Object Oriented Development

and Agile Practices



Designing, Testing and Evolving Software with AI Assistance

by Polla Fattah
and Collection of LLMs

TEXTBOOK AND ENGINEERING MONOGRAPH

Object Oriented Development and Agile Practices

Designing, Testing and Evolving Software with AI Assistance

Dr. Polla Abdulhamid Fattah

with AI Collaboration and Practical Case Studies

FIRST EDITION - ACADEMIC AND PRODUCTION REFERENCE - 2026

Object Oriented Development and Agile Practices

Designing, Testing and Evolving Software with AI Assistance

Author: Dr. Polla Abdulhamid Fattah

Course Context: SE506: Object Oriented Development and Agile Practices (MSc in AI / Software Engineering)

Published By: Academic and Professional Engineering Monograph Series

Publication Date: 2026

Repository and Continuous Updates: github.com/polla-fattah/ood-agile-book

Online Interactive Platform: polla.dev/ood-agile-book/

> **About This Work:**
> This textbook guides postgraduate and advanced students from writing classes to designing, testing, reviewing, and changing maintainable software in a collaborative team workflow. Its central teaching unit is a software change: understand requested behavior, choose an object-oriented design, protect it with automated tests, inspect results, and use feedback to plan subsequent iterations.

Copyright © 2026 Dr. Polla Abdulhamid Fattah.

This work is open access and licensed under the **Creative Commons Attribution 4.0 International License (CC BY 4.0)**. You are free to share (copy and redistribute the material in any medium or format) and adapt (remix, transform, and build upon the material) for any purpose, including commercially, provided appropriate credit is given to the author.

Object Oriented Development and Agile Practices

Designing, Testing and Evolving Software with AI Assistance

Chapter 1: Developing Software That Can Change

Chapter 2: Object-Oriented Development Beyond Syntax

Chapter 3: Agile Development and the Nature of Change

Chapter 4: From Requirements to an Iteration

Chapter 5: Responsibilities, Extension, and Substitutability

Chapter 6: Interfaces, Dependencies, and Composition

Chapter 7: Testing as a Design Practice

Chapter 8: Creating and Connecting Objects

Chapter 9: Designing Changing Behaviour

Chapter 10: From Classes to Application Structure

Chapter 11: Refactoring and Legacy Code

Chapter 12: Measuring Progress, Flow and Technical Health

Chapter 13: Integration, Delivery and Feedback

Chapter 14: Responding to Change and Handing Over

Chapter 15: AI as a Software Engineering Assistant

Appendix A: Programming Readiness and Notation

Appendix B: Tools and Reproducible Setup

Appendix C: Engineering Templates

Appendix D: Project Versions and Glossary

Appendix E: Critical Reading and Project Evidence

Appendix F: Software Engineering for AI and Data Components

Chapter 1: Developing Software That Can Change

> *Why a program that works can still be hard to change, and how this book teaches you to change software safely.*

OO Development and Agile Practices
Designing, Testing and Evolving Software with AI Assistance

Imagine a university that lends cameras, microphones and other equipment to its students. Your team has written a small program that decides whether a student may request another reservation. It looks at two things, the number of reservations the student already holds and the length of the loan requested, and every example you have tried gives the answer you expected. The program works, and for a little while that feels like the end of the story.

Then the equipment officer walks in with a change:

> Staff should also be able to borrow equipment, but their reservation limits will be different. The existing student limits should stay the same.

It sounds small: another condition, two new numbers, an afternoon's work. But the moment you try to start, questions arrive faster than answers. Where do the current limits live in the code? How will the program know whether a borrower is a student or a member of staff? Which of today's behaviours must not change? And when you are finished, what will convince anyone, including you, that the new rule works?

Those questions are the real subject of this book. Writing code that works is the beginning of software engineering, not the end of it. Over the next fifteen chapters we will build a small lending system, but we will spend just as much effort on how software is understood, tested, reviewed and changed, because change is where software spends most of its life. The aim is to make decisions that another developer can inspect and that your team can revise when the requirements move.

This first chapter is a warm-up, and it works the way the rest of the book will. You will read one small Java class closely enough to say what it does and, just as importantly, what it does not do. You will see why a program that works can still be hard to change, and how object-oriented design, iterative delivery, testing and review each help make a change controlled instead of risky. You will meet an AI assistant's explanation of that class and learn to separate the claims the code supports from the ones it does not. And you will finish with a short readiness check that tells you where to spend your effort in the chapters ahead.

A few practical words before you start. This chapter belongs to Week 1 but adds no week of its own, and it only previews ideas that later chapters teach properly, so do not expect it to be complete on any one topic. The class we study, `ReservationRules`, is a warm-up example, separate from the lending system that we build from Chapter 2 onwards. You need no more than basic familiarity with variables, conditions, methods and classes: no design patterns, no testing framework and no project-management tool. If some Java syntax is unfamiliar, read the explanation that follows each example and use the Java tutorial in the source list [1]; Appendix A has a short refresher and a readiness diagnostic.

Sections 1.1 to 1.8 are the heart of the chapter, so read them carefully. Then do the exercises in Section 1.9: Task A, the paper part of Task B (find one input that tells the original rule from the changed one), Task C and Activity 1A. Together they leave you with five things: a table of predictions, an input that exposes a fault, a short change note with a statement of what would convince you, your analysis of an AI explanation, and a readiness checklist. Everything you need is in the chapter, and you can do all of it on paper by tracing the code. If you have a Java Development Kit installed, you can also run the program and the deliberate fault from Task B, and you can answer the review questions and the transfer challenge. Section 15.1 has a longer introduction to working with AI assistants, but Activity 1A does not depend on it.

1.1 When a working program meets a new request

Let us start with the program itself. The equipment office gives us three rules to enforce:

- A student may have at most three active reservations.
- A requested loan lasts from one to seven days, inclusive.
- A request should be rejected if the supplied active-reservation count is negative.

These rules are invented for this book; they are not university policy. By an *active reservation* we mean one that has been confirmed and has not been cancelled or completed. The method will be handed the count of such reservations **before** the new one is added, and it is the caller's job to count them correctly.

Here is the whole class:

```
public final class ReservationRules {
    public boolean mayRequest(int activeReservations, int requestedDays) {
        return activeReservations ≥ 0
                && activeReservations < 3
                && requestedDays ≥ 1
                && requestedDays ≤ 7;
    }
}
```

The class is just a named home for one calculation. Its method, `mayRequest`, takes two integers and answers `true` or `false`, and the `&&` operator means that all four conditions must hold at once. (`public` lets other code use the class and the method, and `final` stops anyone from building subclasses on top of it. Neither keyword says anything about whether the rule is right.)

The comparison `activeReservations < 3` is the one to read twice. Someone who already holds two reservations may ask for a third; someone who holds three may not ask for a fourth. Writing `≤ 3` would quietly allow a fourth.

Existing active reservations	Requested days	Result	Reason
2	7	`true`	One more reservation would reach the limit, and seven days is permitted
3	7	`false`	The borrower is already at the reservation limit
0	0	`false`	A zero-day loan is outside the stated range
0	8	`false`	Eight days exceeds the duration limit
-1	2	`false`	The supplied count is invalid

Now look at what `true` does and does not mean. The method does not know who the borrower is, has no calendar to consult, cannot tell whether the camera is free and stores nothing. It answers one narrow question: do these two numbers satisfy the rules? Everything else has to be checked by the rest of the application before a reservation is confirmed.

That narrowness is easy to forget, because a method's name suggests more than its body delivers. A developer who reads `true` as "the reservation is confirmed" will build a defect on top of a class that behaves exactly as written.

There is one more simplification, and it is better to say it aloud. Returning `false` for a negative count treats an impossible input like an ordinary refusal, so a bug in the caller could pass for a policy decision. We accept that here to keep the example small. Chapter 5 comes back to contracts, and to what code should do when a caller breaks one.

What the new request really asks for

Suppose the officer clarifies the request:

> Students retain the limit of three active reservations and seven days. Staff may have up to five active reservations and request up to fourteen days.

The tempting edit is to replace `3` with `5` and `7` with `14`. That makes the method more generous to everybody, students included, and it still does not distinguish students from staff: the method is never told which kind of borrower is asking, so it has nothing to decide with.

So the first job is not to edit the code but to ask how the program learns a borrower's category, and what should happen when the category is missing or unknown. Do not quietly decide that every unrecognised borrower gets the staff allowance. That is a requirements decision, and nothing in the original code can settle it for us.

There are several ways to build the change. We could add an explicit category and a small conditional. We could pass in a limit chosen elsewhere. If more variations appear, we might give the policy a class of its own. Each approach asks something different of the callers and offers different places for mistakes to hide.

For now we do not need to pick a design pattern. We need to understand the rule, keep the student behaviour intact and choose something proportionate to the problem. Chapters 5 and 6 return to this situation with sharper design tools.

1.2 What does “it works” actually mean?

Ask whether a program works and you will usually get a quick yes. The better question is: works how, and under which conditions? Until we say what behaviour we expect, “it works” means only that nothing has gone wrong yet.

For the reservation rule that is easy to state: a seven-day student request with two reservations on record should be allowed. For a whole application it is not. We also expect it to know whether the equipment is available, to record reservations and to report failures sensibly, and a successful demonstration of one request answers none of those expectations.

There is a second question that matters just as much: how hard will it be to change this software correctly? **Maintainability** is our ability to understand, diagnose, test and modify software over its useful life. We will look for practical signs of it, not treat it as a label that someone attaches to a diagram.

Picture the seven-day limit appearing in a command-line screen, a reservation service and a report generator. A developer updates two of the three and misses the other. Each line, read alone, is perfectly clear, yet the system cannot be changed consistently, because one decision lives in three places. The trouble is in how the decision is spread across the **codebase**, the body of source code and supporting material that a team maintains together. That includes tests, build instructions and configuration, which decide whether a newcomer can work with the code at all. A beautifully clear method inside a project nobody else can build is only part of a maintainable solution.

Turning “good quality” into questions you can answer

When you judge the reservation application, ask questions like these, and look for the evidence that would answer them:

Concern	Question to investigate	Useful evidence
Functional behaviour	Are the stated reservation rules enforced?	Examples and tests linked to those rules
Modifiability	Can one policy change be made without unrelated edits?	A reviewed change and an explanation of affected responsibilities
Understandability	Can another developer explain where the decision is made?	A code walkthrough supported by accurate names and documentation
Testability	Can the rule be checked with controlled inputs?	Repeatable checks that do not depend on a live university service
Failure handling	What happens when required information is unavailable?	Explicit expectations and checks for the failure path

These concerns pull against each other. Moving a rule into its own component can make it easier to test, but a swarm of tiny components makes the program harder to navigate. So a design choice should always be explained by the problem it solves and the cost it brings.

Not every change is the same kind of change, either. Correcting a wrong limit changes behaviour to meet the requirement. Adding staff borrowing introduces new behaviour. Reorganising the code while the visible behaviour stays exactly the same is **refactoring**, using the distinction Fowler describes [3]. It helps to keep these intentions apart, so that a reviewer knows whether a changed result was deliberate or an accident.

Nobody expects you to make a project ready for every possible future. You are expected to say what it assumes, and to respond thoughtfully when one of those assumptions stops being true.

1.3 How design and delivery fit together

So far we have looked at one class. Software never stays one class, and two kinds of thinking take over: how the code is organised, and how the team works on it over time.

Object-oriented development organises software around objects that have state, behaviour and responsibilities. A **responsibility** is something a part of the system is obliged to do: represent a loan period, decide whether a borrower is eligible, record a confirmed reservation. Putting code inside a class does not make a design good. A class can still mix unrelated decisions, expose information that callers misuse, or depend on details scattered across the application; Chapter 2 examines those problems through encapsulation, cohesion and coupling.

The domain gives us a first example of keeping meanings straight. An equipment item and a reservation are different things: the item is something the university owns, and the reservation is an arrangement about that item for a period. If the team keeps those two ideas apart, it can discuss requirements before it decides which fields and methods it needs.

Iterative delivery is about the other kind of thinking, how a team' s understanding and software grow together. An **iteration** is a repeated period of work followed by feedback, and an **increment** adds usable capability to what already exists. A team might improve the reservation workflow over several iterations and deliver a new capability in each one. The Agile principles tie frequent delivery and adaptation to continued care for technical quality, and they ask teams to reflect and adjust how they work [2]. For us that means feedback should change not only what we build but how we build it.

Here is the kind of thing that happens. In an early demonstration the equipment officer turns out to use "reservation" for a confirmed allocation, while the team has been using it for a request nobody has approved yet. That one conversation can change names, states and user messages. Because it happened in a small working increment, there is a concrete example to talk about and only a little code to revise.

Design and process therefore meet at the change itself. Planning decides which behaviour the team intends to deliver. Design decides where its decisions belong. Tests and review help show whether the code matches the intention, and feedback may revise the intention or reveal that the structure was weak.

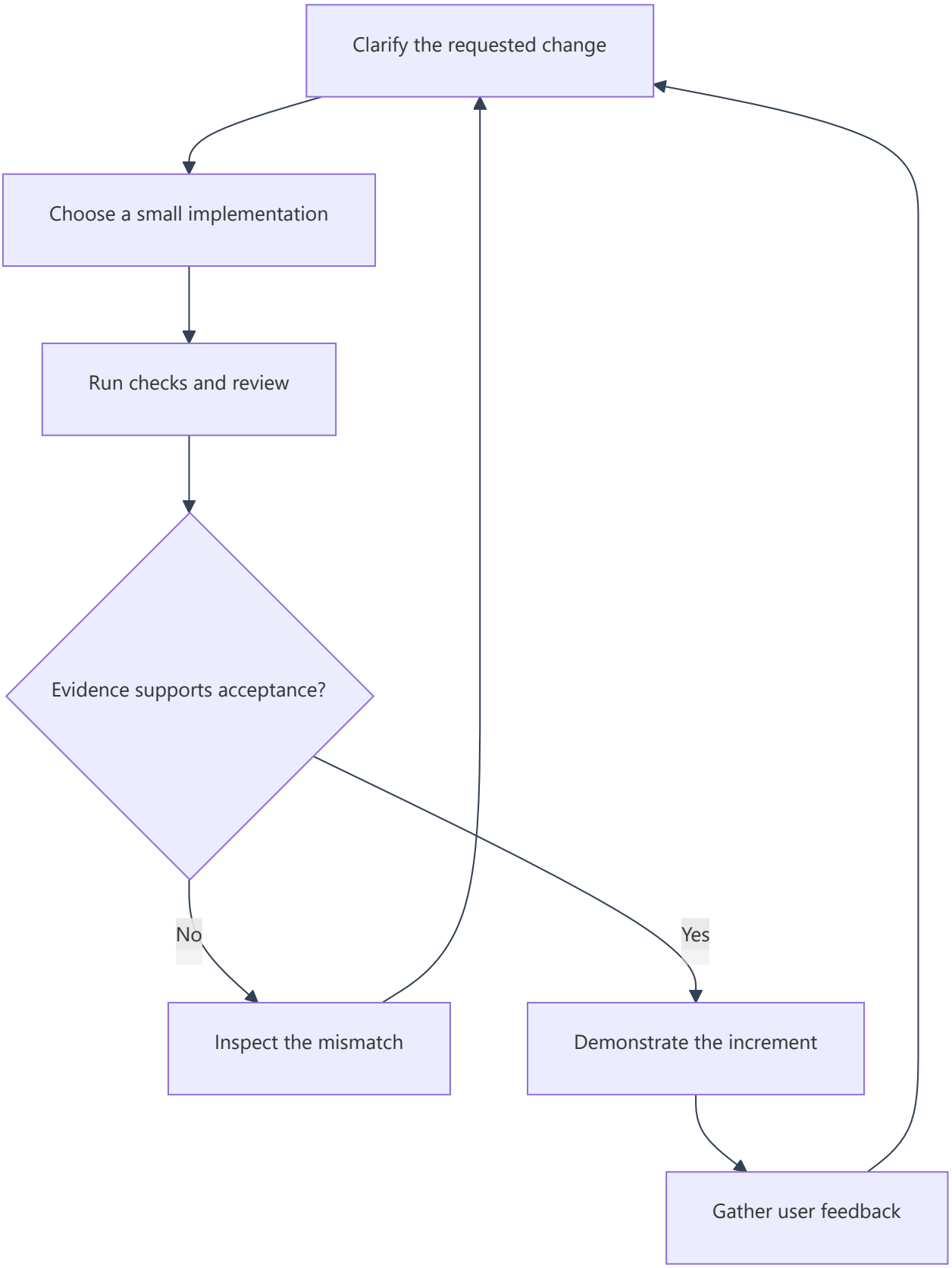


Figure 1.1. A change moves through clarification, implementation and evaluation. A mismatch may reveal a defect, a mistaken expectation or an unclear requirement. User feedback can begin another cycle after the increment is accepted.

Read the figure as a way of thinking, not a script. In practice you may clear up a detail while writing a test, or rethink a design during review. Chapter 3 introduces Agile approaches without asking us to settle those process questions now.

1.4 How do we know a change is right?

Suppose the student and staff rules are clear. What would make us willing to accept the change?

A good place to start is a set of **acceptance examples**: concrete situations with expected outcomes, agreed from the requirement. Before a new reservation is added, a student with two active reservations asking for seven days is eligible, and a student with three is not. A staff member with four reservations asking for fourteen days is eligible, and one with five is not. Writing the examples down has two benefits. It stops us from testing only the staff cases and forgetting that students must behave as before. And it shows what we still do not know, because nothing in these examples settles what to do with a borrower whose category is unknown.

An **automated test** runs a check again and again and compares what the code does with what we expected. A **regression** is an unintended change to behaviour that used to work, and regression checks help catch it, though how well they do depends on what they examine and whether their expectations are right.

Here is a small case. A test that asks for a two-day loan would not notice if someone changed `requestedDays ≤ 7` to `requestedDays < 7` by accident. A test with a seven-day request would. Both versions compile, because the compiler checks the rules of the language and has no idea that the equipment office allows seven days.

What each kind of evidence can and cannot tell you

Evidence	What it can help establish	What it does not establish by itself
Successful compilation	The source satisfies the compiler’s checks	The program implements the requested policy
Passing behavioural tests	The tested cases match their expected results	All relevant cases were identified correctly
Code review	Another person examined decisions, changes and risks	Every defect was discovered
User demonstration	The demonstrated workflow can be inspected against user expectations	Untested failure paths work correctly
Version history	The team can inspect what changed and why	Each recorded change was necessary or correct

A good review connects these kinds of evidence. The reviewer might ask why the staff category is trusted, whether the student boundary is still covered, and whether the explanation matches the actual **diff**, the comparison that shows exactly what changed between two versions of the files. The diff gives the discussion something concrete to point at.

Feedback is only useful when it leads somewhere. “This is bad design” gives the author nothing to investigate. “The fourteen-day limit is repeated in two places, so the next change could leave them inconsistent” names a problem and its consequence.

As the project grows, the evidence grows with it. Chapter 7 develops automated testing and test-driven development, Chapter 11 looks at changing unfamiliar code safely, and Chapter 13 checks how components work together and introduces delivery pipelines. The habit to start building now is simple: for every claim you make about the code, know which check could show that you are wrong.

1.5 Working with an AI assistant without giving up your judgement

An AI coding assistant can explain code, suggest an implementation, propose test cases or review a change. Treat each of those as a candidate to examine, not an answer to accept. An assistant does not decide what the system should do, and a fluent reply is not the same as a correct one.

GitHub’ s documentation for Copilot inline suggestions says as much: a suggestion can look valid and still be incorrect, and users are responsible for reviewing and validating suggestions before accepting them [4]. That is guidance about one tool, not a claim that every model behaves the same way. In this book we apply the same discipline to any candidate output we use.

Activity 1A: evaluate a candidate explanation

	For this activity
Your goal	Decide which statements in an explanation the code actually supports
You start with	The class in Section 1.1 and its three stated rules
What is in bounds	Judge this one method. Do not rewrite it or reason about the rest of the application
How to check	The code itself, a hand trace of two inputs, and the stated rules
How to decide	Accept each claim the evidence supports, reject each it contradicts, and mark the rest as undetermined
No AI account?	The candidate output below is all you need

The candidate output below was written by the author as a plausible AI response. It is a teaching example, not a recorded answer from any particular model.

> The method permits a request only when the active-reservation count is non-negative and below three, and the requested duration is between one and seven days inclusive. It returns a Boolean result. Before returning true, it consults the equipment database to confirm that the requested item is available.

Break the explanation into individual claims. For each claim, note:

1. Whether the code supports it, contradicts it, or gives too little information.
2. The expression, input or operation that justifies your decision.
3. What more you would need to know before making a broader claim about the application.

Then write a corrected explanation in two or three sentences. Keep the accurate parts; do not throw the whole response away because one part is wrong.

The skill being practised is finding where the evidence is. A real application might well have a database somewhere, but that does not show that this method consults it. Equally, the absence of a database call in this class does not prove that nothing else ever checks availability. Keep your conclusion as narrow as your evidence.

If you use a live assistant

The supplied example is enough for the activity. If your course rules allow a live assistant and you have access to one, you can try this prompt:

> Explain only the behaviour visible in this Java class. Identify its inputs, returned result and any state changes. Separate facts visible in the code from assumptions about its caller. Do not invent surrounding services or rewrite the program.

Give it the class and compare its answer with your own trace. A confident explanation still needs checking, and a careful prompt does not guarantee a correct answer. Write down which tool you used, which claim you accepted or rejected, and the evidence behind the decision.

There is one danger worth naming now. If a misunderstood requirement shapes both an implementation and its tests, the two will agree with each other while both disagree with the intended policy. So always compare expected results with the clarified requirement, even when every generated test passes.

Use made-up project data, and follow the module leader’ s current rules about AI in assessments. Every AI exercise in this book has a route that needs no paid service. Chapter 15 gathers the recurring workflows and is meant to be consulted alongside any chapter; the activity in this chapter stands on its own.

1.6 The project we will build, and the road ahead

Throughout the book we will build the **Campus Equipment Lending System** in Java. A first useful version supports a handful of activities: look up availability, request a reservation, confirm an eligible request, hand over the equipment and record its return.

We start with a simple interface and data held in memory, so that the early chapters can stay focused on the rules of the domain. Later chapters introduce the boundaries needed for storage and for outside services. The `ReservationRules` class from the start of this chapter is only a warm-up. It is not the finished application and its structure is not a promise: the lending system proper begins with the Chapter 2 starter, and each later chapter says whether its code joins that project or stays in a separate teaching example. Appendix D lists, for every chapter, where the project stands, how many tests a build should report, and the rules of the domain.

Words we will use

Term	Meaning in the book
Borrower	A person seeking permission to borrow equipment
Equipment item	One individually identifiable piece of equipment
Reservation	A recorded arrangement for an item over a period, whose status matters
Loan	The actual issue of an item to a borrower, followed by its return
Loan period	The intended interval during which the item is borrowed

A reservation and a loan are related but not the same. A confirmed reservation can exist before the borrower ever collects the equipment, and cancelling it is a different act from returning an item that has already been handed over.

We begin with a deliberately small scope: one item per reservation, explicit periods and invented borrowers. Once we use real dates we must decide when two periods overlap, and whether one may begin exactly when another ends. The plain number of days in Section 1.1 does not settle those calendar rules.

Along the way we will keep **invariants**: conditions the domain requires to stay true whenever the system is operating validly. A loan period must be valid, two confirmed reservations must not clash over the same item and period, and an inactive loan cannot be returned. These are goals for the implementation to grow into; the little rule class from Section 1.1 enforces none of them yet.

The road ahead

Chapters	What changes in your understanding
2–4	Objects and their responsibilities, a backlog of work, and a shared delivery workflow
5–7	Better responsibilities and dependencies, then behaviour developed with useful automated tests
8–10	Recurring design solutions, and how to organise a modest application
11–13	Improving unfamiliar code, reading project evidence, and absorbing a changed requirement across integration boundaries
14	Judging the result, documenting decisions and preparing a handover someone else can follow
15	Working with AI assistants, to consult alongside the relevant chapters

Chapters 2 to 14 follow the thirteen teaching weeks; this chapter and Chapter 15 support that sequence without adding weeks of their own. Along the way the project reaches four stages: a kickoff in Week 3, a minimum viable product in Week 6, patterns and application structure in Week 9, and a verified response to a changed requirement in Week 12. The change itself is announced in Week 11.

The book’ s own lending system gives us shared examples, and each practical activity says whether you do it on the book’ s project or on your team’ s project. Teams choose their project domain from those the module offers; equipment lending and an ML model-serving tool are two examples. If your team works on another domain, its change requests will be different. That is why each chapter ends with a note on transfer: learn the

reasoning well enough to apply it, because copying an equipment-related class name into another project does not show that the responsibility belongs there.

The module aims to prepare you to build reliable software around AI and data components, and the project reflects that without training a model. The book takes the data-component route. Its outside collaborator is the **equipment directory**, which answers the question "who is this borrower, and which category are they in?" from beyond the application. The application depends on it only through a contract, so a test can stand in for it and an outage can be told apart from an absent borrower. You will meet it in steps: Chapter 6 previews the contract, Chapter 7 shows how a test stands in for it, Chapter 8 builds the adapter that translates the provider's codes, Chapter 10 places it in the application structure, and Chapter 13 adds what a real provider needs. Each of those terms is explained where it first appears. If your team's project is a model-serving domain, the same questions apply to the component that calls the model. Using AI to help write software is a different subject, and Chapter 15 covers it.

1.7 How to read this book, and any other

When you meet code, predict its behaviour before you run it. Pick one ordinary case and one case near a limit, trace the decisions, write down the result you expect, and only then compare it with what the program prints. A mismatch is useful: it tells you that your understanding, the code or the requirement needs correcting, and you get to find out which.

When the book introduces extra structure, ask what problem made it worth having. If it adds a class, name the responsibility that class receives. If it adds an interface, name the caller that benefits. If you cannot, reread the problem before you copy the solution.

It also pays to keep a small learning record, with three entries for every practical session: the decision you made, the evidence you used and the uncertainty that remains. "I changed the condition" describes an action. "I kept the student limit and checked requests at two and three existing reservations" explains your intent and your evidence.

Reading advice critically

Technical books recommend practices that suit particular circumstances. Read their advice as claims to understand and test, not as rules. Take an imaginary recommendation: "Give every class an interface." Ask which callers need interchangeable behaviour, what coupling the interface removes, and whether the extra files make this example easier or harder to follow. And when you criticise advice, criticise its strongest reasonable reading, not a cartoon of it.

A simple reading record helps:

Entry	Question
Claim and location	What exactly does the author recommend, and where?
Context	What problem and assumptions surround the advice?
Supporting example	How does the example demonstrate the claimed benefit?
Limitation	Under what conditions might the cost outweigh the benefit?
Project application	What would change if we applied it to our system?
Further evidence	Which independently checked source or experiment helps evaluate it?

The module includes an individual critical review of a book from the approved software-design and Agile-engineering list, due in Week 8. You will evaluate the book's principles and examples, discuss their trade-offs and limits, and consider how they apply to engineering AI-enabled systems, supported by further sources. Start reading early, and follow the current assignment brief for exactly what it requires. This textbook does not replace the book you choose, and an AI summary of a book you have not read is no evidence about it. Appendix E develops the review method; the reading record above is enough to begin with today.

1.8 Easy mistakes, and how to spot them

Each of these mistakes can be shown wrong with a small example from this chapter.

Mistake	Counterexample	Better habit
Reading `true` as "the reservation is confirmed"	`mayRequest(0, 1)` returns `true` even if the camera is already reserved for that day	State what a result does and does not establish
Changing the numbers to add the new behaviour	Replacing `3` and `7` with `5` and `14` accepts a student request for eight days	Preserve existing cases while adding the new ones
Treating passing tests as proof	A test that expects students to borrow for fourteen days passes beside an implementation with the same error	Check expected results against the clarified requirement
Trusting a fluent explanation	Activity 1A's explanation mixes accurate claims with an invented database call	Locate the code or requirement behind each claim
Assuming the edit is local	The seven-day limit may also appear in a screen and a report	Search for where each decision is represented before editing

1.9 Try it yourself: a readiness check

This is a chance to find out what you need to practise. It is not a graded assessment. You only need the Java class from Section 1.1, its stated rules and the staff change; no team code is involved.

Task A: read and explain

Without running the code, predict the result of each call below, and explain each answer by naming the condition that decides it.

Call	Your predicted result	Your reason
`mayRequest(0, 1)`		
`mayRequest(2, 7)`		
`mayRequest(3, 1)`		
`mayRequest(1, 8)`		
`mayRequest(-1, 1)`		
`mayRequest(1, 0)`		

Then identify the class name, method name, parameter types and return type, and say whether a call changes a reservation count or records a new reservation. If a keyword is unfamiliar, notice the difference between a gap in your Java syntax and a difficulty in understanding the rule; they need different remedies.

Task B: examine one small fault

A developer changes `requestedDays ≤ 7` to `requestedDays < 7`. Find one input that tells the original and the modified versions apart. State both results and say which one agrees with the original requirement.

If you have a Java Development Kit, you can run it. Save the class as `ReservationRules.java` and create a separate file named `Chapter01Demo.java` containing:

```
public class Chapter01Demo {
    public static void main(String[] args) {
        ReservationRules rules = new ReservationRules();
        System.out.println(rules.mayRequest(2, 7));
        System.out.println(rules.mayRequest(3, 7));
    }
}
```

From the directory containing both files, run:

```
javac ReservationRules.java Chapter01Demo.java
java Chapter01Demo
```

For the original class the output is:

```
true
false
```

Now make the deliberate fault, repeat the same calls and explain the difference, then restore the original expression. Appendix B gives a complete, reproducible setup, but for this exercise a Java Development Kit and the commands above are all you need, and you do not have to install anything to do the paper version.

Task C: plan the staff change

Write a short change note. Do not try to implement a complete redesign. The note should contain:

- The behaviour that must stay the same for students.
- The new staff behaviour, and the extra information needed to select it.
- One open question that should be clarified rather than guessed.
- At least four acceptance examples covering both categories and their limits.

- One simple way to implement the change, and one trade-off it brings.
- A statement, in two or three sentences, of what you would need to see before accepting the change as safe, and what that evidence would still not show.

Do Activity 1A as part of the same exercise, and make your analysis of the AI explanation distinguish what is said about this method from what is said about the whole application.

What to produce

Keep, or hand in as your instructor directs: the completed prediction table, your input that exposes the fault, the change note with its statement of evidence, your analysis of the AI explanation, and the readiness checklist below. If you ran the program, include what you observed and say which version produced it.

You are done when your predictions follow the code, your proposed change keeps the original student rule, your acceptance examples could expose a boundary error, and your evidence statement names checks that could show the change to be wrong. You should also be able to explain why the method alone cannot confirm that equipment is available.

If you want more, once your team has chosen its project domain, repeat Task C for one numeric limit in that domain.

Readiness question	Confident	Need practice
I can trace a Boolean expression with several conditions.	☐	☐
I can distinguish a method’ s inputs, returned result and state changes.	☐	☐
I can identify a case at the edge of an allowed range.	☐	☐
I can separate a confirmed requirement from an assumption.	☐	☐
I can explain what evidence supports a code explanation.	☐	☐
I can describe a small change and the behaviour it must preserve.	☐	☐

Choose one item marked “Need practice” and decide what you will do about it. For syntax, use the relevant Java tutorial [1]; Appendix A has a refresher. For tracing, work through another input with a partner. For requirements, rewrite the rule as concrete examples and see what is left unanswered.

1.10 Questions to think with

Concept questions

1. Give an example of software that produces the expected result today but is difficult to maintain. What makes the next change risky?
2. Explain the difference between an iteration and an increment using the equipment-lending project.
3. Why does passing a set of tests give useful but limited evidence? Include an example of an incorrect expectation.

4. Distinguish a behaviour change from a refactoring. Why should a reviewer know which one is intended?

Applied diagnosis

5. A developer replaces the student limits of three reservations and seven days with five and fourteen. Staff examples now pass. Explain why this is not enough to show the requested change is correct, and give a student example that exposes the problem.

6. An AI assistant writes a method and tests for it. Every test passes, so the team accepts the change without comparing it with the equipment officer’ s rules. Identify the gap in the reasoning and propose a check that closes it.

Transfer challenge

A room-booking application allows each student society to make up to two active bookings, each lasting at most two hours. University departments now need up to four active bookings, each lasting at most three hours. The society rules should stay as they are.

Prepare a change note using the method from Task C. State any assumptions you need about duration values and booking status. Give one example that must still be rejected for a society but is allowed for a department, and name one fact about room availability that these numeric rules alone cannot establish.

No design pattern is needed. Your task is to explain the change, keep existing behaviour and identify the evidence that would support accepting it.

What to take away, and what comes next

A program that works is where engineering begins. To change it responsibly we have to understand the behaviour being asked for, find the decisions that implement it and know what must stay true afterwards.

Object-oriented design helps us organise responsibilities, and iterative delivery gives us regular chances to look at a useful increment and revise our understanding. Tests, reviews, demonstrations and version history each contribute a different kind of evidence, and none of them is a guarantee beyond its own reach.

The little reservation example also showed why working with AI takes judgement. A plausible explanation can mix accurate observations with an invented operation, so we weigh each claim against the code and the requirements, and we record why we accepted or rejected it.

In **Chapter 2: Object-Oriented Development Beyond Syntax** we look inside a small lending system: how objects protect their state, how responsibilities are handed out, and how cohesion and coupling shape the changes we make.

Sources and further reading

The scenario, project rules, code, diagrams and exercises in this chapter are original teaching examples. External sources support the specific concepts identified below. Web sources were checked on 12 October 2026.

1. **Dev.java.** **Classes and Objects**. Official Java learning material on classes, methods, constructors and objects. Use the relevant subsection to address a syntax gap; reading the entire tutorial is not a prerequisite for this chapter.

2. **Authors of the Agile Manifesto.** **Principles behind the Agile Manifesto**, 2001. Supports the discussion of frequent delivery, adaptation, technical quality and team reflection in Section 1.3. This chapter does not yet teach a complete Agile framework.

3. **Fowler, Martin.** **Definition of Refactoring**, 1 September 2004. Supports the distinction between internal restructuring and changes to observable behaviour in Section 1.2.

4. **GitHub Docs.** **Application card: GitHub Copilot inline suggestions**. A vendor's account of its tool's uses and limitations, including the warning that suggestions can look valid while incorrect and the user's responsibility to review and validate them. Used as a specific example in Section 1.5, not as an endorsement or a general evaluation of all AI assistants.

Chapter 2: Object-Oriented Development Beyond Syntax

> *Encapsulation, invariants, snapshot copying, and UML class modelling.*

OO Development and Agile Practices
Designing, Testing and Evolving Software with AI Assistance

Chapter 1 ended with a rule that fits inside a single method: a few numbers, a few conditions, a Boolean answer. Real programs do not stay like that for long. Soon something has to remember which reservations exist, something has to decide whether a new one clashes with an old one, and something has to tell the user what happened.

Picture the equipment officer trying a first small reservation application. It checks a borrower's limit, looks for a conflicting reservation, records a successful request and returns a confirmation message. The demonstration works. Then the developer who wrote it is away, and another team member is asked to change the wording of the message.

They open `ReservationCoordinator` and find one method that counts reservations, applies a borrowing rule, searches for date conflicts, creates a record and assembles the confirmation text. Which lines can they change safely? Why does the conflict check concern an equipment item when the count concerns a borrower? Could somebody outside the class empty the list of stored reservations?

Knowing how to declare a class does not answer any of that. These are questions about design: which responsibilities belong together, which state a class must protect, and how much other code should be allowed to know. This chapter answers them through a small system you can run. You will assign responsibilities to objects and explain the state each one protects, and you will see why private fields and getters are not the same thing as encapsulation. You will diagnose cohesion and coupling from concrete change scenarios, weigh composition against inheritance, and learn to read, draw and check a small UML class diagram so that it matches the code. You will finish by making one behaviour-preserving improvement and presenting it as a reviewed Git change.

The chapter corresponds to **Week 1** and supports LO1 and LO6. It builds on Chapter 1's reservation rule, so you should be comfortable tracing conditions, calling methods and explaining a returned value; Java's official class tutorial [1] and Appendix A are there if you need to revise syntax. The examples use a Java 17 language and API baseline with no preview features or external dependencies, and you need a JDK to compile them. JUnit, Maven and automated CI are not prerequisites for this first design exercise; they arrive with the later work that needs them, and Appendix B records the toolchain and commands for the whole book.

You start from the Chapter 2 starter in the companion file, and you finish with the same code after the confirmation extraction. That finished version is where the running project picks up in later chapters. The fault demonstrations and the notation examples in Section 2.6 are separate teaching material and do not join the project. The companion holds every complete source file, a starting version, the completed refactoring, the checks and suggested answers: use its starting listings and execution instructions first, and look at the solutions only after you have attempted the activity. Java excerpts in this chapter are labelled where they leave out surrounding code.

Read Sections 2.1 to 2.9 as the core, and give the most time to Section 2.6 on UML class diagrams, because that is the notation the midterm assesses. The module's Week 1 reading is Martin [10], the Part I introduction and the object-oriented design fundamentals, with Sections 2.1 to 2.5 as its practical counterpart; use the Moodle links for the exact pages. What you must do is the Section 2.8 AI activity and Parts A to D of the practical activity in Section 2.10: the annotated diagram and scenario model, the responsibility diagnosis, the checked extraction commit with its review, and your team's issue board and branch policy. Parts A to C run on the book's project, and Part D concerns your team's project. If you want more, try the investigation at the end of Section 2.10, then the questions and the transfer challenge in Section 2.11.

2.1 Objects: their knowledge, behaviour and identity

Start with the vocabulary. A class describes a kind of thing, and an object is one particular thing of that kind that exists while the program runs. Two objects built from the same class can stand for different things and hold different information; Java gives us fields, constructors and methods to say all this [1].

In our project a `Reservation` records an arrangement between a borrower, an equipment item and a period. Its information includes an identifier such as `R1`, a borrower identifier such as `B1`, an item identifier such as `CAM1` and a `LoanPeriod`.

Notice that these identifiers are different kinds of identity. Two physical cameras may share a model name but still need separate item identifiers. Two reservations for the same camera on different dates are still two reservations. And a borrower's name would be a poor identifier, because names change and different people share them.

Java adds an identity of its own. Two references can point at the very same object, or at two separate objects that stand for the same domain identifier, and the `==` operator on references only tells you about the first case. Whether two objects count as "equal" in the domain is a decision somebody has to make; matching fields do not settle it automatically. Our small `Reservation` class does not override `equals`, so do not assume it already treats two reservations with the same identifier as equal.

It helps to separate what an object *knows* from what it can *do*. The first is its **state**, the information it holds. The second is its **behaviour**, what it can do through its operations. A reservation can reveal its identifier and period without letting anyone replace them. A coordinator can add a new record once it has checked the request. A period can say whether it overlaps another period.

Not every object needs to change over its lifetime. In the starter, each `Reservation` and `LoanPeriod` is immutable once built: its fields cannot be reassigned, the values they refer to cannot change, and none of its operations modifies it. The coordinator is the exception. It collects reservations in a private list as the program runs.

That difference explains something you may already have noticed. The same request can get a different answer after an earlier request succeeded. Chapter 1's `mayRequest` method only looked at the numbers it was given. A call to the coordinator can change the state that the next call looks at.

From a numeric rule to a small system

We keep Chapter 1's `ReservationRules` exactly as it was: a borrower may hold at most three active reservations and request one to seven days. The staff variation stays a future design exercise, so at this stage the student rules apply to every borrower identifier.

The new starter has four domain and application classes:

Class	Main role	State held
`ReservationRules`	Evaluate the initial numeric borrowing policy	No instance fields
`LoanPeriod`	Represent a valid period and answer overlap questions	Start date and exclusive end date
`Reservation`	Record the identity and details of a confirmed reservation	Reservation, borrower and item identifiers; period
`ReservationCoordinator`	Process requests using policy checks and existing records	In-memory reservations and a rules object

The identifiers stand in for real borrower and equipment records. We have not written full `Borrower` and `EquipmentItem` classes or a catalogue, so a non-blank identifier proves nothing about whether such a person or item exists.

2.2 Who is responsible for keeping things true?

Every class has a job, and the useful way to state a job is as an obligation. "Hold two dates" only describes data. "Represent a period whose end follows its start" is an obligation, and it is what gives those two dates their meaning. We call such an obligation a **responsibility**.

Some obligations can be written as a condition that must always hold, and that kind has a name: an **invariant** is a condition that must be true for valid objects or valid states of the system. Different invariants belong in different places. A single `LoanPeriod` can guarantee that its end comes after its start. It cannot guarantee that nobody else has already reserved the same equipment, because it knows nothing about other reservations.

A period convention we can test

For this stage, dates are calendar dates with no time of day. We write a period as **start inclusive, end exclusive**, or `[start, end)`. A period from 20 October to 22 October covers the 20th and the 21st, and another period may begin on the 22nd. This is an explicit decision for the book; a real lending service might need a cleaning or inspection gap between loans, and that would be another rule.

Java's `LocalDate` is immutable and represents a date with no time and no time zone [2]. We use `isBefore` and `plusDays` to express the convention, and we do not treat these dates as UTC instants or as a fixed number of elapsed hours.

The constructor rejects a missing date or an end that does not come after the start. With that guaranteed, the overlap test is short:

```
// Method excerpt from LoanPeriod.java.
public boolean overlaps(LoanPeriod other) {
    Objects.requireNonNull(other, "other");
    return start.isBefore(other.endExclusive)
            && other.start.isBefore(endExclusive);
}
```

Two periods overlap when each one begins before the other ends. Both comparisons are strict, which is exactly why two periods that merely touch do not overlap. `Objects.requireNonNull` throws a `NullPointerException` for a null argument instead of inventing a replacement period [3].

First period	Second period	Overlap?	Explanation
`[20 Oct, 22 Oct)`	`[21 Oct, 23 Oct)`	Yes	Both include 21 October
`[20 Oct, 22 Oct)`	`[22 Oct, 23 Oct)`	No	The second begins at the first's exclusive end
`[20 Oct, 22 Oct)`	`[20 Oct, 22 Oct)`	Yes	Identical periods overlap
`[20 Oct, 22 Oct)`	`[19 Oct, 24 Oct)`	Yes	The second contains the first

Try comparing only the start dates and you will see that some of these rows go wrong. Concrete examples are the quickest way to check what a short expression really means.

Who protects which condition?

Each condition has a natural owner. The constructor protects the validity of one period. `ReservationRules` protects the numeric policy, provided it is given an accurate count. The coordinator counts its records and checks for item conflicts before it stores anything. No single class can enforce all of this alone, because each one lacks information that another holds.

The starter also makes some deliberately limited assumptions. Every stored record is confirmed and stays active, because cancellation, completion and expiry do not exist yet. Calls arrive one at a time in one process. The data vanishes when the process ends. Identifiers are matched exactly, with no clean-up and no authentication. And the coordinator checks availability only against its own records, so maintenance schedules and other coordinators are out of scope.

These limits shape what we may claim. Inside one sequential coordinator, checking for conflicts before adding a record prevents the overlap we demonstrate. That is not the same as safe booking by several processes at once, a boundary we revisit when we reach integration.

2.3 Keeping state safe: encapsulation and information hiding

You may have been told that making fields `private` is encapsulation. It is a start, and only a start. **Encapsulation** groups state with the operations that govern it and controls who can reach that state. **Information hiding** is about which implementation decisions callers should not need to know. The two support each other, but writing `private` in front of every field achieves neither by itself.

A related word is **abstraction**: showing a caller only what it needs and leaving out the detail. A caller can ask a period whether it overlaps another period without repeating the date comparisons. That is abstraction through an ordinary public method, with no need for Java's `abstract` keyword or an interface. It still needs a clear contract, including our decision that two periods that touch do not overlap.

To see why `private` is not enough, imagine a class with separate setters for a period's start and end. A caller could move the start beyond the current end. If each setter simply assigns a field, the fields are private and the relationship between them is still unprotected.

Our `LoanPeriod` avoids that. Its constructor takes both dates together and checks the relationship, and the class offers readers but no setters. To describe a different period, a caller makes another valid object, so an invalid half-way state never exists. In another domain a controlled way to change an object could be the right choice; immutability is a design decision, not a law.

A getter can hand out the keys

Here is a deliberately unsafe alternative for the coordinator:

```
// Unsafe alternative for diagnosis; not the starter implementation.
public List<Reservation> reservations() {
    return reservations;
}
```

Now any caller can write `coordinator.reservations().clear()` and erase every record without passing through the coordinator's checks. The field is private, yet callers hold a reference to the very list the coordinator uses.

The starter returns this instead:

```
// Method excerpt from the supplied coordinator.
public List<Reservation> reservations() {
    return List.copyOf(reservations);
}
```

`List.copyOf` builds an unmodifiable list, and later changes to the original collection do not show up in it [4]. So callers get a snapshot they cannot modify. It does not copy the elements themselves, and it cannot make a mutable element immutable. Here that is fine, because `Reservation`, `LoanPeriod`, `String` and `LocalDate` are already immutable.

The same reasoning shows why `private final List<Reservation>` alone would not be enough. `final` stops the field from being pointed at a different list, and it does nothing to stop `add` or `clear` on the list it points at.

The coordinator still reveals more of its inner workings than a mature system might, since callers can see that the records are kept as a list. That is acceptable at this stage. If callers later need only a count or an availability answer, a narrower operation would mean they have less to understand.

2.4 Is a long method the problem? Cohesion and coupling

Reviewers often say "this method is too long", and sometimes they are right, but length is a symptom. Two ideas get closer to the cause. **Cohesion** asks how closely the responsibilities inside one component belong together. `LoanPeriod` is cohesive: its validation and its overlap calculation are about the same idea, a period. A coordinator that also knows message wording, storage details and borrowing policy is mixing more varied responsibilities.

Coupling asks how much components depend on each other. One component depends on another when it relies on its type, its operations, its data format or its behaviour. Some dependency is unavoidable, because software works only when its parts cooperate. What matters is whether a dependency expresses a useful relationship or lets unrelated changes spread.

Look at the end of the starter's `reserve` method:

```
// Excerpt from the starting ReservationCoordinator.java.
Reservation reservation = new Reservation(
        "R" + (reservations.size() + 1), borrowerId, itemId, requested);
reservations.add(reservation);
return "Confirmed: " + reservation.id()
        + " for " + reservation.borrowerId()
        + " on " + reservation.itemId()
        + " [" + reservation.period().start()
        + ", " + reservation.period().endExclusive() + ").";
```

It creates and stores a reservation and then builds its confirmation text. Earlier lines, which you can see in the companion's complete starting file, count the borrower's records and check for conflicts.

