1f, %.1f]%n",
            c[0], c[1], c[2], c[3]);
        double sum = 0;
        List<String[]> lite = rows(data + "/umux_lite.csv");
        for (String[] r : lite)
            sum += (Integer.parseInt(r[1]) - 1 + Integer.parseInt(r[2]) - 1)
                / 12.0 * 100;
        double m2 = sum / lite.size();
```

Listing 16.5 (Continued) *Evaluate.java: the same evaluation in Java.*

```
        System.out.printf("UMUX-Lite mean %.1f, SUS-comparable %.1f%n",
            m2, 0.65 * m2 + 22.9);
        System.out.println(
            "\ntask  done  completion  95% CI (adj. Wald)  geo. mean time");
        Map<String, List<int[]>> tasks = new TreeMap<>();
        for (String[] r : rows(data + "/tasks.csv"))
            tasks.computeIfAbsent(r[1], k -> new ArrayList<>()).add(
                new int[] {Integer.parseInt(r[2]), Integer.parseInt(r[3])});
        for (var e : tasks.entrySet()) {
            int ok = 0, n = e.getValue().size();
            double logs = 0;
            for (int[] t : e.getValue())
                if (t[0] == 1) { ok++; logs += Math.log(t[1]); }
            double[] ci = adjustedWald(ok, n);
            System.out.printf(
                "%-4s  %d/%d  %9.0f%%  [%3.0f%%, %3.0f%%]  %14.1f s%n",
                e.getKey(), ok, n, 100.0 * ok / n, 100 * ci[0], 100 * ci[1],
                Math.exp(logs / ok));
        }
        System.out.println("\ncolor pair      ratio  needed  verdict");
        for (String[] p : PAIRS) {
            double ratio = contrast(p[1], p[2]), need = Double.parseDouble(p[3]);
            System.out.printf("%-14s %6.2f  %6.1f  %s%n", p[0], ratio, need,
                ratio >= need ? "pass" : "FAIL");
        }
    }
}
```

16.7.4 The results

Both programs print:

```
SUS per participant
  P1  18-39   92.5
  P2  18-39   87.5
  P3  18-39   75.0
  P4  40-59   72.5
  P5  40-59   87.5
  P6  60+     50.0
  P7  60+     67.5
  P8  60+     47.5
SUS mean 72.5 (SD 17.0), 95% CI [58.3, 86.7]
UMUX-Lite mean 79.2, SUS-comparable 74.4

task  done  completion  95% CI (adj. Wald)  geo. mean time
T1    8/8        100%  [ 63%, 100%]            12.7 s
T2    6/8         75%  [ 40%,  94%]            26.0 s
T3    7/8         88%  [ 51%, 100%]            22.8 s

color pair      ratio  needed  verdict
button text     12.46     4.5  pass
body text       17.74     4.5  pass
tool links       7.56     4.5  pass
urgent note     15.51     4.5  pass
focus, draft     2.27     3.0  FAIL
focus, fixed    12.46     3.0  pass
```

Figure 16.4 draws the SUS scores by participant and age group. The mean, 72.5, is above the average of 68, and UMUX-Lite agrees with a SUS-comparable 74.4. But the 95% interval runs from 58.3 to 86.7: with eight participants the data cannot tell whether the kiosk is ordinary or excellent. The pattern inside the data is clearer than the mean. The three participants aged 60 or

over scored 50.0, 67.5 and 47.5; everyone younger scored 72.5 or more. For the users the head of the puskesmas cared most about, the kiosk is not yet good enough.

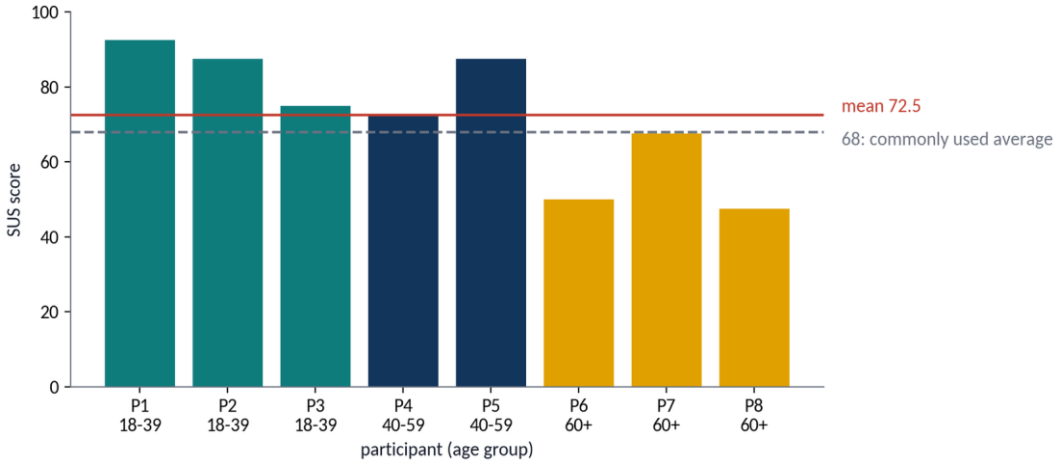


Figure 16.4 SUS scores in the pilot test, colored by age group, with the mean (72.5) and the commonly used average of 68; all three participants aged 60 or over scored below the average.

The tasks show where. Everyone completed T1, taking a number, in a geometric mean of 12.7 seconds. T2, switching language, failed for two of the eight, both older, and took twice as long; in the think-aloud notes both said they "did not see" the language link, a small underlined text at the top. T3, larger text, failed for one older participant who found the link but then did not notice that the page had changed. The contrast check found one failure, the amber focus ring at 2.27:1 against the required 3:1, which had already been fixed in the code shown above. The pattern is the lesson of Chapter 2 and Chapter 11 at once: controls that are small and far from the main action are invisible to people whose eyes and attention are stretched.

16.8 Iterate: From Findings to Changes

The findings become a prioritized list. **Table 16.4** rates each one on Nielsen's severity scale, from 0 (not a problem) to 4 (usability catastrophe) [15], combining how many users it affects, how badly, and whether they can work around it.

Table 16.4 Pilot-test findings with severity ratings and the changes planned for the second iteration.

Finding	Evidence	Severity	Change for iteration 2
Language link not seen by older users	T2: 2 of 8 failed, both 60+; think-aloud	3	Two large buttons "Bahasa Indonesia" and "English" above the services, same size as service buttons
Larger-text change not noticed	T3: 1 failure; "nothing happened"	2	Offer larger text by default to anyone who touches it once; add a short confirmation line
Older adults' SUS below average	50.0, 67.5, 47.5	3	Addressed by the two changes above; retest with at least five older adults
Focus ring contrast 2.27:1	Contrast check	2	Already fixed: navy ring, 12.46:1
Some patients fear touching the kiosk	Research finding 2	2	A volunteer or staff member near the kiosk in the first month; sign "The desk is open as usual"

Two things about this list matter as much as its content. It ties every change to evidence, so that nobody changes the kiosk because of a personal taste. And it ends with a *retest*: the second iteration will be evaluated with the same tasks and at least five older adults, and the success of the changes will be judged by T2 and T3 completion for that group, not by the overall mean.

Chapter 14 would add one more thing once the kiosk is in daily use: the anonymous counts from `/status.json` per hour and per service, compared with the desk's paper slips, show whether the kiosk is actually shortening the line, which is the goal the head of the puskesmas started with.

16.9 Testing the Code

The evaluation arithmetic must be correct before its results are believed. **Listing 16.6** checks the extremes of SUS, the contrast of black on white (exactly 21:1), and that the adjusted Wald interval keeps a lower bound at 8 out of 8.

Listing 16.6 `test_evaluate.py`: tests of SUS scoring, contrast and the adjusted Wald interval.