Having several steps does not make a method wrong; coordinating often means working with several collaborators. The better question is which of those decisions has its own reasons to change. One way to find out is to imagine some likely requests and ask where each would land:

Possible request	Decision it affects	Current location
Change a borrowing limit	Numeric policy	`ReservationRules`
Add a maintenance gap between reservations	Availability policy and period use	Coordinator conflict check and possibly period-related rules
Store records beyond process lifetime	Storage mechanism	Coordinator's list and storage operations
Translate a confirmation message	Presentation wording	Coordinator's final return expression

That table is a better diagnosis than "the method is long". It shows which changes would force an edit to the coordinator, and why. It also separates likely work from imagined generality: a database may be a future boundary, but that is no reason to build a repository framework now.

A small improvement, and exactly what it is worth

We will pull the confirmation expression into a private method called `confirmationFor`. The request method still adds the record first and then returns what the helper produces. The helper receives the existing reservation and returns exactly the same text:

```
// Method added to ReservationCoordinator.java by the extraction.
private String confirmationFor(Reservation reservation) {
    return "Confirmed: " + reservation.id()
            + " for " + reservation.borrowerId()
            + " on " + reservation.itemId()
            + " [" + reservation.period().start()
            + ", " + reservation.period().endExclusive() + ").";
}
```

The request method then ends with `reservations.add(reservation); return confirmationFor(reservation);`, so the record is still stored before the text is built.

It is a modest refactoring. The formatting now has a name, and a reader can follow the steps of a request without wading through every concatenation. It does **not** create a separate presentation component, remove the coupling to storage or make the coordinator thread-safe. A private helper improves readability inside one class; a new component would change how classes collaborate.

It is worth being exact about this, because small edits are easy to oversell. If translation later needs collaborators of its own, a separate formatting responsibility may be justified. For the task at hand a named private operation is enough.

The supplied checks look at both the returned messages and the stored state, and after the extraction both must be unchanged. A version that kept the success message but forgot to store the reservation would not have preserved the behaviour.

2.5 Has-a or is-a? Composition and inheritance

If one class needs the services of another, there are two classic ways to arrange it. The first is **composition**: build behaviour by giving an object collaborators or components. A `Reservation` has a `LoanPeriod` and asks it about overlap, and a coordinator has a `ReservationRules` object instead of inheriting from it.

That fits the domain. A coordinator is not a kind of borrowing rule, and a reservation is not a kind of period. The relationships are about what one object uses or contains, not about one being a subtype of the other.

The second way is **inheritance**, where a derived class is defined in terms of a superclass. Java supports inherited members, overriding and run-time selection of the overridden method [5]. These tools can express a real family of related behaviours, but they also create expectations that callers will come to rely on.

Take an imaginary `SpecialReservation` subclass, introduced only because some borrowers have longer limits. Is that really a new kind of reservation, or is it a policy about whether a reservation may be created at all? If a borrower’ s staff status can change while an existing reservation keeps its identity, forcing that difference into a reservation hierarchy gets awkward quickly.

That does not mean inheritance is always the wrong choice. It means asking what varies, what stays stable and what a caller expects from the base type. Chapter 5 returns to this through behavioural substitutability. For the project at this stage, constructors, fields and collaborators already express every relationship we need, with no custom hierarchy.

Question	Composition can help when	Inheritance deserves consideration when
What is the relationship?	One object uses or contains another	One type has a meaningful subtype relationship to another
What varies?	A collaborator or delegated responsibility	Behaviour shared through a stable base contract
What must be reviewed?	How collaborators are selected and called	Whether derived behaviour preserves caller expectations

Composition has costs too. If you break a readable expression into many tiny collaborators, a request becomes harder to follow. Do not replace one clear expression with several classes unless that reorganisation solves a real problem. The goal is responsibility boundaries that people can understand, not the largest possible number of objects.

2.6 Drawing the design: UML class diagrams to read, write and check

A class diagram is a selective explanation of a design. It answers one question for one audience, such as "which objects take part in a reservation request, and how are they related?" It is not a copy of the source code, and a diagram that tries to show every field and method answers no question well. This section teaches you to read a diagram, to draw one from a written description, and to check one against code. Class diagrams are the only UML notation used in this chapter; later chapters introduce other kinds where they are first needed.

You will often draw these by hand, and a hand-drawn diagram uses the same symbols as the ones in this book. The diagrams here are written as text in a tool called Mermaid so that they can be kept beside the code. Appendix A shows the text syntax.

Reading the classes

For our starter, Figure 2.1 shows the collaborators that matter to understanding a request.

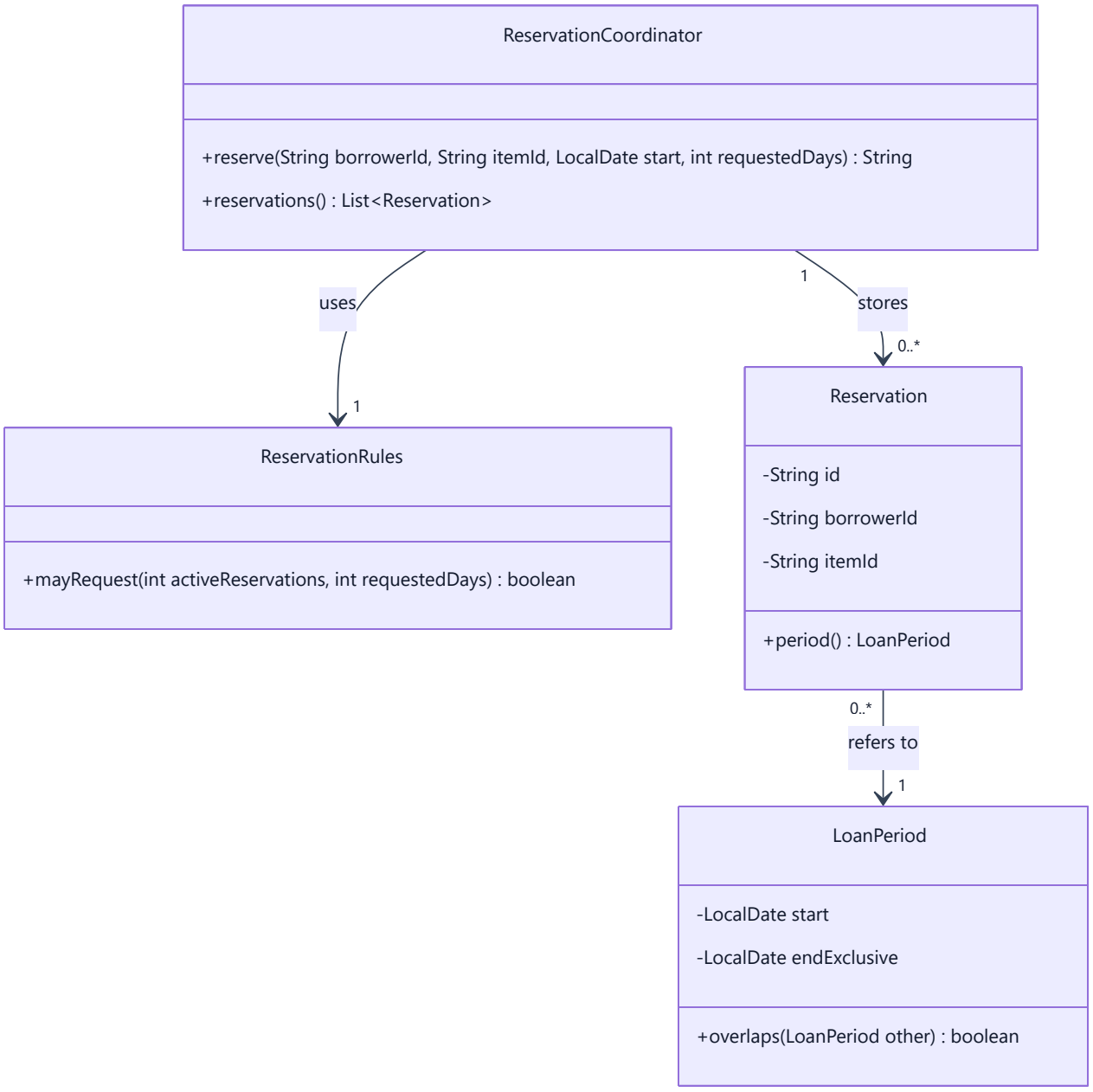


Figure 2.1. Selected classes and navigable associations for one coordinator’ s records. The displayed multiplicities describe this teaching model. The Java types do not prevent a caller from holding additional references to the same immutable objects.

Each box is a class and has up to three compartments. The top holds the class name. The middle holds the **attributes**, the data the class keeps. The bottom holds the **operations**, the methods other code can call. A label in double angle brackets above the name, such as `<<interface>>`, says what kind of class the box is.

Compare the `LoanPeriod` box with the Java it describes:

```java
// Excerpt from LoanPeriod.java; method bodies omitted.
public final class LoanPeriod {
    private final LocalDate start;
    private final LocalDate endExclusive;

    public LoanPeriod(LocalDate start, LocalDate endExclusive) { ... }
    public LocalDate start() { ... }
    public LocalDate endExclusive() { ... }
    public boolean overlaps(LoanPeriod other) { ... }
}
```

Part of the box	In the Java source	How to read it
`LoanPeriod`	`class LoanPeriod`	The class name
`-LocalDate start`	`private final LocalDate start;`	A private attribute called `start` whose type is `LocalDate`
`+overlaps(LoanPeriod other) boolean`	`public boolean overlaps(LoanPeriod other)`	A public operation that takes a `LoanPeriod` called `other` and returns a `boolean`

A `+` marks a public member and a `-` a private one. These diagrams put the type before the name, as Java does, and an operation's return type after its parameter list.

Notice what the box leaves out. The constructor and the two accessors `start()` and `endExclusive()` exist in the code but not in the box, because the explanation does not need them. You can trust a diagram for what it shows. You cannot assume that what it omits does not exist.

A field whose type is a plain value, such as `String`, `int` or `LocalDate`, is normally written as an attribute. A field whose type is another class of your design is normally shown as a line to that class's box instead. Do not show it both ways: a `-LoanPeriod period` attribute beside a line to `LoanPeriod` makes the diagram say one thing twice.

Reading the lines

Every line between two boxes is a statement about two classes. The number at each end is a **multiplicity**: how many objects take part. Read it by standing at one class and looking at the number at the *far* end of the line. That number is how many objects of the far class one object of your class is linked to. `1` means exactly one, `0..1` none or one, `0..*` zero or more, and `1..*` one or more.

The line `ReservationCoordinator "1" ⟶ "0..*" Reservation : stores` in Figure 2.1 reads in two directions. From the coordinator: "each coordinator stores zero or more reservations." From a reservation: "each reservation is stored by exactly one coordinator." The arrowhead shows which object holds the reference.

Try it 1. Write one sentence for each of the other two lines in Figure 2.1, reading from both ends. Then, for each number, find where the Java code enforces it and where the model merely assumes it. The companion gives the answers.

The relationships you need to read and draw

Relationship	Drawn as	Read it as	Use it when
Association	Solid line, with or without an open arrow	One object is linked to another; an arrowhead shows which holds the reference	The link is structural and lasting
Dependency	Dashed arrow	One class uses another without keeping it	A parameter or brief call is the only link
Aggregation	Hollow diamond at the whole	A whole groups parts that can exist without it	Parts can be shared or removed and still exist
Composition	Filled diamond at the whole	A whole owns parts that have no meaning without it	A part's lifetime is tied to its owner
Inheritance	Solid line, hollow triangle at the superclass	The subclass is a kind of the superclass	A real subtype relationship holds
Interface realisation	Dashed line, hollow triangle at the interface	The class implements the interface's contract	Several classes satisfy one contract that callers rely on

Figure 2.2 shows every relationship in the table in one small system. It is a notation sampler for a course-pack printing service and is not part of the reservation project.

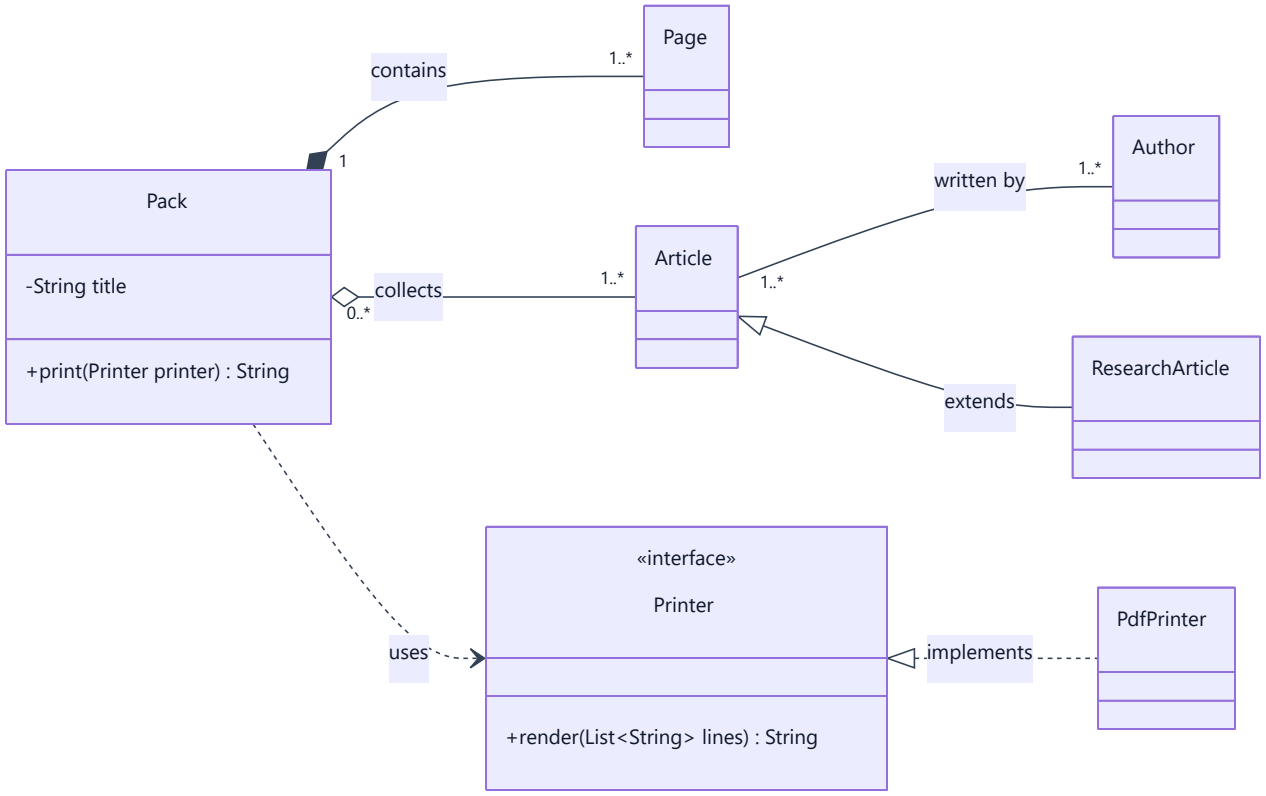


Figure 2.2. A notation sampler. A pack owns its pages, so a page has no meaning without its pack (composition). A pack only collects articles, which exist independently and can appear in many packs (aggregation). Articles and authors are joined by a plain association with no arrowheads, because the description does not say which side holds the reference. A research article is a kind of article (inheritance). A

pack uses a printer only while printing (dependency), and `PdfPrinter` implements the `Printer` interface (realisation).

Choosing a relationship

Choose a relationship by what it means, not by how it looks. For each pair of classes, ask:

1. Does one class implement a contract, an interface, that other classes rely on? That is **realisation**.
2. Is A a kind of B, so that a client of B could be handed an A without noticing? That is **inheritance**. Chapter 5 tests this claim with the Liskov Substitution Principle.
3. Does A keep a lasting reference to B in a field? If not, and A only uses B as a parameter, a local variable or a short-lived object, that is a **dependency**.
4. If A keeps the reference, does B belong to A so completely that it has no meaning without A and is discarded with it? That is **composition**.
5. Does A group Bs that exist independently and could be shared or removed? That is **aggregation**, but only if you can say what the grouping adds. Otherwise it is a plain **association**.

Aggregation is often indistinguishable from association in code, and composition describes a lifetime that the Java types do not enforce, so both are claims about meaning. Figure 2.1 uses plain associations because periods can be shared by reference and the code does not enforce a lifetime. Also distinguish "composition" as a general programming idea, building behaviour from collaborating objects, from UML's filled diamond, which makes a stronger ownership claim.

From code to diagram

When you draw from existing code, each construct has a usual home in the diagram.

In the Java source	In the diagram
`private final LocalDate start;`	Attribute `-LocalDate start`
`private final List<Reservation> reservations`	Association to `Reservation`, with `0..*` at the `Reservation` end
`private final ReservationRules rules` (a class of your design)	Association to `ReservationRules`, with `1` at that end
A parameter, local variable or `new X()` used only inside a method	Dependency on `X`
`class A extends B`	Inheritance, with the triangle at `B`
`class A implements I`	Realisation, with the triangle at `I`
`public` and `private`	`+` and `-`

Two cautions. The Java type does not settle the meaning: a `List<Reservation>` might be an association or a composition, and only the design decides. And a field that can be `null` is `0..1`, not `1`.

A worked example: from scenario to diagram

Here is a short description, and a six-step method for turning it into a diagram.

> The workshop keeps tools in numbered lockers. A locker holds at most one tool at a time, and a tool sits in at most one locker. A technician looks after any number of lockers, but each locker is looked after by exactly one technician at a time; a locker can be reassigned to another technician. Every inspection of a locker is logged as an entry with a date and a note, and an entry is meaningless without its locker. Locker reports are printed as plain text now, and the workshop expects to add other formats later.

Step 1: find the nouns. Workshop, tools, lockers, technician, inspection, entry, date, note, report, plain text, formats.

Step 2: decide which become classes. Keep a noun as a class when it has data or rules the description needs.

Candidate	Decision	Reason
Workshop	Leave out	It is the whole system, and no behaviour is described for it
Locker	Class	It has a number, a log and a report
Tool	Class	A locker holds one
Technician	Class	Looks after lockers
Inspection entry	Class	Has a date and a note and belongs to a locker
Date, note, number, name	Attributes	Plain values describing a class
Report	Operation of `Locker`	Something a locker does, not a thing with its own data
Plain text, other formats	A `ReportPrinter` interface and a `PlainTextPrinter` class	The description expects the format to vary

Step 3: write the attributes and operations. Separate what the description states from what you assume. The description states a locker's number and report, and an entry's date and note. Everything else is a design assumption that you should label as one: here, a technician's name and a tool's name as identifying attributes, and a single `print` operation on the printer, which is one reasonable way to give the report a format-independent contract. A different, justified choice is acceptable, and the answer guidance in the companion accepts it.

Step 4: choose a relationship for each pair, using the questions in the previous subsection.

Pair	What decides it	Result
Technician and locker	Neither owns the other, and a locker can be reassigned	Plain association
Locker and tool	Neither is part of the other	Plain association
Locker and inspection entry	An entry is meaningless without its locker	Composition, with the diamond at `Locker`
Locker and `ReportPrinter`	A locker uses a printer only while reporting	Dependency
`PlainTextPrinter` and `ReportPrinter`	The class implements the contract	Realisation

Step 5: write each multiplicity as a sentence. Each locker is looked after by exactly one technician, so the number at the `Technician` end is `1`. A technician looks after any number of lockers, so the number at the `Locker` end is `0..*`. A locker holds at most one tool and a tool sits in at most one locker, so both ends are `0..1`. A locker has any number of entries and an entry belongs to exactly one locker.

Step 6: read it aloud and delete. Read every line in both directions against the description, and remove anything the description does not say.

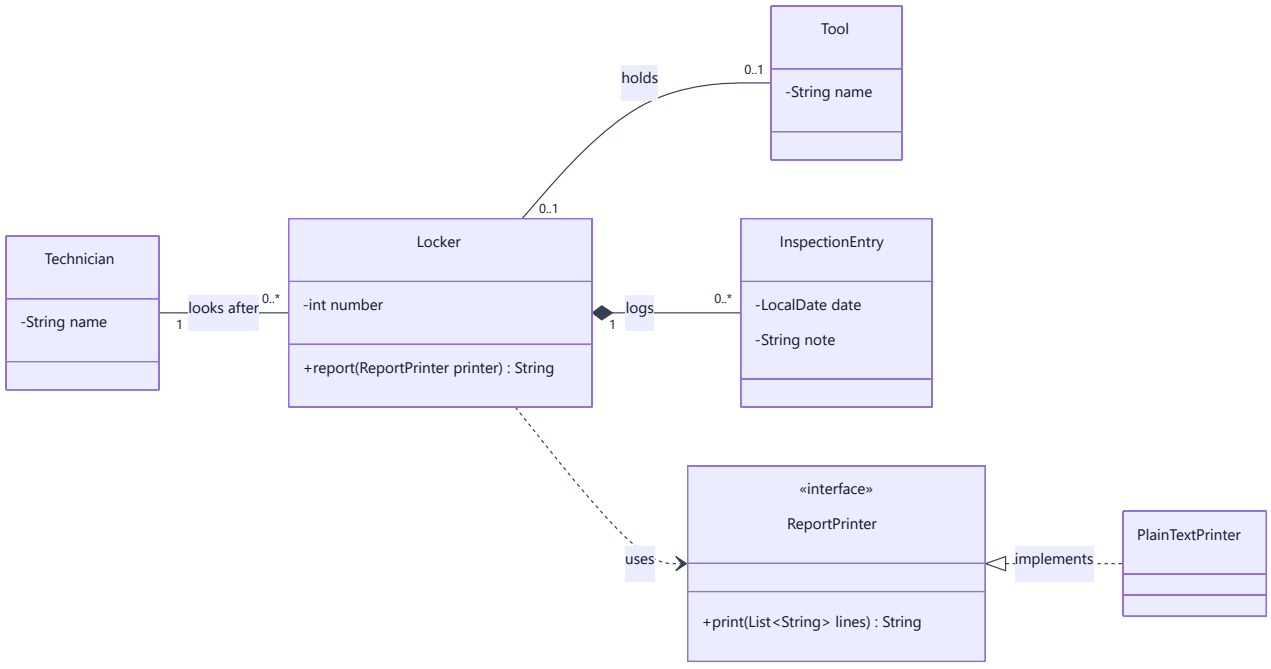


Figure 2.3. The model that results. The two plain associations have no arrowheads because the description does not say which object holds the reference. A model should record what the description says and no more. Other answers are defensible: you might show `Tool` only as an attribute, or add a `Workshop` class if you were asked to model scheduling.

Checking a diagram against the code

A diagram that disagrees with its code is worse than no diagram, because readers trust it. To check one:

- For every box, find the class. For every attribute, find the field with that name and type.

- For every line, find the field, parameter or `extends` / `implements` clause that creates it. If you cannot find one, remove the line or fix the code.
- For every multiplicity, find where the code enforces it (a constructor check, a `final` field, a collection type). If nothing enforces it, mark it as an assumption.
- List what the diagram leaves out, and decide whether each omission is deliberate.

Try it 2. Find two details that are present in the Java for Figure 2.1 but absent from the diagram, and say why leaving each out is acceptable. The companion gives the answers.

The diagram is intentionally incomplete. It omits standard library classes and some accessors. It does not establish the order of operations, display every validation or prove that overlapping bookings are rejected. Read the method body and the period convention for those questions. After extracting a private formatting method, the relationships in Figure 2.1 remain unchanged. This is useful evidence about the scale of the edit: we improved one operation’ s organisation without redesigning the object collaboration.

Spot the errors

Another student drew a first attempt at the workshop model. It contains at least five mistakes.

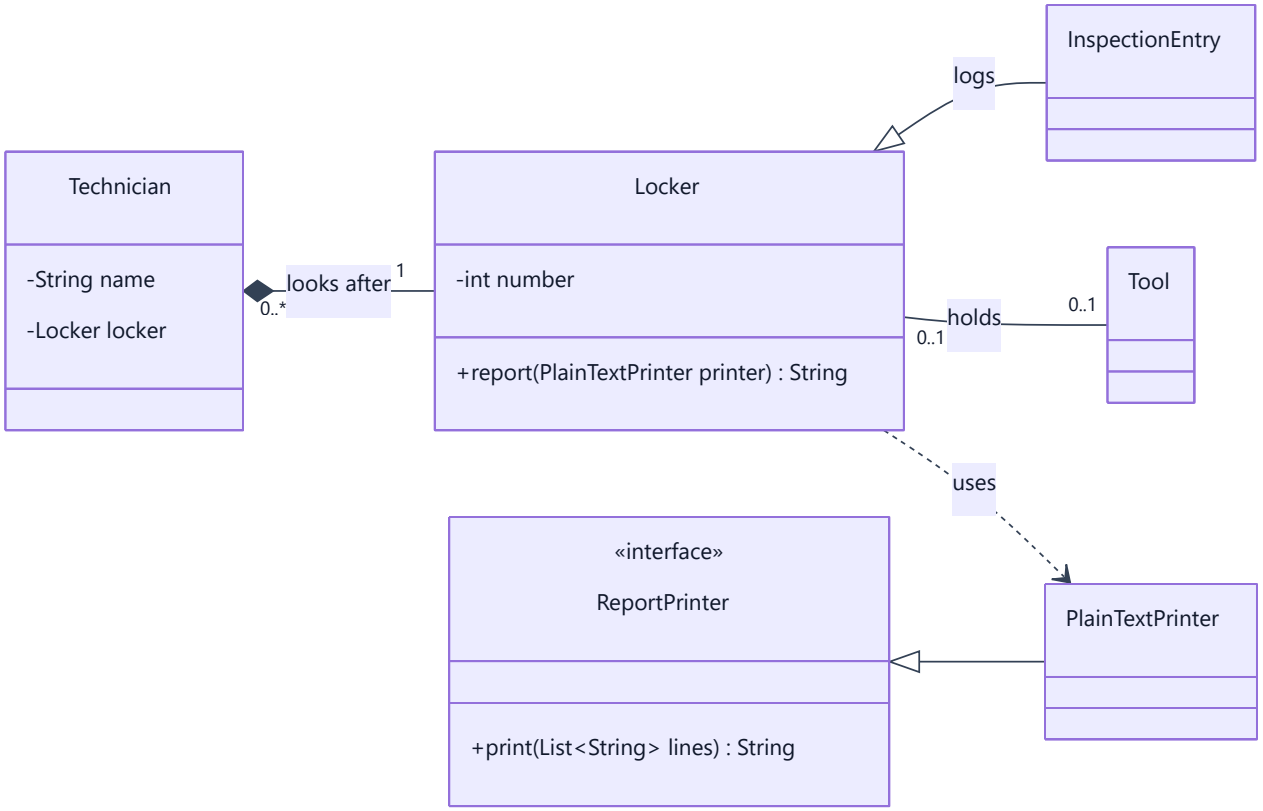


Figure 2.4. A flawed attempt, for diagnosis.

Try it 3. List every mistake you can find. For each, say why it is wrong using the description, and say what the correct notation is. Then redraw the diagram. The companion lists the mistakes.

Common drawing mistakes

Mistake	Why it is wrong	Better habit
Putting a multiplicity at the wrong end	The number at the far end counts the far class, so reversed numbers say the opposite	Read every line aloud in both directions
Treating an arrowhead as ownership	An arrowhead shows which object holds the reference, not who owns whom	Use diamonds, and only with a reason, for ownership
Inheritance for “uses” or “has”	A locker is not a kind of technician, and an entry is not a kind of locker	Test the sentence “A is a kind of B” and its promises
A solid line for interface realisation	Realisation is dashed; a solid line with a triangle means inheritance	Check the line style of every triangle
A filled diamond because the field is `final`	Composition claims a lifetime, and `final` only fixes a reference	Ask whether the part can exist without the owner
Showing a relationship twice	A `-Locker locker` attribute plus a line to `Locker` repeats one fact	Show a field of a design class as a line only
Depending on a concrete class when an interface exists	A client that names `PlainTextPrinter` cannot accept another format	Point the dependency at the interface
Drawing every class and member	The design question disappears among the detail	One diagram, one question, only the members it needs
A diagram that disagrees with the code	Readers trust the diagram	Run the checking steps above before you share it

2.7 Making a change someone else can review

A design improvement is only half the job. The other half is making the change in a way that your teammates can inspect, reproduce and understand. Version control exists for that, although a commit on its own is no proof that a design is good.

Git keeps three things apart: the working files, the staging area and the recorded commits. You edit working files, choose what goes into the next commit, and record that chosen state. Staging matters most when your working files hold several unrelated edits, because the next commit should contain the one change you intend to explain [6].

A **branch** gives a name to a line of development. For the practical activity, put the confirmation extraction on a short-lived branch of its own. A **pull request** (some platforms say **merge request**) is a proposal for review on a hosting platform, and it is a different thing from a local commit. If you have no hosting account, a peer review done at your desk still serves the purpose.

You will hear two styles of working. In trunk-based development, small changes are integrated into one main line often, sometimes through very short-lived branches. In a feature-branch workflow, the work for a feature stays on its branch until it is ready to merge. In practice the two overlap, because a short reviewed branch can be part of frequent integration. What we ask in this course is simple: keep changes small, reviewable and integrated regularly. Git’s own branching discussion shows why topic branches help to keep independent work apart [7].

What a reviewable change looks like

Begin from the supplied starting files and a recorded baseline, and confirm that the checks pass before you edit anything. Then create a branch, make only the extraction, run the checks again and inspect the diff. The companion has the full setup commands.

These are the commands you will use to inspect your work:

```
git status
git diff
git diff --staged
```

The first reports the state of your files. The second normally shows tracked changes that are not yet staged, and the third shows staged changes against the current commit [8]. One trap: an empty `git diff` after staging does not mean there is nothing to commit, and untracked files need a look in `git status` as well.

Appendix B collects the Git commands and the pull-request routine used in this book (B.8). The Pro Git readings [6][7] and the commands above are all you need for this chapter.

A good commit message here would be `Extract reservation confirmation formatting`. Many teams write their convention down; Conventional Commits, for example, puts a type in front of a short description, as in `refactor: extract reservation confirmation formatting` [9]. Use whatever your team agrees, and use it consistently. A good review description states the problem, the limited change and the checks that were run. For example: "The request method interleaves storage and confirmation text. This change moves the text assembly into a private helper. The existing messages and stored records remain unchanged in the supplied checks."

A reviewer should read the diff, confirm that the helper adds no state changes, and check that the record is still added before the confirmation is returned. "Looks cleaner" says less than a sentence about what was inspected and why it matters.

Keep this refactoring on its own. Changing limits, renaming identifiers or touching message punctuation may each be fine, but if they arrive together it is hard to tell why a result changed. And keep compiled class files out of your commits; the companion's `.gitignore` takes care of those.

2.8 Checking an AI’s explanation of the coordinator

Chapter 15 gathers the AI workflows that recur in this book, but this activity is complete on its own and the table below gives you its ground rules.

	For this activity
Your goal	Compare an explanation of the coordinator with its actual control flow, state changes and side effects
You start with	The complete starting `ReservationCoordinator` and its collaborators from the companion
What is in bounds	Judge the explanation only; do not change the code
How to check	Trace the code, run the three-request demonstration and use the supplied checks
How to decide	Keep supported claims, reject contradicted ones and mark claims the code does not establish
No AI account?	The ready-made explanation below is enough

The explanation below was written by the author to imitate a plausible AI answer. It is not a captured response from a real model:

> The coordinator validates identifiers, counts the borrower’s stored reservations and checks the numeric rule. It tests availability using existing reservations for the same item. On success, it adds a reservation before returning a confirmation. It sends that confirmation by email and persists the reservation to a database. Because the list field is final, callers receive a deeply immutable copy and booking is safe under concurrent requests.

Sort each claim into supported, contradicted or not established, and for each one point to the operation in the code, or to the absence of one, that justifies your verdict. Pay particular attention to control flow, state changes and side effects.

Then write a four-sentence replacement that keeps the accurate parts. It should describe input validation, the successful and rejected paths, the state changes and the limits of this version of the project.

If your course allows it, you can repeat the exercise with a live assistant and this prompt; the ready-made explanation is enough if you have no AI account:

> Trace this coordinator and its supplied collaborators. Identify inputs, returned values and state changes on success and rejection. State what is established about storage, notifications and concurrency. Separate facts visible in the code from assumptions. Do not change the code.

Write down one claim you accepted and one you rejected or qualified, with your evidence for each. A second model agreeing with the first is no substitute for tracing the operations yourself.

2.9 Mistakes that look like good design

Mistake	Why it fails in this example	Better question
Private fields mean the object is safely encapsulated	A returned mutable list can still expose a modification path	What can callers change through the references they receive?
A final field makes the referenced object immutable	The coordinator can still add to its final list	Is the reference fixed, the object immutable, or both?
Every noun should immediately become a class	Several concepts have no behaviour or state we need yet	What responsibility would that class own now?
More interfaces automatically mean less harmful coupling	An interface can repeat an unstable contract without solving a caller's problem	Which dependency would this interface improve?
Calling another class means inheriting from it	The coordinator uses a policy; it is not a policy subtype	Is the relationship use, containment or substitution?
A filled diamond just means "has a"	Composition claims the part cannot outlive its owner; `LoanPeriod` objects can be shared by reference	Does the code enforce this lifetime, and would the domain agree?
A formatting refactoring fixes the whole design	The helper leaves storage and other responsibilities in the coordinator	Which improvement did this edit actually make?
Green checks prove the application is production-ready	The starter omits lifecycle, persistence, catalogue validation and concurrency	Which claims have we tested, under which assumptions?

2.10 Try it yourself: improve the design, with evidence

What you start with

Parts A to C use the book's project, and Part D is about your own team's project. Use the companion's common source files and the **starting** `ReservationCoordinator.java`. Save all six Java files in one directory, then compile and run them by following the companion's instructions. The check runner should report 34 passing checks before you change anything. Treat those checks as given evidence; you do not need to understand how the runner itself works yet.

Part A: understand the request

Trace the three requests in `Chapter02Demo`. After each call, record the returned message and the number of stored records. Find the first condition that rejects the overlapping request, and explain why the adjacent request is accepted and why the rejected request does not use up a reservation identifier.

Then draw a small class diagram of the starter that follows the relationships in the code, without copying Figure 2.1. Use the "From code to diagram" table, and run the four checking steps from Section 2.6 on your own drawing. Next to it, mark one invariant that a single object can protect and one condition that needs information from several objects. The notes may sit beside the diagram rather than inside it.

After that, do one separate modelling exercise. It practises turning a written scenario into a class diagram with the six-step method from Section 2.6, and it does not extend the project:

> The equipment office groups laboratory equipment into kits. A kit has a name and holds one or more equipment items. Removing a kit does not delete its items, and an item belongs to at most one kit. Each item keeps any number of maintenance records; a maintenance record belongs to exactly one item and has no meaning without it. A kit summary is produced through an exporter. The office needs CSV output now and expects other formats later.

Draw a class diagram with the classes, their key attributes and multiplicities. Include at least one aggregation, one composition, one interface realisation and one dependency, and state one assumption you had to make. For every relationship, write down the question from Section 2.6 that decided it, then read each line aloud in both directions against the scenario.

Part B: diagnose the design

Write a short responsibility inventory for the coordinator. Name two separate reasons it might have to change, one collaborator that is cohesive and one dependency that deserves attention. Do not judge quality by method length alone.

Part C: make the change

On a short-lived branch, move only the confirmation-string expression into a private `confirmationFor(Reservation reservation)` method. Leave its output, validation order, identifier allocation and storage effects exactly as they were, and do not add an interface or change any public method signature.

Run the checks again, read the diff and ask a peer to review it. If you are studying alone, review it yourself and say honestly that it was a self-review. Make one focused commit after dealing with any useful feedback. A separate hosted pull request is optional for this introductory activity.

Part D: organise your team

Form teams of three or four as the module leader directs, and confirm which of the module's starter domains your project uses. Equipment lending is this book's running example; if your team picks another domain, use each chapter's transfer note.

Set up an issue board with columns that match how your team will really work, and add three cards with concrete completion conditions, for instance "extract confirmation formatting with unchanged output and checks". Agree a branch policy, either very short-lived branches merged often or one feature branch per story, and write down why. Decide who prepares the next demonstration and who reviews the next change, and rotate those roles. Commit counts alone will not explain who contributed what.

What to keep

Keep your annotated diagram, the scenario model, your responsibility diagnosis, the before-and-after check results, the commit identifier, the review note, your analysis of the AI explanation, and your team's issue board and branch-policy note. Add a short statement of the risks that remain after the extraction.

You are done when the diagram matches the code, the extraction preserves the checked behaviour, and your review explains both the improvement and its limits. You should also be able to explain how the returned list is protected without claiming that every copy is a deep copy.

If you want to go further, try this investigation. In a temporary copy, replace `List.copyOf(reservations)` with `reservations`, predict what an outside `clear()` call would do, then run the checks. Put the original back before you continue. It is a deliberate fault demonstration, not an improvement.

2.11 Questions to think with

Concept questions

1. Distinguish a Java object reference from a reservation' s domain identifier. Why are equal identifier strings not enough to make two `Reservation` instances equal through `equals` in this starter?
2. Explain one invariant that a single object can protect and one condition that needs several objects. Why do they need different enforcement points?
3. Explain cohesion and coupling using one likely change to the coordinator. Why is reducing the total number of dependencies not a sufficient design goal?
4. Compare composition and inheritance for the relationship between a reservation and its loan period. What would an inheritance relationship wrongly suggest?

Applied diagnosis

5. A developer adds a getter that returns the private mutable list and says the coordinator is still encapsulated because the field is private. Give a concrete counterexample, and explain the limits of the proposed correction using `List.copyOf`.
6. A refactoring returns the same confirmation text but leaves out `reservations.add(reservation)`. The developer shows the printed text and declares success. Explain the change in behaviour and name the evidence that would expose it.

Transfer challenge

A university room-booking coordinator stores bookings, checks room conflicts and builds confirmation text. Its users now ask for a wording change and nothing else.

Propose a responsibility diagram with at most four domain or application classes. State a period convention, name the state that callers must not modify directly, and describe one small behaviour-preserving improvement. Give two before-and-after observations you would need to accept it, a brief review description and one remaining limitation. You need no new inheritance hierarchy or pattern.

What to take away, and what comes next

Objects become useful design units when their responsibilities, and the ways others may interact with them, are clear. Constructors and controlled operations protect invariants, and immutable values together with careful handling of collections stop callers from bypassing the boundaries you meant.

Cohesion and coupling give us a way to talk about the consequences of change. Extracting the confirmation text made one operation easier to follow without solving every responsibility the coordinator still holds, and being honest about that limit is part of good design reasoning.

Diagrams, checks, diffs and review notes each describe a different side of the same code, and they should all agree with the implementation. An AI explanation is one more description to check against that evidence.

In **Chapter 3: Agile Development and the Nature of Change**, we step back from the code to ask how a team decides what to build next: turning needs into an ordered backlog, writing stories and acceptance criteria, and learning through feedback.

Sources and further reading

The teaching scenario, implementation and exercises are original. Sources below support the identified language, API and Git details; they are not claims that the sample is a production system. Web sources were checked on 12 and 13 October 2026.

1. **Dev.java.** **Classes and Objects**. Java class, constructor and method background for Sections 2.1–2.3.
2. **Oracle.** **LocalDate: Java SE 17 API**. Date semantics and immutability. Our exclusive-end convention is a project decision, not a rule imposed on every application by `LocalDate`.
3. **Oracle.** **Objects: Java SE 17 API**. The `requireNonNull` contract used in constructors and the overlap method.
4. **Oracle.** **List: Java SE 17 API**. The `copyOf` contract and the distinction between an unmodifiable list and mutable elements.
5. **Dev.java.** **Inheritance**. Language mechanisms supporting the comparison in Section 2.5.
6. **Chacon, Scott, and Ben Straub.** **Pro Git: Recording Changes to the Repository**. Working files, staging and commits.
7. **Chacon, Scott, and Ben Straub.** **Pro Git: Branching Workflows**. Topic branches and integration practices. The course' s small-change workflow is a teaching choice.
8. **Git documentation.** **git diff** and **git switch**. Command semantics used in the chapter and companion.
9. **Conventional Commits.** **Conventional Commits 1.0.0**. One published commit-message convention, cited in Section 2.7 as an example. The course does not require this convention.
10. **Martin, Robert C.** *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall, 2002. The module' s required text; the Week 1 reading is the Part I introduction and the object-oriented design fundamentals. Publication details and the exact pages are those listed in the module descriptor and on Moodle.

Chapter 3: Agile Development and the Nature of Change

> *Agile values, backlog refinement, user story writing, and feedback loops.*

OO Development and Agile Practices
Designing, Testing and Evolving Software with AI Assistance

The demonstration goes well. The team shows the reservation coordinator from Chapter 2: a student request succeeds, a conflicting request is rejected, and a booking can begin on the day another one ends. The program does what its developers expected. Then the equipment officer asks a question nobody had planned for: "Can I check whether a camera is available before I create a reservation?"

The team can look inside the stored list, but the application has no such query. That one question has exposed a gap between checking a rule during a request and supporting the officer's real workflow, and the conversation does not stop there. Next come cancellation, a view of one borrower's reservations, and rejecting an equipment identifier that does not exist.

The team cannot do all of it at once, and it should not try. It needs a way to clarify what is being asked, compare the value of each request, be honest about what it does not yet know, and decide what useful work to finish next. That is what agile development is for: not speed for its own sake, but learning quickly while there is still little to undo. In this chapter you will watch that process work on the same small codebase. You will separate iteration from increment, explain the Agile values through concrete project decisions, and compare Scrum's framework with Kanban's attention to flow. You will write small user stories and test them with INVEST, write acceptance criteria and tell them apart from a Definition of Done, and talk about estimates and uncertainty without treating a number as a promise. What you end up with is an ordered backlog, testable stories, shared completion conditions and a visible workflow.

This chapter corresponds to **Week 2** and supports LO5. It assumes you understand Chapter 2's objects, date convention and basic review workflow, and it needs no project-management platform and no new Java framework. It supports the Week 2 readings: the Agile practices and planning chapters of Martin [7], the Scrum Guide [3] and the selected chapters of Rubin [8]. The running project stays exactly where Chapter 2 left it, because this chapter plans future work and does not claim that those features already exist. The companion has worked backlog items, board policies, exercise solutions and an optional executable bridge from acceptance examples to the existing Java implementation.

Sections 3.1 to 3.9 are the core reading. What you must do is Parts A to C of the practical activity in Section 3.10, on your team's project; the equipment-lending backlog in this chapter is the worked model, not your evidence. Everything you keep at the end of Section 3.10 (backlog, story cards, Definition of Done, workflow policy, process note, estimates and AI decision log) comes out of those three parts and needs nothing more than a text editor or a board. If you want more, write the plain-Java acceptance check described at the end of Section 3.10, then work through the questions and the transfer challenge in Section 3.12.

3.1 What do we not know yet? Uncertainty and feedback

At the start of a project we are unsure about many different things. We may not understand what a user needs, how a rule should behave, whether an integration is feasible, or how much work a change will take. It is tempting to file all of these under "requirements problems", but that hides differences that matter.

Uncertainty	Example in the equipment project	A useful next action
User need	Does the officer need an availability enquiry without reserving?	Discuss the workflow and demonstrate a small example
Domain rule	Can a borrower cancel after equipment has been issued?	Ask the responsible stakeholder and record the decision
Technical feasibility	Can the institution's catalogue be accessed?	Conduct a bounded investigation with a concrete result
Delivery	Can the team finish a reviewed query workflow in the next iteration?	Examine remaining work, capacity and assumptions

Each of these needs a different kind of evidence. A bigger estimate will not settle the cancellation policy, and more discussion will not show whether an unavailable service can be reached. Once the team has named what it does not know, it can pick an action that actually reduces the uncertainty.

Feedback comes in different kinds, too. A user can reveal a missing workflow. A compiler can reject an invalid program. A test can show that an observed result differs from what we expected. A reviewer can point out a dependency that will make the next change harder. None of them can answer every question.

Here is an example from our project. The Java implementation currently allows adjacent bookings, which matches a rule invented for this book. Suppose the equipment officer asks for an inspection day between loans. The team has not found a defect in the original comparison; it has found a possible change of policy. So we write down who proposed it, why it matters and which examples would come out differently if we adopted it.

The reason for short feedback cycles is to find out things like this while there is still little work to revise. They do not make change free. Tests, documentation and stored data may all need to change too, and the team should say so openly instead of accepting every request without discussion.

3.2 What the Agile values mean for real project decisions

You have probably heard the Agile Manifesto summed up in a slogan. Here is what it actually says, in paraphrase. It puts more weight on people working together, software that works, continuing involvement from customers and responding to change than on relying mainly on tools, exhaustive documents, contract terms or a fixed plan. It does not throw the second group away; it says explicitly that those things still have value [1].

The table below applies these preferences to our project. The right-hand column is what the first two columns are meant to avoid. None of this is wording from the Manifesto itself:

Project decision	Applying the values thoughtfully	A mistaken interpretation
Choosing a board	Select a tool that makes coordination easier	Assume installing a board creates collaboration
Recording the period convention	Keep a short, clear decision with examples	Discard documentation because the code exists
Handling a new borrowing rule	Discuss its purpose and effect with the equipment officer	Let the team invent policy privately
Planning the next increment	Use current information and revise the plan when needed	Stop planning because requirements may change

The Manifesto's accompanying principles link regular delivery to technical quality, a sustainable pace and regular reflection on how the team works [2]. That link matters in this course. Code that nobody understands makes even a small request expensive, so the design and testing work from earlier chapters is what lets a team respond to change at all.

Take the availability enquiry. A good first move is to ask whether it only has to look at the coordinator's current in-memory records, or whether it must also consult a real inventory and a maintenance schedule. The first can be a small teaching increment, and the second is a lot more work. Both can be called "check availability", but they are not the same job.

Agile work still has commitments, boundaries and evidence. What changes is that nobody expects every detail to be settled correctly on day one. Instead we keep an honest record of what we know, what we plan to deliver and what still needs a decision.

3.3 Iteration or increment? Two kinds of progress

The two words get mixed up all the time, so it helps to separate them. An **iteration** is one repeated cycle of work and feedback. An **increment** is a piece of integrated, usable capability. They describe different things about the same development.

Suppose the team changes its reservation messages after a demonstration. It has iterated on the implementation. Now suppose it adds an availability enquiry that works with the existing records and can be used through the agreed interface. That is an increment, because there is new capability somebody can use.

An isolated database table or a pile of unfinished screens may represent a lot of effort and still not be a usable increment. Technical tasks are often necessary. The point is that the team should be able to say how each one contributes to a result that someone can inspect and use.