```
"""Checks of the evaluation arithmetic; run: python -m unittest test_evaluate"""
import unittest

from evaluate import adjusted_wald, contrast, sus


class EvaluateTest(unittest.TestCase):
    def test_sus(self):
        self.assertEqual(sus([5, 1] * 5), 100)
        self.assertEqual(sus([3] * 10), 50)

    def test_contrast(self):
        self.assertAlmostEqual(contrast("#000000", "#FFFFFF"), 21)

    def test_adjusted_wald(self):
        lo, hi = adjusted_wald(8, 8)
        self.assertEqual(hi, 1.0)
        self.assertGreater(lo, 0.6)


if __name__ == "__main__":
    unittest.main()
```

Listing 16.7 is the Java version.

Listing 16.7 `EvaluateTest.java`: the same checks in Java.

```
/** Checks of the evaluation arithmetic; run: java EvaluateTest */
public class EvaluateTest {
    static int failed = 0;

    static void check(boolean ok, String what) {
        System.out.println((ok ? "ok   " : "FAIL ") + what);
        if (!ok) failed++;
    }

    public static void main(String[] args) {
        check(Evaluate.sus(new int[] {5, 1, 5, 1, 5, 1, 5, 1, 5, 1}) == 100,
            "best answers give SUS 100");
        check(Evaluate.sus(new int[] {3, 3, 3, 3, 3, 3, 3, 3, 3, 3}) == 50,
            "neutral answers give SUS 50");
        check(Math.abs(Evaluate.contrast("#000000", "#FFFFFF") - 21) < 1e-9,
            "black on white is 21:1");
        double[] ci = Evaluate.adjustedWald(8, 8);
        check(ci[1] == 1.0 && ci[0] > 0.6, "8 of 8 still has a lower bound");
        System.out.println(failed == 0 ? "all checks passed" : failed + " failed");
        System.exit(failed == 0 ? 0 : 1);
    }
}
```

`python -m unittest test_evaluate` reports three passing tests; the Java version prints:

```
ok   best answers give SUS 100
ok   neutral answers give SUS 50
ok   black on white is 21:1
ok   8 of 8 still has a lower bound
all checks passed
```

The kiosk servers were tested in a different way: by requesting the same pages from both and comparing the HTML byte for byte, by posting an invalid service code and checking that the answer is a redirect (302 from Flask, 303 from Java), and by the Playwright script, which fails if a button it expects is missing.

16.10 Presenting Design Decisions

A design is only as strong as the case others can check. At the end of the semester the team presented to the head of the puskesmas, Mbak Ratna and the district health office. Their report followed a structure that works for any capstone:

1. **The problem and the people**, in one page: the line at the desk, the four findings, the two personas.
2. **The requirements**, each traced to a finding and a chapter, as in Section 16.4.
3. **The design**, shown with the running kiosk and its screenshots, each decision tied to a requirement.
4. **The evidence**: the pilot test, with its size stated honestly, the results of Section 16.7.4 including the ones that were disappointing, and the contrast check.
5. **What changes next**, the prioritized list of **Table 16.4**, and how the next test will judge it.
6. **What was deliberately not done**, such as AI triage, waiting-time estimates and personal data, with the reason for each.

The sixth point often persuades a sceptical audience more than any other. It shows that the team considered the fashionable options and rejected them for reasons tied to the users, which is the difference between a design and a demonstration.

16.11 Using It for Real

To put the kiosk into a real puskesmas, the team listed what separates a prototype from a service. The tablet runs a browser in *kiosk mode*, which hides the address bar and prevents leaving the page, and it is mounted at a height usable from a wheelchair. A small thermal printer is added so that the number can be taken away on paper, because a number on a screen disappears when the next patient touches it; the printer is driven by the server when `/take` issues a number. The queue counts are stored in a small database and reset each morning, and the desk's application calls a `/call` endpoint to mark numbers as served (Exercise 6). The status endpoint feeds the waiting-room television, whose number is sized with the method of Chapter 15. Finally, the Flask development server is replaced by a production server, as Flask itself warns, and the kiosk is restarted automatically if it stops.

The same approach serves other places where people take a number: a village office, a bank branch, a post office, a vehicle-registration office. The four services change, the words change, and the tests must be repeated with the people of that place. The process, and the rule that nobody may be worse off than before, do not change.

16.12 HCI in Practice: Profitina

Profitina, the AI Visibility (GEO) platform at app.profitina.com, is built and released in the same cycle as the kiosk, at a larger scale. Its research question is how a brand owner who is not an engineer can understand why AI answer engines do or do not cite the brand's website, and its design answer is a dashboard that explains each measured signal in plain words rather than a single opaque number. Its requirements include UU PDP 27/2022 compliance that users can operate themselves: explicit consent, data export, email and password change and a complete account purge, all in the product, not by email to support. Its AI runs locally, and when no model is available the app says so, which is the honesty principle of Chapter 13 and of this chapter's refusal to show an estimated waiting time.

Its evaluation is continuous. The full automated suite of well over a thousand tests, including JavaScript guards for the interface, must pass before any release, and a staged release reports success only after the deployed version answers its health check. User feedback flows through a transparent pipeline (submit, curate, dataset, train and measure, release) whose status each user can see in the dashboard. The cycle of **Figure 16.1**, research, define, develop, deliver and back again, is the same for a puskesmas kiosk built by three students and for a product used by brands across Indonesia.

16.13 Common Mistakes, and How to Avoid Them

- **Starting with the solution.** "We will build a kiosk" came from the head of the puskesmas; the research could have shown that a second desk at peak hours was better. Keep the goal (a shorter line, nobody worse off) separate from the proposed solution until the research is done.
- **Designing for the average user.** The mean SUS of 72.5 looked fine and hid the older adults' scores. Report results by the groups that matter, as **Figure 16.4** does.
- **Treating a small test as precise.** Eight participants find problems; they do not measure averages precisely. Always report the interval, such as [58.3, 86.7] for SUS.