A small route through the project

One possible route runs like this. First make the current reservation flow reproducible. Then add an enquiry that does not change any records, and after that a borrower's list of active reservations. Cancellation can come once its policy and state changes are understood. This is a suggested learning route, not a new assessment schedule.

Each addition still has to work with everything before it. If the enquiry gives the right answer but creates a reservation as a side effect, it has not delivered what was wanted. If cancellation removes an unrelated record, a friendly "cancelled" message is not enough evidence.

An increment is also not the same as a public release. A small teaching application can be demonstrated to its intended reviewers and still be unfit for the institution to deploy, so its scope and limitations should be stated openly. Releasing software to real users involves more decisions than finishing a classroom exercise.

Chapter 4 will turn the chosen work into an iteration plan. For now, practise spotting small outcomes and asking what evidence would make each one believable.

3.4 Scrum: who is accountable, what happens when, what it produces

Scrum is a framework with defined parts, and it is easy to describe loosely and get wrong. We use the November 2020 Scrum Guide as the formal reference [3]. Here is its basic structure in one table:

Element	Meaning
Product Owner	Accountable for product value and effective Product Backlog management
Developers	Create usable increments and plan their work
Scrum Master	Establishes Scrum and supports team effectiveness
Sprint	A fixed-length event of one month or less containing the other events
Sprint Planning	Establishes why the Sprint matters, selected work and an approach
Daily Scrum	Developers inspect progress toward the Sprint Goal and adapt their plan
Sprint Review	Inspect the outcome with stakeholders and consider adaptations
Sprint Retrospective	Examine how the team worked and choose improvements

Scrum ties each of its three artefacts to a commitment: the **Product Backlog** to the **Product Goal**, the **Sprint Backlog** to the **Sprint Goal**, and the **Increment** to the **Definition of Done**. The Sprint Backlog holds the goal, the selected work and the plan for delivering it. User-story templates and story points are optional practices, not parts of Scrum [3].

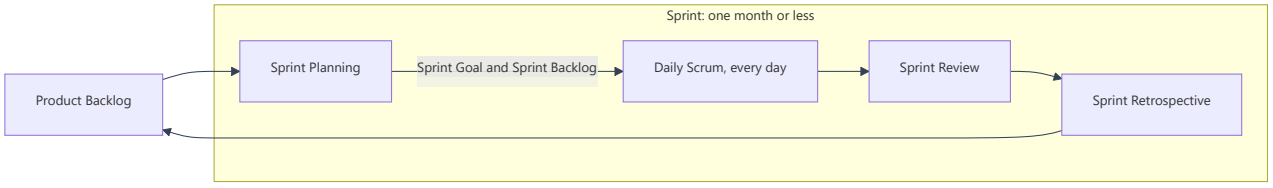


Figure 3.1. A Sprint contains all of the Scrum events. Planning selects work from the Product Backlog and sets the Sprint Goal; the Developers inspect progress at each Daily Scrum; the Review inspects the outcome with stakeholders; the Retrospective improves how the team works, and what is learned feeds the Product Backlog and the next Sprint.

What Scrum looks like for a student team

Definitions only go so far, so let us see them applied. A possible Product Goal for our learning project is: "Enable the equipment office to manage a small set of student reservations with visible rules and reliable records." It describes a useful future state without fixing every screen or method.

A possible Sprint Goal is narrower: "Let the officer check availability without changing reservations." It tells the team why its selected work belongs together. Compare it with "Finish US-02, update the README and edit three classes", which lists tasks but never says what outcome they add up to.

Picture a short coordination conversation. A developer says, "The enquiry works, but its result depends on a date convention we have not documented. I will pair with the reviewer to resolve that before adding another feature." That is the team talking about its own progress. The Scrum Guide defines the Daily Scrum as a 15-minute event where the Developers inspect progress toward the Sprint Goal and adapt their plan. It is not a status report to a supervisor [3].

At a review, the equipment officer might try the enquiry and ask whether a damaged item should ever show as available. That is a conversation about the product and what to do next. The Scrum Guide calls the Sprint Review a working session and advises against turning it into a presentation [3].

At a retrospective, the team might notice that reviews keep arriving too late. It could agree to ask for a review as soon as a small behaviour first works, and then look later at whether that helped. The retrospective is about raising the team's quality and effectiveness, including how it collaborated and checked its work [3].

When the classroom bends the rules, say so

In a course, things happen that Scrum does not describe. The module leader may supply stakeholder answers, teams may take turns facilitating, and reviews may happen inside teaching sessions. These are course arrangements. They do not change Scrum's accountabilities, and they do not make every classroom meeting a formal Scrum event. A submission date is not automatically a Sprint boundary either.

The module descriptor's later "Sprint 0" label refers to kickoff work. Treat it as a local convention for preparation, not as another Scrum event. When you write about the process your team really followed, keep that distinction visible.

3.5 Kanban: making the workflow visible

Scrum organises work around goals and time boxes. Kanban asks a different question: how does work move through the team, and where does it get stuck? The May 2025 Kanban Guide describes three jobs: define and visualise a workflow, manage work within it, and improve it. A workflow definition includes the work items, start and finish points, states, limits on work in progress, policies for moving work, and a service-level expectation. That expectation combines elapsed time with a probability, and it should be informed by historical evidence when you have it [4].

For our introductory board we use five columns:

Column	Local meaning
Backlog	Candidate work, not yet selected to start
Ready	Small enough and clear enough to begin under the team’ s local policy
Building	Implementation and associated checks are under way
Review	Work is awaiting or receiving review and verification
Done	The agreed completion conditions have been met

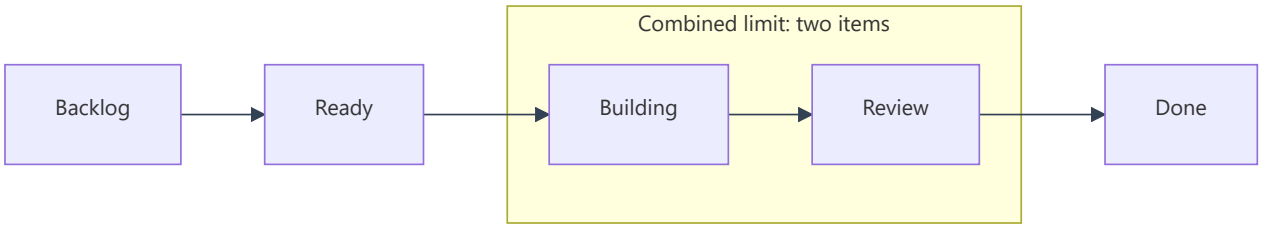


Figure 3.2. The five-column board. The start point is entry into Building and the finish point is entry into Done. Building and Review share one limit of two items, and a blocked item stays inside it.

These column names are only our example; your team can choose others. We define the start point as entry into Building and the finish point as entry into Done. For this exercise, the **combined** number of items in Building and Review may not exceed two. Blocked work stays inside that count. Moving it to a “Blocked” column that nobody counts would hide unfinished work instead of finishing it.

Imagine one item in Building and another waiting in Review. The team has reached its limit. Before starting a third item, it should ask what would help an existing one finish: reviewing together, answering an open question, or making a smaller change. A limit is only worth having if it changes what people decide to do.

A board by itself is not yet a Kanban system. The team also needs workflow policies, active management and evidence it can use to improve. For practice, a team might provisionally expect 85% of its small items to finish within five working days after starting. With no historical data, that is an unvalidated hypothesis and not a measured promise. Chapter 12 develops flow measures and forecasting, and we will not derive a delivery guarantee from this invented example.

Scrum or Kanban? What each pays attention to

Question	Scrum emphasis	Kanban emphasis
What structures the work?	Goals, Sprints and defined accountabilities/events	Explicit workflow and control of work in progress
What is inspected?	Product outcomes and progress toward goals	How work moves, waits and finishes
Can a board help?	Yes, to make selected work visible	Yes, to visualise the workflow and its policies
Can they be used together?	A Scrum team can also manage flow explicitly	Flow practices can operate within a Sprint cadence

Resist the two easy summaries: "Scrum has meetings and Kanban has none", and "Scrum is planned and Kanban is unplanned." Neither is true. What we want right now is to make the work and its completion conditions visible. Whichever approach you pick should help the team decide things, and not just give it new labels.

3.6 From "manage equipment" to a story: epics, user stories and INVEST

"Manage equipment" is far too broad to tell anyone what to implement next. It could mean stock records, borrowing, maintenance, losses, reports or purchasing. To make progress, we need a particular person, a particular outcome and a reason.

An **epic** is a convenient name for a large area of work that still has to be split up. In this book, "Manage student equipment reservations" is an epic. A **user story** is a smaller need, written from the user's point of view. A familiar form looks like this:

> As an equipment officer, I want to record an eligible, non-conflicting reservation so that the selected item is allocated for the borrower's requested period.

That sentence starts a conversation; it does not finish one. It does not say what "eligible" means, what counts as a conflict, how dates work or what happens on failure. Ron Jeffries describes a story as a short written reminder joined to conversation and confirmation [5]. We will do the same: keep a card, write down the decisions that come out of the discussion, and attach examples someone can observe.

A story is not a task

"Create a `HashMap`" is an implementation task. It says nothing about what a user needs. A story can lead to tasks about data structures, checks and interface work, but each task should stay tied to the outcome it serves.

Not every item on a backlog needs an invented user-story sentence. A task that sets up a build check can be labelled a technical enabler and justified by the delivery problem it solves. In this book `US-` marks a story and `EN-` marks an enabler. These prefixes are our local identifiers and not Agile rules.

Using INVEST as a set of questions

Bill Wake introduced INVEST as a prompt for the qualities a good story tends to have [6]. Here are the questions it raises in our domain:

Quality	Question to ask about this story
Independent	Can it provide an identifiable outcome without another unfinished story? If not, what is the dependency?
Negotiable	Which implementation details can still be discussed while preserving the need?
Valuable	Who benefits from this behaviour, and how?
Estimable	Is the team clear enough about scope and uncertainty to discuss its size?
Small	Can the team implement and verify a useful version within its intended work period?
Testable	Can we describe observations that distinguish success from failure?

Treat INVEST as a way to reason, not a six-point score that certifies a story. Real dependencies should still be written down. "Independent" should not tempt us into pretending that cancellation needs no existing reservation, and "negotiable" should not mean brushing aside a rule that the team agreed for safety or for the domain.

For the availability story, a small version could look at the coordinator's current records for one known item and one period, and leave the records unchanged. If we also added live inventory, maintenance calendars, permissions and email delivery in the same story, nobody could say clearly what was being delivered.

A backlog is an ordered list, not equal priorities

The Scrum Guide describes the Product Backlog as an emergent, ordered list, and refinement as the continuing work of breaking items into smaller, more precise ones [3]. Ordering and refinement together keep upcoming work understandable. Because the team inherits code from Chapter 2, we will also record what the code already supports next to what remains to be done.

Order	ID	Outcome or purpose	Relationship to existing code
1	US-01	Record an eligible, non-conflicting student reservation	Core behaviour exists; delivery and completion evidence must be reviewed
2	US-02	Inspect an item's availability without reserving it	No dedicated query operation exists yet
3	EN-01	Make the existing checks reproducible in an automated build workflow	Planned enabler for the kickoff work in Chapter 4

This is the start of a draft backlog, not a report that three items are finished. Review what the code already does instead of pretending to build it a second time. Once a story's conditions are fully met, keep the evidence in the history of completed work and do not count it again as new work.

3.7 Is the story finished? Acceptance criteria and Definition of Done

Two ideas answer that question, and they work at different levels. **Acceptance criteria** describe the behaviour expected from one particular item, and they should be precise enough that a second team could tell whether an implementation meets them. The **Definition of Done** describes the quality conditions that every increment must meet. Scrum makes that definition the commitment attached to the Increment [3].

Here are criteria for US-01. They use the rules from earlier chapters. All counts are taken before the new reservation is added, and every stored reservation is currently active. Dates use the exclusive-end convention from Chapter 2.

Criterion	Required observable behaviour
AC-01	An eligible request for an available item creates one stored reservation and returns its confirmation
AC-02	A request overlapping an existing reservation for the same item is rejected without changing stored records
AC-03	A request that begins at an existing reservation’ s exclusive end, or ends exactly where an existing reservation begins, is permitted when the other rules hold
AC-04	A borrower with three active reservations cannot add a fourth, even for another item
AC-05	Durations of one through seven whole days are allowed by the duration rule; zero, negative and eight-day requests are rejected without adding records

These criteria describe a controlled demonstration with non-blank, agreed sample identifiers and dates that can be represented. They say nothing about whether an item exists in a catalogue, about authentication, about persistence or about safety under concurrent use. Keep those gaps in view when you judge what kind of product the story describes.

Turning a criterion into an example

A criterion becomes much clearer once it has a concrete example. Here is one for AC-02:

```
Given CAM1 is reserved for B1 from 20 October to 22 October,
using a start-inclusive, end-exclusive period,
when B2 requests CAM1 from 21 October for one day,
then the request is rejected as unavailable,
and the original reservation remains the only stored record.
```

The format separates the starting situation, the action and what we observe afterwards. Here it is just plain text and not something a tool can run. The companion shows a short Java check against the existing coordinator. That gives you a first link between an example and real behaviour, though it is not yet a full TDD or BDD method.

Look at both observations: the rejection, and the unchanged state. A program that prints the correct rejection but still adds a reservation has not met the criterion.

A working Definition of Done to start with

Acceptance criteria say what one story must do. What must be true of every piece of work before the team calls it finished? Here is a first answer that a team could adopt as **DoD version 0.1 for its controlled teaching increment**:

1. Relevant acceptance examples have been checked, including important rejection paths.
2. The code compiles and runs using the documented commands in the agreed environment.
3. The change has been reviewed and acceptance-blocking findings have been resolved.
4. The work is integrated with the team's shared codebase and existing checks still pass.
5. Necessary usage notes, domain decisions and known limitations are current.
6. The increment can be reproduced and demonstrated within its explicitly stated scope.

This is a proposal for a team to agree on, not a university grading policy. It relies on checks you have now; automated CI gates can join it once the infrastructure exists. Do not claim that a pipeline has passed before it exists, and do not lower the agreed standard just to move a card to Done.

AC-03 belongs to the reservation story, because it defines how dates behave. Reproducible setup belongs to the shared completion agreement. Both matter. If a story's examples pass but the change is unreviewed and not integrated, it does not meet this example DoD. Appendix C has blank forms, each with a worked example, for a story card (C.2), an acceptance example (C.3) and a Definition of Done (C.4).

"Ready" answers a different question: does the team understand an item well enough to start it? Our Ready-column policy is a local agreement. It is not an extra formal Scrum artefact or event.

3.8 Estimating without pretending to know: relative estimation

Teams estimate so they can talk about work and make forecasts. An estimate is not a guarantee that a particular person will finish at a precise time. Before anyone estimates, agree what is included: only the implementation, or also the review, checking and documentation needed to reach Done?

We use **story points** as an optional, local, relative scale. A team might use 1, 2, 3, 5 and 8 to compare items, weighing effort, complexity and uncertainty together. Points are not hours, and the same number in two teams need not mean the same amount of work.

Say a familiar display adjustment is the team's reference item at two points. A small availability query may seem bigger, because it needs a read-only operation, several date examples and a new way of interacting. One learner estimates three and another eight. Averaging those two numbers straight away would throw away the most useful thing we have, which is the reason they disagree.

The second learner may be assuming the query covers maintenance schedules. Once that assumption is spoken aloud, the team can narrow the story or record a separate requirement. If the uncertainty is about an outside catalogue whose interface nobody knows, a short investigation is often more useful than a bigger number.

An investigation like that should have a bounded question and a defined output, for example: "Determine whether a read-only sample catalogue can be accessed and record the fields it provides." It should not turn into an open-ended task called "research the system".

When you practise, estimate only the items you have clarified, and write down the assumption behind each estimate. If an important question is still open, leave the item unestimated. Different, well-reasoned estimates are fine. Claiming certainty without evidence is not.

Do not rank students by the points they complete, turn points into individual productivity targets, or assume that doubling the team doubles the delivery. Chapter 12 looks at measured progress and flow, and Chapter 4 combines the selected work with the team's real capacity to make an iteration plan.

3.9 Challenging a story with AI without letting it invent policy

An AI assistant can suggest assumptions you missed and criteria you could use, but it has no authority over the equipment office's rules. The exercise is to judge what it proposes against the context your team has agreed.

Chapter 15 gathers the AI workflows that recur in this book. This activity is complete on its own, and the ready-made suggestions and the table below are all you need.

	For this activity
Your goal	Decide which suggested assumptions and criteria belong in the story, which do not, and which need a stakeholder
You start with	US-01, AC-01 through AC-05, and Chapter 2's scope
What is in bounds	Classify the suggestions; do not change the policy, the code or the story's scope on your own
How to check	The stated rules, the Chapter 2 code and checks, and the story's scope
How to decide	**Accept**, **reject for this story**, or **clarify** with a named person or decision
No AI account?	The ready-made suggestions below are enough

The suggestions below were written by the author for this exercise. They are not a captured response from a real model:

1. A request for the same item and an identical period must be refused.
2. A request that begins exactly when an existing period ends must be refused.
3. Every confirmation must be emailed within thirty seconds.
4. A refused request must leave stored reservations unchanged.
5. Specify how start dates that are already in the past are treated.
6. A deposit payment must be taken before any reservation can be confirmed.
7. Specify whether cancelled reservations count toward the borrower's limit.

Sort each suggestion into one of the three groups and give a reason. To "reject" a suggestion can mean that it contradicts a rule or adds scope nobody approved. It does not have to mean the idea could never be useful. And a question about a future feature does not automatically block today's small story.

If your course allows it, you can also try this prompt with a live assistant:

> Review this story against the supplied policy and current project scope. Suggest missing assumptions, edge cases and candidate acceptance criteria. Label each suggestion as derived from a stated rule, an open question or a possible scope extension. Do not silently change the policy or claim a feature is already implemented.

Keep a record of what you accepted, rejected and deferred, and point to the rule or the code as your evidence, not to how confident the assistant sounded. The ready-made suggestions mean you can do the whole activity without an AI account.

3.10 Try it yourself: build a draft backlog

This activity is about your team's project. If your team uses equipment lending, start from the Chapter 2 coordinator, the three backlog entries in Section 3.6 and the rules from earlier chapters. If your team chose another of the provided domains, apply the same steps to that domain's starter codebase and its first user-visible capability. For an ML model-serving tool, for example, you might write "as an analyst, I want a prediction for a supplied record so that I can check the result". The transfer challenge shows how to adapt the steps. The work prepares you for the Week 3 kickoff; it does not change the module's assessment dates or weights.

Part A: define and refine the work

Write a short proposed Product Goal. Extend the draft backlog to six items by considering a borrower's active-reservation list, cancellation and catalogue-identifier validation. Give the items stable IDs, and note any dependencies or unresolved decisions. Put the items in order and explain why the first two come first.

Record which behaviours the code already supports, which need new implementation and which you cannot yet judge. Do not label an inherited feature Done just because a demonstration prints the text you wanted.

Refine US-02 into a small story with at least four acceptance examples. State that asking about availability must not create or remove reservations. Decide which identifiers and period rules belong to its controlled scope, and mark any policy that is still unresolved instead of guessing it.

Part B: agree on completion and workflow

Propose a Definition of Done and explain how it differs from your story criteria. Draw a board with the five example columns, or with an alternative you can justify. Define its start and finish points, its movement policies, its WIP control and how it handles blocked work. If you have no data, mark the service-level expectation as provisional. Then write a short process note that says which Scrum elements your team uses as the Guide defines them, which it adapts for the course, and which optional practices (stories, points, a board) it adopts.

The snapshot below is **hypothetical** and only gives you a decision to practise on. It says nothing about how far your project has really got:

Ready	Building	Review	Done
US-03, US-04	US-02	EN-01, blocked by a setup problem	US-01

The combined Building-plus-Review limit is two. Decide what the team should look into next, and explain why moving the blocked card outside the counted area would misrepresent its work.

Part C: estimate and critique

Pick two clarified items and propose relative estimates with their assumptions. Name one item that needs clarification or an investigation before anyone can size it. Do the AI activity and record what you decided about each suggestion.

Finally, compare these two invented pieces of advice:

> Advice A: Freeze every detailed story before any implementation begins.

> Advice B: Start implementation immediately and leave acceptance conditions until the demonstration.

For each one, say what problem it is trying to solve, what risk it carries in this project, and what working agreement would be easier to defend. This practises the critical reading introduced in Chapter 1; it is not a claim about any particular author's position.

What to keep, and how to know you are done

Keep the proposed goal, the ordered six-item backlog, two refined story cards, the acceptance examples, the Definition of Done, the workflow policy, the process note, the estimates with their assumptions and the AI decision log. Then swap your top story with another team.

You are done when the other team can explain the intended outcome, choose meaningful inputs and predict what they would observe, all without inventing any essential policy. Write down any question they could not answer, and either revise the story or deliberately put the decision off.

If you want more, write a plain-Java check for your top story's examples in the style of the companion's `Chapter03AcceptanceExamples`, and confirm that it fails when you introduce a deliberate fault.

3.11 Habits that sound agile but are not

Mistake	Consequence	Correction
Treating Agile as the absence of planning	Dependencies and deadlines remain invisible	Plan at useful detail and revise from evidence
Treating a story template as the full requirement	Eligibility and failure behaviour remain ambiguous	Add discussion, decisions and examples
Calling every item highest priority	The team has no usable order	Explain why one outcome should be pursued before another
Treating acceptance criteria as the whole DoD	Correct examples hide unreviewed or unintegrated work	Check behaviour and shared quality conditions
Starting new work whenever review is blocked	Unfinished work accumulates	Make the blockage visible and help existing work finish
Treating AI suggestions as stakeholder decisions	The scope or policy changes without agreement	Classify each proposal and justify its disposition
Counting inherited working code as newly delivered work	Progress becomes misleading	Record the baseline and the remaining work honestly

3.12 Questions to think with

Concept questions

1. Explain how an iteration can improve an existing feature and how an increment adds usable capability. Give an equipment-lending example of each.
2. Choose one Agile value and explain a decision that respects both sides of its preference rather than discarding one side.
3. Distinguish Sprint Review, Sprint Retrospective and Daily Scrum using the decisions each helps the team make.
4. Explain the difference between acceptance criteria, Definition of Done and the local Ready-column policy.

Applied diagnosis

5. A team marks an availability story Done because its method returns a correct answer for one date. The enquiry also adds a reservation, and the change has not been reviewed. Identify two separate completion failures and the evidence needed to investigate them.

6. Two items already occupy the team' s limited active workflow. One is blocked in Review. A developer moves it to an uncounted column and starts a third item, claiming the WIP limit is still respected. Explain what the board now hides and propose a useful next action.

Transfer challenge

A laboratory room-booking team receives the request "Make booking easier." Develop two small user stories and one justified technical enabler. Give one story four acceptance examples, including a failure case and an observation about unchanged state. Propose a shared Definition of Done and a simple WIP policy.

Identify one question that needs a stakeholder answer and one technical uncertainty that might need an investigation. Explain how the team could obtain feedback without pretending that every room-booking rule is already known.

What to take away, and what comes next

Agile development pairs a willingness to learn with visible work and clear quality expectations. Short cycles pay off when feedback changes what the team understands, and good design and checks are what let it act on that feedback safely.

Scrum and Kanban are two ways of organising the work. Their vocabulary is there to clarify who is responsible for what and which decisions are being made. A board, a meeting or an estimate has little value if the team cannot explain how it changes the next thing they do.

What you now have is a draft backlog with clear stories, observable examples, shared completion conditions and a workflow. It separates what the code already does from new work, and it records the decisions that are still open.

In **Chapter 4: From Requirements to an Iteration**, we use these documents to define project scope, slice features, prepare for the kickoff milestone, and set up a reproducible repository with an initial CI workflow.

Sources and further reading

All project scenarios, backlog entries, board policies, exercises and ready-made AI examples are original teaching material. The numeric rules and Java baseline continue the preceding chapters. Sources were checked on 12 and 13 October 2026; the named guide versions keep the conceptual baseline explicit.

1. **Authors of the Agile Manifesto.** **Manifesto for Agile Software Development**, 2001. Section 3.2 paraphrases its preferences; read the original wording alongside the project applications.
2. **Authors of the Agile Manifesto.** **Principles behind the Agile Manifesto**, 2001. Delivery, technical quality, sustainable work and reflection.
3. **Schwaber, Ken, and Jeff Sutherland.** **The Scrum Guide**, November 2020. Formal accountabilities, events, artefacts and commitments, including the Daily Scrum, Sprint Review, Sprint Retrospective and Product Backlog sections cited in Sections 3.4 and 3.6. The guide mentions neither user stories nor story points. Local teaching conventions and optional practices are identified separately.
4. **Kanban Guides contributors.** **The Kanban Guide**, May 2025. Workflow, WIP controls and flow. The example limits and provisional expectation in this chapter are invented for practice, not measured recommendations.

5. **Jeffries, Ron.** **Essential XP: Card, Conversation, Confirmation**, 2001. Stories as more than written cards.

6. **Wake, Bill.** **INVEST in Good Stories, and SMART Tasks**, 17 August 2003. Original explanation of INVEST.

7. **Martin, Robert C.** *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall, 2002. The module' s required text; the Week 2 reading is its Agile practices and planning chapters. Publication details as listed in the module descriptor.

8. **Rubin, Kenneth S.** *Essential Scrum.* Addison-Wesley, 2012. A recommended text; read the selected chapters named for Week 2. Publication details as listed in the module descriptor.

Chapter 4: From Requirements to an Iteration

> *Acceptance criteria, Definition of Done, Arrange-Act-Assert testing, and capacity planning.*

OO Development and Agile Practices
Designing, Testing and Evolving Software with AI Assistance

By now the team has a backlog of six items and a working Java reservation coordinator, and it is time for the kickoff meeting. Everyone arrives with an idea. One member wants a mobile interface. Another proposes a database. A third would rather finish every story before anyone outside the team sees a line of it. Everyone is busy and everyone is keen, and yet nobody in the room can say what the equipment officer will actually be able to see at the end of the next iteration.

The trouble is not a shortage of ideas. It is that the team has no shared boundary and no credible route from an idea to evidence. A backlog says what might be valuable; an iteration plan has to say what the team will try to achieve, what it can realistically finish, and how another person will know that it worked.

This chapter builds that route. You will start by writing a project vision with explicit scope, assumptions and constraints, and by slicing one feature into a small increment that gives a user something observable. You will turn the slice into an iteration plan that pays for implementation, review and verification out of realistic capacity, and you will agree how the team shares a repository and reviews each other's changes. Then you will make it concrete: run a starter build and its checks, see for yourself that the checking process notices a deliberate failure, and gather traceable evidence for the kickoff milestone.

The running project remains the **Campus Equipment Lending System**, and its reservation behaviour is preserved. Availability enquiry is still proposed work. Preparing delivery evidence is real progress, but it must not be reported as newly implementing behaviour that the starter already has.

The chapter corresponds to **Week 3** and supports LO5 and LO6, as preparation for Milestone 1. The descriptor's term **Sprint 0** is used here as the local name for the kickoff work; it is not a formal Scrum event. The chapter gives preparation guidance and creates no new assessment weights, submission rules or module wording. It supports the Week 3 readings: the planning and testing chapters of Martin [6] and the sprint-planning chapters of Rubin [7].

You will need the Java code as Chapter 2 left it, Chapter 3's ordered backlog and Definition of Done, and basic Git operations, plus a Java 17 JDK, Git and Bash (on Windows, use Git Bash or WSL). You do not need prior TDD experience. The separate companion supplies all Java listings, repository files, commands, model answers and an offline AI exercise solution.

Read Sections 4.1 to 4.7 in order, and run the supplied checks while you read 4.6. What you must do is steps 1 to 8 of the practical activity in Section 4.10, on your team's project; together they produce the Milestone 1 evidence. The supplied lending repository is a worked model and a starting point, not evidence of your team's own work: your team's availability plan, board, review record and hosted CI runs are what count. Hosted CI needs a repository your team can use, so start steps 6 and 7 early. If you want more, try the AI activity in Section 4.8, the questions in Section 4.11, the transfer challenge in Section 4.12 and the extension at the end of Section 4.10.

4.1 What are we building? Vision, scope and constraints

Start with an outcome you can finish

A vision says whose problem the project addresses and what improvement would matter to them. Its job is to guide choices when time is short. “Build a complete equipment platform” guides almost nothing, because nearly any feature can be justified under it.

For this stage of the book’s project, we will use the following working vision:

> Help an equipment officer demonstrate clear reservation decisions for a small, controlled set of student borrowers and equipment, so that accepted requests can be inspected and conflicting requests do not create overlapping records within one running coordinator.

This is deliberately narrower than a production service. It names an actor, an observable improvement and the setting in which we can judge the claim. It does not promise authentication, a maintained inventory or safe operation by several officers at once. The wider product can include borrowing and returns later, while the first evidence stays small enough to inspect.

A Product Goal describes a future state of the product that the team is working toward, and an iteration goal gives the next stretch of work some coherence. Neither is a list of every task. In Scrum, the Sprint Backlog combines the Sprint Goal, the selected backlog items and an actionable plan for delivering them [1]. Our worked plan borrows that distinction without assuming that a classroom team has adopted every element of Scrum.

Draw the boundary where everyone can see it

Scope helps most when it makes decisions easier. A short table can separate what the system does now, what we have selected to work on, and what we have deliberately put off.

Area	At this point in Chapter 4	Planning consequence
Reservation request	Existing Java behaviour accepts eligible, non-conflicting requests	Verify and demonstrate it; do not rebuild it for credit
Availability enquiry	Proposed US-02; no query method or interface yet	Refine and plan a thin next slice
Storage	In-memory records in one coordinator	Restarting loses records; demonstrate in one process
Identity and equipment	Nonblank identifiers; controlled synthetic examples	No claim of directory or catalogue validation
Loan period	Whole calendar days; start inclusive, end exclusive	Label dates and test adjacency
Cancellation and completed loans	Not implemented; all stored reservations count as active	Clarify lifecycle before adding those stories
User interface	Scripted console demonstration; proposed wireframes	No web or mobile implementation required for kickoff
Concurrent use	Sequential teaching example	No claim that a check prevents races across processes

Constraints and assumptions are not the same thing. Java 17 compatibility is a technical constraint we chose for this starter. Using synthetic borrower identifiers is a teaching boundary. Assuming that an officer can recognise a sample item is an assumption about the demonstration. An unresolved rule about past dates is an open question. If we call all four "requirements", we lose track of who owns each one and what follows from it.

Every assumption should come with a reason to revisit it. For example: "For the controlled demo, the operator uses CAM1 and MIC1 from a supplied list of sample items. Before opening input to arbitrary equipment, clarify the catalogue authority and unknown-item behaviour." The trigger ties a temporary simplification to the moment it stops being good enough.

Write down decisions, and when to revisit them

A useful decision record says what the question was, what was chosen, why, what follows from it and when to look at it again. It does not need to be a long report. "Use in-memory records for kickoff because the immediate question concerns reservation decisions; records disappear on restart; reconsider when a demonstration must survive restarting" is enough to guide a reviewer.

Compare that with "Database later." It records neither the limitation today nor the condition that would make the work necessary. The team might put off essential persistence for ever, or bring it in at once for no reason. A clear trigger lets the plan change when the evidence changes.

You do not have to settle every future policy before you run the starter. Resolve the questions that decide how the next slice behaves, and keep the others visible with an owner and a date by which someone will decide. That keeps work moving and stops a temporary assumption from quietly becoming a permanent rule.

4.2 Sketching the screens and walking through examples

A wireframe is a question made visible

Before anyone spends time on presentation, a lightweight wireframe lets users and developers talk about the information, actions and outcomes. In this book a wireframe may be a table, a sketch or a simple sequence of views. What matters is the questions it brings up, not how polished it looks.

Think about the journey of a reservation request. What does the officer enter? Does "end date" include that day? What appears when the equipment is unavailable? Does a failed request leave a record? Can the officer tell a successful reservation from an availability observation?

The tables below are **proposed interface sketches**. They are not screenshots of screens that exist.

View A: Request a reservation

Element	Example or behaviour
Borrower identifier	B1
Equipment identifier	CAM1
Start date	2026-10-20
Duration in days	2
Period explanation	Includes 20 and 21 October; ends before 22 October
Action	Request reservation
Scope note	Controlled demo; records last only for this session

View B: Request outcome

Outcome	Information shown
Confirmed	Reservation R1, B1, CAM1, `[2026-10-20, 2026-10-22)`
Item unavailable	Request rejected; no new reservation stored
Limit or duration failure	Request rejected; no new reservation stored
Malformed input	Proposed interface should explain the field to correct

The current coordinator returns confirmation or rejection strings and throws exceptions for certain invalid arguments. A future interaction layer will have to turn malformed input into a useful response, and drawing that response does not mean it is implemented. Notice, too, that the coordinator today gives one message for both a borrower-limit failure and a duration failure. A sketch that shows two precise explanations would introduce new work, and the backlog should say so.

View C: Availability enquiry, proposed US-02

Element	Example or behaviour
Equipment identifier	CAM1 from the controlled set of sample items
Start date	2026-10-22
End date, exclusive	2026-10-23
Action	Check availability
Result	Available, unavailable, or an identifiable invalid-period outcome
Explanation	An enquiry creates no reservation and does not guarantee a later request will succeed

Someone can understand this sketch without agreeing on a particular toolkit. That leaves the team free to use a thin console interaction while it learns about the domain. If we chose a web framework now, we would add configuration and presentation tasks and still not have answered the date-boundary question.

Walking through the examples

Let us follow one example from the sketch down to the stored state. Start with no reservations. Submit B1, CAM1, 20 October, two days, and expect one stored reservation that ends, exclusively, on 22 October. Submit B2, CAM1, 21 October, one day, and expect a rejection with the original record kept. Submit B2, CAM1, 22 October, one day, and expect a second stored reservation.

The third step matters because “no overlap” is ambiguous without it. If an officer expects a collection on 22 October to conflict with the earlier booking, then either the period convention or the operational turnaround rule needs discussion. Do not quietly add a cleaning interval to the current rule. Record it as a candidate requirement, with examples.

For an availability enquiry, record both the answer shown and the unchanged state. A program that says “available” after quietly creating a reservation has not delivered the story. Seeing that an item is available is also different from being allowed to reserve it, because a borrower may already have three active records. The proposed query does not establish whether a borrower is eligible.

Wireframes, examples and code are three descriptions of the same intended behaviour. When they disagree, that is useful evidence. Settle the disagreement openly and do not pick whichever description is easiest to implement.

4.3 Prioritisation and vertical slicing: what first, how thin?

Where the backlog stands now

Chapter 3’ s order covered US-01, US-02 and EN-01, and its Part A asked you to extend it to six items. This chapter uses the extended order below. It keeps Chapter 3’ s identifiers and says more clearly what already exists and what remains.

Order	Item	Current interpretation
1	US-01: Record an eligible reservation	Core behaviour exists; confirm interaction and delivery evidence
2	US-02: Inspect availability without reserving	New user outcome to refine and forecast
3	EN-01: Automate build and checks	Enabling work supplied in this chapter; hosted evidence still required
4	US-03: Inspect a borrower’ s active records	Filtering and interaction remain new work
5	US-04: Cancel a confirmed reservation	Authority, lifecycle and count effects need clarification
6	US-05: Reject unknown equipment identifiers	Depends on a catalogue source and failure policy

The order shows our current judgement. It is not an instruction to finish every task of one item before looking at another. EN-01 supports reliable delivery of US-02, so it makes sense to prepare automation during kickoff, as long as you explain that dependency openly. What you should not do is quietly move every technical improvement ahead of feedback from users.

Catalogue validation has a low priority for now, and that is defensible only inside the controlled set of sample items. A real deployment would make the team reconsider the order. “Low priority” is a decision for a particular context; it does not say that unknown equipment is harmless.

Cut the feature by what the user sees

A **vertical slice** goes through every part of the system it needs in order to deliver one outcome someone can observe. It does not have to use a graphical interface, a database or a network. For this project, a console request that reaches the coordinator and shows a verified decision is already a thin journey through interaction, domain behaviour and storage.

Compare a horizontal split such as “all screens first, then all services, then all storage”. It can leave a lot of finished-looking work and nothing to demonstrate. Those activities may be real implementation tasks, but putting them on separate tickets does not turn them into useful increments.

For US-02, a defensible first slice is:

> The officer supplies one sample item and a valid date interval, receives an availability answer based on current reservations, and can verify that enquiry left every stored reservation unchanged.

The slice also needs a recognisable response when the interval is empty or reversed. It can put off calendar styling, searching several items, live inventory integration and availability across several servers. It cannot put off overlap semantics or keeping state intact, because those decide whether the promised outcome is true.

The examples below carry Chapter 3's query criteria forward.

ID	Given and enquiry	Expected outcome
AQ-01	Empty coordinator; CAM1 `[20 Oct, 22 Oct)`	Available; records unchanged
AQ-02	CAM1 reserved `[20 Oct, 22 Oct)`; enquire `[21 Oct, 23 Oct)`	Unavailable; records unchanged
AQ-03	Same reservation; enquire `[22 Oct, 23 Oct)`	Available; records unchanged
AQ-04	Same reservation; enquire about sample item MIC1 for `[20 Oct, 22 Oct)`	Available; records unchanged
AQ-05	Equal or reversed start and end	Identifiable invalid-input outcome; records unchanged

All dates above are in 2026. "Records unchanged" means that the contents are kept, not just the count. A query that swaps one reservation for another can leave the size the same and still be wrong. The exact query API and the way invalid input is reported are design choices to settle before implementation; the existing `reserve` method is not a query implementation.

What has to come with the slice

A feature is small enough when the team can understand it and verify it within the iteration, integration included. You can make a feature smaller by dropping meaningful rejection behaviour, but the result then looks like success when it is not. A reservation feature that accepts requests and ignores conflicts is not a safe reduction of this outcome.

Keep the necessary work apart from the attractive extras. For US-02, interpreting the interval, inspecting the relevant reservations, reporting the result and checking that nothing changed are necessary. A calendar widget is optional. A persistent store becomes necessary only if the chosen outcome needs data to survive a restart. Drawing this line lets the team negotiate scope without treating quality as negotiable.

Use epics to group the direction of travel, not to make the iteration bigger. "Reservation decisions" can hold US-01 and US-02, and "Record lifecycle" can hold US-03 and US-04. EN-01 supports delivery across them. An epic does not call for an architecture layer or a Java package of the same name.

4.4 What can we really finish? Release outlook and iteration planning

An outlook is not a promise

A release outlook tells people the likely steps toward the product goal, and it should show the uncertainty. For the book's project, kickoff makes the baseline reproducible; the MVP stage develops a small reservation and loan flow; later stages bring in integration boundaries and respond to a changed requirement. These are the book's teaching stages. They are not evidence that every team can deliver the same scope on the same dates.

If precise forecasts would be invented, use a short outlook with "now", "next" and "later". Leave the module's official milestone schedule as it is. A planning table cannot create an assessment deadline, or promise that a poorly understood story will fit into a particular week.

The goal for the current kickoff is:

> Make the reservation baseline reproducible and reviewable, and prepare a bounded availability slice for the next iteration.

The **candidate goal for the next iteration**, carried from Chapter 3, is:

> Let the officer inspect availability for one sample item and interval without changing reservations.

These are two different goals. Chapter 4 provides the starter materials for the first and plans the second; it does not claim to have implemented US-02.

Counting the hours the work will really take

The figures below are **invented planning data**. They are not official contact hours or the measured productivity of a team. Suppose four members report 6, 6, 5 and 5 person-hours available for the next iteration, which gives 22 person-hours. Keep four of them for agreed coordination, integration support and routine project administration, which leaves a planning budget of 18 person-hours.

Forecast work for the next iteration	Person-hours
Resolve the query contract and update examples	2
Implement the read-only enquiry behaviour	4
Add the thin interaction and invalid-input response	3
Add and run query checks alongside regression checks	3
Review, integrate, update notes and rehearse demonstration	2
Selected work	**14**
Unallocated capacity within the 18-hour budget	**4**

The estimates are inputs for discussion, each expressed on its own in person-hours. They are not converted from Chapter 3’ s story points. Two people pairing for one hour use two person-hours. Review and correction have to fit somewhere in the budget, and calling them “after development” does not make them free.

The four spare hours are room for uncertainty, and they are not a promise to pull in another feature. If the team finds that the query needs a policy decision nobody expected, it can spend that time resolving the risk while protecting the goal. If everything finishes early, improve the evidence or choose additional work on purpose.

Does the calendar, and who knows what, allow it?

Eighteen hours in total does not prove the plan is feasible. Suppose the only person who can explain the coordinator is unavailable until the last day, and the reviewer is free only at the start. The work cannot flow the way the arithmetic suggests. So ask when people can actually work together, and whether knowledge has to be shared earlier.

Make ownership visible without creating permanent specialists. A task owner makes sure that progress and questions stay visible; it does not mean nobody else may help. Agree on an early pairing session about the query contract, and a review window before the final demonstration. A little coordination can stop the whole slice from waiting until the end.

Look at the plan again when the evidence changes. If four hours of implementation turn into nine, talk about the forecast and the scope that is left. Do not hide the overrun by putting off checks or calling the interaction "almost done". In Scrum, scope can be clarified and renegotiated with the Product Owner as the team learns more, while the Sprint Goal is protected [1]. A classroom adaptation should make its decisions visible in the same way.

When you have to replan, a practical question is: "What complete, still useful outcome can we verify with the capacity left?" Sometimes the answer is a smaller interface. Sometimes the behaviour that is still unresolved means the item cannot honestly be called complete. Being honest about unfinished work is more useful than a green status that conceals it.

4.5 How the team works together: agreements and review

Agreeing how work moves

Chapter 3's board used Backlog, Ready, Building, Review and Done, with a combined work-in-progress limit of two across Building and Review. Keep that policy for the worked team. A blocked item still takes up a slot, and if review is blocked, helping it finish is usually more useful than starting a third item.

A team agreement should answer practical questions: where requirements live, how questions are raised, who reviews a change, what evidence reviewers need, and how unfinished work is reported. The tool for the board can be swapped for another. A shared Markdown table works if updates are visible and ownership is clear.

Start from what your team has already decided: the issue board and branch policy from Part D of Chapter 2, the Definition of Done from Chapter 3, and your process note saying which Scrum elements you use as defined and which you adapt for the course. Change them where experience shows a problem, and record why.

Use small, short-lived branches for this chapter's review exercise. One branch should hold one coherent change that another person can understand. "Set up every tool and rewrite the coordinator" is hard to review, because functional changes and workflow changes get tangled together. A branch that adds the checking script and its documentation has a clearer purpose.

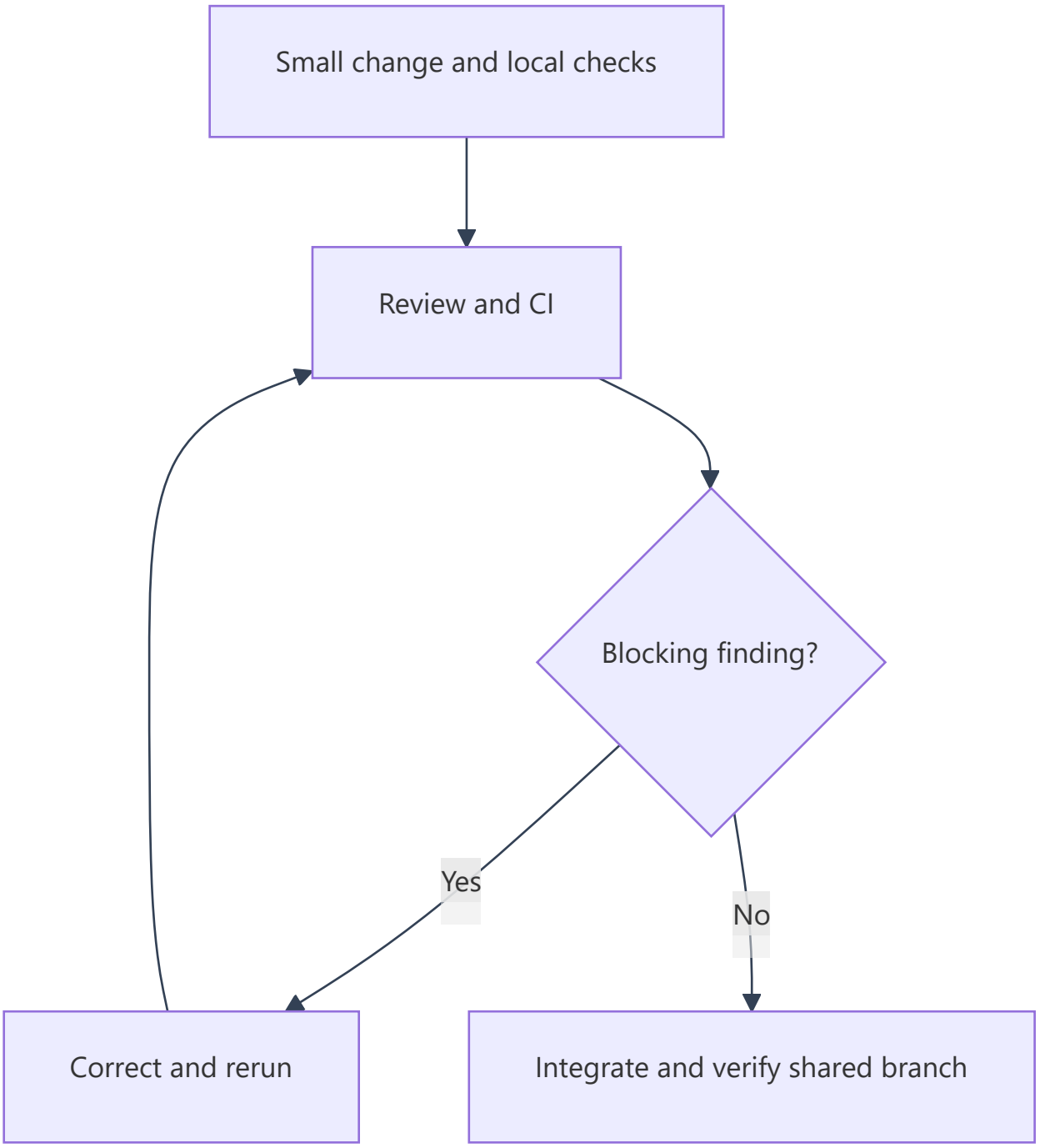


Figure 4.1. The team's review-and-CI loop for one small change. A blocking finding sends the change back to the author and the checks run again; only a change with no blocking finding is integrated, and the shared branch is verified after integration.

The figure shows this chapter's team policy, and Git does not guarantee it automatically. A repository does not enforce review just because the README asks for it. Where your hosting service supports it, configure the shared branch to require the agreed check and a review. Where that is not possible, say that the team follows a manual policy, and keep the evidence of its reviews.

What a good review looks at

A useful review starts from the behaviour that was intended. The reviewer then follows the relevant code or documentation and asks whether the evidence supports the claim. “Looks good” on its own does not say which risk anyone examined.

For a CI change, ask whether a compilation failure stops the job, whether the check programs really run, and whether a failed assertion reaches the result of the job. When US-02 comes later, look at whether the query changes state or depends on a borrower’s eligibility. A comment about code style should not distract from a defect in behaviour.

The author resolves blocking findings and reruns the affected checks, and the reviewer looks at the correction. Keep the real feedback and the real response; do not invent a review conversation to fill in a milestone template. If you work alone, a documented self-review is fine for practice, but it is not an independent review and should be labelled as such.

Growing the Definition of Done

Carry forward Chapter 3’s requirements: checked examples, reproducible execution, a real review, integrated code, updated notes and a demonstration you can repeat. For this stage, add that the agreed automated check has passed on the integrated revision and that the team can find its result.

Acceptance criteria stay specific to a story, while the Definition of Done covers the quality and delivery conditions that apply to all work. A passing pipeline is one part of those conditions. It does not show that a wireframe was understood, that a product decision was approved or that a useful outcome was delivered. Appendix C has forms for the iteration plan of Section 4.4 (C.6), a team agreement (C.7), a pull-request description (C.8) and the evidence note of this chapter’s companion (C.9).

4.6 A starter build and CI you can run

One command that means something

Continuous integration needs a check that can run without depending on one developer’s open IDE. Our starter uses Java 17, Bash and the supplied executable Java checks, and it has no external Java dependencies. Maven and JUnit remain the planned build and testing tools for the fuller testing stage; for now we keep the existing plain-Java checks so that kickoff does not require learning a test framework first. On Windows, run the script from Git Bash or WSL, because PowerShell cannot interpret it. Three terms will come up below. Temurin is one distribution of the OpenJDK. YAML is the plain-text format in which a hosted workflow is written. An exit status is the number a command returns to the shell, where zero means success and anything else means failure, which is how a script or a CI job learns that a step failed. Appendix B has the setup.

The code file supplies a conventional source layout and a script. If you are continuing from the flat Chapter 2 directory, the move is mechanical: application classes go to `src/main/java`, the check classes go to `src/test/java`, and their contents do not change. The classes stay in the unnamed package, so this is only a change of layout. It is the directory layout Maven expects, so a later move to Maven changes the build file and the checks, not where the sources live. From the repository root, run:

```
bash scripts/check.sh
```

The script clears generated class files, compiles the listed source directories, runs Chapter 2′ s 34 checks and Chapter 3′ s five acceptance example groups, then prints a completion message. Compilation targets Java 17 through `--release 17`, and generated classes go into a separate output directory [2]. The script must stop when compilation or either checking program fails, and the final message must be reached only after success.

The five acceptance examples cover eligible reservation, overlap rejection, adjacency, borrower limit and duration boundaries. They do not test US-02, because that feature is not implemented. So a passing run tells us something limited: the supplied reservation checks passed against this revision in the recorded environment. It is not a claim of complete correctness.

After a successful build, run the scripted journey:

```
java -cp build/classes Chapter04Demo
```

It confirms the first request, rejects the conflict, confirms the adjacent request and reports two stored reservations. The values are fixed synthetic examples; this is not an interactive application. Run it in a fresh process to get the same starting state again.

Line endings matter for shell scripts. Git on Windows may convert LF to CRLF when it checks files out, and a CRLF script fails under WSL or Linux with an error such as `set: pipefail: invalid option name`. The supplied repository therefore includes a `.gitattributes` file that keeps `*.sh` files in LF form. Check that it is committed before a teammate tries a fresh clone.

Reading a check: Arrange, Act, Assert

You do not have to write checks this week, but you should be able to read one. Every supplied check has the same three steps. Here is the AC-02 block from `Chapter03AcceptanceExamples`, with the steps labelled:

```
// Excerpt from Chapter03AcceptanceExamples.java; the comments are added here.
// Arrange: the coordinator already holds B1's reservation of CAM1 (made by AC-01).
// Act: B2 asks for CAM1 on an overlapping day.
String conflict = coordinator.reserve("B2", "CAM1", day.plusDays(1), 1);
// Assert: the request is refused, and nothing new was stored.
require(conflict.equals("Rejected: item unavailable."), "AC-02 rejection");
require(coordinator.reservations().size() == 1, "AC-02 unchanged count");
```

Arrange builds the situation, Act performs the one operation under test, and Assert states what must be observed, including what must not have changed. A check that asserted only the message would pass for a request that was refused but still stored. Chapter 7 names this structure Arrange–Act–Assert and builds on it.

Running the same checks on a hosted service

The separate code file supplies a GitHub Actions workflow that checks out the repository, installs Temurin Java 17 and runs the same Bash command. GitHub stores workflows under `.github/workflows`, and its documentation also supports running the project's own build commands [3]. The action versions in the supplied example were checked against their official repositories [4, 5].

GitHub Actions is one concrete teaching route. If your institution has another CI service, that is fine as long as it checks the revision you intend, runs the agreed command and keeps results someone can inspect. A badge or a YAML file alone is not a passing run. Confirm that the workflow really starts and that its compilation and checking steps run.

Record the revision, environment, command, result and a link or exported log. For a pull request, identify the revision that was tested; after integration, keep the passing result of the shared branch too. A check on a branch can be valid and still become stale after another change is integrated.

The example uses a Java major version and major-version tags for the actions, because that is easier to read. Those selectors can resolve to newer releases, so record the versions they resolved to in your evidence. Stricter pinning of dependencies, wider coverage of environments and delivery stages belong in Chapter 13.

Making sure a failure really shows up

Start with a passing run. On a clearly labelled disposable branch or copy, change one acceptance check so that it expects two records after the first successful reservation. That expectation is wrong on purpose, because the valid behaviour stores exactly one. Run the same checks and expect a nonzero result that names AC-01.

This shows that an assertion failure travels through the runner and reaches the result. Put the correct expectation back and run again. On a hosted practice branch, keep both the failed run and the restored passing run, and never merge the deliberate defect into the shared baseline.

A failed download or a missing JDK does not prove the same thing. It only shows an environment failure that happened before any assertion was reached, so read the log and find the step that really failed. In the same way, replacing a test command with `echo passed` can give you a green workflow that checks nothing.

What the author has run: the supplied compilation, 34 checks, five acceptance groups, demonstration and deliberate failing-check experiment were executed locally. The GitHub Actions workflow is supplied for learners to run; no hosted execution or independent team review is claimed. Completing the milestone's hosted CI evidence requires an observed run in the team's repository.

4.7 Getting ready for Milestone 1

Pulling the evidence together

Milestone 1 joins the direction of the project to a starting point that can actually run. Its documents should agree with one another. A vision that promises live inventory, a backlog that assumes sample identifiers and a demonstration that uses invented equipment without explanation would leave a reviewer unable to tell what success means.

The map below is a way to find your way around the supplied milestone, and it is not an extra assessment rubric.

Evidence	Question it should answer
Vision and scope	Whose outcome matters, and under which boundaries?
Epics, stories and ordered backlog	What is already supported, what is next, and what remains uncertain?
Wireframes and acceptance examples	What will the officer do, see and change?
Capacity and iteration plan	Why is the selected work plausible, including verification and review?
Team agreement and review record	How will changes be inspected and integrated?
Repository and README	Can another person obtain the intended revision and run it?
Passing CI result	Did the configured service execute the gate successfully?
Deliberate failure and restored pass	Does a failed check produce failure, and is the baseline healthy again?
Decision and limitation notes	Which assumptions restrict the claims being made?

Trace a small number of links that matter: story to acceptance examples, examples to checking program, change to review, revision to CI result. Links are worth having when they let another person answer a question. A large spreadsheet of repeated identifiers adds little if the linked check does not exercise the behaviour that was promised.

Let someone else try it from scratch

Ask a teammate to follow the README from a fresh checkout, and watch where they need help that is not written down. Does the command assume a shell they do not have? Do they have a development kit or only a runtime? Does the demonstration need records created in an earlier session?

Correct the instructions and repeat the step that was affected. A handoff that works is evidence of shared understanding. Do not report a command you have not run as verified because it looks familiar. If the hosted service is unavailable, keep the local evidence and name the missing hosted result as a gap. An honest gap gives the team a concrete next action.

By the end of this chapter you should have a bounded plan, sketches someone can inspect, and a baseline you can repeat, with checks that gate delivery. You do not need to have implemented every backlog item or designed the final architecture. Chapter 5 will look at the responsibilities in the coordinator, using these checks to protect the behaviour that already exists.

4.8 Challenging an overambitious plan with AI

Chapter 15 gathers the AI workflows that recur in this book. This activity is complete on its own and tells you the task, context and constraints you need.

	For this activity
Your goal	Evaluate a proposed iteration plan against scope, capacity, behaviour and delivery evidence
You start with	The Chapter 3 backlog, the capacity figures in Section 4.4 and the Chapter 4 starter
What is in bounds	Review and replace the plan; do not implement any feature
How to check	Your own capacity arithmetic, the code (is there a query method? does `reserve` change state?) and the actual gate output
How to decide	A verdict and reason for each distinct proposal, then a bounded replacement plan
No AI account?	The ready-made output below is enough

The goal is to judge proposed work against scope, capacity, behaviour and delivery evidence. The text below is **fictional AI output written by the author** for this offline exercise. It is not a record of a conversation with a real model.

> Four students have 22 person-hours, so plan 22 hours of features. Deliver US-02, cancellation, a mobile app, a database and email confirmations. Assign one student to each layer and integrate on the final evening. Convert the three-point availability story into three hours. Show "available" by calling `reserve` and deleting its record afterwards. Skip failure cases until next sprint. Mark EN-01 complete when the YAML file is committed; a green badge proves the whole story works.

Make a review table with a verdict and a reason for each separate proposal. Then replace the plan with a bounded goal, a selected slice, a capacity calculation and clear evidence. Find at least one domain error, one estimation error, one integration risk and one unsupported claim of completion.

If you use an AI service, give it only the synthetic scope and backlog. Ask it for risks or alternatives instead of accepting a plan from it as authority. Record the prompt, the claims you accepted or rejected, and the independent checks you used. The ready-made output alone is enough to reach the same learning outcome.

4.9 Planning mistakes that look reasonable

Mistake	Why it fails	Better action
Count starter behaviour as newly delivered	Hides the actual change	Separate existing capability from new evidence or functionality
Slice only by technical layers	Leaves no usable end-to-end result	Select one observable journey and its necessary tasks
Fill all nominal capacity with features	Leaves review, coordination and uncertainty unfunded	Budget those activities explicitly
Use a wireframe as proof of implementation	Confuses an agreed sketch with working behaviour	Label proposal and implementation separately
Treat an availability answer as a reservation	Omits eligibility and later state changes	Keep query and command outcomes distinct
Create CI configuration without running it	Provides no evidence that the gate executes	Inspect an actual run against the intended revision
“Fix” a red job by ignoring its exit status	Conceals the defect	Correct the cause and preserve failure propagation
Invent review or meeting evidence	Makes the project state misleading	Report actual observations and remaining gaps

4.10 Try it yourself: prepare for the kickoff

This activity is about your team’s project. If your team uses equipment lending, the supplied listings are your starting repository. If your team chose another of the provided domains, apply each step to that domain’s starter codebase, and use the supplied files as the model for layout, script and workflow. Do not change domain without following the module’s arrangements. You start from the Chapter 3 backlog, the current Java baseline and the supplied Chapter 4 repository listings.

If your instructor runs a demonstration, it walks through the three reservation requests and then compares an ordinary passing run with the deliberately wrong acceptance check. Ask yourselves what each result establishes, precisely. For guided practice, write a one-page scope statement and a wireframe, have another pair explain the journey back to you, and resolve one difference between their reading and the behaviour you intended.

Then work through these steps as a team:

1. Write the vision, boundaries, assumptions and two decision records with review triggers.
2. Group the backlog into epics, preserve story identifiers, and distinguish existing capability from new work.
3. Refine US-02 or an equivalent thin slice with normal, boundary and rejection examples, including state effects.
4. Create an iteration goal and a capacity plan using your actual availability. Show coordination, review and uncertainty explicitly.

5. Agree ownership, review and board policies. Apply them to one small repository change.
6. Create the starter files, run the gate and reproduce the scripted demonstration.
7. Enable the agreed hosted workflow. Retain a passing run, a deliberate assertion-failure run on a practice branch, and a restored passing run. Record the integrated baseline’ s result.
8. Assemble the evidence map and ask another team member to follow the README from a fresh checkout.

What you should end up with is: the vision and scope notes, the ordered backlog and selected story, the wireframes, the capacity plan, the team agreement, the review record, the repository instructions, the verification evidence, and the unresolved questions with owners.

You are done when another person can explain the outcome you chose, tell what is implemented from what is only planned, run the baseline, find the real CI results and trace at least one acceptance example to a check that was executed. A missing hosted run stays on the page as an explicit gap. For the assessed hand-in, follow the module leader’ s submission requirements.

If you want more, review whether the query slice should use a duration or an exclusive end date at its interaction boundary. Compare the input errors each choice creates, and update the sketch and examples. Do not implement a second interface just for this extension.

4.11 Questions to think with

Concept questions

1. How do a vision, an iteration goal and an implementation task differ? Use the equipment project for each.
2. Why can a console-based journey be a vertical slice even when it has no database?
3. Why is total available person-time insufficient to establish a feasible iteration plan?
4. What does a passing starter check establish, and what does it leave unproven?

Applied diagnosis

5. A team reports the availability story done because the screen says “available”. Its implementation calls `reserve`, then removes the new record. Identify the violated requirement and the state evidence you would inspect.
6. A CI job compiles successfully, runs `java -cp build/classes Chapter03AcceptanceExamples || true`, then prints “all checks passed”. Explain how this can report success after an assertion failure and how you would verify the correction.

4.12 A different setting: the transfer challenge

A small clinic wants reception staff to inspect appointment-slot availability and make bookings. It has one supplied in-memory booking routine, a draft backlog and a four-person student team with uneven availability.

Create a bounded first-iteration goal, a wireframe and a capacity-aware plan. Identify two assumptions that should be revisited before real use. Give one adjacency example and explain why equipment-lending date rules cannot simply be copied into a time-based appointment domain. Show what the first automated gate would establish and what evidence would still require human review.

Use synthetic appointment data. The exercise concerns planning and software behaviour; it does not ask you to determine clinical policy or process real patient records. A team working on another provided domain may instead apply the same steps to that domain's first user-visible capability.

What to take away, and what comes next

A credible iteration ties a bounded outcome to a thin journey, a plan the team can afford and evidence that someone else can inspect. Scope notes bring assumptions into the open, wireframes raise questions about interaction, and acceptance examples make behaviour testable. Review and automation help the team keep a shared baseline while it learns what to build next.

The application behaviour in Chapter 4 is still the reservation code from Chapters 2 and 3. What is new is the repository layout, a command that runs the checks repeatably, a scripted demonstration and a template for a hosted workflow. US-02 is still planned. Chapter 5 uses this protected starting point to look at responsibilities, extension and substitutability.

Sources and further reading

Sources checked 12 and 13 October 2026. Process templates, capacity figures, wireframes and classroom policies are author-created examples; they are not requirements imposed by the sources.

1. Schwaber, K. and Sutherland, J. (2020), **The Scrum Guide**. Formal reference for Product Goal, Sprint Goal, Sprint Backlog and adapting scope. "Sprint 0" is a local descriptor label.
2. Oracle, **The `javac` command, Java SE 17**. Compilation, output directories and the release target.
3. GitHub, **Building and testing Java with Maven**. Workflow location and reuse of project build commands; this starter runs its Bash gate rather than Maven. The page's own examples show older action versions; the supplied workflow uses the current major versions from the action repositories [4, 5].
4. GitHub Actions, **setup-java**. Java installation inputs; the supplied workflow uses `setup-java@v6` and Temurin 17.
5. GitHub Actions, **checkout**. Repository checkout action; the supplied workflow uses `checkout@v7`.
6. Martin, R. C. (2002), *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall. The module's required text; the Week 3 reading is its planning and testing chapters. Publication details as listed in the module descriptor.
7. Rubin, K. S. (2012), *Essential Scrum*, Addison-Wesley. A recommended text; read the sprint-planning chapters named for Week 3. Publication details as listed in the module descriptor.

Read the relevant current action documentation before adopting the workflow in a different runner environment. A template is a starting point whose actual run must still be observed.

Chapter 5: Responsibilities, Extension, and Substitutability

> *SOLID principles, Single Responsibility, Open-Closed, and Liskov Substitutability.*

OO Development and Agile Practices
Designing, Testing and Evolving Software with AI Assistance

The equipment officer asks for a different confirmation layout. In the same week the team is asked to demonstrate a second borrowing policy for staff. One developer proposes a subclass of the reservation coordinator. Another asks an AI assistant to "make the code SOLID" and gets back interfaces for almost every class and a rejection turned into an exception.

The code certainly looks more elaborate. But the team has not yet answered the questions that matter. Which parts of this program change for different reasons? Which behaviour is meant to vary? What may existing callers keep relying on? And how will anyone tell an internal improvement from a new feature?

Three old principles help with those questions: the **Single Responsibility Principle (SRP)**, the **Open–Closed Principle (OCP)** and the **Liskov Substitution Principle (LSP)**. Treat them as ways to reason about change and about contracts, not as a checklist to satisfy; a design is not better merely because it has more classes, interfaces or principle names in it. In this chapter you will learn to find the independent reasons a class has to change, to apply SRP to separate one responsibility and weigh the cost of the new boundary, and to justify an extension point for a variation you have actually observed, without pretending that existing code must never change. You will describe a behavioural contract and diagnose a subtype that breaks what its clients expect. And you will make a small, protected refactoring, show that it is not a feature change, and report the evidence.

This is the first SOLID chapter. It sits in Week 4, supports LO1 and LO4 and the formative smell analysis, and does not change the module's assessment arrangements. Chapter 6 covers interface segregation, dependency direction and injection, Chapter 7 develops testing practice, and Chapter 11 gives a systematic treatment of smells and legacy refactoring. The Week 4 readings are the SRP, OCP and LSP chapters of Martin [6] and "Bad Smells in Code" from Fowler [7].

You start from Chapter 4's Java 17 reservation starter, its 34 regression checks and its five acceptance example groups. The availability enquiry (US-02) is still planned. The staff rule in this chapter is a small variation that the book introduces for teaching: it does not mean US-02 is done, and it is not an approved university policy. The companion holds complete listings, intermediate versions, commands, model answers, teaching notes and verification results. Follow the two stages separately: **A, extract formatting without changing existing behaviour; B, add the documented staff variation while preserving the old student entry point.**

Sections 5.1 to 5.7 are the core reading. What you must do is steps 1 to 8 of the practical activity in Section 5.10, including the review in step 6 (with another person, or labelled honestly as a solo self-review). Steps 1 to 7 run on the book's project and step 8 applies the diagnosis to your team's project. The Week 4 lab uses a static-analysis tool, PMD (version 7.28.0 in this edition), which needs a one-time download described in

Section 5.1, and comparing its findings with your own reading is part of the lab. If you want more, try the AI activity in Section 5.8, the extension at the end of Section 5.10, the questions in Section 5.11 and the transfer challenge in Section 5.12.

5.1 Why would this code change? Change pressure on the service

Start from the code we have

The current `ReservationCoordinator` validates arguments, counts a borrower's reservations, asks `ReservationRules` about count and duration, checks overlap, stores a new reservation and constructs a confirmation. Chapter 2 already moved string construction into a private method. Chapter 4 made the checks repeatable. We now have a useful baseline for asking whether a further boundary is justified.

The coordinator is not automatically wrong because it performs several steps. Coordinating a reservation necessarily involves several collaborators and decisions. The more useful question is whether unrelated changes repeatedly force edits into the same place.

Consider these proposed changes:

Change pressure	Main knowledge involved	Current location
Change confirmation wording	Presentation of a completed reservation	Coordinator's private formatting method
Add a second count/duration policy	Borrower eligibility rules	Existing rule plus new selection behaviour
Change interval overlap semantics	Meaning of a loan period	`LoanPeriod`
Preserve records across restart	Storage lifetime and retrieval	Coordinator's in-memory list
Reject unknown equipment identifiers	Authority for equipment identity	Not implemented

These are different concerns, but the table is not an instruction to create five interfaces immediately. The first two are the chapter's actual teaching changes. Persistence and catalogue integration remain future questions. Their possible existence is a reason to document a limitation, not sufficient evidence for building an extensibility framework now.

Say what the new rule is before choosing a design

For the worked example, adopt this **illustrative requirement, invented for this book**:

Category	Existing active reservations allowed before a successful request	Requested duration
Student	0, 1 or 2	1 through 7 days
Staff	0, 1, 2, 3 or 4	1 through 14 days

A successful student request may therefore create the third active reservation; a successful staff request may create the fifth. Both policies reject negative counts and nonpositive durations. They decide only count and duration. Both still require a valid period and no overlapping reservation for the same item.

This is a feature change. A staff request for fourteen days can now succeed where the existing four-argument operation would reject it. Moving the existing formatter is a refactoring only if its observable results remain the same. Combining those changes in one unexplained edit would make failures harder to interpret.

In the new example the caller supplies the category, and we trust the caller to do so. Nothing here authenticates staff status, persists a borrower profile or infers a category from an identifier. Passing `STAFF` is not proof that a person is staff. A production boundary would need an authoritative source for the category before accepting such a claim. That is part of the scope of this example, and not an implementation task that we silently completed by adding an enum.

What counts as success at each stage

Stage A succeeds when the formatting responsibility has moved and existing outputs, exceptions, state changes and signatures remain compatible for existing callers. Stage B succeeds when the explicit category operation implements the new table while the old operation continues to mean student borrowing.

All stored records still count as active. Storage remains in memory, calls remain sequential and intervals remain half-open. No cancellation, return processing, catalogue, email delivery or availability enquiry is added. Those boundaries stop a design exercise from turning by accident into an expansion of the system that nobody could review.

Evidence from three sources: compiler, reading, analyser

The change pressures and the two-stage plan above say what we intend to do. Before changing anything, collect evidence about where the code is hard to change today, from three different sources.

A static-analysis tool examines source code without running it and reports patterns that often indicate problems. The Week 4 lab uses one for smell diagnosis. You will gather three kinds of evidence in this order: the compiler’ s, your own, and the tool’ s. Setting them side by side is the point of the lab.

The compiler. `javac -Xlint:all` asks the compiler for every optional warning. From the Chapter 4 repository root:

```
javac --release 17 -Xlint:all -d build/lint src/main/java/*.java src/test/java/*.java
```

On the Chapter 4 starter this prints nothing. That result is informative: the compiler checks language-level hazards such as unchecked conversions and fall-through, not whether message wording and eligibility belong in one class.

Your own reading. Do the manual smell review before you run any analyser, so that the tool’ s output has something to be compared with. Read `ReservationCoordinator` once for each of these four smells and write down each place where you find one, with the line or method:

Smell	What to look for
More than one reason to change	A method that mixes eligibility, conflict search, record creation and message wording
Hidden construction	A class that creates its collaborators with `new` inside its own methods
Repeated or similar logic	Two loops over the stored list that differ only in the test applied, or rejection outcomes returned as plain strings that a caller must compare
Primitive values standing for concepts	Strings or integers that carry meaning (an identifier, a number of days, a rejection reason) without a type of their own

For each finding, record where it is, what you observed, whether it matches a change pressure in the table above, and whether you would act on it. Also name one pressure from the table that the compiler did not report and that you found only by reading.

The analyser: PMD. The book uses PMD, version 7.28.0 [8], which runs on Java 8 or above [8]. Download `pmd-dist-7.28.0-bin.zip` from the project's GitHub release page, compare the file's SHA-256 digest with the one the release page shows for it (the companion records the digest that was checked), unzip it, and set the variable `PMD_HOME` to the unzipped `pmd-bin-7.28.0` folder. This is the only download the chapter asks for.

Before you read the output, here are five terms. **Cyclomatic complexity** counts the independent paths through a method: each `if`, loop or `case` adds one. **Cognitive complexity** is a variant that penalises nested conditions. **NPath** counts the possible execution paths through a method. A **God class** does too much and knows too much; a **Data class** holds data and little behaviour. In the Data-class line below, WOC, NOPA, NOAM and WMC are PMD's own measures of how much of a class is behaviour (the share of its public methods that are not accessors, its public attributes, its accessor methods and the summed complexity of its methods); you do not need them to read the result.

PMD ships many rules in categories, and the choice of rules is part of using it. On this five-file starter, the quickstart rule set reports 5 findings, all the same one: classes without a package, which the book chooses deliberately until Chapter 10. The code-style category reports 34, for example parameters that "could be final". Neither helps a smell diagnosis. The lab therefore uses a small rule set of eight design rules from PMD's design category, kept in `config/pmd-smells.xml`, with PMD's default thresholds [9]: cyclomatic complexity from 10, cognitive complexity from 15, NPath from 200, God class, Data class, more than 10 methods, 10 or more parameters, and 30 or more imports. Run the compiler gate first, so that PMD can read the compiled classes, and then run the lab script:

```
export PMD_HOME=/path/to/pmd-bin-7.28.0
bash scripts/check.sh
bash scripts/pmd.sh
```

On the Chapter 4 starter it prints the following. The recorded run was on Windows, so the paths contain backslashes; elsewhere they use your operating system's separator:

```
src\main\java\Reservation.java:3:       DataClass:      The class 'Reservation' is suspected to b
src\main\java\ReservationCoordinator.java:10:   CyclomaticComplexity:   The method 'reserve(Strin
[INFO] Found 2 violations.
```

Now interpret each finding the way you interpreted your own. The second one is **useful**: `reserve` has twelve independent paths because it validates identifiers, counts the borrower's records, applies eligibility, searches for a conflict and creates the record, which is your first smell seen from another side. The first one is **low-value here**: `Reservation` is deliberately an immutable record of an accepted booking, so holding data and little behaviour is its job. It does carry a question, though: where does the behaviour that uses its fields live? In this starter, it lives in the coordinator's confirmation wording. And PMD **missed** things you found by reading: it did not report the `ReservationRules` that the coordinator constructs for itself, and it cannot know that "borrower limit or duration" is one string for two different reasons. Placing each finding in one of the three groups (useful, low-value in this context, a concern the tool missed) is the comparison the lab asks for.

A finding is evidence to investigate, not an instruction. At this scale a tool may report items that do not matter, and it cannot know which changes your team expects. Rerun PMD after each stage of the practical activity: the complexity finding stays at 12 after Stage A and after Stage B, because moving wording and policy out changed who decides, not how many branches the sequence of checks has. The number moves only when you split that sequence. Extract Method alone can lower it while leaving every responsibility where it was, which is why a number is not a verdict. Record only output you actually produced.

5.2 One class, one reason to change? Responsibility and SRP

A reason to change is more useful than a method count

SRP asks us to look at changes that come from different responsibilities. Martin's explanation ties those responsibilities to the people or groups whose needs drive the changes [1]. In this project, the people who decide borrowing limits may care about quite different things from the people who decide how an officer reads a confirmation.

That is more informative than counting methods. A class with several methods can serve one coherent responsibility, and a class with one method can mix policy, storage and presentation. Several requests from the same person may also concern different responsibilities, so an organisational chart is evidence about change and not a mechanical way to generate classes.

Suppose the coordinator's method stays short because it hands work to twenty tiny helpers. If those helpers all reach into the same mutable list and repeat the same decision logic, the design may still be hard to change. By contrast, `LoanPeriod` has a constructor, accessors and an overlap operation, and yet those methods describe one coherent idea of an interval.

Pulling out a collaborator with a clear input

For Stage A, move the confirmation construction into `ConfirmationFormatter`. Its input is the immutable `Reservation` that has already been accepted. Its output is a string. It does not approve a request, add a record or send a message.

The coordinator's final step becomes:

```
reservations.add(reservation);
return formatter.format(reservation);
```

This keeps the existing order. Formatting is still done after the reservation is stored. The extraction does not add transaction handling or a rollback in case some future formatter fails. With the supplied deterministic formatter and a valid stored reservation, the ordinary path stays the same. If formatting later gains external dependencies, what happens on failure will need a design decision of its own.

Passing one coherent reservation object saves us from copying the formatting logic's inputs into a long list of unrelated arguments, because the object already carries the identifier, borrower, item and period. The formatter must not be handed the coordinator's mutable list just because it could find the same data there. That would give presentation access to storage decisions it has no need for.

The original private method was already a useful seam. A separate class now gives a formatting change a named home, and lets us examine it directly without running a reservation scenario. The price is another file, another object and another call to follow. If the formatting were a stable one-line expression with no change pressure of its own, keeping the private method could be a sensible decision.

What stays together, and why

The coordinator still creates collaborators directly and owns the list. This chapter does not claim that one extraction finishes its design. It makes one justified improvement and leaves the request workflow readable. Chapter 6 will look at how collaborators are supplied and which dependencies belong at the application boundary.

The numeric rules were already outside the coordinator, so do not claim to have separated them just now. What has really changed is that eligibility decisions and confirmation presentation now each have a name and a home, while the coordinator stays responsible for putting the steps of the reservation in order.

A short rationale might say:

> Confirmation wording can change independently of eligibility and conflict handling. Move it to a formatter that accepts an accepted reservation. Keep the returned text and storage order unchanged, and run existing exact-output checks.

That rationale names a reason, a boundary and evidence. "Applied SRP" names none of them.

Keeping decisions out of the formatter

It is tempting to write a formatter that checks the requested duration and returns either a confirmation or a rejection. That brings a policy decision back into presentation and risks disagreeing with `ReservationRules`. Another formatter might allocate the reservation identifier because the identifier appears in the message. But presentation uses an identifier; that does not make presentation responsible for assigning it.

Ask which object has the information and the authority to make each decision. The coordinator knows the stored records and whether a request was accepted. The formatter knows how to show the accepted record. If we keep that distinction clear, later changes to wording are less likely to alter how reservations behave.

5.3 Making room for variation: OCP and anticipated change

Which change are we protecting against?

OCP encourages a design where an anticipated kind of extension can be added without repeatedly rewriting stable behaviour [2]. The useful question is "closed against which change?" There is no helpful promise that every future requirement you can imagine can be added without touching existing code.

Here the variation we have actually seen is a count-and-duration decision for two borrower categories. It is not storage technology, notification transport, date arithmetic or authentication. A small interface expresses exactly that decision:

```
public interface BorrowingPolicy {
    boolean mayRequest(int activeReservations, int requestedDays);
}
```

An interface gives the client a common type. A class declares `implements BorrowingPolicy` and supplies its public method, and `@Override` lets the compiler check that the declaration implements or overrides a matching method [3]. The interface does not supply the meaning of the operation by itself, so we also have to document its contract.

`ReservationRules` keeps its existing student predicate and implements the interface. `StaffReservationRules` implements the new staff predicate. The coordinator obtains a policy and asks it the same question each time. Overlap detection and storage are not copied into separate student and staff coordinators.

Keeping the old entry point working

The completed coordinator keeps:

```
reserve(borrowerId, itemId, start, requestedDays)
```

It hands over to a new overload that uses `BorrowerCategory.STUDENT`. The new overload also accepts an explicit category. Existing code therefore still compiles and keeps its student meaning, and you have to call the new form explicitly to choose staff behaviour.

Do not infer staff status from `borrowerId.startsWith("S")`. That would hide a new rule about identity in a convenient string convention. The synthetic identifier S1 in the demonstration is just a label, and the category argument is what selects the policy.

Every record for the same borrower identifier counts toward the active total, whichever overload created it. The implementation does not keep separate student and staff buckets, and it does not remember a borrower's category. If a caller supplies inconsistent categories, each request applies the category it was given to all of that borrower's existing records. This limitation comes from how the example is set up, and it is not a model of how a real system would authorise changes of category.

The selector that is still there

The example keeps a small `switch` inside the coordinator to choose between the student and staff policies that it creates itself. Adding another enum category would still mean updating that selector, so the coordinator is not completely closed to new categories.

The narrower improvement is that the reservation algorithm hands the varying rule to a policy. A change to the staff duration limit belongs in the staff policy, while overlap and storage stay as they are. The selector shows a decision about composition that Chapter 6 can move outwards when it discusses dependency injection. This deliberate middle step is easier to explain than pretending we need a factory framework for two rules.

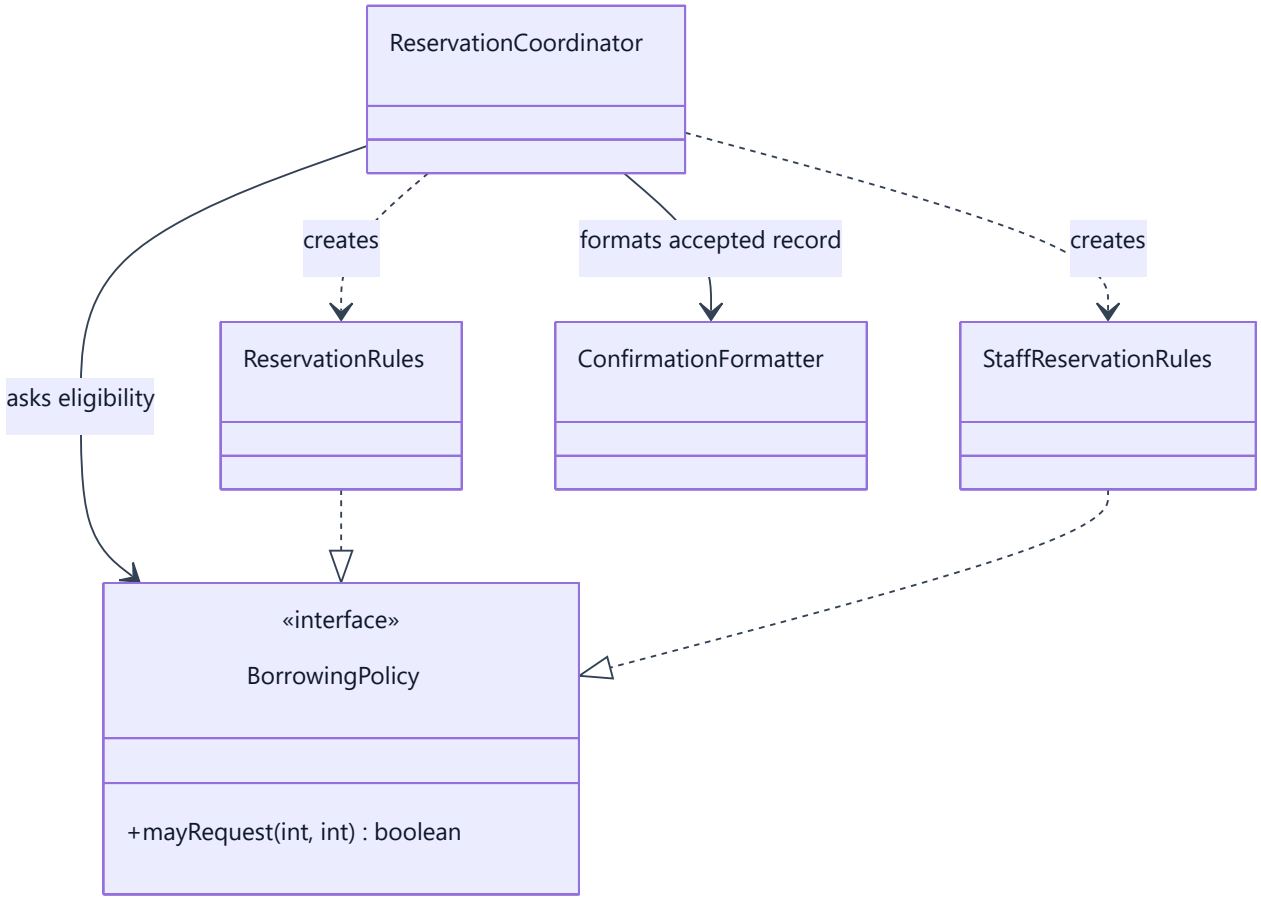


Figure 5.1. The collaborations after Stage B. The solid arrows are associations: the coordinator holds a `BorrowingPolicy` reference and a `ConfirmationFormatter`. The dashed lines ending in a hollow triangle are interface realisations: `ReservationRules` and `StaffReservationRules` each implement `BorrowingPolicy`. The two plain dashed arrows are dependencies: the coordinator creates the concrete policy objects itself.

Those last two arrows are drawn on purpose. The interface gives the coordinator’s algorithm one type to ask, but it does not remove the coordinator’s dependence on the concrete classes. Chapter 2’s notation table explains each symbol. When Chapter 6 moves object creation outward, the two dashed arrows should disappear from this diagram, which is a visible way to check that dependency direction really improved.

Would something simpler do?

For these exact numeric rules, a single `BorrowingLimits` class with validated `maximumActive` and `maximumDays` parameters could also be reasonable. The two policies differ only in constants. A compact conditional could even be acceptable in a short-lived demonstration with no further variation expected.

The interface approach earns its place here by making the decision contract explicit and allowing separate policy behaviour to be examined. It becomes more valuable if policies later differ in more than limits. It also adds names and selection work. Do not justify it by asserting that two constants always require polymorphism.

Likewise, do not invent sponsor approval, network checks or a rule engine to make the abstraction look necessary. Those changes need their own requirements. The book's later changed-requirement exercise remains later work; this chapter uses only the small second rule stated above.

5.4 What can a caller rely on? Behavioural contracts

Writing down what callers may rely on

A method signature tells us the types of the parameters and the result, and leaves many questions open. Can the method reject input with an exception? Does it change anything? Is the result the same every time for the same input? What exactly does `true` mean?

For `BorrowingPolicy`, use this contract:

Aspect	Promise
Accepted input domain	Every pair of Java `int` values can be passed
Invalid numeric values	Negative count or nonpositive duration returns `false`
Decision	`true` exactly when the inputs satisfy that implementation's documented count/duration limits
Side effects	Does not create reservations, change a counter or perform external I/O
Repeatability	Same inputs and fixed policy configuration produce the same result
Scope of approval	Only count and duration; no assertion of item availability or borrower identity

The student and staff implementations have different documented limits, so they can legitimately return different booleans for `(0, 14)`. A client of the common interface must not assume that every policy uses seven days. That limit belongs to the student policy's specific contract.

The interface is still meaningful because all implementations answer the same kind of question under common input and side-effect rules. An interface saying "do anything and return a boolean" would give clients almost nothing to reason with.

Preconditions, postconditions and invariants

A **precondition** states what must hold before an operation for its promised behaviour to apply. A **postcondition** states what the operation guarantees afterwards. An **invariant** is a property maintained by an object's valid observable states. These terms help turn vague expectations into statements a reviewer can check.

For `LoanPeriod`, construction requires non-null dates with the start strictly before the exclusive end. Successful construction establishes that invariant. For the borrowing policy, negative counts are part of its accepted input domain and have a specified `false` result. Treating them as a forbidden precondition would change this particular contract.

Chapter 1 noted that returning `false` for a negative count treats an impossible input like an ordinary refusal, so a caller bug could look like a policy decision. This contract settles that question deliberately instead of leaving it implicit, and the two options have different costs. Returning `false` keeps every existing client working, because they branch on the result and never handle an exception, and it keeps the policy defined for every pair of integers. Treating a negative count as a precondition violation and throwing would signal a caller bug loudly, but it changes a result that existing callers and checks already rely on, and a new exception becomes part of every client's contract. Here, preserving behaviour decides the matter. In new code whose callers you control, the exception can be the better choice. Either way, write the choice into the contract.

For a successful reservation, the postconditions include one new stored record and a confirmation that identifies it. When a request is rejected for eligibility or overlap, the stored records stay unchanged. The order of the decisions matters too: the existing method checks count and duration before it builds the requested period and before it checks overlap. If you reorder those steps, the failure a caller sees can change.

Contracts include exceptional outcomes when callers rely on them. The existing four-argument operation rejects blank identifiers with `IllegalArgumentException`, null dates with `NullPointerException`, and an unrepresentable end date through Java's date exception. The new explicit-category overload rejects a null category. A refactoring is not the moment to tidy those outcomes quietly. A more consistent error model in the future would be a separate change to the API.

Checking behaviour, not just return values

Suppose an implementation returns the expected rejection string but first inserts a reservation. A check on the string alone would miss the broken state guarantee. Suppose it returns `false` for a negative count on the first call and throws on the second. One isolated assertion would miss the repeatability problem.

Choose checks that correspond to real promises: inspect state after a rejection, cover accepted and rejected boundaries, and call policies through the common type. The companion includes direct policy checks and coordinator-level checks because each one observes a different risk. Direct checks make the boundaries of a policy easy to diagnose, and integrated checks confirm that the coordinator chooses and applies the right policy.

Passing examples are evidence, not a proof for every possible integer or call sequence. For predicates this tiny, reading the whole expression is a good complement to boundary tests. The absence of fields and external calls also supports the claim that the supplied implementations are stateless. A larger policy would need a different way of verifying it.

5.5 When a subtype breaks its promises: LSP and unsafe inheritance

A subtype makes promises to clients

LSP asks whether clients that use a supertype can keep relying on that type's behavioural promises when they are handed a subtype. Liskov and Wing's treatment covers constraints on what inputs are required, what outputs are guaranteed, invariants, and how an object's state may change over its history [4]. Matching Java signatures is necessary for many substitutions, but it does not establish any of those behavioural properties.

In practice, a review asks whether a subtype requires more from the caller, guarantees less afterwards, breaks a property of valid state, or changes state in ways the supertype's contract ruled out. Different implementations need not give identical results in every situation; their differences must stay within what the common contract allows.

The staff policy is not a subclass of the student policy. Both implement the broader `BorrowingPolicy` contract. That matters: a client that was promised the student's seven-day maximum should not be handed an object that approves fourteen days while still claiming to fulfil that exact student contract.

A counterexample you can run

The reservation classes do not need an inheritance hierarchy just to illustrate LSP. Use a separate teaching example, `IntHistory`, whose contract is to accept any `int`, append it once and keep earlier entries in order.

A client is entitled to do this:

```
history.add(5);
history.add(-2);
```

It then expects `[5, -2]`. Now suppose someone writes `PositiveOnlyHistory extends IntHistory`, overriding `add` so that it throws when the value is zero or negative. It compiles, but the original client now fails on `-2`.

The subtype has strengthened the input requirement. That extra restriction may make sense for some other domain object, but it breaks the contract of this supertype. Calling the exception "validation" does not repair the substitution. The key evidence is the valid supertype client and the changed observable result.

A second faulty variant might accept `-2` but ignore it. That avoids the exception while weakening the guarantee that every value is appended. A third might erase earlier entries whenever a new value arrives, which breaks the promise to preserve history. These are different failures, even though all of them can share the same `void add(int)` signature.

Repairing the relationship

One repair is to keep `IntHistory` accepting every integer, and to add a separate `PositiveHistory` type with an `addPositive` method. That method openly requires a positive value. It may hand storage to an internal `IntHistory`, but it does not present itself as a substitute for the broader type.

Composition alone is not a magic guarantee. If `PositiveHistory` still claimed to implement an interface that promises to accept every integer, the conflict would remain. The repair works because the public contract and the type relationship it advertises now agree.

Another possible repair is to redefine the common abstraction so that clients are told which values an operation may reject. That changes the contract with clients and may force changes in them. It is not a behaviour-preserving fix for callers that already depend on every integer being accepted. Decide whether the broad contract or the restrictive type is the abstraction you really mean before you edit.

Exceptions and history need context

Do not memorise "throwing an exception violates LSP". An exception is compatible when the supertype's contract allows that outcome for the input and state in question. The defect in our example is that the supertype promised to accept `-2`, and the subtype rejects it.

In the same way, adding a method to a subtype is not automatically safe. A new method that changes state can invalidate promises that callers observe through inherited methods. If a type promises that entries stored earlier stay present, a subtype-specific `eraseAll` operation can contradict that promise about history when the same object is also held through the supertype reference.

In a review, state the promise, show the sequence of calls and inspect the outcome. "This violates LSP" should be the conclusion of that explanation, and not a substitute for it.

5.6 Small refactorings and the checks that guard them

Change the structure and the behaviour in separate steps

Refactoring changes the internal structure and keeps the observable behaviour [5]. Stage A follows that idea: introduce the formatter, move the existing expression, replace the call and run the existing checks. Do not rename domain concepts, change the order of validation and add staff behaviour in the same step.

A practical sequence is:

1. Run the Chapter 4 gate and record the starting result.
2. Add the concrete formatter with the existing confirmation expression.
3. Replace the private formatting call and remove the now-unused private method.
4. Compile and rerun the original regression and acceptance checks.
5. Inspect the diff for unrelated changes and record what behaviour was preserved.
6. Begin Stage B in a separate change, with the staff rule and its new checks stated explicitly.

The companion supplies the starting, extracted and extended coordinator versions. Keep one coordinator version in a given source directory. Compiling several classes with the same name together is a setup error, not evidence about the design.

Reading the checks alongside the change

The original exact confirmation assertions protect punctuation, identifiers and date formatting. Overlap, adjacency and count checks protect the reservation algorithm. Snapshot checks protect observation of stored state. These existing checks should remain unchanged through Stage A and the compatibility portion of Stage B.

Stage B adds checks for staff's accepted fifth reservation, rejected sixth, accepted fourteen-day duration and rejected fifteen-day duration. Cross-category scenarios show that a staff booking still blocks an overlapping student request. Another check confirms that the original overload still rejects a student's eight-day request.

The query tests AQ-01–AQ-05 from earlier planning are not part of this result: no availability API has been implemented. State the new checks' scope accurately. The existence of a `BorrowingPolicy` does not fulfil every backlog item involving equipment.

When a check fails, diagnose before editing

If the old seven-day check fails after you introduce staff rules, look at how the policy is selected before you change the expectation. The old entry point was promised student behaviour. Updating the test to accept fourteen days would hide a compatibility defect.

If an exact confirmation check fails during extraction because the formatter says "Reserved" instead of "Confirmed", the extraction changed a result that clients can see. Either restore the old text for Stage A, or separate the wording change and get it approved on its own. A test is not sacred, but replacing it needs a reason grounded in the intended contract.

The LSP demonstration deliberately includes an invalid subtype. Its harness observes the expected failure and reports the counterexample. That success means the teaching demonstration behaved as intended; it does not mean the invalid subtype passed the base contract. Keep this example outside the running reservation application.