- **Checking accessibility at the end.** The focus-ring failure was cheap to fix because it was found early. Put WCAG criteria into the requirements (**Table 16.2**) and check them with code.
- **Adding technology because it is available.** AI triage or face recognition would have added risk and data without helping any persona. Write down what you decided not to do, and why.
- **Collecting data you do not need.** A queue number needs no name. Every field you do not collect is a consent screen you do not need and a breach that cannot happen.
- **Stopping after one iteration.** A test that finds problems is a success only if the problems are fixed and retested. Plan the second iteration before presenting the first.
- **Presenting only the good news.** Decision makers trust teams that show the disappointing results and what they will do about them.

16.14 Chapter Summary

- A complete interactive system includes the service around the screen: the place, the people who do not use it, the staff and the rules. ISO 9241-210 describes its design as a cycle of understanding the context, specifying requirements, designing and evaluating [1]; the Double Diamond gives the same path four phases (**Figure 16.1**) [2].
- Every chapter of this book contributed a decision to the queue kiosk (**Table 16.1**), including the decision not to use AI where a guess could harm.

- Research at the desk produced four findings, two personas and a scenario; requirements were written as testable statements traced to findings.
- Hick's law (Equation (16.1)) kept the choice to four services, and Fitts's law (Equation (16.2)) justified large, full-width buttons [3], [4]; WCAG 2.2 success criteria were requirements from the start (**Table 16.2**) [5], as were Indonesian accessibility law [6], [7] and data minimization [9].
- The kiosk was built as a Flask application and as a JDK HTTP server serving identical HTML, with two languages, a larger-text mode, a skip link, status messages and an urgent-help handoff, and captured with Playwright (**Figure 16.2**, **Figure 16.3**).
- The pilot test with eight participants measured SUS (Equation (16.3)), UMUX-Lite (Equation (16.4)), task completion with adjusted Wald intervals (Equation (16.5)) and color contrast (Equation (16.6)): SUS 72.5 with a wide interval [58.3, 86.7], all older adults below 68, language switching the weakest task, and one contrast failure already fixed.
- The findings became a prioritized list with severities (**Table 16.4**) and a planned retest with older adults; the presentation showed the evidence, the disappointments and the options deliberately rejected.

16.15 Exercises

1. Compute the SUS score of participant P6 by hand with Equation (16.3), from the answers 3, 3, 3, 3, 3, 2, 3, 3, 3, 4. Then say which two items pulled the score down most.
2. Compute the adjusted Wald interval for T2 (6 successes out of 8) by hand with Equation (16.5), and compare it with the naive interval $\hat{p} \pm 1.96\sqrt{\hat{p}(1-\hat{p})/n}$ with $\hat{p} = 0.75$. Why is the adjusted interval preferred here?
3. Compute the contrast ratio of the tool-link color `#0B5E5E` on white with Equation (16.6), showing the linearized channels, and confirm the 7.56 printed by `evaluate.py`.
4. A manager asks for eight services on the first screen instead of four. Use Equation (16.1) and Equation (16.2) to explain the cost, and propose a design that serves the extra purposes without hurting the main task.
5. Plan the second iteration's test: participants, tasks, measures and the criterion that will decide whether the new language buttons work. How many older adults would you recruit, and why?
6. Add an endpoint `POST /call` with a form field `service` that marks the next waiting number of that service as served, in both `kiosk_app.py` and `KioskServer.java`. What must it do when nobody is waiting, and how does "People ahead of you" change?

16.16 Solutions to the Exercises

1. Odd items 1, 3, 5, 7, 9 are all 3, contributing $5 \times (3-1) = 10$. Even items 2, 4, 6, 8, 10 are 3, 3, 2, 3, 4, contributing $(5-3)+(5-3)+(5-2)+(5-3)+(5-4) = 2+2+3+2+1 = 10$. SUS $= 2.5 \cdot 20 = 50$, as `evaluate.py` printed. The two items that cost most are item 10 ("I needed to learn a lot of things before I could get going with this system"), answered 4, and the uniformly neutral positive items: nothing was clearly easy for this participant, which matches the failed T2.

2. $z^2 = 3.8416$. $\hat{p} = (6 + 1.9208)/(8 + 3.8416) = 7.9208/11.8416 = 0.6689$. Half width $= 1.96\sqrt{0.6689 \cdot 0.3311/11.8416} = 1.96 \cdot 0.1368 = 0.268$, so the interval is $[0.401, 0.937]$, printed as [40%, 94%]. The naive interval is $0.75 \pm 1.96\sqrt{0.75 \cdot 0.25/8} = 0.75 \pm 0.300$, that is $[0.45, 1.05]$, which runs past 100% and is too narrow in its true coverage for small samples. The

adjusted interval stays inside 0 to 1 and has close to the promised 95% coverage even with eight participants [13].

3. Red: $0x0B = 11$, $c = 0.0431 > 0.04045$, so $c_{\text{lin}} = ((0.0431 + 0.055)/1.055)^{2.4} = 0.0930^{2.4} \approx 0.00335$. Green and blue: $0x5E = 94$, $c = 0.3686$, $c_{\text{lin}} = (0.4015)^{2.4} \approx 0.1119$. $L = 0.2126 \cdot 0.00335 + 0.7152 \cdot 0.1119 + 0.0722 \cdot 0.1119 = 0.0007 + 0.0800 + 0.0081 = 0.0888$. White has $L = 1$. Ratio $= (1 + 0.05)/(0.0888 + 0.05) = 1.05/0.1388 = 7.56$, well above 4.5.