Reporting what you found, and what it does not show

A useful change record separates the baseline run, the unchanged checks of Stage A and the additional checks of Stage B. Include the command, the revision or version, the output you observed and the relevant limitations. Do not claim an independent review or a hosted CI run when only local checks ran.

The author ran the supplied Java checks locally. The Chapter 4 hosted workflow can keep invoking the updated gate in a team's repository, but an actual hosted run is still the team's own evidence. A clean local result supports a review; it does not replace one, and it does not show that the code is ready for production.

5.7 When is an abstraction too early? Avoiding premature abstraction

Weigh the cost against the change it supports

Every boundary has a cost. A new interface needs a name, a contract, implementations and a place where one implementation is chosen. An abstraction that hides a real variation can reduce future edits. One that only mirrors a single class adds navigation without changing which decisions are coupled.

For the worked example:

Decision	Benefit	Cost or remaining limitation
Extract a concrete formatter	Gives independent presentation changes a home	Another object and file; no configurable presentation yet
Use a small policy interface	Names the count/duration contract shared by two rules	Extra types and selection; numeric parameterisation is also possible
Keep a selector in the coordinator for now	Makes the intermediate stage easy to inspect	New categories still require a coordinator edit
Retain direct construction	Keeps focus on SRP, OCP and contracts	Dependency assembly remains coupled until Chapter 6
Avoid coordinator inheritance	Prevents duplicating or overriding the reservation workflow	Variation must be expressed through explicit collaborators

A reasonable review might conclude that, for a particular project, one of these costs outweighs its benefit. Principles help us explain a judgement; they do not remove the need to make one.

Warning signs worth noticing

Useful signals include repeated edits to the same policy conditional, message changes that break reservation logic, and subclasses that disable inherited operations. Look at the change history, or build a counterexample from a client's point of view, before you prescribe a repair. A long method alone does not tell you which boundary is needed.

Do not create `IReservation`, `ILoanPeriod`, an abstract coordinator factory and a generic notification bus just because an automated suggestion calls them "SOLID". Ask what concrete variation each one supports and which client contract it improves. If there is no answer, that is a reason to postpone the abstraction.

The next chapter moves from having a useful contract to the direction of dependencies and the way objects are put together. That is a new question, and keeping it separate stops students from treating SRP, OCP, LSP, ISP and DIP as interchangeable names for adding interfaces.

5.8 Reviewing a "make this SOLID" suggestion with AI

Chapter 15 gathers the AI workflows that recur in this book. This activity is complete on its own, and the ready-made output below is all you need.

	For this activity
Your goal	Judge a "make this SOLID" proposal against concrete requirements, contracts and costs
You start with	The Chapter 4 source, the staff-rule table in Section 5.1 and the policy contract in Section 5.4
What is in bounds	Review the proposal; do not apply it wholesale
How to check	Compilation (is `ReservationRules` `final`?), the exact-output checks, and the rejection and state checks
How to decide	A verdict with a concrete defect or cost for each proposal, then the smallest defensible two-stage change
No AI account?	The ready-made output below is enough

The text below is **fictional AI output written by the author** for an offline exercise. It is not an actual model transcript.

> Create an interface for every class, including immutable reservation and period values. Add a coordinator factory and an abstract base coordinator. Replace all rejection strings with exceptions because exceptions are cleaner. Make `StaffReservationRules` extend `ReservationRules` and override the seven-day limit. Remove all category conditionals so no existing file ever changes. While extracting the formatter, shorten the confirmation to "Reserved". The green build proves the design follows SOLID.

Review each proposal using a concrete requirement, contract or cost. Identify at least one proposal that would not compile against the current source, one that changes existing behaviour, and one whose benefit has not been demonstrated. Propose the smallest defensible two-stage change.

If you use a live AI service, give it only the synthetic source and the stated rules. Ask it to separate behaviour-preserving edits from feature changes, and to point out the dependencies that remain. Record which claims you accepted, rejected or left unresolved, and how you checked them. The ready-made output gives you the same activity without an account.

5.9 Design habits that look principled

Mistake	Counterexample or consequence	Better reasoning
SRP means one method per class	A one-method service can still mix policy and presentation	Identify independent change pressures
OCP forbids editing existing code	A new category still needs an explicit selection decision	State the protected axis and the remaining edit boundary
Same signature means safe subtype	`PositiveOnlyHistory.add(-2)` rejects valid base input	Evaluate the behavioural contract
Any different policy result violates LSP	Student and staff intentionally have different limits	Compare with the shared contract, not a stronger concrete policy
Better wording is part of a refactoring	“Reserved” breaks the preserved confirmation result	Separate the presentation feature from extraction
A passed LSP demo validates the subtype	The harness may have successfully detected a violation	State what the demonstration actually observed
A category parameter verifies authority	A caller can supply `STAFF` without proving identity	Label the trusted-caller boundary
More interfaces mean better design	Unused variations add navigation and decisions	Relate every boundary to a concrete need
Switch on every analyser rule	Dozens of low-value findings hide the one that matters	Choose a small rule set for a stated purpose and record why
Read a finding that did not change as a failed refactoring	A tool measures one thing, such as branching; a refactoring may change another	Say what the tool measures and what the change targeted

5.10 Try it yourself: a protected design change

You start from Chapter 4’s source and checks, this chapter’s staff-rule table and the supplied companion listings. Use Java 17, and keep the LSP micro-example separate from the application. Steps 1 to 7 run on the book’s project; step 8 runs on your team’s project.

If your instructor demonstrates the task, they will trace the existing reservation operation and point out three independent change pressures, and then run one valid base-type client against both `IntHistory` and the invalid subtype and explain the difference. For guided practice, do Stage A together: look at the formatter’s input, output and lack of authority over reservations, run the unchanged checks and read the diff.

Then work through these steps:

1. Produce a responsibility table for the coordinator, rules, period and proposed formatter. Identify a concern you will deliberately leave unchanged and explain why. Run `javac -Xlint:all`, the manual smell review and PMD with the lab rule set from Section 5.1 on the starting code. Place each PMD finding in one of three groups (useful, low-value here, or a concern your own reading found that the tool missed), and compare all three kinds of evidence with your table.

2. Record a passing baseline, extract the formatter and rerun the original checks without altering their expected behaviour.
3. Add the explicit staff variation while preserving the existing student overload. Document the policy contract and the source of the category argument.
4. Add or use the supplied boundary and cross-category checks. Explain what each group establishes.
5. Run the separate LSP example. Identify the violated promise and compare the composition-based repair with changing the base contract.
6. Review the complete diff with another person where available. Retain actual findings and corrections; label solo review honestly.
7. Write a short rationale connecting each new class or interface to a change pressure and naming the remaining dependency-selection limitation. Say what PMD reported after Stage A and after Stage B, and why.
8. On your team's project, produce the same kind of responsibility table for one class in your team's starter codebase, run the same compiler, manual and PMD checks on it (PMD works on any Java codebase), and name one change pressure you can point to in its backlog or history.

What you should end up with is a smell and responsibility diagnosis with the tool findings compared against it, plus the step 8 table for your team's project; the before, refactored and extended versions or commits; the check results; a contract table; an explanation of the subtype counterexample; and a concise design rationale.

You are done when Stage A preserves the existing behaviour, Stage B implements the staff boundaries and keeps the student overload compatible, rejections leave the records unchanged, and you can explain why the policy interface is useful and why the selector still changes for another category. Your rationale must also tell a rule invented for this book from an approved institutional rule.

If you want more, work in a scratch copy of the Stage B code. Extract two private methods from `reserve` (one that counts a borrower's reservations and one that tests whether an item is unavailable), run the gate and rerun PMD, and explain what changed and what did not. Then implement the two numeric policies using a validated limits object in a separate experiment, and compare files changed, clarity and likely future variation. Do not merge both alternative designs into the running application just to demonstrate them.

5.11 Questions to think with

Concept questions

1. Why is counting methods an unreliable way to apply SRP? Give one example from the running project.
2. Against which variation is the policy design relatively protected, and which selection code remains open to change?
3. How can two implementations return different booleans and still satisfy the same policy interface contract?

4. Why is extracting the formatter a refactoring while introducing staff limits is a feature change?

Applied diagnosis

5. A policy implements the right Java signature but throws for a negative active count and increments an internal request counter on every call. Compare it with this chapter’ s policy contract. What evidence would expose each problem?

6. A developer subclasses a history that promises to append every integer. The subclass silently ignores negative numbers rather than throwing. Explain why avoiding an exception does not make the subtype safe, and propose an honest repair.

5.12 A different setting: the transfer challenge

A coursework feedback tool accepts a score and produces feedback text. A new requirement adds a second grading policy and a different feedback layout. Its current class validates input, selects a grade and formats the message.

Write the change pressures, one proposed responsibility boundary and a contract for the grading decision. Give a small input/output table for two explicitly invented policies. State what remains stable and where a policy is selected. Then construct a subtype counterexample that breaks a stated client promise and explain how you would repair the relationship.

Keep the grading rules illustrative; do not present them as university assessment policy. Explain which edits preserve existing behaviour, which introduce a new feature, and which checks distinguish them.

What to take away, and what comes next

Responsibility boundaries are most useful when they answer distinct pressures to change. An extension point should protect a variation you can name. A subtype must keep the promises that clients of the broader type rely on. Small steps, each checked separately, make these design claims much easier to judge.

The running project now has a concrete confirmation formatter and student and staff count-and-duration policies, with the original student entry point preserved. The subtype example stays separate, and the availability enquiry is still planned. Chapter 6 turns to client-specific interfaces, the direction of dependencies and assembling objects by hand.

Sources and further reading

Sources checked 12 and 13 October 2026; PMD sources 8 and 9 on 13 October 2026. The reservation policies, code, counterexamples and teaching decisions are author-created applications of the ideas below.

1. Martin, R. C. (2014), **The Single Responsibility Principle**. A reason-to-change perspective on responsibility.

2. Martin, R. C. (2014), **The Open Closed Principle**. Discussion of extension and stable boundaries. This chapter deliberately evaluates a modest intermediate design rather than claiming universal closure.

3. Oracle, **Java Language Specification, Java SE 17, Chapter 9: Interfaces**. Interface declarations and implementation rules.

4. Liskov, B. H. and Wing, J. M. (1994), **A Behavioral Notion of Subtyping**, *ACM Transactions on Programming Languages and Systems*, 16(6), 1811–1841. DOI: 10.1145/197320.197383. Primary treatment of behavioural subtyping; the executable history example here is original teaching material. The linked PDF is a scanned copy, and the bibliographic details were confirmed through Crossref.

5. Fowler, M. (2004), **Definition of Refactoring**. The distinction between internal restructuring and observable behaviour change.

6. Martin, R. C. (2002), *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall. The module's required text; the Week 4 reading is its SRP, OCP and LSP chapters. Publication details as listed in the module descriptor.

7. Fowler, M. (2018), *Refactoring: Improving the Design of Existing Code*, 2nd ed., Addison-Wesley. A recommended text; the Week 4 reading is its "Bad Smells in Code" chapter. Publication details as listed in the module descriptor.

8. PMD, **Installation and basic CLI usage**. Java 8 or above to run PMD; the `check` command with `-d`, `-R`, `-f` and `--aux-classpath`. The release used here, 7.28.0 (25 September 2026), is on the project's GitHub release page.

9. PMD, **Java design rules**. Descriptions and default thresholds of the eight rules used in the lab.

Chapter 6: Interfaces, Dependencies, and Composition

> *Interface segregation, dependency inversion, and educational dependency injection.*

OO Development and Agile Practices
Designing, Testing and Evolving Software with AI Assistance

The reservation coordinator now uses separate borrowing policies and a confirmation formatter, and a new demonstration requirement arrives: each accepted reservation must be passed to a notification output. One team member simply creates a console sender inside the coordinator. It works. But another member cannot inspect the notifications without watching the console, and swapping the output means editing the reservation workflow itself.

An AI assistant offers a tidy-looking fix: introduce an interface. The coordinator's field becomes a `NotificationSender`. Yet the constructor still creates one particular sender and reads provider settings. The diagram looks different; the code still chooses its own environment.

That gap, between naming an abstraction, depending on a useful contract and having a collaborator supplied from outside, is what this chapter is about. You will learn to design an interface around what one particular client needs and to see, through a concrete dependency, what the **Interface Segregation Principle (ISP)** warns against. You will separate the **Dependency Inversion Principle (DIP)** from dependency injection and from inversion of control, which are easy to blur. You will build a small object graph with constructor injection and an explicit composition root, and a tiny container walkthrough will make object lifetime visible without installing a framework. And you will weigh what an interface costs and what it does and does not guarantee.

The chapter belongs to Week 5 and supports LO1 and LO4. Quiz 1 revision points appear later and refer to Chapters 2 to 5, matching its stated Weeks 1 to 4 coverage; this chapter's new material is not silently added to it. The Week 5 readings are the DIP and ISP chapters of Martin [8] and, for dependency injection, the articles listed on Moodle, of which Fowler's [3] is one standard source.

You start from Chapter 5's Java 17 code: the student and illustrative staff policies, the extracted formatter, the original 34 checks, the five acceptance example groups and the 44 Chapter 5 checks. Formatting a string has never meant sending email; notification output is a new feature here, deliberately kept small. The companion supplies complete code, the deliberately coupled intermediate version, the checks, solutions and teaching notes. Testing terminology and frameworks receive fuller treatment in Chapter 7.

Read Sections 6.1 to 6.6 as the core; Section 6.6 compares manual assembly with a small educational container. What you must do is steps 1 to 9 of the practical activity in Section 6.10, including the container walkthrough in step 7. Steps 1 to 8 run on the book's project and step 9 applies the same analysis to your team's project. If you want more, try the AI activity in Section 6.8, the extension at the end of Section 6.10, the questions in Section 6.11 and the transfer challenge in Section 6.12.

6.1 Why dependencies make change hard

Start with a real requirement

A dependency exists when one part of the program needs knowledge or behaviour from another. Dependencies are necessary: the coordinator needs dates, reservations and borrowing decisions. The design problem is choosing which knowledge should remain stable and where changing details should live.

Chapter 5 left two visible decisions inside the coordinator: construction of concrete policies and selection of a policy by category. The formatter was also constructed there. This was a deliberate halfway design. We now move assembly out of the request workflow while introducing a notification boundary.

The new demonstration requirement is:

> After storing an accepted reservation, attempt one notification through the configured output. Rejected or invalid requests produce no notification. The configured output may record the accepted reservation in memory or print a notice to a supplied stream.

These are local outputs for teaching. Neither is an email service, a durable queue or proof that a borrower received anything. When an output returns normally, that means it completed its own local operation. It does not mean anything was delivered remotely.

Looking at the coupled version

The companion first adds a deliberately coupled version containing a field initialiser like this:

```
private final NotificationSender notifications =
        new ConsoleNotificationSender(System.out, new ConfirmationFormatter());
```

The field has an interface type, but the coordinator still names the console implementation, chooses the global output stream and constructs its formatting dependency. A change to those choices requires changing the coordinator’ s source. The interface alone has not separated configuration from use.

This halfway version is new teaching code, and it does not describe Chapter 5. Adding notification is a feature change. Replacing direct construction with an injected collaborator is a later structural change, and it comes with an explicit change to the constructor.

The strongest reason to change the design is concrete: one caller wants console output while another wants to inspect the accepted records in memory. The same reservation algorithm should support both. There is no need to invent a remote provider, credential store or messaging platform to demonstrate that pressure.

Which decisions should stay where they are?

The coordinator should still decide when a reservation has been accepted and therefore when notification is eligible. Moving that decision into the sender would duplicate policy and conflict handling. The sender should decide how its local output handles the accepted record, without granting or rejecting the reservation itself.

The coordinator also needs an application-level answer to output failure. Object injection does not settle that question. The chapter adopts an explicit rule: retain the accepted reservation and return its normal confirmation with a failure suffix. No automatic retry occurs. Section 6.3 explores why that rule matters.

A useful dependency diagnosis lists both the unwanted choice and the decision that belongs in the client. "Remove all dependencies" is not a useful goal. "Remove the coordinator's knowledge of which notification provider to construct, while keeping its acceptance sequencing" is reviewable.

6.2 Designing an interface around what the client needs

Design for the caller's role

So what should the coordinator ask of a notification output? Very little. ISP warns against making a client depend on operations it does not need [1], and in the reservation workflow the needed role is small: accept an already confirmed reservation for notification output.

The contract is:

```
public interface NotificationSender {
    void send(Reservation reservation);
}
```

Compare it with a proposed all-purpose `EquipmentPlatform` interface containing `send`, `registerBorrower`, `exportInventory`, `deleteAccount`, `configureMailServer` and `rotateCredentials`. The coordinator uses only the first operation. The larger interface exposes unrelated responsibilities and encourages implementations to supply meaningless methods or throw unsupported-operation exceptions.

A smaller client-facing role lets a console output and an in-memory recorder implement what they genuinely support. It does not require every capability of a future provider to be copied into the coordinator's interface. An administration tool may need a different contract if that requirement actually exists.

Small is not the same as one method

The notification contract happens to have one method. ISP does not prescribe one-method interfaces. A client that genuinely needs several cohesive operations may benefit from a role containing all of them. Splitting them mechanically can make construction and state coordination harder.

For example, a future document-writing client might need both writing and closing a session. Whether those belong together depends on who owns the resource and what the client must guarantee. Separating them only to minimise method count could allow a write without the required lifecycle management.

For our recorder, `received()` is an inspection operation exposed by the concrete implementation. The coordinator does not need it, so it is not placed on `NotificationSender`. A check can keep a reference to the recorder while supplying the same object through the narrow sender role. That is a deliberate use of two views of one object.

Saying what happens, and what changes

A signature cannot say everything. The chapter's sender contract specifies a non-null accepted reservation; it must not modify that record or the coordinator; expected output failures use `NotificationException`. The recorder appends the immutable record to its own local list. The console implementation prints a notice to its supplied stream.

Both implementations have output effects, so this is not a pure policy operation. Chapter 5's `BorrowingPolicy` remains deterministic and side-effect-free under its contract. Do not apply one collaborator's restrictions blindly to another. The common idea is that callers need accurate promises, not that every interface should forbid mutation.

The sender receives a `Reservation` rather than raw borrower identifiers and dates assembled afresh. This keeps the notification tied to the accepted record. It also means the boundary deliberately knows that small domain type. If an external transport later needs a different representation, an adapter can translate it; that future need does not require a transport-specific payload now.

Leaving unused capabilities out

A provider SDK might expose account management, diagnostics and bulk operations. A client-specific interface should express the application's needs, not mirror the SDK. Otherwise a provider change can still force application changes even though every reference is labelled as an interface.

The book does not implement those external operations here. The important review question is whether the coordinator's contract can be explained without referring to a vendor or transport setting. "Send this accepted reservation to the configured output" can. "Send this vendor request with the provider's retry token" would pull provider details into the workflow.

6.3 Who decides what, and what happens when notification fails

Three different decisions

Before we handle failure, it helps to see who is deciding what. There are three different kinds of decision in the worked example:

Decision	Owner in this design	Reason
Whether the request is acceptable	Coordinator and selected borrowing policy	Depends on reservation rules and current records
How notification is represented locally	Sender implementation and formatter	Depends on the chosen output
Which concrete objects are used	Application composition root	Depends on the demonstration environment

The first decision should not depend on console configuration. The last decision has to be made somewhere, because a running program eventually needs concrete objects. Moving the assembly to the boundary makes that decision explicit instead of pretending it disappears.

Likewise, "policy" can mean different levels of decision. A numeric borrowing policy decides limits. The coordinator implements the broader application policy about storing before notifying and handling expected notification failure. Both are distinct from how a console stream writes bytes.

Spelling out the partly successful case

The completed operation stores the reservation, formats the confirmation and calls the sender. On an expected `NotificationException`, it returns the confirmation followed by:

```
Notification failed; reservation retained.
```

The leading space joins the suffix to the existing confirmation. The record remains stored and counts as active. The method does not delete it, claim that the reservation was rejected or attempt notification again.

That rule is the author's own choice for this book. Another application might choose a different outcome, but it would need to make the storage and communication consequences explicit. A result that simply says "failed" can encourage callers to repeat the reservation request even though it already succeeded.

Repeating the same request here normally produces an overlap rejection. It is not a notification retry operation. Nor does the suffix mean that a background worker will retry later: there is no queue or worker in this version. Do not describe failed notification as "pending" unless the system actually retains and processes such work.

Calling a sender is not the same as delivering a message

The coordinator attempts the sender once per successful reservation call. This is a local control-flow fact, not an exactly-once delivery guarantee. A future remote sender might deliver a message and then fail before confirming it. Retrying could duplicate the delivery.

Even the console may partially write before reporting an error. Its implementation uses `PrintStream.checkError()` because `PrintStream` normally records I/O errors rather than throwing `IOException` to the caller [6]. The sender translates that expected local error into the application's notification exception.

One limitation follows from how `PrintStream` works: its error state is sticky. Once the stream has reported an error, `checkError()` keeps returning `true` even if a later write succeeds, so every later notice through the same console sender would be reported as failed. For a console demonstration that is acceptable, and the companion shows it with a short experiment. A long-lived sender would need a fresh stream or a different way to detect failure.

The coordinator catches that specific exception. Unexpected programming failures are not converted into apparent success. Such failures may still occur after storage; the example does not claim transactional atomicity across formatting, storage and output. A richer result model and reliable delivery design belong to later integration work.

Writing the examples down before refactoring

Example	Expected reservation result	Notification observation
Eligible available request	Existing exact confirmation; one record stored	One accepted record supplied to sender
Same-item overlap	Existing rejection; records unchanged	No call
Invalid duration	Existing rejection; records unchanged	No call
Invalid identifier	Existing exception; records unchanged	No call
Expected output failure after acceptance	Confirmation plus explicit failure suffix; record retained	One attempt; no automatic retry
Repeat of the already accepted request	Overlap rejection	No second attempt from that rejected request

These examples pin down the feature whether assembly is done by hand or by a container. They also show what the new checks must observe beyond the return strings we already had.

6.4 Who owns the abstraction? Dependency inversion

Which way do the source dependencies point?

Having a small interface is one thing; deciding who owns it is another. DIP keeps higher-level policy apart from changing implementation details by putting an abstraction between them [1, 2]. In the worked example, the coordinator names `NotificationSender`, an application-owned role. Concrete outputs implement that role. The application's contract does not import a provider's SDK type or inherit its configuration vocabulary.

At runtime, the coordinator still calls a concrete object's method. Dependency inversion does not reverse that call. It changes what the coordinator's source needs to know and which side defines the abstraction.

Figure 6.1 draws the finished design as a class diagram, using the notation from Chapter 2 and the style of Figure 5.1.

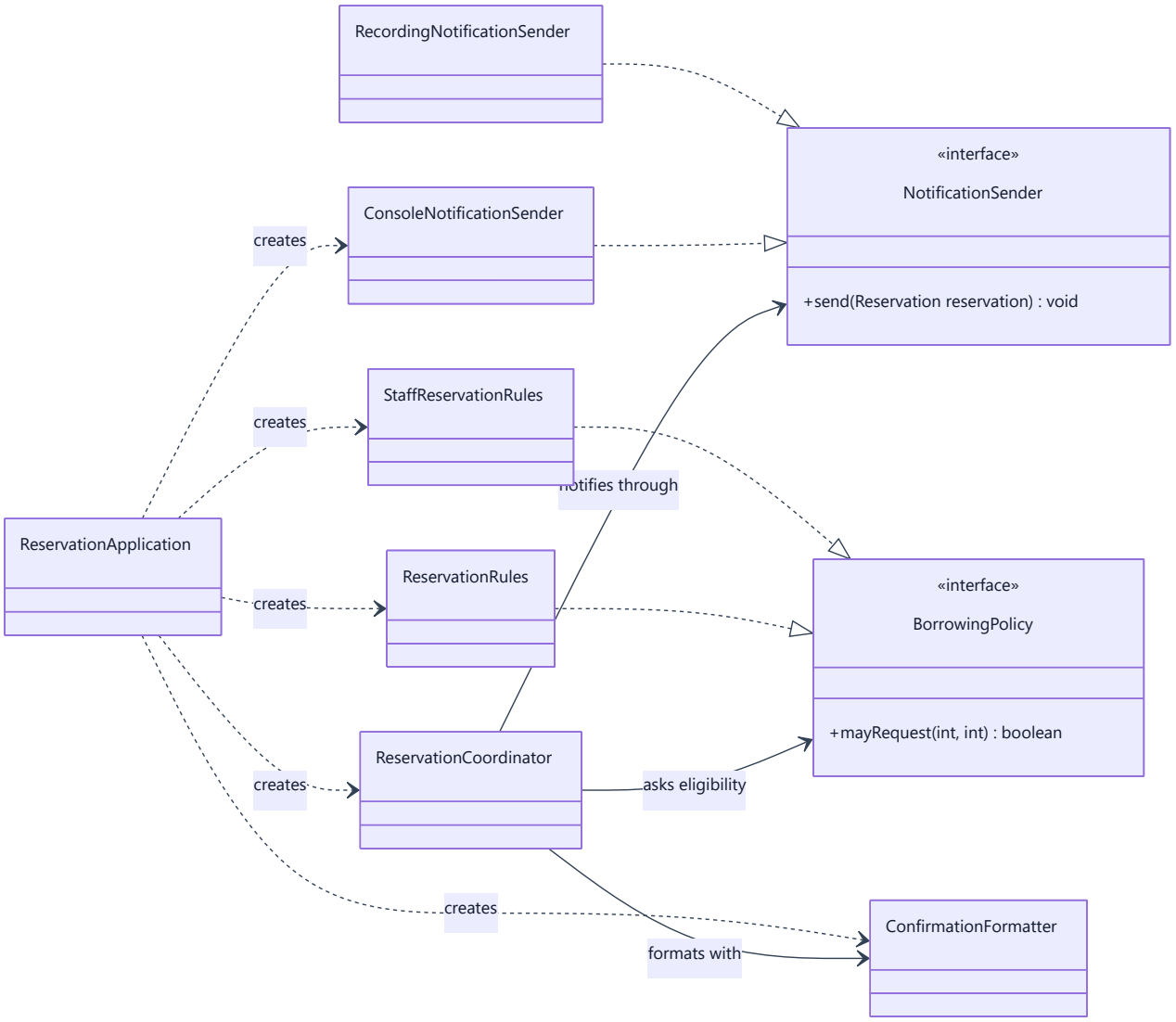


Figure 6.1. Source-level dependencies after the refactoring. Solid arrows are associations: the coordinator holds a `NotificationSender`, a `ConfirmationFormatter` and a `Map<BorrowerCategory, BorrowingPolicy>`. The map is drawn as one association to `BorrowingPolicy` to keep the figure simple: it means the coordinator asks whichever policy the map holds for the borrower's category. Dashed lines ending in a hollow triangle are realisations. The dashed arrows labelled "creates" show which class's source names which concrete class.

Compare it with Figure 5.1. There, two dashed "creates" arrows ran from the coordinator to the concrete policy classes. Here, every "creates" arrow starts at `ReservationApplication`. The coordinator's outgoing arrows now end at roles, apart from the concrete `ConfirmationFormatter`, which stays concrete on purpose (see below). That change in arrow direction is the visible evidence that its source no longer chooses its environment.

The arrows show **source-level knowledge**, not time order. The composition root knows the concrete console implementation because choosing it is the root's job. The recorder is an alternative implementation, not an additional mandatory destination for every notification, so the application creates only the console sender. Figure 6.2 in Section 6.5 shows a different view, the objects assembled at run time.

The files still use the unnamed Java package for now. Who owns each contract is a design decision here, and it is not a claim that the build enforces module boundaries. Chapter 10 will organise larger application boundaries more explicitly.

Injection without inversion

Suppose the coordinator receives `ConsoleNotificationSender` through its constructor. That is dependency injection: somebody else supplies the collaborator. The coordinator still depends on the concrete provider type. Some construction coupling has been removed, but the intended provider boundary has not yet been inverted.

Conversely, a field typed as `NotificationSender` that is assigned `new ConsoleNotificationSender(...)` inside the coordinator uses an abstraction while retaining concrete construction knowledge. This is precisely the intermediate defect we are examining.

For the final design, use the small application contract and pass the selected implementation from the application root. The two decisions support each other, but they answer different questions: what type of behaviour the client requires, and who supplies the object.

A preview: outside data behind a contract

The same reasoning applies when a collaborator brings data from outside the application. Chapter 8 adds the book's data-backed collaborator, an equipment directory that answers "who is this borrower?". The reservation code should not know how that directory is implemented, only what it promises. Its contract, which Chapter 8 builds and tests, is one method:

```
/** Empty means absent or inactive, never a provider outage. */
@FunctionalInterface
public interface BorrowerDirectory {
    Optional<Borrower> find(String borrowerId);
}
```

This listing is only a preview. It is not part of this chapter's code, and `Borrower` is a small type that Chapter 8 introduces. Notice three promises in it. An empty answer means "no such active borrower", which is an ordinary result. An outage is a different outcome, and Chapter 8 expresses it as an exception, so that "not found" and "could not ask" can never be confused. And the contract speaks in the application's own types, so the provider's codes stay behind it. The application owns this interface, in the same way that it owns `NotificationSender` here. That is dependency inversion applied to data.

Not every concrete dependency is a problem

The coordinator still constructs accepted `Reservation` values and `LoanPeriod` values. It uses an in-memory list. The chapter does not inject a factory for every `new` expression. Creating a small domain value as part of an operation differs from selecting an environmental output provider.

The concrete `ConfirmationFormatter` is supplied in the constructor, but no interface is introduced for it. Its separate responsibility was justified in Chapter 5; a substitutable formatting role has not been required here. A concrete dependency can be acceptable when its stability and role fit the client.

Likewise, using Java's `Map` interface does not by itself demonstrate DIP. The useful design decision is that policy mappings are assembled outside the coordinator. The collection interface is an existing representation that makes this small mapping explicit.

6.5 Handing collaborators in: injection and the composition root

Making required collaborators visible

The completed constructor requires a policy map, a formatter and a sender:

```
public ReservationCoordinator(
        Map<BorrowerCategory, BorrowingPolicy> policies,
        ConfirmationFormatter formatter,
        NotificationSender notifications)
```

The constructor checks for null values and requires an entry for every current borrower category. It copies the map with `Map.copyOf`, which produces an unmodifiable map and rejects null keys or values [5]. A caller cannot change the coordinator's category mapping by editing its original map later.

This is a shallow copy. It does not clone the policy objects. The supplied policies are stateless, satisfying their existing contract; putting a mutable or dishonest policy into an unmodifiable map would not make that policy safe. Construction checks validate assembly, not every collaborator's behaviour.

The coordinator then uses `policies.get(category)` instead of its former category switch. The category-to-implementation mapping moves to the application root. Adding a future enum category still requires a corresponding mapping; missing configuration fails at construction rather than halfway through a request.

Assembling once, at the edge of the application

`ReservationApplication.create(sender)` builds the student and staff policies, places them in the map, creates the formatter and passes all three collaborators to the coordinator. The main method chooses a console sender with a supplied stream and starts the demonstration.

A check chooses a recorder instead:

```
RecordingNotificationSender inbox = new RecordingNotificationSender();
ReservationCoordinator coordinator = ReservationApplication.create(inbox);
```

The check can later inspect `inbox.received()` because it retained the concrete reference. The coordinator only sees the sender role. This arrangement demonstrates replaceability without a mocking library; Chapter 7 will develop the vocabulary and uses of test doubles.

Two views of the same design

Figure 6.1 showed classes and which source files know about which. Figure 6.2 shows the objects that `ReservationApplication.main` actually builds when the program runs.

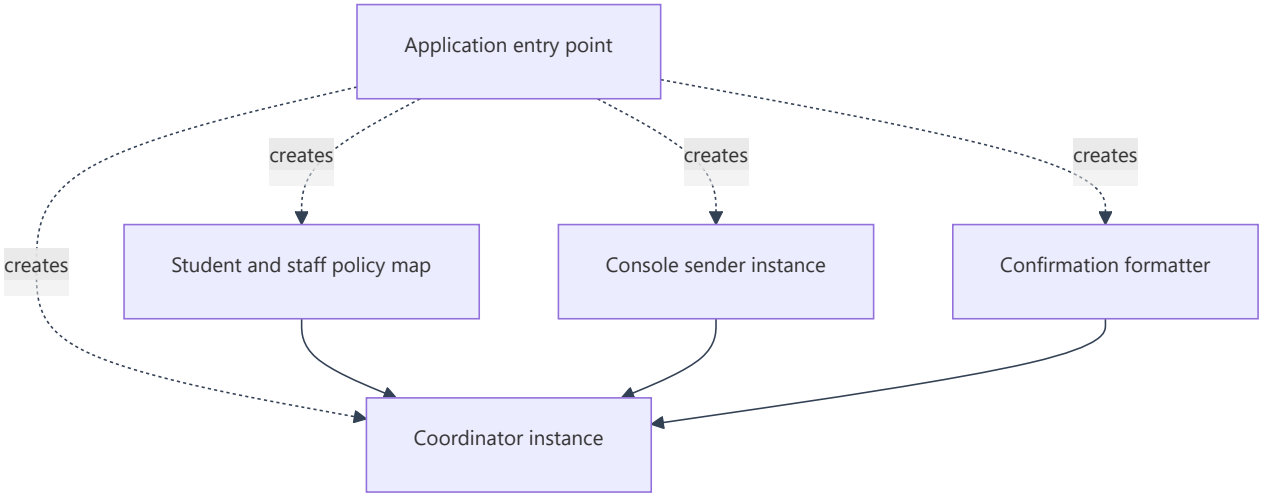


Figure 6.2. An object-assembly view. The boxes are objects created at run time, not classes. A dotted arrow labelled "creates" means the application entry point constructs that object. A solid arrow means "is passed to the constructor of". The sender also receives its own formatter and output stream, which are left out to keep the central relationship readable.

Keep the two views apart. Figure 6.1 says which classes know about which; Figure 6.2 says which objects were wired to which. Neither shows the order of calls during a request: the coordinator checks eligibility and conflicts, stores a record, formats the result and calls the sender that was already supplied. Do not read event order from either diagram.

A **composition root** is the small area at the application's entry point where concrete collaborators are put together. It may call helper factory methods. Those helpers do not become a service locator merely because they are static; the important point is that the coordinator is passed its dependencies and does not call back into a global registry to find them.

The constructor change, stated honestly

The chapter removes the no-argument coordinator constructor. Callers must now supply dependencies, normally through `ReservationApplication.create`. Therefore the completed source is not a drop-in constructor-compatible replacement for Chapter 5.

Existing checks and demonstrations are migrated only at their construction expressions. Their reservation assertions remain the same. This preserves the old operation's tested domain behaviour under a recording sender while making setup explicit. Claiming that every old source line is unchanged would be inaccurate.

The four-argument `reserve` method still means student borrowing. The explicit category overload still uses the trusted category supplied by the caller. Student/staff limits, date boundaries, count behaviour and existing rejection messages remain as before on their ordinary paths. The new notification side effect and documented failure suffix are separate feature behaviour.

Constructor or setter?

How should the coordinator receive its collaborators? For anything it cannot work without, the constructor is the natural place, because an object cannot be used before those dependencies are supplied. With later setters, the program may temporarily contain an unusable coordinator, and every operation may need to ask whether configuration has happened yet.

That does not make setters universally wrong. Optional or intentionally replaceable runtime collaborators can have a different lifecycle. This coordinator has no requirement to swap its sender while reservations are being processed, so construction is the simpler place to establish the dependency.

A very long constructor can expose a class that has too many responsibilities, but parameter count alone does not prove it. Do not hide ten unrelated services inside a generic context object solely to shorten the signature. The dependencies would still exist, with less visibility.

6.6 Inversion of control and containers: what do they add?

Telling the terms apart

You will meet three terms around this topic, and they are easy to blur, so it is worth fixing them before we look at a container. Dependency injection is a way of supplying collaborators from outside an object. Inversion of control is a broader idea: another component controls some aspect of execution or assembly that the object would otherwise control itself [3, 4]. A dependency-injection container is a tool that uses registration or configuration to assemble and often manage objects.

The manually assembled coordinator already uses DI. It does not need annotations or a framework to qualify. A container can automate repeated assembly, but it cannot decide whether the contract is well chosen or whether the failure policy is correct.

Term	Question answered in this example
DIP	Does the coordinator depend on the application’s notification role rather than a provider detail?
DI	Who supplies the sender, formatter and policy mapping?
IoC	Which component controls assembly or later calls into application code?
DI container	What mechanism resolves registrations and manages selected object lifetimes?
Service locator	Does application code fetch collaborators from a registry rather than receive them explicitly?

Walking through a small container

The companion includes an **educational single-threaded container**, `DemoContainer` in the file `ContainerWalkthrough.java`, kept apart from the running application. It is the container this book’s Week 5 lab uses. It registers a provider function for a type and lazily creates one instance of each registered type. It caches those instances, reports a missing binding and detects a construction cycle.

The walkthrough registers a formatter, a notification role backed by the recorder, and a coordinator provider. That provider resolves the formatter and sender and passes them to the same ordinary constructor. Only configuration code uses the container. `ReservationCoordinator` has no container import, field or lookup call.

Trace the first resolution of the coordinator. The container calls its provider, which requests the formatter and sender. Those providers run, and their instances are passed to the coordinator. A second resolution returns the cached coordinator. The reservation stored through the first reference is visible through the second because both references identify the same object.

This makes lifetime a design decision. Sharing the coordinator means sharing its in-memory reservation list. A container does not make that list thread-safe. Creating a fresh coordinator for every request would instead discard earlier reservations unless storage were shared elsewhere. Neither lifetime is automatically correct for a future server application.

What configuration failures teach

A missing sender registration should fail with an identifiable configuration error. A cycle in which a provider tries to resolve itself should also fail. The educational program demonstrates both outcomes without pretending that they are business-rule rejections.

Its limitations are deliberate: no reflection, annotation scanning, concurrent resolution, scopes beyond one cached instance per registered type, or resource cleanup. Do not extend it into a home-grown production framework as part of this chapter. Its purpose is to expose registration, resolution and lifetime in ordinary Java.

Real frameworks supply much more machinery. Spring’ s reference documentation describes a bean as an object that its IoC container instantiates, assembles and manages, with the dependencies among beans reflected in configuration metadata [7]. The manual graph in this chapter gives you a basis for reading such configuration. A framework container also arrives as libraries with their own dependencies, which the plain `javac` gate used so far does not manage. The executable code in this chapter uses no framework. The Week 5 lab uses the educational container above, not a library container, so everything the lab needs is in this chapter and its companion. A library container does the same three jobs (registration, resolution and lifetime) with much more machinery and with libraries of its own to manage. Treat the educational container as a way to see those three jobs in plain Java, and the manual graph and the container’ s configuration as the map you will use when you meet a library container in a project.

Whichever container you use, identify its registrations, supplied constructor arguments and instance lifetimes before adding annotations. The library or framework name should not replace an explanation of the object graph.

6.7 What abstraction costs

Give each boundary a job

Every interface we added should be able to say what it is for. The new notification interface earns its place because the same client needs two outputs and a predictable failure boundary. The borrowing-policy interface already expresses a genuine variation. The formatter remains concrete. No interface is added to immutable `Reservation` or `LoanPeriod` values.

The root is allowed to know concrete classes. If every assembly choice is hidden behind another factory, the program can become difficult to trace without protecting any additional change. Good separation makes the changing decisions easier to locate; it should not make it impossible to find which object is actually used.

The policy map removes a category switch from the coordinator but adds a configuration invariant: every category must have a non-null policy. Constructor validation makes that cost visible. A design explanation should include it, rather than presenting fewer conditionals as an unconditional improvement.

Spotting a disguised dependency

Sometimes a design looks injected and is not. An injected `ApplicationContext` that offers arbitrary lookups can hide all the coordinator's real needs. A method that calls `context.get("mail")` is still locating its own collaborator. The constructor may be shorter, but reviewers must inspect the entire class to discover what is required.

Similarly, an interface whose `send` method requires server addresses, passwords and transport-specific objects transfers provider setup into every caller. The interface exists syntactically while the application remains coupled to implementation details.

Use concrete review questions: can I choose the recorder without editing reservation logic? Can I see the required dependencies at construction? Can a provider-specific change remain at the boundary? Can I explain the state after expected output failure? Those questions reveal more than counting interface declarations.

What this chapter has not solved

The application is still an in-memory, sequential teaching program. It has no authoritative borrower categories, persistent notification history, retry queue, remote delivery, cancellation or read-only availability enquiry. It also does not implement the whole MVP reservation-and-loan flow.

The new structure makes selected changes and tests easier. It does not manufacture those features. Keep backlog claims aligned with actual behaviour, and carry explicit limitations into the next iteration. Chapter 7 will use these replaceable collaborators to examine test design and evidence more deeply.

6.8 Reviewing an interface that hides very little, with AI

Chapter 15 gathers the AI workflows that recur in this book. This activity is complete on its own, and the ready-made output below is all you need.

	For this activity
Your goal	Judge whether a generated refactoring really separates configuration and policy from the reservation workflow
You start with	The coupled coordinator from Section 6.1, the notification requirement and Figure 6.1
What is in bounds	Evaluate the proposal and draw a corrected object graph; do not apply it wholesale
How to check	The coordinator's source (which concrete classes does it name?), the recorder-based checks, and the failure case
How to decide	A verdict with a concrete reason for each proposal, then a corrected constructor
No AI account?	The ready-made output below is enough

The text below is **fictional AI output written by the author** for an offline exercise:

> Add `NotificationSender` and type the coordinator field with it. Inside the coordinator constructor, read the provider settings and create `ConsoleNotificationSender`. Add `configure`, `exportHistory` and `deleteBorrower` to the same interface so it can support future features. Catch every runtime exception and still return the ordinary confirmation. Pass a global container into the coordinator so future dependencies can be found when needed. The new interface means DIP and DI are both complete.

Evaluate each proposal. Identify where configuration still occurs, which operations the client actually needs, and what a caller would learn about notification failure. Explain the difference between receiving a sender and receiving a registry from which the coordinator finds a sender.

Produce a corrected object graph and a small constructor. If you use a live AI service, give it only synthetic code and requirements, keep the output you evaluated and record how you verified it independently. The ready-made output gives you the same activity without an account.

6.9 Dependency habits that mislead

Mistake	Why it is misleading	Better action
An interface-typed field proves DIP	Concrete construction or provider-specific types may remain	Inspect the actual source dependencies
DI means a framework	Ordinary constructors can receive dependencies	Explain assembly before adding tooling
Every class needs an interface	Many stable values have no relevant implementation variation	Name the client need and change pressure
A sender must expose provider administration	The coordinator does not need those operations	Keep the application role narrow
A container singleton is thread-safe	Lifetime and concurrent access are separate questions	Inspect mutable state and execution model
Notification failure means reservation failure	Storage may already have succeeded	State the partial-success outcome explicitly
One call means exactly-once delivery	Output may fail after partial or remote effects	Describe only the guarantee actually implemented
Old checks passed, so all compatibility is preserved	Construction changed and notification adds effects	Separate API migration from retained domain behaviour

6.10 Try it yourself: assemble a replaceable output

You start from Chapter 5’ s finished code, this chapter’ s notification requirement and the companion listings. Keep using Java 17 and the existing plain-Java gate. The container walkthrough is separate from the application. Steps 1 to 8 run on the book’ s project; step 9 runs on your team’ s project.

If your instructor demonstrates the task, they will compare the Chapter 5 baseline, the deliberately coupled notification version and the injected version, and ask you to point to every concrete provider choice and say which one belongs at the application entry point. For guided practice, inject a recorder, accept one reservation, reject an overlapping request and inspect both the stored reservations and the recorded

notifications. Then use an output that throws the documented notification exception and explain the result.

Then work through these steps:

1. Define the sender role and identify at least two operations it should not contain.
2. Move output construction and category-to-policy mapping to the composition root. Require valid collaborators in the constructor.
3. Migrate existing caller construction while preserving existing reservation assertions.
4. Demonstrate both console and recording outputs using the same coordinator implementation.
5. Verify no notification on rejection, one attempt after acceptance, and the chosen failure outcome without automatic retry.
6. Draw separate source-dependency and object-assembly views for your solution, in the style of Figures 6.1 and 6.2, labelling what each arrow means. In the source-dependency view, check that every "creates" arrow starts at the application class.
7. Run the educational container walkthrough and explain shared state, missing registration and cycle detection.
8. Review the change and write a rationale for every new boundary. Retain real review findings rather than a fictional approval.
9. On your team's project, find one environment or provider choice that a class constructs for itself, such as an output, a storage mechanism, an external service or configuration. Sketch where it would be assembled instead, using both views.

What you should end up with is: the injected notification contract; two implementations; explicit assembly code; migrated and additional checks; source-dependency and object-assembly diagrams; an explanation of the failure policy; a short design rationale; and the step 9 sketch for your team's project.

You are done when swapping the provider needs no edit to the reservation algorithm, setup failures can be told apart from reservation rejections, the old rule expectations still hold, and an expected notification failure keeps the record and reports its outcome accurately. For any formal submission, follow the module leader's assessment requirements.

If you want more, compare the supplied map with a dedicated policy-resolver object in a separate design sketch. Explain the benefit to the client before you add another interface, and do not implement a general rule registry without a requirement.

6.11 Questions to think with

Concept questions

1. Why does `NotificationSender` omit the recorder's `received()` method?
2. How can constructor injection be present while the intended DIP boundary is still missing?
3. What does the composition root know that the coordinator should not know?

4. Why can sharing a coordinator through a container change application behaviour even when its methods are unchanged?

Applied diagnosis

5. A coordinator catches a sender failure, deletes the accepted reservation, and returns "Rejected". The sender may already have printed or delivered a notice. Diagnose the inconsistent outcome and compare it with this chapter's explicit failure rule.

6. A constructor copies a map of policies. A reviewer says this proves that policy behaviour can never change. Explain what the copy protects and what still depends on the policy objects' contracts.

6.12 A different setting: the transfer challenge

A document conversion service validates a request, converts a file and sends a completion notice. A desktop demonstration displays notices in a window; an automated run records them in memory.

Define a client-specific notification role, identify the assembly boundary and describe what remains in the conversion workflow. Decide what callers should observe if conversion succeeds but notification fails. Draw a dependency diagram and give two acceptance examples that distinguish conversion state from notification state.

Use synthetic filenames and local demonstrations. Do not assume a background retry queue or reliable remote delivery unless you explicitly include and justify that additional requirement.

6.13 Revising for Quiz 1: where to look

Quiz 1 is held in Week 5 according to the module descriptor, but its stated coverage is **Weeks 1–4**: OO foundations, Agile practices and user stories, Git workflows, and the first SOLID principles. Revise Chapters 2–5 rather than treating all of Chapter 6 as newly included quiz content.

Chapter	Revision focus
2	Encapsulation, object responsibilities, cohesion/coupling, composition and inheritance, UML class diagrams, Git history, branches and small reviewed changes
3	Agile values in decisions, iterative/incremental delivery, Scrum/Kanban distinctions, stories, acceptance criteria and Definition of Done
4	Vision and scope, vertical slicing, capacity-aware planning, review workflow and meaningful CI evidence
5	SRP, OCP, behavioural contracts, LSP counterexamples and protected refactoring

Use the module leader's confirmed quiz guidance if it provides further detail. These pointers do not add assessment rules or predict particular questions.

What to take away, and what comes next

A useful interface describes the role a client needs. DIP is about the direction and ownership of dependencies; DI supplies collaborators from outside; a container is one possible way to do the assembly. The running example now makes its provider and policy choices explicit at its entry boundary, and it defines what you observe when an expected notification fails.

The running project now includes local console and recording outputs, explicit construction and 28 new checks. The existing reservation rules are still covered by their earlier expectations, with only the caller setup migrated. Chapter 7 develops test design, controlled substitutes and a complete red–green–refactor cycle.

Sources and further reading

Sources checked 12 and 13 October 2026. The notification contract, failure rule, source examples and educational container are author-created teaching material.

1. Martin, R. C. (2000), **Design Principles and Design Patterns**. The author's own discussion of ISP and DIP, hosted by the University of Turku. The link is a text PDF whose text was extracted and read on 14 October 2026: it states ISP as a preference for many client-specific interfaces over one general-purpose interface, and DIP as depending upon interfaces or abstract classes rather than concrete ones. [8] is the module's required treatment.
2. Martin, R. C. (2016), **A Little Architecture**. Policy, details and dependency direction.
3. Fowler, M. (2004), **Inversion of Control Containers and the Dependency Injection Pattern**. Separating configuration from use and distinguishing injection from service location.
4. Fowler, M. (2005), **Inversion of Control**. Broader IoC terminology.
5. Oracle, **Java SE 17** `Map`. `Map.copyOf` behaviour and its unmodifiable, shallow nature.
6. Oracle, **Java SE 17** `PrintStream`. Error-state reporting and `checkError()`.
7. Spring, **Introduction to the Spring IoC Container and Beans**. Framework-container context; no Spring installation is required for this chapter's executable walkthrough.
8. Martin, R. C. (2002), *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall. The module's required text; the Week 5 reading is its DIP and ISP chapters. Publication details as listed in the module descriptor.

Chapter 7: Testing as a Design Practice

> *Test-Driven Development (TDD), test doubles, fakes, stubs, and Mockito.*

OO Development and Agile Practices
Designing, Testing and Evolving Software with AI Assistance

A developer adds availability enquiry to the equipment system and reports that the build is green. During the demonstration, the officer asks about a camera that is clearly reserved and the system answers "available". The test behind that green build had checked only that the method returned a boolean without throwing.

It is not hard to imagine that test's neighbours. Another one rejects every enquiry and still passes the overlap example. A third accepts adjacent dates incorrectly, because it treats the end of the earlier reservation as inclusive. Every one of these implementations can look plausible when its tests ask incomplete questions.

So what are tests for? Not for collecting green indicators. They are there to create evidence about specific behaviour, to expose misunderstandings and to support changes in design. In this chapter you turn the availability examples from Chapters 3 and 4 into executable tests, moving from an acceptance example to a focused test with a meaningful expected outcome. You will go through a complete red–green–refactor cycle and learn why the first failure has to be the intended one. You will make data, time and collaborators controllable so that tests are repeatable, choose a useful test double, and tell state checks from interaction checks. You will learn to recognise weak or brittle tests and to say what a passing result does not show. Along the way you assemble the evidence for Milestone 2: a working MVP, its tests, a demonstration and a retrospective.

This is Week 6, supporting LO3 and preparing Milestone 2. The midterm practice near the end draws on Chapters 2 to 7, corresponding to Weeks 1 to 6. The chapter adds no assessment weights, deadlines or marking rules. The Week 6 readings are the testing chapter and "A Programming Episode" in Martin [9], and Part I of Beck [10].

Because Milestone 2 falls in this week, its checklist comes up front rather than at the end. Use it from the start; do not wait until the end of the chapter. By your demonstration you need:

1. a working MVP flow that someone else can run from written instructions;
2. acceptance examples mapped to named tests;
3. one red–green–refactor sequence recorded from your own code;
4. a short rationale for your object-oriented and SOLID choices;
5. a justified test double or controlled clock;
6. a build result for the revision you demonstrate;
7. real review feedback and a short retrospective with a next action; and
8. a statement of what the MVP does not do.

Section 7.8 explains each item. Start with items 1 and 3.

You start from Chapter 6’ s Java 17 application, with its explicit collaborator construction, student and staff policies and local notification outputs; the previous executable checks remain available. This chapter introduces **Maven, JUnit Jupiter and Mockito**, with an explicitly versioned configuration in the separate code file. Appendix B records the setup (B.4 and B.5), and the companion’ s `pom.xml` and commands are the exact reference. The companion also contains the full source of the running project, test listings, three TDD stages, solutions and observed verification results.

Sections 7.1 to 7.8 are the core reading, and Section 7.5 introduces hand-written doubles before Mockito. What you must do is steps 1 to 6, 8 and 9 of the practical activity in Section 7.11. Steps 1 to 6 run on the book’ s project; steps 8 and 9 run on your team’ s project, which is where your Milestone 2 evidence comes from, including the demonstration rehearsal, real review feedback and retrospective in checklist item 7. If you want more, there is step 7 (the same demonstration rehearsal on the book’ s project), the AI activity in Section 7.9, the questions and the transfer challenge in Section 7.12. The midterm practice set in Section 7.12 is optional too, but it is well worth doing: it practises the class-diagram, refactoring and unit-test work the midterm covers.

7.1 What does a test actually show? Behaviour and test evidence

Start from a promise, not an implementation

Chapter 3 proposed US-02: an officer can inspect availability without creating a reservation. Chapter 4 refined five acceptance examples. Those examples are now implemented, rather than left as planning material.

Example	Setup and action	Required observation
AQ-01	Empty schedule; enquire about CAM1 for a valid interval	Available; records and notifications unchanged
AQ-02	CAM1 reserved `[20 Oct, 22 Oct)`; enquire `[21 Oct, 23 Oct)`	Unavailable; no side effects
AQ-03	Same booking; enquire `[22 Oct, 23 Oct)`	Available because the end is exclusive
AQ-04	Same booking; enquire about MIC1 for the same interval	Available for the different sample item
AQ-05	Equal or reversed interval	Identifiable invalid-input outcome; no side effects

All dates in these examples are in 2026. The selected Java API is `isAvailable(itemId, start, endExclusive)`. It returns a boolean for a valid interval. Blank or null item identifiers cause `IllegalArgumentException`; equal or reversed dates do likewise; null dates cause `NullPointerException` through `LoanPeriod`. These choices settle the earlier open question about the API for this stage of the book’ s project.

Availability means no conflicting reservation for the item and interval. It does not mean that a particular borrower is eligible. The query accepts any positive interval representable by the dates, including more than seven days; student reservation limits still apply when a reservation is requested. It also does not know whether an item is physically on loan or present in a real catalogue. Those are distinct questions.

Looking at the result and the state together

The overlap example checks a false result, unchanged reservations and unchanged recorded notifications. An implementation that calls `reserve` and then undoes it would also send a notice, which the unchanged-notices check exposes. One that adds a temporary record directly and removes it again is harder to catch: with identifiers taken from the list size, nothing remains afterwards for a sequential test to see, and only a concurrent caller could observe the intermediate state. We therefore reject that design by reading it, not by testing it. A separate example confirms that an enquiry does not consume the next reservation identifier. With size-based identifiers it overlaps the unchanged-records check, but it protects the promise if identifiers later come from their own counter.

Choose observations according to the promise. For an immutable snapshot, comparing the before and after lists can expose replacement, deletion or reordering. Checking only the count misses changes to contents. Conversely, inspecting a private list field through reflection binds the test to representation when a public snapshot already exposes the behaviour of interest.

The expected answer must come from an agreed example or rule. Calculating it by calling the same method under test is circular. Copying the implementation’ s overlap expression into a test can reproduce its mistake. Concrete dates with an independently understood boundary provide a better starting point.

Adding a test runner without losing earlier evidence

So far our checks have been plain Java programs. For the new tests we want a runner. JUnit provides annotations, assertions and reporting for them [1]. Maven compiles the application and runs JUnit through the configured Surefire plugin [2]. The standard command becomes:

```
mvn --batch-mode verify
```

The existing `scripts/check.sh` now delegates to that command. The former plain-Java check classes remain intact. A `LegacyRegressionTest` invokes their four `main` methods from four JUnit tests, so the earlier checks are actually executed. Merely placing a class named `Chapter06Checks` in a test directory would not make its `main` method run automatically.

The wrapper reports four JUnit cases while their internal output still reports 34 Chapter 2 checks, five Chapter 3 example groups, 44 Chapter 5 checks and 28 Chapter 6 checks. Do not merge those counts as if they were identical units. The useful question is which behaviours execute and what failures reach the build.

7.2 What a test looks like: Arrange–Act–Assert

Making the test read like an argument

Most good unit tests follow a simple shape: Arrange–Act–Assert. You establish the relevant state, perform the operation being examined, and inspect the outcome. It is a reading aid, not a requirement to place exactly one statement in each phase.

The first availability test arranges a two-day CAM1 reservation and takes snapshots of records and notices. It acts by enquiring about an overlapping interval. It asserts that the answer is false and both snapshots still match.

```
boolean available = coordinator.isAvailable(
        "CAM1", start.plusDays(1), start.plusDays(3));

assertAll(
        () → assertFalse(available, "An overlapping item must be unavailable"),
        () → assertEquals(recordsBefore, coordinator.reservations()),
        () → assertEquals(noticesBefore, inbox.received()));
```

The test name describes the behaviour: `overlappingEnquiryIsUnavailableWithoutChangingRecordsOrNotifying`. A precise name saves the next reader from having to reverse-engineer why the setup contains particular values. Avoid names such as `test1` or `worksCorrectly` when the failure should tell another developer what promise was broken.

Several assertions can belong together when they describe one outcome. Splitting this test into three methods with identical setup may obscure the relationship between a query result and its non-mutation promise. Combining every query, loan and notification scenario into one giant method would create the opposite problem: a failure would be hard to locate.

Assertions that could actually fail

`assertTrue(true)` cannot reject an implementation. "No exception occurred" may be valuable for a specific promise, but it does not establish availability. A meaningful assertion must distinguish the behaviour you want from a plausible defect.

Use `assertThrows` for a specified exceptional outcome and inspect state afterwards when preservation matters. A catch block that simply prints an error and continues does not establish that the right exception occurred. Likewise, wrapping a test in a broad catch can turn a failure into a false success.

The companion uses parameterised tests for small tables of policy boundaries. Each row states category, count, days and expected decision. The test runner reports individual invocations, making a particular boundary failure visible. The table is deliberately authored from the rule: it is not generated by evaluating the policy itself.

Keeping setup small and independent

Each scenario constructs fresh mutable collaborators unless sharing is part of the behaviour being tested. Reusing a static coordinator across tests makes results depend on which test ran first. The same test may then pass alone and fail in the suite.

A helper such as `reserved()` can make a repeated one-reservation setup clear. A large universal setup that creates borrowers, loans, notices and dates for every test can conceal which state matters. Prefer the smallest setup that makes the scenario meaningful, while using real simple domain objects rather than mocking their accessors.

7.3 Red, green, refactor: one cycle in detail

Red: make the missing behaviour visible

Test-driven development, or TDD, works in short cycles: express an expected behaviour, observe its failure, implement enough to satisfy it and improve structure while preserving the tests [3]. This is a way of developing, not a label for tests added after a complete feature.

The decision at red matters: a failing build is useful only when the test reaches the behaviour it is intended to examine.

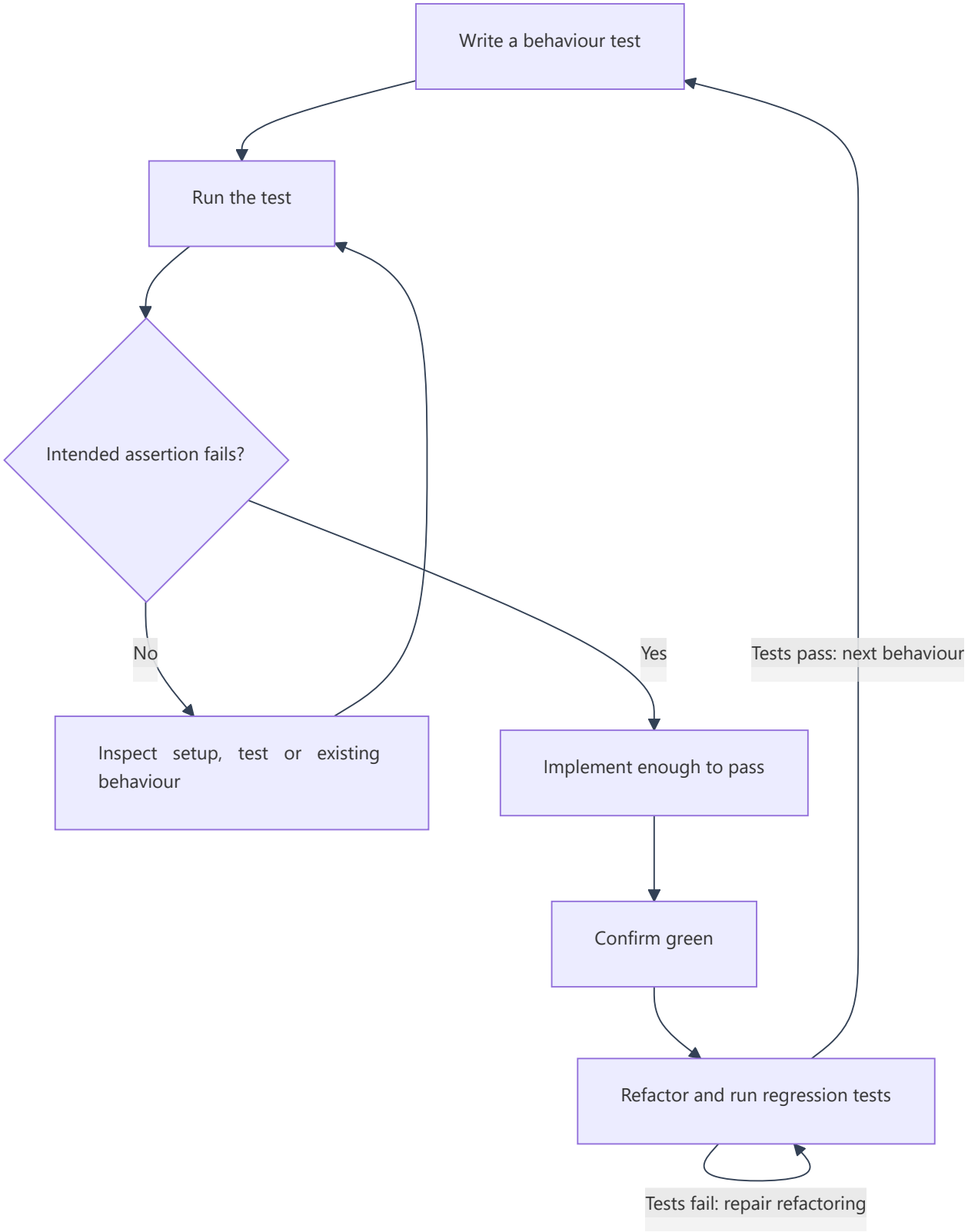

Write a behaviour test
Run the test
Intended assertion fails?
No
Yes
Tests pass: next behaviour
Inspect setup, test or existing behaviour
Implement enough to pass
Confirm green
Refactor and run regression tests
Tests fail: repair refactoring

Figure 7.1. The red–green–refactor cycle. The diamond is the test of the red stage: only the intended assertion failing lets the cycle move on to implementation. A different failure (a compile error, a missing dependency) sends you back to repair the setup, and a refactoring that breaks a test is repaired before the next behaviour starts.

The supplied worked cycle begins with a compiling `isAvailable` method that validates its arguments but always returns true. This intentionally incomplete method supplies an API for the first test without pretending to implement overlap detection.

Run only the new test:

```
mvn --batch-mode -Dtest=AvailabilityTddTest test
```

The observed red result is one executed test with one failure and zero errors. Its assertion reports that an overlapping item was expected to be unavailable but the method returned true. The unchanged-state assertions do not fail. This makes the missing behaviour precise.

A missing library, syntax error or inability to resolve a dependency would not be this red result. Those failures prevent or interrupt the experiment. Fix the environment or compilation issue, then rerun until the intended assertion exposes the missing behaviour.

Green: write just enough to pass

The green version constructs a `LoanPeriod`, inspects existing reservations and returns false if the same item has an overlapping period. If no conflict exists, it returns true. It does not call `reserve`, allocate an identifier or invoke the sender.

The same test then passes. This is useful evidence for that example, not permission to stop thinking. An implementation that always returned false would also satisfy the first overlap test. Additional examples are needed to distinguish overlap rejection from universal rejection.

The companion adds empty-schedule, adjacency, different-item, invalid-input, identifier-preservation and eligibility-distinction cases. These examples extend the specification. The complete author workflow includes one explicitly recorded TDD cycle; the chapter does not claim that every supplementary test or loan feature was developed test-first.

Refactor: share the conflict check

The green query duplicates conflict scanning already present in `reserve`. The refactoring extracts a private `hasConflict(itemId, requestedPeriod)` operation and uses it from both public methods.

The public reservation operation retains its validation and eligibility order. It still stores exactly one record after passing the conflict check. The query uses the same conflict meaning without acquiring reservation’ s mutation or borrower-limit behaviour.

Rerun the TDD test and the legacy bridge after extraction. The observed refactor run executes five JUnit cases: the new test and four legacy wrappers. They pass. This connects the structural improvement to both the new query and the previous reservation behaviour.

The extraction is not justified because duplicated text is always forbidden. It is justified because the two callers need the same meaning of a reservation conflict. If one caller later needs physical stock or maintenance status, that new meaning should not be forced into this helper merely to avoid duplication.

Keeping a trail you can show

The companion includes red, green and refactored coordinator snapshots plus their commands and results. In a team's repository, separate commits or a clearly recorded diff sequence can provide equivalent evidence. Keep the actual failing assertion and correction, not just a final screenshot saying tests passed.

A deliberately wrong implementation supplied after the fact can demonstrate test sensitivity. It does not reconstruct a historical test-first workflow that never happened. Be precise about whether evidence shows an actual development sequence or a later diagnostic experiment.

7.4 Edges, clocks and a small lending flow

Testing the edges that matter

Borrowing rules have useful boundaries: a student with two existing records may add a third; one with three may not. Seven days are allowed for a student, eight are not. The illustrative staff limits are five total records and fourteen days. Test values around those boundaries rather than a large number of unrelated ordinary values.

Overlap also has boundaries. An enquiry beginning at the exclusive end is available if no other reservation conflicts. Equal start and end is invalid. A different item does not conflict solely because its dates match. Each example distinguishes a plausible error in the rule.

Choosing boundaries is a way of reasoning. It is not an exhaustive proof over every possible integer or date. Combine examples with source inspection and, where justified, broader techniques later. Do not introduce a coverage target as a substitute for an explanation of the risks.

Taking control of time

Availability does not depend on the current date in this version, so it needs no clock. The new loan checkout operation does: checkout is allowed only on a date within the confirmed reservation's half-open period.

`LoanService` therefore receives a Java `Clock`. It obtains `LocalDate.now(clock)`. Tests use `Clock.fixed`, or a small controlled clock when the scenario must advance to a later date. Java's clock abstraction includes an instant and a zone, allowing a test to supply both [4].

A fixed UTC instant is not sufficient to define a local date without a zone. One test uses 19 October at 22:30 UTC with the Asia/Baghdad zone and expects the local checkout date to be 20 October. This is a deliberate example of zone-sensitive date conversion, not reliance on the machine's default zone.

Do not wait until midnight or call `Thread.sleep` to test a date boundary. Change the supplied clock. The application remains quick to test, and the relevant time is visible in the setup.

A small lending flow

The clock example above used checkout as its subject. This subsection states the rules of that small lending flow, which is a new feature and not a testing technique, so that the tests which follow rest on stated rules.

The worked MVP increment adds a small `LoanService` linked to one coordinator in the same process. It records checkout and return against a confirmed reservation. Its rules, which we invented for this book, are explicit:

Operation	Rule and state effect
Checkout	Reservation must exist; today’s date must be within its period
Repeat checkout	One reservation can create at most one loan, including after return
Item already out	A second active loan for the same item is rejected
Return	Only an active loan can be returned; repeated return is rejected
Return date	Cannot precede the recorded checkout date
Observation	Loan snapshots are unmodifiable and contain immutable loan values

A `Loan` is a Java record with validated identifiers, a checkout date and an optional return date represented by null. Here null has one documented meaning: the loan is active. Records provide concise value-oriented data declarations, but validation still has to be written [8].

The service replaces an active loan value with a returned value instead of mutating earlier snapshots. Tests can therefore compare current state while an earlier snapshot continues to describe the earlier state.

Not inventing lifecycle rules by accident

Returning a loan does not delete or cancel its reservation. The existing reservation-count rule still counts every stored reservation, and reservation availability remains based on those records. An early return therefore does not free a reserved interval or restore a borrower’s reservation allowance in this version.

This inherited limitation is explicit and tested. A production lifecycle would need agreed definitions for completed reservations and active counts. Introducing that rule now without revisiting earlier contracts would disguise a feature change as testing work.

The demonstration uses one coordinator and one loan service for the session. Multiple independent loan services would have independent loan maps; the example provides no shared persistent or concurrent protection. It is a bounded in-memory MVP flow, not a complete lending operation.

7.5 Test doubles: dummies, stubs, fakes, spies and mocks

Naming the role a substitute plays

Sometimes a test needs a collaborator that is hard to control, so we stand in a substitute for it. A test double replaces a collaborator for a testing purpose. The vocabulary below follows the commonly used distinctions described by Meszaros and summarised by Fowler [5]. Roles can overlap; the tool or class name alone does not determine the role.

Role	Purpose	Example in this project
Dummy	Supplies an argument that the scenario should not use	A sender passed to an immediately rejected request
Stub	Produces a controlled answer or failure	A sender that throws the expected notification exception
Fake	Provides a working simplified substitute	An in-memory output standing in for an external delivery destination
Spy	Records calls for later inspection	The recorder when a test examines received reservations
Mock	Supports verification of expected interactions	A Mockito sender checked for one call or no calls

The coordinator itself is the subject under test in notification scenarios. Mocking its `reserve` method would replace the behaviour we want to examine. Keep the real coordinator, rules and small immutable values, and replace the output boundary that would otherwise complicate observation.

A hand-written recorder is often clearer than a mocking framework. Its list makes the effect visible. A configured failing sender is similarly easy to understand. Learn those roles first so a framework's fluent syntax does not become a substitute for reasoning.

Bringing in Mockito for one specific interaction

The companion then creates a `NotificationSender` using Mockito. After a successful reservation, it verifies one `send` call and captures the supplied reservation to inspect its identifiers and period. Another test verifies no interaction for an invalid duration. A third makes `send` throw `NotificationException`, then checks the retained record and explicit failure suffix.

These checks concern public boundary behaviour, not private helper calls. The exact number of sender invocations matters because the selected feature says one attempt per acceptance and no automatic retry. Counting how often a private list is inspected would be a different, usually brittle kind of assertion.

Mockito can configure results and verify calls [6]. The supplied setup uses its interface-only proxy mock maker. All mocked types here are interfaces, so the mocks are Java dynamic proxies and no class mocking or static mocking is needed. This keeps the first framework example narrow.

One practical warning. On newer JDKs, Mockito may still attach its ByteBuddy agent when it starts, and the JVM then prints "A Java agent has been loaded dynamically". This appeared with the Mockito version printed in the companion POM, even with the proxy setting, and the tests still passed; passing `-XX:+EnableDynamicAgentLoading` to the test JVM removed the message. Other JDK and Mockito combinations may behave differently. The JVM states that dynamic loading will be disallowed by default in a future release, so check Mockito's documentation for the setup recommended for your JDK.

A preview: doubles for an outside data source

Chapter 6 previewed the contract of the book's data-backed collaborator, the equipment directory, and Chapter 8 builds it. You can already see how a test would substitute it. The contract has one method, so a lambda is enough, and each double below answers a different question that a future test will ask:

```
BorrowerDirectory knownStudent = id -> Optional.of(new Borrower(id, BorrowerCategory.STUDENT));
BorrowerDirectory nobody = id -> Optional.empty();
BorrowerDirectory offline = id -> { throw new DirectoryException("provider offline"); };
```

`knownStudent` is a stub with a fixed answer: it lets you test what the service does with a borrower it can trust. `nobody` is a stub that says "absent or inactive": it tests the ordinary rejection. `offline` is a throwing stub for the failure case: it tests that an outage is reported as an outage and not as "not found". None of them needs a framework, and none of them proves that a real provider behaves this way. Those three lines were compiled and run against Chapter 8's types; they print `STUDENT`, `false` and `failure: provider offline` when the three doubles are asked about borrower `B1`. This chapter's code does not contain these types, so the snippet is illustrative here and becomes real code in Chapter 8.

Matching the double to the question

Use a recorder when you want to inspect accumulated notices. Use a throwing stub when you need a predictable output failure. Use an interaction-verifying mock when the contractual call itself is the observation. A fixed clock supplies controlled time; it does not prove the real operating-system clock is correct.

Neither the recorder nor the mock proves that email is delivered. A fake may accept inputs a real provider rejects. Later integration checks must examine real adapters and their contracts. Chapter 13 covers that boundary more fully.

Avoid checking every possible interaction automatically. A blanket "nothing else was called" assertion can make harmless implementation changes fail. Verify only absence or ordering that is meaningful to the requirement. Here "no notification after rejection" is meaningful; "exactly three calls to the snapshot getter" is not.

7.6 What testing tells you about design

Difficult setup can reveal a design problem

Chapter 6's constructor injection lets tests choose a recorder, failing sender and policy mapping without altering the coordinator. The new loan service receives a clock for the same reason: time-dependent behaviour can be examined without editing production logic or waiting for a real date.

A difficult test may expose hidden global configuration, mixed responsibilities or an operation with unclear results. It may also expose a poorly chosen test. Before adding another interface, ask whether the scenario is trying to inspect a private implementation detail that should be irrelevant.

The availability query creates no records and has no notification side effects. Its clear contract made it easy to test result and state separately. Attempting to implement it by calling `reserve` would immediately entangle eligibility, identifiers and notifications. Test difficulty would then be useful feedback about the proposed design.

Using real, simple collaborators

`LoanPeriod` is cheap, deterministic and central to the rule being examined. Mocking it to say "overlaps" in every scenario would prevent query tests from exercising actual period semantics. The companion uses real dates and period objects.

Similarly, an ordinary list or map should usually remain real in these tests. Replacing it with a mock can lead to assertions about `add`, `get` and call ordering rather than whether the application preserved the right records. A substitute should remove an obstacle or expose a meaningful boundary, not erase the very behaviour under examination.

Keeping regression evidence during the migration

Moving to JUnit does not require rewriting every old assertion at once. The bridge makes the earlier check runners visible to the new build. New tests are focused JUnit methods with readable names and parameterised cases.

This bridge is transitional. Its four coarse test results give less precise runner-level reporting than individual migrated methods, even though internal assertion messages remain useful. Later migration can improve diagnosis without changing the expected behaviour. Do not discard earlier checks simply because their syntax predates the chosen framework.

7.7 Tests that look good but are weak or brittle

A test can execute code and still ask nothing useful

Coverage records which code was exercised under a particular measurement. It does not establish that the expected answer was correct or that the assertions would reject a defect. A test can execute every line of a method and accept every result.

Compare a query test that only calls `isAvailable` with one that expects an overlapping interval to be false and preserves the recorded state. The second distinguishes a meaningful defect. A further adjacency example distinguishes a different defect that the first overlap example would miss.

One useful diagnostic is to introduce a small, labelled wrong implementation in an isolated copy. The companion changes the end-boundary comparison so adjacent intervals incorrectly overlap. The selected adjacency test must fail. Restoring the correct comparison must return the test to green.

This is a targeted mutation experiment. It shows that one test detects one plausible fault. It does not prove that the suite detects every defect or justify changing tests until an incorrect implementation passes.

The companion repeats the idea with sixteen faults. Fourteen were caught at once. One revealed a real gap: the rule table says loan snapshots are unmodifiable, but no test checked it, so one assertion was added. The last could not be caught by any sequential test at all, because the faulty enquiry leaves nothing behind. A surviving fault is information to examine, not always a defect in the tests.

Tests must not repeat the same misunderstanding

An AI-generated implementation and AI-generated tests may agree because both assume inclusive end dates. Agreement is not independent evidence. Check expected values against the human-reviewed interval convention and concrete examples.

Also avoid expected values derived from the result itself. Asserting that a returned identifier equals the identifier in the same returned object establishes no independent fact. For the first accepted reservation, an explicit expected R1 is meaningful under this version's identifier contract.

Not tying tests to the implementation

A test asserting that the query calls private `hasConflict` exactly once would fail if the implementation used a different correct structure. The public result and state contract should normally survive that refactoring.

Exact text assertions can be useful when text is part of the existing contract, as it is for reservation return strings. They also reveal a cost: wording changes require deliberate expectation updates. A future structured result might separate domain outcome from presentation more cleanly, but that is a separate design change rather than an excuse to weaken current checks.

When a test fails, classify the failure before fixing it: requirement misunderstanding, implementation defect, outdated expectation, environment/configuration problem or test fragility. A green build obtained by suppressing the assertion's exit status answers none of those questions.

7.8 Preparing the Milestone 2 evidence

Demonstrating an outcome and its boundaries

The Chapter 7 demonstration now follows a complete small journey: inspect reservation availability, create a reservation, observe the changed availability, check out the item within the reserved period and return the loan. It records one accepted-reservation notice. It does not send additional checkout or return notices because those were not part of the selected requirement.

This gives teams a concrete example of a bounded MVP increment. Their assessed domain and scope may differ. The module leader's approved Milestone 2 requirements remain authoritative; the book supplies preparation rather than a new rubric.

Evidence	What it should establish
Working MVP demonstration	A coherent user outcome runs from documented setup
Acceptance mapping	Selected examples connect to executed tests
TDD sequence	Intended failure, implementation and protected refactoring are visible
OO/SOLID rationale	Collaborators and responsibilities have concrete reasons
Double choice	Controlled behaviour or interaction is justified
Build result	Actual tests ran against the intended revision
Review and retrospective	Real feedback, a change in understanding and a next action are recorded
Limitations	In-memory storage, lifecycle boundaries and missing integrations are explicit

A test report is not the whole milestone. Another person should be able to reproduce the demonstration and explain what remains outside its claim. A hosted CI run remains separate evidence from the local Maven runs performed for this chapter.

Making the retrospective useful

A retrospective note should connect an observation to an action. For example, a team might discover that availability tests had silently assumed the machine's current date, then decide to make all relevant time dependencies explicit. Record that only if it actually happened.

Avoid pre-filled success narratives. "We collaborated well" without an example does not explain how work changed. A useful note might identify a delayed review, its effect on integration and a concrete earlier point for review in the next iteration. Distinguish a proposed improvement from an improvement already tested.

7.9 Keeping implementation and expected behaviour apart, with AI

Use the reviewed AQ-01–AQ-05 examples as the authority. Ask an AI assistant for a candidate query implementation, then separately ask for omitted boundary cases. Evaluate both outputs rather than letting them validate each other.

Chapter 15 gathers the AI workflows that recur in this book. This activity is complete on its own, and the reviewed AQ-01–AQ-05 examples are all you need.

	For this activity
Your goal	Separate a proposed implementation from the evidence that would accept or reject it
You start with	AQ-01–AQ-05 as reviewed, and Chapter 6's coordinator
What is in bounds	Evaluate the candidate implementation and its tests; do not treat either as proof of the other
How to check	The reviewed examples, the labelled wrong-boundary mutation, and the build result
How to decide	A verdict with independent evidence for each claim
No AI account?	The ready-made output below is enough

For offline use, the text below is **fictional AI output written by the author** for this exercise:

> An enquiry can call `reserve` and remove the temporary record. Treat the end date as inclusive so adjacent bookings conflict. A test is sufficient if the method returns without throwing. Since the implementation and generated tests agree, coverage will prove correctness. For time-sensitive loan tests, use the current date and wait until tomorrow to test the next state.

Identify each unsupported claim, propose an independent expected result and describe the evidence that would reject the candidate. Run the supplied labelled boundary mutation in an isolated copy. Retain its actual failure and the restored pass.

A paid AI service is unnecessary. If you use a live service, give it only synthetic data and record the prompt, the relevant output, the claims you accepted or rejected and how you verified them independently. Do not manufacture a transcript or claim a test-first sequence that was not performed.

7.10 Testing habits that mislead

Mistake	Consequence	Better action
Count a dependency-download error as TDD red	Behaviour was never examined	Reach the intended assertion failure
Write expected values by calling the implementation	Reproduces the same mistake	Use independently reviewed examples
Mock the object under test	Replaces the behaviour being investigated	Keep the subject real
Use real time for a deterministic date rule	Results change with date or zone	Inject an explicit clock
Treat every recorder as the same double role	Hides why it is used	Describe its role in the particular test
Verify private call counts	Breaks harmless refactoring	Observe public behaviour or meaningful boundary interactions
Remove old checks during tool migration	Loses existing evidence	Bridge or migrate them explicitly
Call the bounded demo production-ready	Conceals lifecycle and integration gaps	State what the MVP actually demonstrates

7.11 Try it yourself: a test-first change and Milestone 2 evidence

You start from Chapter 6’ s finished code, the reviewed query criteria, the new loan-flow rules and the companion’ s Maven and JUnit setup. Steps 1 to 7 run on the book’ s project; steps 8 and 9 run on your team’ s project.

If your instructor demonstrates the task, they will run the labelled red stage, point out the precise assertion failure, apply the green implementation, extract the conflict detection and rerun both the new test and the legacy bridge. For guided practice, convert one policy boundary table into parameterised examples and discuss which incorrect result each row distinguishes, then compare a hand-written recorder with the Mockito interaction test.

Then work through these steps:

1. Reproduce the supplied TDD cycle or conduct a small new one for an agreed missing behaviour. Record the actual sequence.
2. Complete the remaining query examples, including side effects and adjacency.
3. Demonstrate the loan flow with controlled dates and rejection of an invalid return.
4. Choose a substitute for one collaborator and justify its role.
5. Run the labelled wrong-boundary implementation in a separate copy, identify the failing assertion and restore the correct source.
6. Run the full build and map selected acceptance examples to named tests.

7. Rehearse the MVP demonstration from documented instructions, obtain actual review feedback and write a short retrospective with a next action.
8. On your team's project, take one behaviour from your backlog and develop it test-first. Record the actual red failure, the green change and the refactoring, as the chapter's worked cycle does. Do not reconstruct a sequence afterwards.
9. On your team's project, work through the Milestone 2 checklist at the top of this chapter. Rehearse your own MVP demonstration from written instructions, obtain actual review feedback and write a short retrospective with a next action, as step 7 does for the book's project. Record which checklist items are complete, which are gaps and who will close each one.

What you should end up with is: runnable source; focused tests; TDD and mutation evidence; a rationale for your double; the actual build result; demonstration notes; review and retrospective notes; explicit limitations; and the step 8 and 9 records for your team's project.

You are done when the failure you saw was behavioural and you understood it, query results and state effects match the examples, time-sensitive tests do not depend on the actual day, the whole regression suite runs, and the demonstration includes reservation, checkout and return. For submission, use the approved module requirements.

If you want more, sketch how completed reservations should affect borrower counts and early-return availability. List every earlier contract and check that would need revisiting, and do not silently implement that policy as part of this exercise.

7.12 Questions to think with, and midterm practice

Concept questions

1. Why is "the method did not throw" insufficient evidence for an availability query?
2. What distinguishes the red stage of the worked cycle from a missing-dependency failure?
3. When does the notification recorder act as a spy, and why can a framework mock serve a similar purpose?
4. Why do both the instant and zone matter when testing a local-date checkout rule?

Applied diagnosis

5. A test calculates its expected availability by calling the same overlap helper used by production code. Both treat adjacency as overlap. Explain why the green result is weak and replace it with an independent example.

6. A loan test passes alone but fails after another test, and both use a static shared loan service. Diagnose the likely cause and propose a setup strategy that still allows intentional multi-step scenarios.

Transfer challenge

A room-booking service permits availability enquiry and check-in only during the booked interval. Design three acceptance examples, one clock-controlled test and one collaborator substitute. Explain which expectations transfer from equipment lending and which rules need stakeholder clarification, such as cleaning time or late check-in.

Midterm practice set (Chapters 2–7)

The midterm covers Weeks 1–6, which are Chapters 2–7. According to the module descriptor, it asks you to model a scenario with UML class diagrams, identify SOLID violations and smells in supplied code, refactor short snippets by introducing interfaces and dependency injection, and write unit tests with test doubles for a small component. The five tasks below practise those four things. They are separate from the running project. Do not add Chapter 8' s patterns, which the midterm does not cover even though they are taught that week. Whether Mockito may be used in the exam is the module leader' s decision, and it has not yet been confirmed for this book; follow the module leader' s instructions about what the exam permits. The book teaches Mockito as a working tool, separately from what an assessment allows, and hand-written doubles work in any setup.

Task 1: UML. Draw a class diagram for this scenario. Include multiplicities, one composition, one interface realisation and one dependency, and write down one rule your diagram cannot express.

> A library lends laptops. Each laptop has an asset tag and a model. A loan links one borrower to one laptop and records a start date and a due date. A borrower and a laptop can each appear in many loans over time. A loan can be renewed any number of times; a renewal records a new due date and has no meaning without its loan. When a loan is overdue, a notice is delivered through a channel. The library uses email today and expects other channels later.

Task 2: diagnosis. Read the class below. List every smell or SOLID violation you can find. For each, name a concrete consequence, such as something you could not do or test, instead of naming a principle alone.

```
// Supplied for practice. MailClient is a supplied class that opens a real
// network connection; assume you cannot call it from a test.
import java.time.LocalDate;

public class OverdueReminder {
    public boolean remind(String borrowerEmail, LocalDate dueDate, String itemName) {
        if (LocalDate.now().isAfter(dueDate)) {
            MailClient client = new MailClient("smtp.campus.example", 25);
            client.send(borrowerEmail, "Overdue: " + itemName, "Please return " + itemName + ".")
            System.out.println("Reminder sent to " + borrowerEmail);
            return true;
        }
        return false;
    }
}
```

Task 3: refactor. Rewrite the class so it can be tested without a mail server. Introduce an interface and inject what it needs. State which behaviour you preserved and which part of the class' s API changed.

Task 4: tests. Write JUnit tests for your refactored class. Use a hand-written test double and a fixed clock. Cover the due date itself, an earlier date and the day after, and state which role your double plays.

Task 5: trace. Trace a rejected reservation through the coordinator. Explain why it produces no notification, choose the test observations that would show this, and identify the role of an injected sender. A good answer connects design, behaviour and evidence rather than listing acronyms.

The companion gives the supplied `MailClient`, a model solution for each task and a way to run the tests. Attempt the tasks before reading it.

What to take away, and what comes next

Meaningful tests tell intended behaviour apart from plausible mistakes. The worked cycle makes one missing rule fail visibly, implements it and protects the structural change that follows. Controlled collaborators and clocks make important outcomes reproducible, and carefully chosen assertions keep the tests useful while you refactor.

The running project now includes reservation availability enquiry and a bounded checkout and return flow. Its limits on lifecycle and integration remain explicit. Chapter 8 looks at construction and compatibility patterns, building on these tests and not replacing design judgement with pattern names.

Sources and further reading

Sources checked 12 and 13 October 2026. Version pins are the configuration the chapter specifies, not a claim to use every tool' s newest release. Examples and policy choices are author-created teaching material.

1. JUnit Team, **JUnit 5 User Guide, version 5.13.4**. Test methods, assertions, parameterised tests and build integration.
2. Apache Maven, **Using JUnit Platform with Surefire**, and **Maven 3.9.11 release notes**.
3. Fowler, M., **Test Driven Development**. Short development cycles and refactoring.
4. Oracle, **Java SE 17** `Clock`. Controlled instants and zones.
5. Fowler, M. (2006), **Test Double**. Summary of Gerard Meszaros' s test-double vocabulary.
6. Mockito Team, **Mockito API documentation**, for the basic mocking, stubbing and verification concepts. That hosted page identifies itself as 2.2.7; the executable chapter pins and verifies **Mockito 5.20.0**, rather than assuming the URL labelled "current" gives the selected version.
7. Apache Maven, **Compiler Plugin**. The supplied configuration pins 3.14.1 and Java release 17.
8. Oracle, **Java 17 Record Classes**. Record declarations and compact constructors.
9. Martin, R. C. (2002), *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall. The module' s required text; the Week 6 reading is its testing chapter and "A Programming Episode". Publication details as listed in the module descriptor.
10. Beck, K. (2002), *Test-Driven Development: By Example*, Addison-Wesley. A recommended text; the Week 6 reading is Part I. Publication details as listed in the module descriptor.

Chapter 8: Creating and Connecting Objects

> *Creation and connection patterns: Factory Method, Abstract Factory, Builder, Adapter and Decorator, and when to decline them.*

OO Development and Agile Practices
Designing, Testing and Evolving Software with AI Assistance

The equipment office can now reserve, check out and return equipment, and it wants to take the next step: borrower categories should come from the university directory instead of being supplied by whatever code makes the call. There is a catch. The proposed directory answers in codes (`S`, `T`, `Y` and `N`), returns null when a person is missing and throws its own exception when a lookup fails.

One developer proposes to put all of this inside `ReservationCoordinator`. Another proposes a factory, a builder, an adapter and a decorator around every class. Both proposals need a better argument. The first spreads an outside system's conventions through the reservation rules. The second adds structure before anyone has said what actually varies.

This chapter asks the more useful question: **which responsibility needs a boundary, and what is the smallest design that makes that boundary clear?** For the directory the answer is one justified Adapter, and the running project gains exactly that. Along the way you will meet the creational and structural patterns through small separate examples, so that you can identify a concrete construction or compatibility problem and say which pattern, if any, answers it. You will tell Factory Method, a simple factory and Abstract Factory apart by their responsibilities, build a validated immutable request through a staged builder, translate provider records, absence and failures into a contract the application owns, and compose decorators while predicting what order and exception handling do. Most of all you will practise justifying either adopting or declining a pattern, using tests and the cost of change as evidence.

The chapter belongs to Week 7 and supports LO2. The midterm remains on Weeks 1 to 6, corresponding to Chapters 2 to 7, and these new patterns are not added to that scope; assessment rules and the module descriptor stay under the module leader's control. The Week 7 readings are the Factory and Adapter chapters of Martin [7] and the creational and structural patterns in Gamma et al. [1].

You will need Chapters 5 to 7 (contracts, interfaces, constructor injection, composition and meaningful tests), Java 17 and the Maven/JUnit configuration from Chapter 7. All provider records are synthetic; no real university directory, network service or personal data is used. A second Markdown file holds the complete source listings, tests, reconstruction commands, solutions and teaching notes.

Sections 8.1, 8.2, 8.5 and 8.7 are the core reading, with Factory Method and Adapter as the two suggested live demonstrations. Sections 8.3 (Abstract Factory), 8.4 (Builder) and 8.6 (Decorator) are working knowledge: they give limited but required depth through runnable guided examples. Read each, run its example and, for the decorator, predict both audit histories before you run the tests. Working knowledge is not an optional extra, but it asks for understanding rather than independent implementation, and those implementations stay separate from the running application. What you must do is steps 1 to 3, 5 and 9 of the practical activity in Section 8.10. Steps 1 to 3 and 5 run on the book's project and step 9 on your team's project. This is

midterm week and the midterm does not cover this chapter, so do those steps first. If you want more, there are steps 4, 6, 7 and 8 (the second-scenario comparison, the AI activity in Section 8.8 and the Builder and Decorator exercises), the questions and the transfer challenge in Section 8.11; the solutions to the Builder and Decorator exercises sit in the companion's Section 8.4 as a separate set outside the 63-invocation baseline.

8.1 When does creating an object become a design problem?

Start with the ordinary constructor

Creating a `LoanPeriod` with its two required dates is clear. The constructor validates a small invariant and produces an immediately useful object. Replacing this with a hierarchy of factories would not improve the rule. Likewise, Chapter 6's `ReservationApplication.create(sender)` is a useful assembly function: it connects known policies, a formatter and an explicitly supplied sender.

Construction becomes troublesome when the decision about **which object**, **which compatible group**, or **which configuration** is repeated across places that should not own it. A caller may need only an export operation yet repeatedly inspect an output-format flag. Another may choose a heading renderer independently from a body renderer and accidentally combine incompatible output formats. A third may pass six positional arguments whose meanings are difficult to review.

These are different problems. Giving all three solutions the name "factory" conceals which responsibility changed. A pattern name should shorten a precise explanation after the problem is understood; it should not replace that explanation.

Creating, connecting and adding behaviour are different jobs

Factory Method, Abstract Factory and Builder concern construction decisions. Adapter connects an existing interface to the one a client needs. Decorator attaches an optional responsibility while retaining the component interface. These categories organise discussion, but actual code must still be examined for intent and consequences [1].

Consider the directory request. The awkwardness is not that constructing a directory object requires many arguments. It is that its data and failure meanings differ from the application's. A builder would not answer what `T` means or whether an outage should be treated as a missing borrower. Those are translation decisions.

Similarly, an audit wrapper around a sender may add behaviour without choosing the sender's concrete class. Calling that wrapper a factory would draw attention away from call order and errors, which are the important design questions.

Name the pressure before reaching for a structure

A useful design note names the current simple solution, the change it handles poorly, the selected boundary and the cost of adding it. For this project: passing a trusted category worked for local examples; provider codes now need central translation; an application-owned directory interface isolates that translation; additional types and mapping tests are the cost.

This reasoning does not require predicting every future directory. There is already one incompatible interface. Conversely, inventing five hypothetical provider families is not evidence that an Abstract Factory is needed today.

Keep construction visible at the composition point. Hiding it in a global registry can make dependencies harder to discover even if individual classes contain fewer `new` expressions. The goal is understandable ownership, not a ban on constructors.