4. Hick: $\log_2 9 = 3.17$ bits instead of $\log_2 5 = 2.32$, about 37% more information to process for *every* patient, most of whom need one of the four clinics. Fitts: eight buttons on the same screen must be about half as tall, 32 pixels, raising the index of difficulty from 2.51 to 3.37 bits for the farthest button and falling toward the 24-pixel minimum of WCAG 2.2. A better design keeps the four clinics as the first screen and adds one smaller, clearly labeled button “Other services and questions” at the bottom, which issues a desk number; the extra purposes are served, and the main task is untouched.

5. Participants: at least five aged 60 or over, since that is the group whose problems the changes target, plus three younger participants to check that nothing got worse for them; at least two older participants who come alone. Tasks: the same T1–T3, so the results can be compared. Measures: completion and time per task by age group, SUS and UMUX-Lite, and think-aloud notes. Criterion: the new language buttons work if at least four of five older participants complete T2 without help and none says they did not see the language choice. With five participants the interval will still be wide, so the criterion is about problems found, not about a precise rate.

6. In Python, add a route `@app.post("/call")` that reads `code = request.form.get("service", "")`, checks `code in queues`, and increments `served[code]` only if `served[code] < queues[code]`; it returns the status JSON so that the desk sees the new counts. In Java, add a branch for the path `/call` with method POST that does the same with `served.merge(code, 1, Integer::sum)` after the same check. When nobody is waiting, the endpoint must do nothing and say so (for example `{"called": null}`), never let `served` exceed `given`. On the `/take` page, “People ahead of you” is computed as `queues[code] - served[code] - 1`, so every call reduces it by one for everyone who takes a number afterwards; a patient who already holds a number sees the change only on the waiting-room television, which reads `/status.json`.

References

[1] ISO 9241-210:2019, *Ergonomics of human-system interaction — Part 210: Human-centred design for interactive systems*. Geneva, Switzerland: International Organization for Standardization, 2019.

[2] Design Council, “Framework for Innovation: Design Council’s evolved Double Diamond,” 2019. [Online]. Available: https://www.designcouncil.org.uk/our-resources/framework-for-innovation/

[3] W. E. Hick, “On the rate of gain of information,” *Quarterly Journal of Experimental Psychology*, vol. 4, no. 1, pp. 11–26, 1952.

[4] I. S. MacKenzie, “Fitts’ law as a research and design tool in human-computer interaction,” *Human-Computer Interaction*, vol. 7, no. 1, pp. 91–139, 1992.

[5] W3C, “Web Content Accessibility Guidelines (WCAG) 2.2,” W3C Recommendation, Oct. 5, 2023; approved as ISO/IEC 40500:2025, see W3C WAI, “WCAG 2.2 approved as an ISO standard,” Oct. 21, 2025. [Online]. Available: https://www.w3.org/TR/WCAG22/ and https://www.w3.org/WAI/news/2025-10-21/wcag22-iso

[6] Republik Indonesia, *Undang-Undang Nomor 8 Tahun 2016 tentang Penyandang Disabilitas*. Jakarta, 2016.

[7] Republik Indonesia, *Peraturan Pemerintah Nomor 42 Tahun 2020 tentang Aksesibilitas terhadap Permukiman, Pelayanan Publik, dan Pelindungan dari Bencana bagi Penyandang Disabilitas*. Jakarta, 2020.

[8] European Union, "Directive (EU) 2019/882 on the accessibility requirements for products and services (European Accessibility Act)," *Official Journal of the European Union*, L 151, Jun. 7, 2019; applicable from Jun. 28, 2025.

[9] Bisnis.com, "Aturan Turunan UU PDP: Persetujuan Data Konsumen Lebih Ketat," Aug. 29, 2026, on Peraturan Pemerintah No. 33 Tahun 2026 implementing UU No. 27 Tahun 2022. [Online]. Available: https://teknologi.bisnis.com/read/20260829/84/1999753/aturan-turunan-uu-pdp-persetujuan-data-konsumen-lebih-ketat

[10] Microsoft, "Playwright for Python," documentation. [Online]. Available: https://playwright.dev/python/ (accessed Sep. 2026).

[11] J. Nielsen and T. K. Landauer, "A mathematical model of the finding of usability problems," in *Proc. INTERCHI '93*, Amsterdam, The Netherlands, 1993, pp. 206–213.

[12] J. Brooke, "SUS: A 'quick and dirty' usability scale," in *Usability Evaluation in Industry*, P. W. Jordan *et al.*, Eds. London, UK: Taylor & Francis, 1996, pp. 189–194.

[13] J. Sauro and J. R. Lewis, *Quantifying the User Experience: Practical Statistics for User Research*, 2nd ed. Cambridge, MA, USA: Morgan Kaufmann, 2016.

[14] J. R. Lewis, B. S. Utesch, and D. E. Maher, "UMUX-LITE: When there's no time for the SUS," in *Proc. CHI '13*, Paris, France, 2013, pp. 2099–2102.

[15] J. Nielsen, "Severity ratings for usability problems," Nielsen Norman Group, Nov. 1, 1994. [Online]. Available: https://www.nngroup.com/articles/how-to-rate-the-severity-of-usability-problems/