8.2 Factory Method: letting subclasses choose what to create

The problem, and the simple attempt first

The isolated export example has a small workflow: validate an input, obtain an exporter and execute it. A simple factory can choose a plain-text or HTML exporter from a flag. That is often enough when one assembly location owns the choice.

The separate Factory Method scenario assumes that subclasses must specialise product creation while keeping the workflow inherited from a common creator. The change pressure is therefore an extension point in that workflow, not merely the existence of two product classes.

The structure: a creation step that subclasses override

`ExportJob.run` owns the workflow and calls `createExporter`. `PlainJob` and `HtmlJob` override that creation method. Each returns an object implementing `Exporter`. The workflow uses the product interface without inspecting the concrete creator's type.

```
public abstract static class ExportJob {
    protected abstract Exporter createExporter();

    public final String run(String value) {
        Objects.requireNonNull(value, "value");
        return createExporter().export(value);
    }
}
```

The complete imports and surrounding class appear in the code file. The final `run` method keeps this example's workflow fixed. The protected hook is the deliberate variation point.

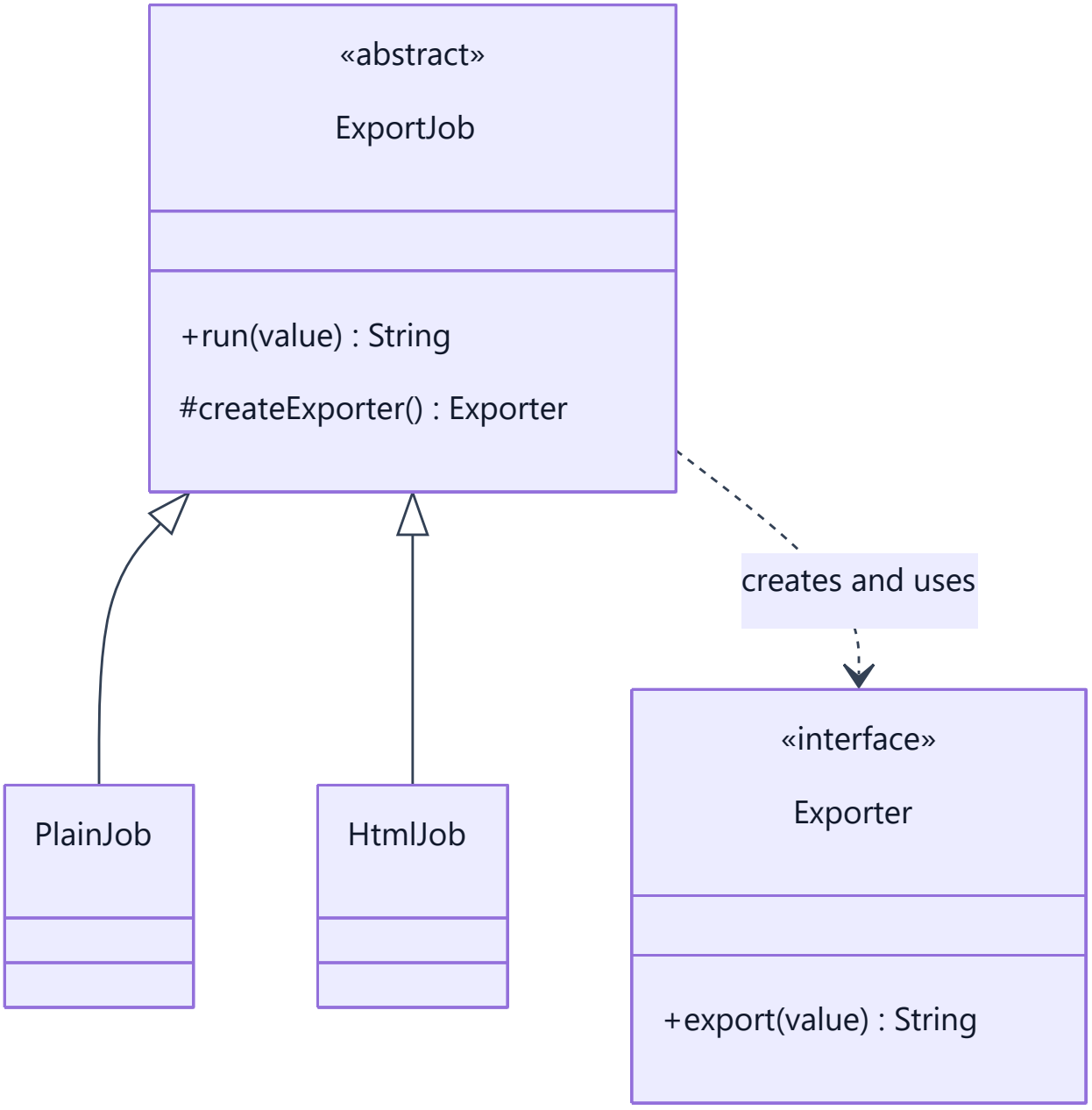


Figure 8.1. Factory Method. `ExportJob` owns the workflow and declares the creation hook, and each concrete job overrides it.

Read the notation with Chapter 2's table in mind. `#` marks a protected member: subclasses can override `createExporter`, but outside callers cannot call it. `<<abstract>>` marks a class that cannot be instantiated on its own. The solid line with a hollow triangle is inheritance. The dashed arrow is a dependency: `ExportJob` creates and uses exporters without keeping one in a field.

The diagram shows why the design is Factory Method: subclasses vary creation inside an inherited workflow. The concrete product implementations are lambdas in this compact example; they still implement the product interface. A separate named class is unnecessary for a one-operation product with no extra state.

How is it different from a simple factory?

`simpleExporter(boolean html)` contains an ordinary selection expression returning the same kinds of products. It is a simple factory, not the subclass-based Factory Method demonstrated above. A static method named `create` is not sufficient evidence of the latter pattern.

Tests can show that both designs produce `TEXT: R1` or an HTML paragraph. Identical output does not make their extension mechanisms identical. One central selection function changes when a new option is added; the other introduces a concrete creator subclass. Code structure and change requirements settle the classification.

Testing it, what it costs, and when not to use it

The tests exercise both concrete jobs and an instrumented creator whose hook returns a distinctive exporter. They check that the inherited workflow uses the chosen product and that null input is rejected before export. The hook-count assertion documents this tiny creator example; it is not advice to verify private call counts throughout the application.

Factory Method provides an explicit subclass extension point, but creates inheritance coupling. Readers must follow the inherited workflow and its override. A constructor accepting an `Exporter` can be simpler when callers already know the product and no framework-style creation hook is required.

For that reason, the running project keeps `ReservationApplication.create(sender)` as a simple assembly function. We do not convert it into a creator hierarchy solely to display the pattern. The separate example demonstrates the mechanism; the design decision explains when to decline it.

8.3 Abstract Factory: products that must belong together

The problem: parts that must match

A receipt has two product roles: a heading renderer and a body renderer. A plain-text receipt needs plain versions of both; an HTML fragment needs HTML versions of both. Independent format flags can accidentally select an HTML heading and a plain body.

A simple attempt is one renderer that switches format for every part. For two very small operations, that may remain the best implementation. The teaching scenario adds a requirement to choose the whole related set once while allowing the rendering workflow to use stable product interfaces.

The structure, and keeping the family consistent

`ReceiptFamily` exposes `heading()` and `body()`. `PlainFamily` supplies plain implementations of both roles; `HtmlFamily` supplies HTML implementations. The receipt function obtains both products from the supplied family and combines their outputs. It does not contain a format switch.

The essential distinction is the related set of product roles [2]. Abstract Factory is not just a factory declared with Java' s `abstract` keyword. An interface implemented by concrete family providers can serve the role. Factory Method instead focused on an overridable product-creation operation inside an inherited workflow.

Our types encourage family consistency by accepting one factory for a receipt. They do not mathematically prevent someone from writing a defective factory that returns a mixed pair. Tests and implementation review still establish that the concrete family fulfils its promise.

Testing the relationship, not just the construction

The tests render a complete receipt with each family. They expect `Loan` and `R1` as separate text lines in one case, and an HTML heading and paragraph in the other. An angle-bracket input checks the HTML text-node escaping used by these products. The example creates fragments, not a complete website or general sanitisation library.

A test that merely asserts both product references are non-null would miss a mixed-format family. The expected complete result observes the relationship that motivated the design.

What it costs, and when not to use it

Adding another family can leave the client workflow unchanged. Adding a new product role, such as a footer, changes the factory contract and requires every family to supply it. The design makes one axis of extension convenient by making another more coordinated.

Do not infer a family merely because two objects are constructed near each other. A clock and borrower directory do not automatically form a compatible product family. Nor does one sender with two implementations justify two product roles invented for the diagram. In the lending project, the small existing assembly function remains sufficient.

The isolated family example is intentionally compact. If the real receipt stays this small, replacing it with one formatter per output style is a reasonable conclusion. The learning outcome is to recognise the trade-off, not to preserve every example’ s abstraction in production.

8.4 Builder: configuring a value you can trust

The problem: configuration that is hard to read

Imagine a report request with a required borrower identifier, an optional date interval, a returned-loan option and an ordered collection of requested column labels. A positional constructor could work, but calls containing nulls, a boolean and a list become difficult to review as the options grow.

A sequence of setters on the final request is another simple attempt. It allows clients to observe an incomplete request and to change it after it has been submitted. If the required outcome is an immutable configuration value, construction and later use should be separated.

Structure: configure, validate, publish

`ReportRequest.forBorrower("B1")` creates a mutable builder. Named operations configure it, and `build()` produces an immutable request.

```
var request = ReportRequest.forBorrower("B1")
        .during(start, start.plusDays(7))
        .includeReturned(true)
        .column("item")
        .column("date")
        .build();
```

This is a fluent builder variant with one product representation. The classical Builder discussion also includes separating construction steps from representations and, when useful, a director controlling those steps [1]. Our example does not claim to implement that entire arrangement. Chaining methods alone would also be insufficient: what matters here is staged construction followed by a validated, independent product.

Making defaults and rules visible

The required identifier must be nonblank. Omitting dates means no date restriction; supplying dates requires a positive half-open interval. Returned loans are excluded by default. Column labels form an ordered list, empty by default; the example stores labels but does not implement report generation or a column-name registry.

Validation belongs before publication of the product. Each `build()` checks the interval, and the constructor copies the builder's column list with `List.copyOf`. Further builder changes therefore cannot change a previously built request. The request exposes the immutable snapshot.

The builder is mutable and intended for one construction sequence at a time. An immutable product does not make its builder thread-safe. Reusing the builder is supported sequentially in this example, but callers must recognise that settings remain until changed.

Testing it, what it costs, and when not to use it

Tests check defaults, explicit options, invalid dates and the independence of two products built before and after another column is added. Attempting to modify a product's columns must fail. Merely checking that chained calls compile would not detect an alias to mutable builder state.

The costs include another type, more methods and a distinction between temporarily configured state and a valid built value. A small constructor with two meaningful required parameters is often clearer. `LoanPeriod(start, endExclusive)` therefore remains a constructor call.

Builder also does not perform the report or authorise access to it. Combining configuration construction with database queries or directory lookups would mix responsibilities. Keep this example separate until the running application has an actual report requirement.

8.5 Adapter: translating an outside system into your own terms

The problem, and the tempting shortcut

The illustrative provider exposes `lookup(key)` and returns a `Person(key, kind, status)`. The application wants `find(borrowerId)` returning a domain `Borrower` when an eligible record exists. The methods differ, but the important mismatch includes data meaning and failure behaviour.

Putting provider codes directly in the coordinator would make reservation logic depend on their spelling. Repeating translation in callers would risk inconsistent defaults. An object Adapter instead implements the application interface while holding the incompatible provider collaborator [3].

The provider interface in this chapter is a local model of an external contract, not an implemented HTTP client. Its map-backed stand-in makes all cases reproducible. Replacing that stand-in with a real client will require provider-specific integration tests later.

Agreeing the translation first

Provider outcome	Application outcome
Same identifier, kind `S`, status `Y`	Present student borrower
Same identifier, kind `T`, status `Y`	Present staff borrower
Null row	Empty result: borrower absent
Recognised kind, same identifier, status `N`	Empty result: borrower inactive
Unknown/null kind or status; mismatched identifier	`DirectoryException`: invalid provider data
Provider's declared `Failure`	`DirectoryException` retaining the cause
Unexpected programming exception	Propagates; never converted into absence

Codes are exact and case-sensitive. `T` means staff only because this teaching provider contract says so. The adapter does not trim identifiers, normalise codes or infer a default category. Those changes would require an agreed mapping policy.

The adapter validates the returned identifier and category before interpreting inactivity. Thus an inactive row with an unknown kind is still malformed. Ordering is part of the selected contract and is covered by a test; another policy would need an explicit decision.

The structure, and the new entry point

`BorrowerDirectory` belongs to the application's vocabulary. `LegacyDirectoryAdapter` implements it and delegates the lookup to `LegacyDirectoryClient`. A `Borrower` is an immutable identifier/category value. An `Optional<Borrower>` expresses a legitimate empty lookup, while an exception expresses the inability to obtain a trustworthy result. Java's `Optional` represents presence or absence, and its documentation says a variable of type `Optional` should never itself be null; it does not define which failures should become absence [4].

`DirectoryReservationService` is the new application entry point. It validates the local identifier/item/date arguments, looks up the borrower, and calls the existing coordinator only after obtaining a valid domain borrower. The caller supplies no category to this entry point. The coordinator continues to own borrowing limits, interval conflicts, storage and notification.

Figure 8.2 is a **sequence diagram**. Where a class diagram shows what exists, a sequence diagram shows what happens, in time order from top to bottom. Each box at the top is a participant, and the vertical line below it is its lifeline. A solid arrow is a call and a dashed arrow is a return. A frame labelled `alt`, split by `else`, shows alternative paths, of which exactly one happens in a given run. A note spans the participants it concerns. Read it as a story: the service asks the adapter, the adapter asks the provider, and what follows depends on what the provider returns.

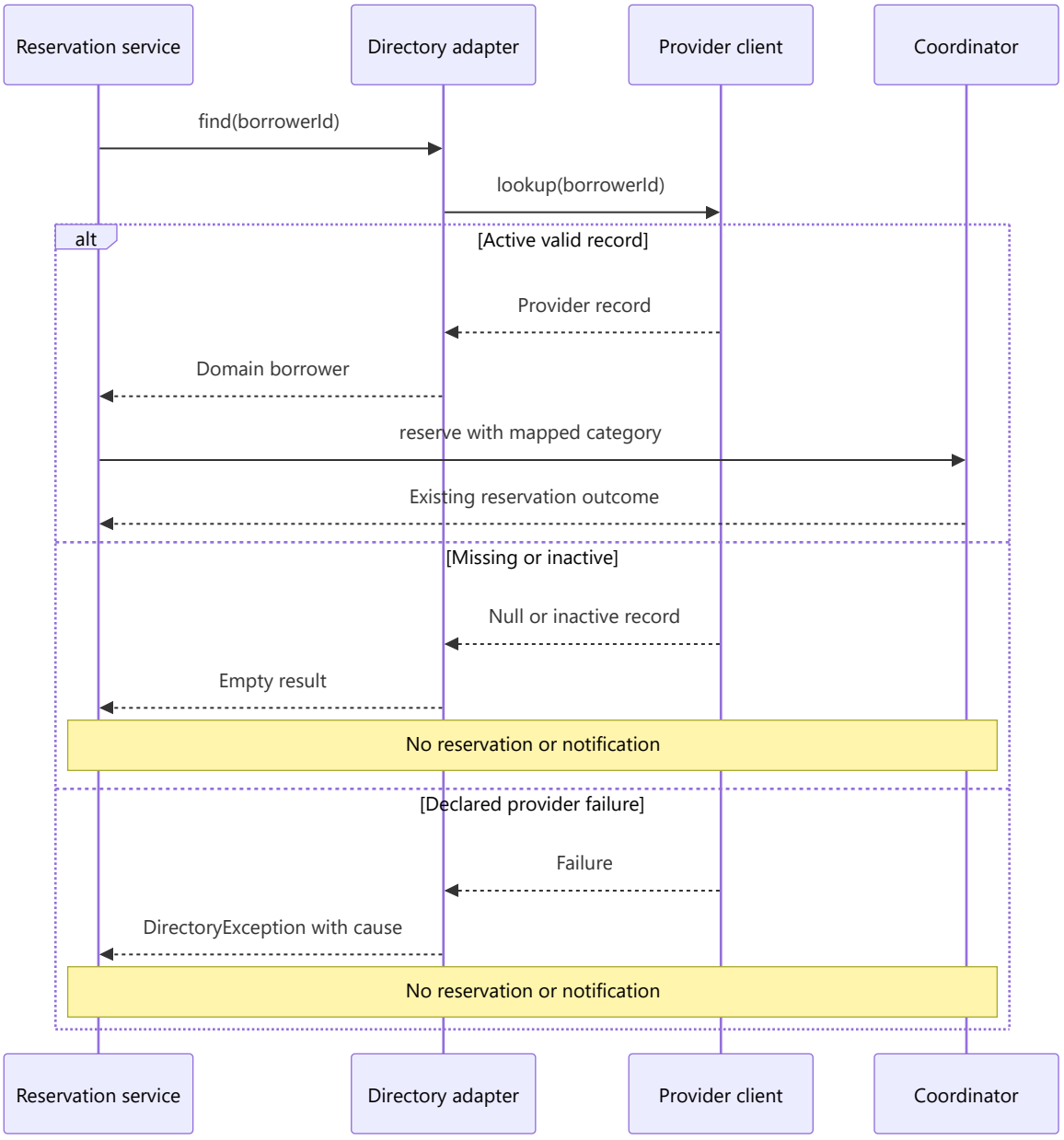


Figure 8.2. The directory lookup before a reservation. The sequence highlights the point before which reservation state must remain untouched. Malformed provider data follows the same no-mutation boundary as a provider failure, although its exception describes a different cause, and the diagram omits it to stay readable.

A data-backed collaborator, and the same questions for a model

The directory is the book's data-backed collaborator. The module is about software built around AI and data components, and this book takes the data-component route: the equipment directory is the component that stands outside the application and answers from its own data. Chapter 1 promised it, Chapter 6 previewed its contract, Chapter 7 showed three doubles for it, this chapter builds the adapter, Chapter 10 places it in the application structure, and Chapter 13 adds what a real provider needs: timeouts and an explicit fallback-or-failure policy. Here the provider is a synthetic, in-process stand-in, so there is no transport, timeout or storage to fail.

The directory is the book's first collaborator whose answers come from outside the application. It can be absent, wrong or unavailable, and its data uses its own vocabulary. Any component that returns data or predictions is in the same position, including one that calls a trained model. The questions are the same: what does an empty answer mean, what counts as a failure, which outputs will you refuse to trust, and how will a test substitute the component? If your team's domain is an ML model-serving tool, your adapter boundary sits between your service and the model runtime. Map its label codes to your own types, treat a missing prediction and a timeout as different outcomes, and reject values outside the documented range.

Keeping a rejection and a failure apart

A missing or inactive borrower produces the explicit rejection string. A directory outage throws `DirectoryException`. Returning "borrower not found" for both would make an operational failure look like a valid eligibility decision. Defaulting to staff would be worse: it would permit a longer loan without a trusted category.

The service still delegates duration policy to the coordinator. It therefore performs directory lookup before deciding whether ten requested days are allowed. Local blank identifiers and a null start date fail before lookup. This order is deliberate and documented rather than an accidental consequence of catch blocks.

The new entry point is not authentication. Knowing an identifier does not establish who the caller is or whether they may reserve for that person. The earlier direct coordinator APIs remain available for previous examples and trusted internal use; callers can bypass the directory if they use them. A real exposed application would need a controlled authenticated entry path and reviewed authorisation rules.

Testing the boundary, and what it costs

The tests use synthetic provider records to check both category mappings, absence, inactivity, unknown codes, null fields, identity mismatch and expected failure translation. A separate service test shows that a staff record permits ten days while a student record rejects ten days using the same existing policies.

Failure tests inspect state and notices, then perform a valid request to show that failure did not consume R1. Unexpected exceptions are not broadly caught and hidden. These observations test semantics, not whether a method happened to have "adapter" in its name.

The cost is a mapping layer and a contract that must evolve with provider changes. If two components already share the same interface and meaning, direct injection is sufficient. If they disagree on essential capabilities, an adapter cannot magically supply them. This implementation does not cache, retry, authenticate, persist or contact a real provider.

8.6 Decorator: adding behaviour around an operation

The problem: optional behaviour

Suppose an isolated notification exercise needs optional audit observations and a single retry after a declared delivery failure. Subclasses for every combination would multiply quickly. Adding flags to one large sender is simpler at first, but combines responsibilities and hides order.

A Decorator implements the same component interface as its wrapped delegate and adds behaviour around delegation. Java's `FilterInputStream` illustrates the underlying wrapping mechanism: it is an input stream containing another input stream, with subclasses able to add behaviour [5].

The structure, and why order matters

The exercise defines a separate `Sender.send(String)` interface. `AuditedSender` records an attempt, delegates and records either normal completion or a declared failure. `RetryOnceSender` retries once only for `DeliveryFailure`. Both accept another `Sender`.

For a delegate that fails once and then succeeds, composition order changes the observations:

Composition	Recorded events	Delegate calls
Audit outside retry	`attempt`, `sent`	Two
Retry outside audit	`attempt`, `failed`, `attempt`, `sent`	Two

In the first composition, audit observes one logical call whose internal retry succeeds. In the second, each attempt passes through audit separately. Neither event sequence is universally correct. Decide whether the audit requirement concerns operations or individual attempts.

Failure is part of the contract

If both attempts fail, the second failure propagates and no third attempt occurs. An unexpected `IllegalArgumentException` is not retried. Catching every exception would confuse invalid input or programming defects with the declared retryable condition.

The audit list is an in-memory record kept for observation. It is not durable audit storage. The example assumes list recording succeeds; if a real audit sink can fail, the design must decide whether to propagate that failure, preserve the original delivery failure, or report it separately. A same-signature wrapper is not automatically behaviourally substitutable under every error policy.

A real sender may deliver a message and then lose the acknowledgement. Retrying can duplicate the effect. This toy example has no idempotency key or delivery guarantee, and "sent" means only that the delegate returned normally.

Decorator: test, trade-off and non-use case

Tests assert both event sequences, the two-attempt bound and propagation of expected and unexpected failures. These are meaningful interaction observations because order and attempt count are the feature under discussion.

The decorators remain isolated. Chapter 6's notification contract specifies one attempt after acceptance, no automatic retry and retention of the reservation after the expected notification failure. Installing this retry example there would change that contract. A pattern name does not authorise the change.

Use one straightforward sender when optional combinations are unnecessary. Decorators add another path to trace and can obscure which layer handles an error. They also require clear ownership of closeable resources when the wrapped components own files or connections; this string-based exercise has none.

8.7 When the simpler option is the better one

Compare by the decision each one isolates

Observed pressure	Candidate	Simpler alternative to consider
Subclasses must create a workflow’ s product	Factory Method	Inject a ready product into the workflow
Related product roles must vary together	Abstract Factory	One formatter or one assembly function
Many named options configure an immutable value	Builder	A clear constructor or named creation method
Provider interface and meanings differ	Adapter	Direct injection if the contracts already match
Optional responsibilities compose around one interface	Decorator	One explicit implementation with fixed behaviour

The table is a decision aid, not a rule that each pressure has only one solution. Explain who owns the variation, how failures travel and how the selected structure changes future edits. Prefer the design that makes the current requirement understandable and testable.

What the running project gained

Chapter 8 adds the directory-backed reservation service, domain borrower and provider adapter. The Chapter 7 coordinator, policies, availability query, loan service and notification implementations remain unchanged. The original test suite remains in the build.

A staff borrower in the examples can now request ten days through a category-free entry point; a student cannot. Missing or inactive borrowers are rejected before storage. Provider failures remain identifiable and leave reservation state untouched. These are the new claims the tests support.

The inherited lifecycle limits also remain: returned loans do not remove reservations, free their intervals or restore counted reservation allowances. The project still runs in memory within one process. Pattern work does not resolve these separate requirements.

Record a decision another developer can challenge

A short problem–solution–consequence note is enough: “Provider codes and null/error conventions differ from the application. An object adapter translates them behind `BorrowerDirectory`. This keeps provider details out of reservation policy, at the cost of mapping tests and explicit provider-contract maintenance.”

Then name what was declined: the existing assembly function needs no creator hierarchy; no related provider product family is required; the core value objects do not need builders; retry would change the notification promise. Declining those structures is part of the design outcome.

Tests establish behaviour for the selected examples. They cannot alone prove that a pattern is economical or necessary. Review the dependency direction and the likely edit for one realistic change. If the explanation is only “this chapter teaches the pattern,” keep it in the teaching examples rather than the running system.

8.8 Classifying an AI’s pattern suggestions before accepting them

Chapter 15 gathers the AI workflows that recur in this book. This activity is complete on its own, and the fictional answer below is all you need.

	For this activity
Your goal	Classify each suggested structure by what it actually does, and decide which are justified
You start with	The isolated pattern examples, the directory contract in Section 8.5 and the Chapter 6 notification contract
What is in bounds	Classify, correct and decline suggestions; do not wire any pattern into the application unless a requirement supports it
How to check	The source (is there a subclass hook?), the mapping table, and the tests for unknown codes, outage, immutability and decorator order
How to decide	A verdict with a structural or behavioural reason for each claim
No AI account?	The ready-made answer below is enough

The text below is a **fictional AI answer written by the author** for offline use:

> Rename the static `create` function to `FactoryMethod` and it becomes Factory Method. Add an Abstract Factory even though there is only one product role. The adapter should catch every exception and return a staff borrower to keep the system available. Wrap all senders in retry decorators; order does not matter. A builder means the finished object is immutable even if it retains the builder’s mutable list.

Identify the unsupported claims. For each, specify the missing requirement or structural evidence and a concrete test where applicable. Compare a corrected simple factory with the genuine subclass hook example. Keep one justified boundary in the application and explain which suggested structures you decline.

If you use a live AI service, give it only synthetic contracts and examples. Keep the proposed answer, your classifications and the outcomes you checked independently. Do not accept generated tests that assume the same unsafe fallback as the generated implementation. A paid service is unnecessary; the ready-made answer supports the full activity.

8.9 Pattern habits that mislead

Mistake	Consequence	Better question
Classify by names such as `create` or `Factory`	Misidentifies the extension mechanism	Which operation varies, and how?
Put provider codes inside borrowing policies	Couples business rules to provider representation	Where should translation live?
Treat absence and outage identically	Hides failure as an eligibility decision	Was a valid lookup completed?
Assume a family interface prevents all mixing	Overstates guarantees	Do concrete factories return compatible products?
Keep the builder's mutable list	Previously built values can change	Does construction copy mutable state?
Assume decorator order is irrelevant	Changes audit meaning and failure visibility	Which calls pass through each layer?
Install retries without revisiting the contract	Introduces duplicate attempts or effects	What outcome does the existing promise allow?
Require all patterns in the semester project	Adds complexity without a problem	Which real change justifies this structure?

8.10 Try it yourself: the directory boundary and pattern exercises

You start from Chapter 7's finished code, the synthetic provider contract and mapping table, and the complete Chapter 8 code file. Steps 1 to 8 run on the book's project; step 9 runs on your team's project. Steps 4 and 6 to 8 are the extras described in the opening route.

If your instructor demonstrates the task, they will first compare the simple factory and the overridden creation hook, and then trace an active borrower and a failed lookup through the adapter boundary. For guided practice, add a malformed provider case that you chose independently and explain why it produces a failure and not an absence. Then run the decorator tests and predict both audit histories before you look at the results.

Then work through these steps:

1. Integrate the borrower-directory boundary using the selected mapping contract.
2. Show staff and student duration outcomes without accepting a caller-supplied category at the new entry point.
3. Prove that missing, malformed and failed lookups do not create reservations or notices.
4. Compare two designs for a second scenario: injected exporter versus Factory Method, or constructor versus Builder.
5. Write a problem–solution–consequence decision, including a justified non-use case.
6. Inspect the fictional AI answer and retain independent verification of your corrections.

7. **Builder exercise.** In the isolated `patterns` package, add an optional maximum-row setting to the report builder. It must be absent by default, must reject zero and negative values, and must not change a request that was already built. Write the tests first.
8. **Decorator exercise.** Add a third decorator that counts the attempts passed through it. Predict the count for one failure followed by success when it sits outside the retry decorator and when it sits inside, then check your prediction with tests.
9. On your team's project, find one place where your code uses an external or mismatched interface directly, such as a service, a library or a data source with its own codes. Decide whether an adapter is justified, using the problem–solution–consequence form, and say what you would decline to build.

What you should end up with is: runnable source and tests; a mapping table; a reproducible demonstration; a short design decision; a record of the AI claims you accepted or rejected, with reasons; the two exercise solutions with their tests; and the step 9 note for your team's project.

You are done when these things hold:

- known codes map correctly;
- unknown codes never grant a category;
- a provider outage is distinct from absence;
- the earlier behaviour is still tested;
- the comparison of two designs for the second scenario identifies costs rather than counting classes.

If you draw a diagram, it must match the boundary you implemented.

If you want more, propose a changed provider code for staff, or a second provider contract. Identify the adapter changes and contract tests it would need, and do not add automatic fallback between providers without defining conflict, outage and identity semantics.

8.11 Questions to think with, and a transfer challenge

Concept questions

1. What distinguishes the worked Factory Method from `simpleExporter`?
2. Why are two unrelated construction methods insufficient justification for Abstract Factory?
3. Which property makes the built report request independent of subsequent builder changes?
4. How do Adapter and Decorator differ even though both may contain another object?

Applied diagnosis

5. A directory adapter catches `RuntimeException` and returns an empty result. Describe at least two different failures this hides and the resulting problem for the application.

6. A team changes `Audit(Retry(sender))` to `Retry(Audit(sender))` because both satisfy `Sender`. Predict the audit histories for one failure followed by success and explain the requirement that selects between them.

Transfer challenge

A library application receives patron records from an external system using numeric category codes and a separate inactive flag. Design the application-owned lookup contract, an adapter mapping and three failure cases. Then decide whether constructing its receipt output requires a simple factory or a family factory. Explain what additional requirement would change your answer.

What to take away, and what comes next

Creation and connection patterns earn their place when they isolate a specific decision. Factory Method varies product creation through a subclass hook. Abstract Factory groups related product roles. Builder stages construction. Adapter translates incompatible contracts, and Decorator composes optional responsibilities whose order still matters.

The running project now has a tested directory-backed entry point. Chapter 9 moves on to changing behaviour, looking at further patterns through concrete pressures to vary and the same demand for evidence.

Sources and further reading

Checked 12 and 13 October 2026. The project policies, provider contract, code, test data and teaching scenarios are original examples. The design patterns used here are treated in this chapter and, for the behavioural patterns, in Chapter 9.

1. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. **Publisher catalogue**. Read the five named pattern entries for their full discussions and variations.
2. Gamma et al., **Abstract Factory, publisher excerpt**. Related product families and the intent of the pattern.
3. Gamma et al., **Adapter, publisher excerpt**. Interface conversion and object composition.
4. Oracle, **Java SE 17 Optional**. Presence and absence in the return type.
5. Oracle, **Java SE 17 FilterInputStream**. A stream that delegates to a contained stream; useful for examining wrapper behaviour.
6. JUnit Team, **JUnit 5.13.4 User Guide**. Assertions and parameterised tests used in the code file.
7. Martin, R. C. (2002), *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall. The module' s required text; the Week 7 reading is its Factory and Adapter chapters among the pattern chapters. Publication details as listed in the module descriptor.

Chapter 9: Designing Changing Behaviour

> *Behavioural patterns: Strategy, State, Observer, and Command.*

OO Development and Agile Practices
Designing, Testing and Evolving Software with AI Assistance

Three requests land on the equipment team in the same week. Borrowing limits should vary with the borrower's category. A reservation should be cancellable before checkout but not once the item is on loan. And a successful reservation should tell several local observers about it, even if one expected notification attempt fails.

All three are about changing behaviour, but they change for different reasons. Category selects a rule. Lifecycle history decides which action is legal. Subscribers decide who reacts when a reservation is accepted. One big conditional method could tangle them together, and a single vague idea of a "behaviour pattern" would hide how different they are.

Then comes a fourth wish: the team would like to represent a reservation request as an object that a caller can run later. That raises a question of its own, namely when the request is captured and when its side effects happen. This chapter pulls the four apart. You will learn to tell an interchangeable policy from lifecycle-dependent behaviour; to explain and test Strategy using the borrowing policies you already have; to define a transition table and implement a small State-based lifecycle; to specify what Observer promises about order, subscription changes and failure; and to represent a request with Command and explain why its side effects belong to execution time. Each time you will compare the pattern with a simpler alternative, and justify adopting or declining it with a concrete change and some evidence.

This is Week 8, supporting LO2 and LO4, with Chapters 7 and 8 as the entry. The module descriptor is not changed and no assessment rules are invented here. The Week 8 readings are the Strategy, Command, Observer and State chapters of Martin [6] and the behavioural patterns in Gamma et al. [1]. The same week carries the individual critical book review: follow the module leader's approved instructions and reading list, see **Section 9.12** and **Appendix E**, and remember that evidence must come from the selected book and verified sources, not from an AI summary of an unread text.

You start from Chapter 8's Java 17 application, its directory adapter and its complete Maven test suite. The code-and-solutions file supplies a complete new version of the code. The original borrowing policies already demonstrate Strategy; a notification publisher is an optional collaborator in the running project; and the proposed lifecycle is a separate exercise that does not replace the project's reservation or loan rules.

Strategy and State are the core reading (Sections 9.1 to 9.3, 9.6 and 9.7: Strategy, State, testing and selection). Observer and Command (Sections 9.4 and 9.5) are working-knowledge topics: understand the structure, the ordering and failure rules and the cost, and see each pattern work in the supplied examples, but you do not implement either on your own. What you must do is steps 1 to 4 and 6 of the practical activity in Section 9.10: steps 1 and 2 cover State, step 3 covers Observer and Command, and step 4 covers Strategy and the choice among patterns. Steps 1 to 4 run on the book's project and step 6 on your team's project. If you want more, try the AI activity in Section 9.8 (step 5), the questions and the transfer challenge in Section 9.11.

9.1 What changes, and why? Variation in behaviour

Ask what changes and what stays responsible

The existing `BorrowingPolicy` interface answers whether a count and duration are permitted. The coordinator selects a policy by category and uses its answer alongside identifier validation and conflict checking. Changing the policy does not change what a reservation is or whether a camera is already reserved.

Lifecycle behaviour asks a different question. The same cancellation request may be legal for a confirmed reservation and illegal after checkout. A completed transition changes which later actions are possible. Merely selecting another duration algorithm cannot capture that history unless transition rules are added somewhere else.

Observer concerns interested recipients. A desk display and a local audit observer can respond to the same confirmed reservation without becoming additional borrowing rules. Command concerns a request as a value-bearing action object: it captures the receiver and arguments, then invokes the receiver when executed.

These distinctions correspond to different design pressures. The classic pattern vocabulary gives them names, but the requirements determine which structure is useful [1]. Before drawing classes, write one example where the behaviour varies and one outcome that must remain unchanged.

Four questions for one domain

Question	Primary variation	Candidate
Which count/duration rule applies?	Chosen algorithm or policy	Strategy
Which action is legal now?	Current lifecycle position and transitions	State
Who reacts to this occurrence?	Registered interested recipients	Observer
What action should this request object execute?	Captured receiver and arguments	Command

The same application can use all four, but it need not. A two-branch conditional may remain clearer than an abstraction if the decision is small and stable. A direct method call is often sufficient if the request never needs to be stored, passed or handled independently.

Avoid the claim that all conditionals violate good design. The question is whether decisions are duplicated, mixed with unrelated responsibilities or difficult to extend safely. Pattern-based code still contains conditions; it relocates particular decisions so their ownership is clearer.

Keeping structural and behavioural changes apart

The State exercise compares two implementations of the same proposed transition table. That is a structural comparison within an isolated example. Integrating those transitions into the current lending application would instead introduce new lifecycle behaviour and require new decisions about counts, availability and existing loans.

The optional publisher adds several observers behind the existing notification boundary. Its ordering and failure policy are new, explicit behaviour. The coordinator still makes one call to its selected sender after storage, and the publisher makes no automatic retries. Treat each claim at the appropriate boundary rather than saying that every change is “only refactoring.”

9.2 Strategy: swapping one rule for another

You already have a Strategy

Strategy supplies interchangeable implementations of a behaviour through a shared contract. The existing student and staff policies already serve that role. `ReservationRules` permits a student count below three and a duration from one through seven days. `StaffReservationRules` uses the illustrative limits below five and up to fourteen days.

The coordinator depends on `BorrowingPolicy`, not on the calculations inside those classes. Its category-to-policy map is supplied through construction. Chapter 8's directory supplies the category at the application entry point; it does not choose arbitrary policy code or rewrite duration limits.

A strategy need not be replaced while an operation is running. The supplied policy map is fixed for a coordinator instance, while each request selects an entry by category. Configuration-time selection and per-request lookup are enough to demonstrate the role.

Comparing with the plain conditional

The teaching test includes a small conditional implementation: if student, evaluate the student limits; otherwise, for the supported enum values, evaluate the staff limits. This can be understandable for two rules in one place. The pressure for Strategy appears when algorithms acquire distinct reasons to change or repeated callers need consistent selection.

The comparison uses independently written expected results at the boundaries. It checks both the conditional and policy objects against those results. Testing only that the two implementations agree would be weaker: both could contain the same mistaken maximum.

The conditional is kept only as a teaching example. No duplicate policy engine is added to the running application. The complete student/staff policy tests from earlier chapters continue to run.

The contract every strategy shares

Every policy accepts the same two integers and returns a deterministic boolean without changing application state. Negative counts and nonpositive durations return false. A true answer means the count/duration rule permits a request; it says nothing about identity, physical stock or interval conflicts.

That last distinction matters. If a new strategy performs directory lookup, sends a message or stores a reservation, it no longer behaves like the current policy contract. Matching the Java method signature would not justify substituting it.

The new test supplies a temporary staff strategy permitting at most one counted reservation and at most two days. The coordinator itself is unchanged. Three days are rejected; two days are accepted; another request for that borrower is rejected; student behaviour still follows its supplied strategy. This is an isolated configuration test, not a change to the book's default staff policy.

What it costs, and simpler alternatives

Strategy separates variation from the surrounding use case and makes the chosen behaviour easy to supply in a test. It adds a selection responsibility: someone must choose a compatible policy, and the constructor must reject an incomplete map. Those responsibilities do not disappear because the interface is small.

If the only difference is two numeric limits, one parameterised policy class may be simpler than two implementations. If rules later differ substantially, separate strategies can make their algorithms clearer. Neither form should pretend to be a reservation lifecycle.

Do not add a `setPolicy` method just to make the pattern look more dynamic. Mutable policy selection can create questions about requests in progress and reproducibility. Use the narrowest variation mechanism the requirement needs.

9.3 State: behaviour that depends on where you are in a lifecycle

Start with the rules

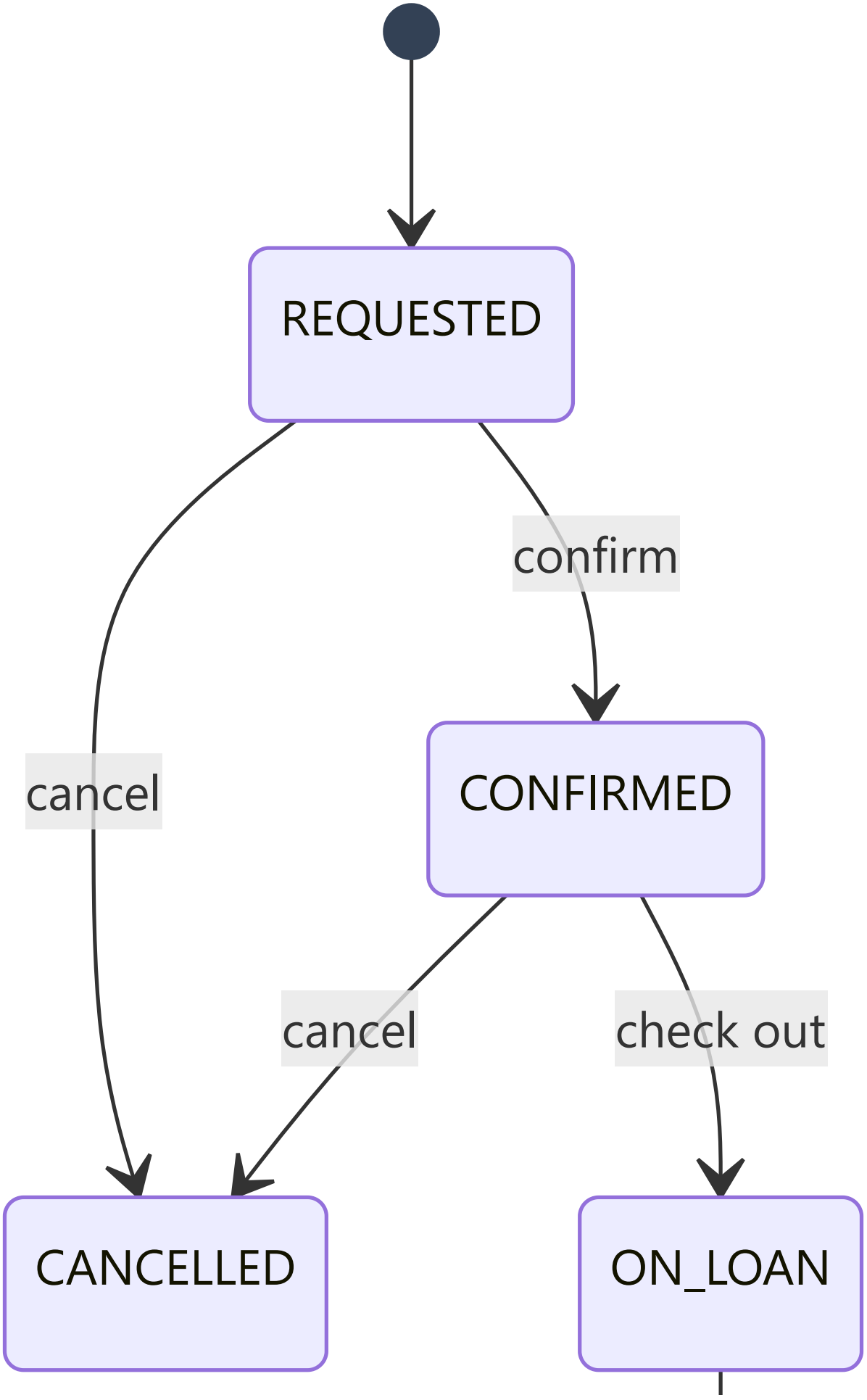
The separate lifecycle exercise begins at REQUESTED. It may be confirmed or cancelled. A confirmed reservation may be checked out or cancelled. An on-loan reservation may be returned. RETURNED and CANCELLED are terminal for the four supported actions.

These are rules invented for an isolated teaching model. The running system currently confirms accepted reservations immediately and tracks loans separately. It has no implemented cancellation operation. The proposed lifecycle therefore cannot be silently attached to existing records as if it were already agreed behaviour.

Current status	Confirm	Check out	Return	Cancel
REQUESTED	CONFIRMED	Reject	Reject	CANCELLED
CONFIRMED	Reject	ON_LOAN	Reject	CANCELLED
ON_LOAN	Reject	Reject	RETURNED	Reject
RETURNED	Reject	Reject	Reject	Reject
CANCELLED	Reject	Reject	Reject	Reject

Every rejection throws `IllegalStateException` and leaves the current status unchanged. Repeating cancellation is a rejection, not an idempotent success. That choice belongs in the table before it appears in Java.

Figure 9.1 draws the same rules as a **state diagram**. A state diagram shows the positions an object can be in and the events that move it between them, and nothing about how it is implemented. Each rounded box is a state. An arrow is a transition, labelled with the event that triggers it. The filled circle marked `[*]` is the start point, here REQUESTED. A state with no outgoing arrow, such as RETURNED, allows none of the four actions. A transition can also carry a guard, a condition in square brackets that must hold before it fires; this model has none.


REQUESTED
confirm
cancel
CONFIRMED
cancel
check out
CANCELLED
ON_LOAN

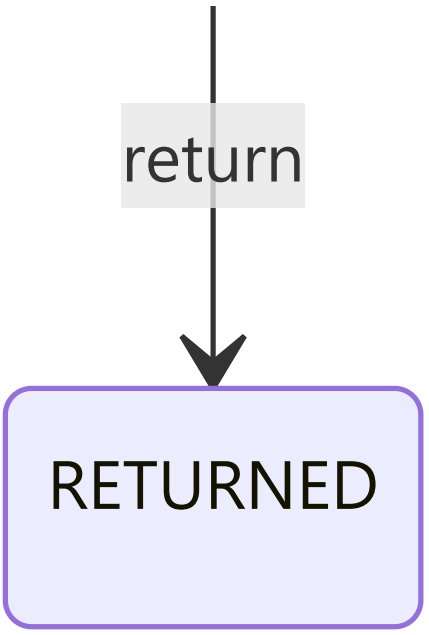


Figure 9.1. The proposed reservation lifecycle. The diagram shows permitted paths only. An action with no arrow from a state is not allowed there, which is why the table remains the complete specification: it also states the absent edges, and those are essential for testing. Terminal means no supported outgoing action; it does not mean deleting a persisted record.

A conditional version, and where it strains

`ConditionalReservation` stores a status enum and switches on it inside four operations. The repeated selection is easy to inspect at this size. As operations and state-specific rules grow, however, understanding one state's complete behaviour requires visiting several methods.

`StateReservation` instead holds a current state object and delegates each operation to it. Each state provides its allowed successors; default implementations reject unsupported operations. The context assigns the returned successor only after the state operation completes successfully.

Figure 9.2 compares the two designs as classes. Both implement the same `Lifecycle` interface, so a caller cannot tell them apart.

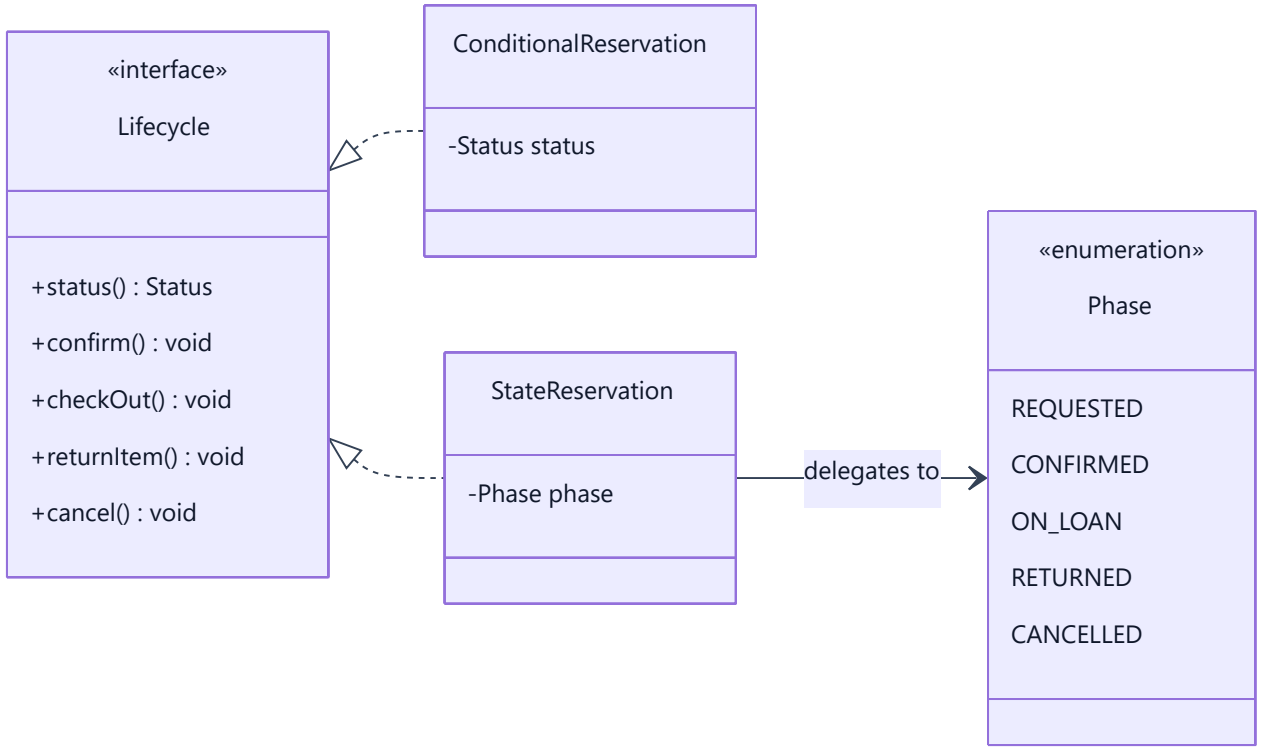


Figure 9.2. Two implementations of one lifecycle contract. `ConditionalReservation` keeps a status value and switches on it inside each operation. `StateReservation` holds a `Phase` and delegates each operation to it, so the enum constants are the state objects. In the code `Phase` is private, so callers never see it; the diagram shows it because it explains the design. Compare this with the earlier figures in this book: the realisation arrows say both classes keep the same promise, and the association says where the State design keeps its current position.

```
public void cancel() {
    phase = phase.cancel();
}
```

If `phase.cancel()` throws, assignment never occurs. This is the local reason the current status remains unchanged after rejection. It is not a general transaction mechanism for arbitrary side effects inside state operations.

State objects without five public classes

The implementation uses a private enum whose constants provide state-specific method bodies. These constants act as the State objects. They share the operation contract, and the context dispatches to the current constant. An enum used only as a tag in a central switch would be the conditional design instead.

The state objects contain no reservation-specific mutable data. Two contexts can therefore refer to the same REQUESTED constant without sharing their progress. Each context owns its own current-state reference. A test advances one context while leaving another unchanged.

No caller can assign a new phase directly. A public setter would let a caller skip confirmation or resurrect a returned reservation without passing through the transition rules. Read-only status is sufficient for observation.

Testing it, what it costs, and where it stops

Twenty parameter rows cover five statuses and four actions. Each row runs against both implementations and checks an independently specified outcome. There are five permitted transitions and fifteen rejected combinations. The tests also check that rejected actions preserve state.

This is comprehensive for the small status/action table, not for a real reservation process. Time windows, permissions, conflicts, cancellation effects and loan storage are outside the exercise. Chapter 7′ s clock-controlled checkout rules remain in the `LoanService` of the running project.

State groups behaviour by lifecycle position but introduces indirection. With this small fixed table, keeping the conditional implementation is defensible. If integration is proposed later, first decide whether completed or cancelled reservations stop counting, whether intervals become available, and how existing loan records stay consistent. Those are requirements, not consequences automatically supplied by State.

9.4 Observer: telling interested parties without knowing who they are

The problem, and the simple attempt

After a reservation is accepted, several local components may need to react. Hard-coding each observer inside the coordinator would couple reservation decisions to the changing recipient list. A direct call remains simpler when there is only one fixed collaborator; the existing sender interface already supports that case.

The optional `NotificationPublisher` implements `NotificationSender`. Observers register a callback receiving the immutable confirmed reservation. The coordinator can therefore use the publisher through its existing boundary. Registration and dispatch are the publisher′ s responsibility, not new methods on the coordinator.

Java provides listener-management examples such as `PropertyChangeSupport` [2]. (The older `java.util.Observable` class is deprecated and should not be used as a model.) This chapter implements a deliberately smaller publisher with its own documented contract. It does not inherit another library′ s ordering, concurrency or failure guarantees merely because both use listeners.

Deciding what observers can count on, before dispatching

The selected publisher is synchronous, sequential and in memory. It calls observers in registration order using a snapshot taken at the start of publication. Each observer in that snapshot is attempted once unless an unexpected exception stops dispatch. There are no retries.

Situation	Selected outcome
No observers	Normal return; no delivery claim
Same observer object registered twice	Reject duplicate registration
Subscription closed	Remove that registration; repeated close is harmless
Subscribe or unsubscribe during a callback	Change subsequent publications, not the current snapshot
Expected `NotificationException`	Retain failure and continue to later observers
Unexpected exception	Propagate immediately; later observers are not attempted
Recursive publication through the same publisher	Reject with `IllegalStateException`

Observer identity uses the same object reference, not custom equality. Separate callback objects can still perform equivalent work. The publisher does not deduplicate recipients or effects by business identity.

A subscription handle owns one registration. Closing an old handle again must not remove a later registration of the same observer. The implementation retains a distinct entry object per registration to make that ownership precise.

Publishing happens after storage

The coordinator stores the accepted reservation before calling its sender. If a publisher observer fails with the expected notification exception, the publisher continues, collects failures and finally throws one aggregate `NotificationException`. The coordinator then returns its existing suffix: the notification failed and the reservation was retained.

Figure 9.3 uses the sequence-diagram notation introduced in Chapter 8: participants across the top, solid arrows for calls, dashed arrows for returns. Here a dashed arrow can also carry an exception instead of a normal result, and the notes mark what is already true or what happens next.

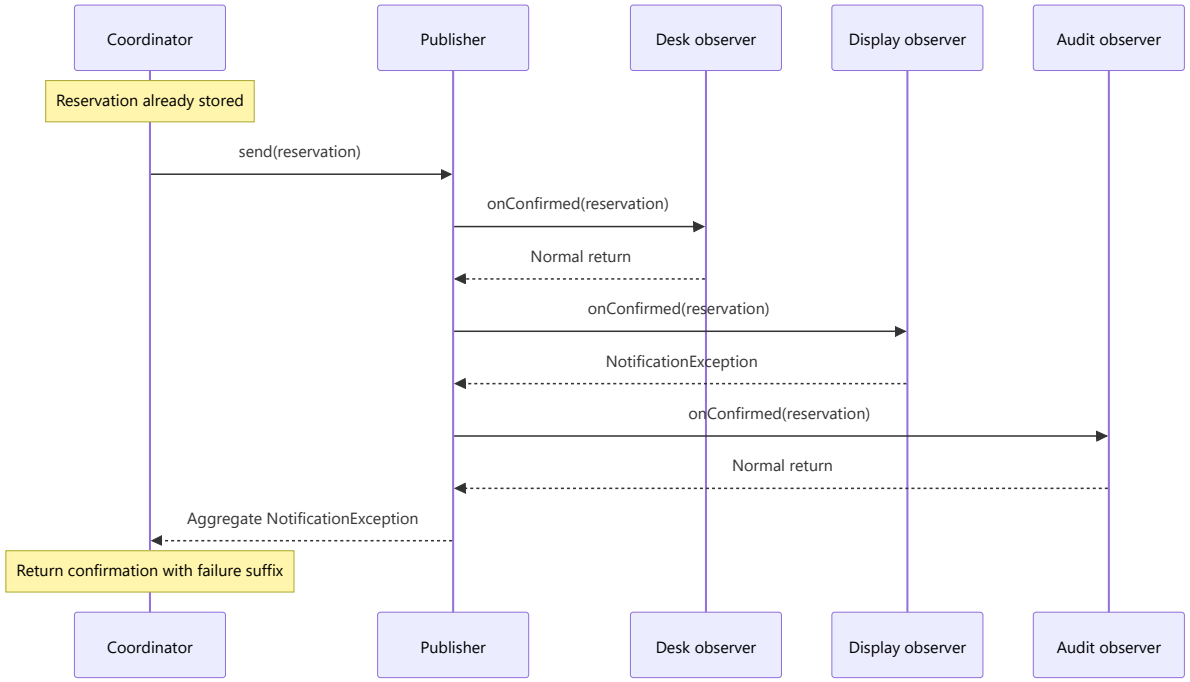


Figure 9.3. One accepted reservation published to three observers, one of which fails with the expected exception. The example shows partial success. It does not roll back earlier observer effects, and the suffix does not mean every observer failed. The aggregate retains individual exceptions as suppressed exceptions for direct inspection; Java's `Throwable` API supports that association [3]. The current coordinator returns only its established message and does not expose detailed observer diagnostics.

Unexpected failures and subscription lifetime

An unexpected programming exception stops dispatch and escapes the coordinator, which catches only `NotificationException`. The reservation still exists because storage happened earlier. Tests explicitly demonstrate that partial outcome. A caller must not infer "nothing happened" from the fact that the operation threw.

The publisher resets its recursion guard in a `finally` block. This avoids leaving it permanently marked as publishing after a failure. The guard is not a concurrency lock or rollback system. Callbacks must respect the existing sender contract and must not mutate the coordinator or start new reservation operations from inside notification.

Keep subscription handles where their lifetime can be managed. Otherwise, a long-lived publisher can retain observers longer than intended. Snapshot iteration defines callback-time registration changes; it does not make this ArrayList-based publisher safe for concurrent calls.

What it costs, and when not to use it

Observer reduces direct knowledge of recipients but makes execution less obvious: work happens inside registered callbacks. Ordering dependencies can become hidden. If one observer must complete a business step before another begins, explicit orchestration may express that requirement better than a loosely described event.

Use the publisher only where a changing set of local observers is useful. Keep a single sender for a fixed output. The example establishes no durable delivery, broker, asynchronous execution or exactly-once guarantee. Adding those mechanisms is outside this chapter, not an automatic next step.

9.5 Command: a request you can carry before running it

The problem: carrying a request before running it

A direct call to `DirectoryReservationService.reserve` immediately performs lookup and reservation processing. A user interface action or small invoker may instead need an object containing the request to be executed when selected. Command separates that request representation from the caller that triggers it.

The example defines `Command<R>` with `R execute()`. `ReserveEquipmentCommand` captures the service, borrower identifier, item identifier, start date and days. Construction validates required references and identifiers but does not perform lookup, allocate a reservation or notify anyone.

The receiver remains responsible for the use case. On execution, the command delegates to the directory-backed service. It neither reproduces borrowing policy nor stores its own reservation list. A method reference or lambda can be sufficient when no named request data is needed; Java's `Runnable` is a familiar no-result action interface, though using it does not itself make a whole Command architecture [4].

When you run it changes the outcome

Between constructing and executing a command, another request may reserve the item. The command must evaluate the current receiver state when executed. Capturing an earlier "available" result and using it as permission would bypass the actual conflict check.

A command record is immutable in its captured fields, but it holds a reference to a mutable service. It is not a snapshot of the application's state. A test constructs the command, creates a competing reservation, and verifies rejection when the command later executes.

Repeated execution also matters. In the supplied example the first execution creates a reservation and the second sees a conflict, producing a different result without another notice. That particular outcome does not mean all commands are idempotent. The implementation has no request identifier, deduplication record or replay protocol.

Failure, undo and simpler alternatives

A directory failure propagates unchanged through the command. The command does not convert it into success, retry it or run a compensating action. The receiver's distinction between absence and outage remains intact.

Undo requires its own semantics. Deleting a record would not necessarily undo an observer's side effect, and the current project has no cancellation contract. Naming a class `Command` does not supply undo, transactions, queueing or persistence.

The test suite checks lazy construction, execution-time state, repeated execution and failure propagation. Use a direct call when there is no reason to represent the action separately. Introduce the command object when carrying the request itself makes an invoker or interaction clearer, and retain the receiver's responsibility for all meaningful validation.

9.6 Testing behaviour that patterns organise

Let the requirement supply the expected result

A test that checks only that an object implements `BorrowingPolicy` cannot establish correct limits. A test that constructs a state object cannot establish a valid transition. A test that counts subscribers cannot establish what happens after one throws.

For Strategy, use independent boundary examples for each supported policy. For State, use the transition table, including rejected edges. For Observer, use ordered event histories and state checks after both expected and unexpected failures. For Command, distinguish construction from execution and observe the real receiver's outcome.

These examples use hand-written test data and the existing JUnit setup. No real university data, remote delivery service or system-clock wait is needed. The unchanged earlier suite remains in the build so new demonstrations do not silently replace regression evidence.

Comparing before and after without circular proof

The lifecycle tests send the same action to the conditional and State implementations, but both are checked against a fixed table. Equality between the two alone would be insufficient. If both accidentally allowed cancellation on loan, they would agree while violating the requirement.

The table covers this exercise's complete state/action space. It still does not prove arbitrary operation sequences with time, external state or permissions that the exercise does not model. Keep the claim proportional to the model.

State identity is deliberately hidden. Tests observe `status()` and operations, not the private enum field. They can survive replacing enum-based states with separate state classes as long as the public contract remains the same.

Testing partial effects directly

The publisher's unexpected-exception test checks that earlier callbacks ran, later callbacks did not, and the coordinator retained its record. That is more informative than checking only the thrown exception. The expected-failure test checks that later observers still run and the recorded aggregate retains both failures in order.

Subscription tests also examine time: removal during a callback affects the next publication, while the removed observer still appears in the current snapshot. These are deterministic sequence tests, not concurrent-race tests.

The command replay test documents one concrete repeated execution. It makes no guarantee about future commands or alternate receiver states. Passing evidence should narrow uncertainty about the selected promise, not expand into an unsupported general assurance.

9.7 Choosing a pattern, and knowing its limits

Compare the options by what they cost

Situation	Reasonable first option	Pattern when justified
Two small fixed limit rules	One conditional or parameterised policy	Strategy when behaviour needs independent implementations
A small fixed transition table	Enum and explicit operations	State when grouping state-specific behaviour improves clarity
One fixed notification output	Direct injected sender	Observer for a changing set of local interested recipients
One immediate reservation request	Direct service call	Command when the request must be represented and passed separately

No row says that increasing class count is progress. A useful explanation identifies where a future edit will occur, which contract stays stable and which new obligations arise. For example, Observer centralises registration but requires decisions about order and failure. State centralises each state's actions but requires consistency between the model and its tests.

Being precise about what this chapter changed

The running project retains Chapter 8's directory boundary and all previous application sources. `NotificationPublisher` is an additional sender option; the existing factory still accepts whichever sender the caller supplies. The new demonstration explicitly chooses the publisher. No previous demonstration is silently rewired.

The command exercise calls the existing application service. The State model remains in the separate `behaviour` package and never updates real reservation or loan storage. The existing borrowing policies are recognised as strategies rather than rewritten for presentation.

The inherited limitations remain visible: in-memory storage, trusted internal APIs, no authenticated identity enforcement, and reservation records that still count and block intervals after loan return. Solving any of those requires a reviewed behaviour change, not another pattern label.

Write a decision others can challenge

Record the concrete problem, selected mechanism, simpler alternative and consequences. "Several local observers need to react; a publisher owns registration and dispatch; one fixed sender remains simpler where recipients do not vary; failures can leave partial effects." Another developer can test or challenge that statement.

"Observer decouples everything" cannot be assessed. Recipients still share an event contract and may acquire temporal dependencies. "State eliminates invalid operations" is likewise too broad unless the interface, transition rules and mutation boundary actually enforce the selected model.

A pattern is a way to organise responsibilities under a particular pressure. It is neither a substitute for requirements nor evidence that operational reliability has been solved.

9.8 Strategy or State? Judging AI proposals

Chapter 15 gathers the AI workflows that recur in this book. This activity is complete on its own, and the fictional proposals below are all you need.

	For this activity
Your goal	Decide whether each proposal fits a policy-selection problem or a lifecycle problem, and what each allows a caller to do
You start with	The lifecycle table in Section 9.3, the existing borrowing strategies, and the failure and replay behaviour in Sections 9.4 and 9.5
What is in bounds	Compare the proposals and qualify the reliability claim; do not adopt any of them without the table
How to check	The transition table, the Strategy and State tests, and the publisher and command tests
How to decide	A verdict with evidence for each claim
No AI account?	The ready-made proposals below are enough

Use these two **fictional AI proposals written by the author** for offline comparison:

Proposal A: "Implement `RequestedPolicy`, `ConfirmedPolicy` and `ReturnedPolicy` as strategies. Expose `setPolicy` so callers can choose any policy before each operation. Calling a return method should simply select the returned policy."

Proposal B: "Store the current phase inside a reservation context. Delegate confirm, checkout, return and cancel to phase-specific behaviour. Allow only transitions from the reviewed table and preserve the current phase after a rejected action."

For the lifecycle requirement, compare what each proposal permits a caller to do. Do not select the answer because it uses more confident wording. Then ask whether the same design is appropriate for student/staff duration rules. Identify the different variation pressure.

A third fictional claim says: "Observer guarantees that all recipients receive the event once, and Command makes retries safe." Refute or qualify it using this chapter's partial-failure and replay tests.

A live AI service is optional. If you use one, give it synthetic inputs, keep the relevant actual output and document your independent checks. Do not fabricate a transcript or use generated implementation and generated expectations as mutual proof.

9.9 Behaviour-pattern habits that mislead

Mistake	Consequence	Better action
Treat category policies as lifecycle positions	Confuses selection with permitted transitions	Identify what changes the behaviour
Expose an unrestricted state setter	Allows invalid jumps	Change state only through reviewed operations
Store context data in shared state objects	One reservation may affect another	Keep progress in each context
Catch all observer errors as expected failures	Hides programming defects	Specify and catch the declared failure type
Assume a thrown notification means no reservation exists	Misreads partial effects	Check the storage-before-publication sequence
Remove observers from a live iteration	Makes dispatch behaviour fragile	Define and test snapshot semantics
Treat a captured command as earlier permission	Bypasses current receiver state	Validate the use case at execution
Assume a pattern supplies retries or undo	Invents unimplemented guarantees	Define those semantics separately

9.10 Try it yourself: lifecycle, publisher and command

You start from Chapter 8′ s finished code, the lifecycle transition table, the conditional and State examples, and the publisher contract. Steps 1 to 5 run on the book′ s project; step 6 runs on your team′ s project.

If your instructor demonstrates the task, they will locate the existing strategies, trace permitted and rejected lifecycle actions before showing the two implementations, and run one expected observer failure to show the retained reservation and the later callback. For guided practice, choose a row of the transition table and explain its expected result independently of the code, predict the effect of unsubscribing during dispatch, and compare a direct service call with constructing and executing a command.

Then work through these steps:

1. Replace the repeated conditional structure in the isolated lifecycle example with State delegation, preserving the agreed table.
2. Show both permitted and rejected actions, including unchanged status on failure and independent contexts.
3. Demonstrate the publisher′ s order and its expected-failure continuation rule. Then construct a `ReserveEquipmentCommand` without executing it, show that construction performs no reservation, and execute it.
4. Choose a second scenario and justify adopting or declining Strategy, Observer or Command compared with a simpler implementation.
5. Evaluate the fictional AI proposals and retain independent evidence for your conclusions.

6. On your team’ s project, find one conditional that selects behaviour by type or tracks a lifecycle. Write a test that protects its current behaviour, then decide whether Strategy, State or neither is justified. Record what you would decline to build and why.

What you should end up with is: the before and after source; a reviewed transition table; runnable tests; a reproducible demonstration; a concise problem–solution–consequence note; and the step 6 decision for your team’ s project.

You are done when these things hold:

- there is no unrestricted phase setter;
- terminal states behave correctly;
- transition outcomes match independently stated expectations;
- an observer failure does not imply a rollback;
- constructing a command performs no reservation;
- the old regression suite still runs.

This exercise does not ask you to wire the proposed lifecycle into the running application.

If you want more, propose cancellation for the real project. Specify its effects on reservation counts, availability, active loans, identifiers and notifications before you change any code, and keep that proposal separate from the behaviour-preserving State exercise.

9.11 Questions to think with, and a transfer challenge

Concept questions

1. Why is category-based borrowing policy a Strategy example while the proposed reservation lifecycle is a State example?
2. Why can stateless enum-based State objects be shared while reservation progress must remain separate?
3. What does snapshot iteration mean for an observer removed during publication?
4. What does a Command capture, and which decisions still belong to execution time?

Applied diagnosis

5. An unexpected observer exception occurs after two callbacks succeed. The caller assumes no reservation was stored and immediately repeats the request. Explain why that assumption is unsupported and identify the observations needed before choosing a recovery action.
6. An AI-generated State implementation has `setStatus(RETURNED)` and a happy-path test. Explain the missing guarantee and design tests that reject an invalid jump or action sequence.

Transfer challenge

A document workflow has draft, approved and published stages. Review policy differs by department, several local views react after publication, and a user interface can hold a publish request before executing it. Identify the four variation pressures, choose only the justified patterns, and specify one invalid transition, one observer failure and one stale-command outcome. State which requirements need clarification before

implementation.

9.12 The critical book review

Your individual critical review of a book from the approved list is due in Week 8. The current assignment brief is the authority on length, sources, format and deadline, and **Appendix E** develops the review method in full (a reading record, a question for each part of the task, and prompts for checking sources). This chapter's patterns give you one concrete angle for a pattern discussion: ask whether the book's example establishes the extension benefit it claims, which simpler alternative the author considered and what cost the author acknowledges.

Read the selected book directly. Keep three things apart: what the author claims, what you infer, and what an AI tool suggested. Reading this textbook does not satisfy the assignment, and an AI summary of a book you have not read is not evidence about it.

What to take away, and what comes next

Strategy selects a behaviour. State governs behaviour through transitions. Observer connects interested recipients to an occurrence, and Command represents an action for later invocation. Each structure brings obligations as well as opportunities, especially around state, ordering and partial effects.

The project now has an optional tested local notification publisher and a request-action example, while the proposed lifecycle stays clearly separate. Chapter 10 looks at how these responsibilities fit into application structure and dependency direction.

Sources and further reading

Checked 12 and 13 October 2026. Code, test data, transition rules and publisher policies are original teaching examples. The behavioural patterns used here are treated in this chapter.

1. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. **Publisher catalogue**. Read the Strategy, State, Observer and Command entries in an authorised copy.
2. Oracle, **Java SE 17 PropertyChangeSupport**. A concrete listener-management API; its documented guarantees should not be assumed for the chapter's custom publisher.
3. Oracle, **Java SE 17 Throwable**. Suppressed exceptions used to retain multiple observer failures.
4. Oracle, **Java SE 17 Runnable**. A no-result action interface for comparison with the chapter's result-bearing command.
5. JUnit Team, **JUnit 5.13.4 User Guide**. Parameterised examples and assertions in the companion code.
6. Martin, R. C. (2002), *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall. The module's required text; the Week 8 reading is its Strategy, Command, Observer and State chapters. Publication details as listed in the module descriptor.

Chapter 10: From Classes to Application Structure

> *Layered architecture, package cohesion, stability metrics, and architectural fitness.*

OO Development and Agile Practices
Designing, Testing and Evolving Software with AI Assistance

The equipment application has come a long way. It has borrowing policies, a directory adapter, notification collaborators, loan operations and several demonstrations, and each class has a responsibility you can explain. Yet most of the production code still sits in Java' s unnamed package, and that organisation says nothing to a newcomer. May a borrowing policy depend on a directory client? Where should a storage implementation be constructed? The code cannot tell you.

The next change request is deliberately modest: make those boundaries visible while keeping the demonstrated reservation behaviour exactly as it is. To do that the team has to decide which classes express domain rules, which coordinate operations, which talk to the outside world and which assemble the application. Moving files around will not answer those questions by itself.

This chapter reorganises the existing single-process program and shows how to check the result. It introduces a small storage contract for reservations, moves concrete storage out of the coordinator, and writes down dependency rules that a test can check against compiled code. There is no database, web framework or distributed system in sight. Along the way you will learn to assign responsibilities to presentation, application, domain and infrastructure code, and to tell runtime calls from the direction of source-code dependencies. You will see the difference between an entity' s identity, its immutability and its value equality, locate a focused repository contract and its implementation, and restructure a use case while checking its observable behaviour and its compatibility limits. You will compare two package designs by cohesion, coupling and stability, reading the counts without mistaking them for a measure of quality, and you will finish by recording an architecture decision and preparing evidence for peer review and Milestone 3.

This is Week 9, supporting LO2 and LO6 and preparing Milestone 3. The weekly readings are Martin' s principles of package design [7] and selected chapters of Evans' s domain-driven design [8]; use the Moodle links for the exact pages. The approved module requirements remain authoritative, and the chapter creates no new deadlines, weights or marking rules.

Milestone 3 covers design patterns and layered architecture, with a demonstration and a retrospective. Do not wait until the end of the chapter: use this checklist from the start. By your demonstration you should be able to show, on your team' s project:

1. at least one design pattern from Chapters 8–9 used where it removes a real problem, and one place where you declined to use one;
2. a compact package diagram saying which packages may depend on which;
3. one restructuring change, with moves kept separate from logic changes;
4. passing checks, including a dependency check or a documented review of imports;
5. a short architecture decision record with alternatives, consequences and a revisit trigger;

6. real peer review of that change, with the feedback addressed;
7. a demonstration and a retrospective with a next action; and
8. a statement of what the architecture does not do.

Section 10.9 explains each item. Start with items 2 and 5, which need no code. The approved Milestone 3 instructions decide what is submitted; this list is preparation, not a rubric.

You start from Chapter 9′ s tested Java 17 project, including the stronger assertions added when that chapter was reviewed (the companion lists them in Section 2.5). The finished version preserves its existing use-case results and earlier demonstrations, but package names, imports, launcher names and direct coordinator construction change, so no Java source or binary compatibility is claimed. The separate code-and-solutions file contains the complete finished version, tests, migration guidance and worked answers.

Sections 10.1 to 10.5 establish the structural model, including package design; 10.6 and 10.7 examine domain semantics and storage; and 10.8 and 10.9 connect the implementation to an architecture decision and review. Read 10.1 to 10.5, 10.7, 10.8 and 10.9 as the core, leaving the Ports and Adapters comparison at the end of 10.4 for later. Section 10.6 (entities and value objects) is working knowledge: understand the difference between identity and value meaning, and why Java equality is a separate decision. You do not implement either one. What you must do is steps B1 to B3 and T1 to T5 of the practical activity in Section 10.12. Steps B1 to B3 run on the book′ s project and are reading, tracing and diagnosis, because the migrated version is supplied and there is nothing to rebuild; steps T1 to T5 run on your team′ s project and produce your Milestone 3 evidence. If you want more, there is the Ports and Adapters comparison, the AI activity in Section 10.10, steps B4 to B6 (break the dependency rules on purpose, and reproduce the migration yourself), the questions and the transfer challenge in Section 10.13, and the extension at the end of Section 10.12.

10.1 Why designing good classes is not enough

Responsibilities exist at more than one scale

A well-designed `LoanPeriod` can reject an invalid interval while the surrounding application still has poor dependency organisation. A correct directory adapter can be called directly from several user interfaces, each repeating eligibility logic differently. Class-level correctness does not by itself establish a coherent application boundary.

Chapters 5–9 separated several responsibilities. Borrowing policies decide count and duration eligibility. The coordinator combines those decisions with conflict checks and accepted-record storage. The directory service resolves borrower categories before invoking the coordinator. Notification implementations handle local output. We now make those distinctions visible across packages.

Layering is one way to narrow the responsibilities a developer must consider together. Presentation, domain logic and data access are a familiar separation, with an application service boundary useful where operations need coordination [1]. The chapter′ s exact packages are a worked design decision, not a universal folder template.

The architecture questions you already have

The small program already has architecture: someone chooses its dependencies, owns its state and decides when effects occur. Those decisions exist even without a server or database. The question is whether they are explicit enough to review and change.

For example, the coordinator currently owns an ArrayList of reservations. That is easy to follow, but ties use-case orchestration to one storage mechanism. Extracting a narrow storage contract lets the chapter demonstrate dependency direction and storage ownership without inventing persistence requirements.

This extraction has a cost. It creates another interface and requires a contract for ordering, snapshots and duplicate identifiers. If a program will remain a tiny script with one private list, the extraction may not be worthwhile. Here it supports the growing project' s structural learning goal and a concrete, independently testable boundary.

What must stay the same

The refactoring preserves accepted and rejected reservation strings, policy limits, half-open overlap rules, directory failure distinctions, loan behaviour and notification timing. Storage still precedes notification. An expected notification failure still retains the reservation. The optional publisher still has its declared partial-failure semantics.

The Java integration surface does change. Callers import named packages. Direct `ReservationCoordinator` construction requires an additional repository argument. Command-line launchers use fully qualified names. These changes must be documented even when all user-facing demonstration transcripts remain identical.

Treat the before/after boundary precisely: behaviour is protected within the supplied sequential, in-memory version. Nothing here promises compatibility for an unseen application that reflects on class names, serialises Java objects by type name or depends on the old constructor signature.

10.2 Layers: giving each boundary a reason

A reason for each boundary

The new package root is `edu.se506.lending`. Under it, domain code expresses the equipment vocabulary and rules; application code coordinates operations; infrastructure implements technical boundaries; presentation adapts an interaction to the application; bootstrap code constructs the chosen object graph.

Package	Responsibility	Representative types
`domain`	Values, records and borrowing decisions	`LoanPeriod`, `Reservation`, `BorrowingPolicy`
`application`	Use-case coordination and existing result formatting	`ReservationCoordinator`, `DirectoryReservationService`, `LoanService`
`application.port`	Contracts needed by application operations	`ReservationRepository`, `BorrowerDirectory`, `NotificationSender`
`infrastructure.directory`	Provider-specific translation	`LegacyDirectoryAdapter`, `LegacyDirectoryClient`
`infrastructure.notification`	Selected local notification mechanisms	`ConsoleNotificationSender`, `NotificationPublisher`
`infrastructure.persistence`	Concrete reservation storage	`InMemoryReservationRepository`
`presentation`	Thin console interaction	`ReservationConsole`
`bootstrap`	Assembly and executable demonstrations	`ReservationApplication`, `Chapter10Demo`

The persistence package name identifies a storage responsibility. Its current implementation is explicitly in memory and loses data when the process ends. Naming the package does not make it durable.

Packages show the boundaries; rules enforce them

A package is a namespace and organisation mechanism. Public Java types can still be imported from another package. A folder named `domain` does not prevent a developer from constructing a console sender inside it.

The chapter therefore states permitted production dependencies and adds a test that examines compiled class dependencies. Tests may cross boundaries to assemble their setup and observe outcomes; their dependencies are not treated as production dependencies. Startup wiring also has a deliberate exception because it must choose concrete implementations.

Source package	Permitted local targets
`domain`	`domain`
`application.port`	`domain` and `application.port`
`application`	`domain`, `application.port` and `application`
`infrastructure`	`domain`, `application` (including its ports) and `infrastructure`
`presentation`	`domain`, `application` and `presentation`
`bootstrap`	every application package, and explicitly demonstrated isolated examples

Ordinary production code must not refer to the isolated `patterns`, `behaviour` or `examples` packages. Figure 10.1 in Section 10.4 draws the same rules.

The existing `patterns` and `behaviour` packages remain separate teaching examples. The new value-equality demonstration lives in `examples.structure`. Ordinary application, domain, presentation and infrastructure code must not depend on these exercises. A bootstrap demonstration may show an isolated example explicitly, as Chapter 9 already did.

Don’t multiply layers mechanically

A class does not need an interface merely because it crosses a package boundary. `DirectoryReservationService` remains a concrete application service. A caller can depend on that stable operation without adding a second interface that repeats every method.

Likewise, not every domain object needs a repository or a matching service. `LoanPeriod` is constructed and validated as a value-shaped object; storing it through its own repository would obscure its role as part of a reservation. The loan service’s internal map remains where it was because this chapter extracts one justified boundary, not a generic storage framework.

10.3 What each layer does: the four layers in practice

Presentation translates an interaction

`ReservationConsole` receives a directory-backed application service and a `PrintStream`. Its `reserve` method invokes the service and prints the existing result. It neither selects a borrower category nor duplicates the seven-day student rule.

The test supplies a byte buffer instead of the real console and checks exact success and rejection output. A directory exception propagates without printing a success line. This thin adapter is sufficient for this version; it is not an interactive menu, input parser or complete user interface.

The bootstrap demonstration supplies synthetic directory rows and chooses the sender. Presentation itself does not construct the adapter or storage implementation. This makes it possible to examine output behaviour without hiding construction decisions inside it.

Application coordinates the use case

`DirectoryReservationService` performs lookup and delegates a valid borrower to `ReservationCoordinator`. The coordinator applies policy, checks reservation conflicts, creates an accepted record, stores it and invokes its sender. `LoanService` coordinates checkout and return using the supplied clock.

These are application operations because they arrange several responsibilities to complete a use case. Service Layer discussions describe this kind of application boundary and coordination [2]. The label does not require a remote service, dependency-injection framework or one service per entity.

`ConfirmationFormatter` remains in the application package because existing returned strings are part of this version' s observable contract. It is also used by the console notification adapter. A future structured result could separate domain outcomes from wording, but doing that now would mix a behaviour/API redesign with this restructuring.

Domain expresses meaning and rules

Borrower categories, borrowing policies, reservations, loans and loan periods belong to the domain vocabulary. Policy implementations contain count/duration decisions. `LoanPeriod` validates its interval and supplies overlap behaviour. These classes do not know about console output, directory provider codes or repository implementations.

Domain code may use appropriate Java standard-library types such as `LocalDate`, collections and null checks. Independence does not mean rewriting every useful standard type. It means avoiding dependencies on application orchestration and external mechanisms that the domain rules do not require.

The model is deliberately modest. We do not introduce a full set of Domain-Driven Design concepts or declare every class an aggregate. First explain the actual identity, invariants and ownership in the existing program.

Infrastructure supplies the mechanisms

The directory adapter translates the synthetic provider contract into `BorrowerDirectory`. The in-memory repository implements reservation storage. Notification adapters implement output or local callback dispatch. Their code depends on the application-owned contracts and domain types it handles.

Infrastructure does not own borrower eligibility simply because it sees borrower records. Moving the seven-day limit into the directory adapter would make a policy depend on the chosen provider representation. Similarly, the repository should not send a notification when a record is added: application orchestration already determines when that effect occurs.

The bootstrap package chooses and connects these pieces. Its concrete imports are intentional. Keeping construction there makes selection visible without requiring a framework or global service registry.

Where an outside collaborator sits

The equipment directory from Chapter 8 is the book's data-backed collaborator, and it shows where any outside collaborator belongs, whether it is a data source or something that scores or predicts. Its contract, `BorrowerDirectory`, is owned by the application and lives in `application.port`. Its failure type, `DirectoryException`, sits beside it, because callers must tell an outage from an absent borrower. Its implementation, `LegacyDirectoryAdapter` with the provider-shaped `LegacyDirectoryClient`, lives in `infrastructure.directory` and translates provider codes into the domain's `Borrower`. Bootstrap constructs the adapter and passes it to `DirectoryReservationService`.

If your team's project has a model-backed collaborator instead, it would occupy the same three positions: a narrow port with bounded outcomes and a defined failure, an adapter in infrastructure, and construction in bootstrap. The application would not import the adapter, and a test could substitute the port exactly as the directory tests do.

10.4 Which way do dependencies point, and how do we check?

Reading arrows as source dependencies

Figure 10.1 groups the packages. An arrow means that classes in one group may refer to classes in another. The application node includes its port contracts. Bootstrap is the explicit composition point.

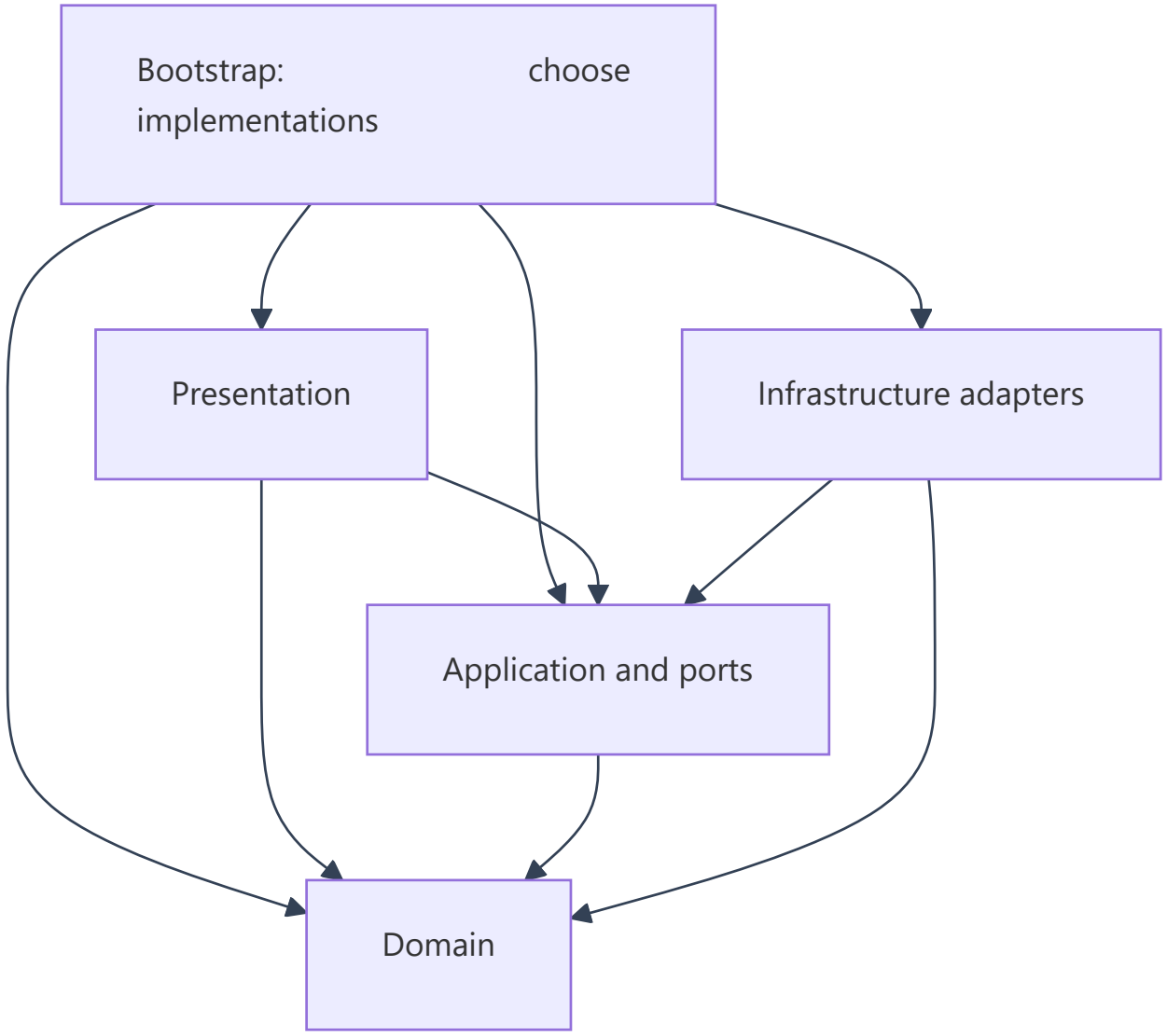


Figure 10.1. Permitted source dependencies. Each box is a group of packages, and an arrow means that classes in the first group may refer to classes in the second. A missing arrow means the reference is not permitted: `Domain` refers to nothing else in the application, and `Presentation` and `Infrastructure` may not refer to each other or to `Bootstrap`. The figure shows source references only, not the order of calls. It draws the rows of the matrix in Section 10.2, including the two arrows into `Domain` from `Bootstrap` and `Presentation`.

The application has no arrow to infrastructure. `ReservationCoordinator` imports `ReservationRepository`, which is an application-owned contract. `InMemoryReservationRepository` imports and implements that contract. Bootstrap constructs the implementation and passes it to the coordinator.

Within application, the port package has a narrower rule: it may depend on domain types and other ports, not concrete application services. This keeps a storage contract from requiring the coordinator it exists to support.

A runtime call can go towards an adapter

At runtime the coordinator calls an object that happens to be an in-memory repository. That does not require its source to import the concrete class. Interface dispatch connects the call to the implementation selected during assembly.

The sequence below focuses on an accepted reservation after borrower lookup. It omits the directory call to keep the storage and notification ordering visible.

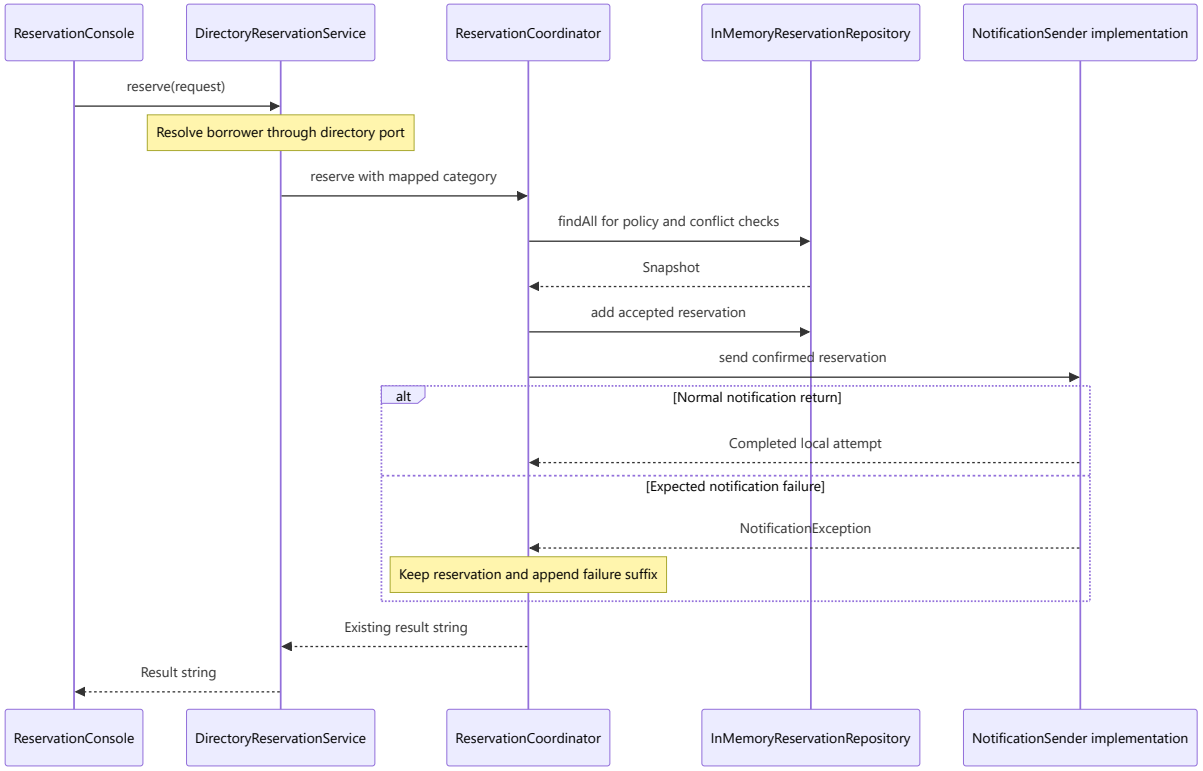


Figure 10.2. One accepted reservation at run time. Each box at the top is an object, time runs down the page, solid arrows are calls and dashed arrows are returns. Chapter 8 explains this notation.

The actual coordinator may obtain more than one snapshot during a request. The diagram groups those reads; it does not claim one storage call or a transaction. The application still assumes one sequential session.

Checking what the code really references

`DependencyRulesTest` invokes the JDK' s `jdeps` analyser on compiled production classes [3]. It rejects local dependency edges that contradict the package policy, including forbidden references to isolated examples outside bootstrap. It also checks that dependencies were actually discovered, avoiding a vacuous pass on missing output. A second test in the same class builds the graph of package-to-package references from the compiled classes and searches it for a cycle (the next subsections explain why).

This test can detect a fully qualified concrete-class reference even if no import statement was added. It does not detect reflective class names in strings, runtime configuration relationships or a semantically misplaced rule that uses only permitted types. Source review and behaviour tests remain necessary.

The test requires a full Java 17 JDK with the `jdk.jdeps` module. It is part of the supplied Maven test run. Package rules are project decisions, not universal rules inferred by the analyser.

Optional comparison: Ports and Adapters

The directory, sender and repository interfaces can also be discussed as ports, with infrastructure implementations as adapters. The console drives the application, while storage and notifications are collaborators the application calls. This inside/outside emphasis is consistent with Cockburn's discussion of isolating application behaviour from runtime mechanisms [4].

The terminology provides another view of the same small program. It does not require six adapters, a hexagonal deployment topology or a new package for every method. Use it only when it clarifies which side owns the contract and which mechanism may be replaced.

10.5 Cutting packages well: cohesion, coupling and stability

Section 10.4 asked which package may refer to which. This section asks whether the packages themselves are well cut. It applies three questions to the same version of the code (what changes together, what clients depend on, and whether dependencies point towards stability) and gives counts that you can reproduce.

Package cohesion: what should change together

Package design asks the questions of class design at a larger scale. A cohesive package holds types that change for the same reason and are used together. A package that mixes unrelated reasons to change is edited more often and by more people. Martin's principles of package design put this as two rules [7][9]. The Common Closure Principle (CCP): things that change together belong together. The Common Reuse Principle (CRP): things that are used together belong together, so a client is not forced to depend on classes it never uses. A third rule, the Reuse/Release Equivalence Principle (REP), says that the unit you reuse is the unit you release: a package that other teams adopt needs a version, release notes and a stable contract, so it should be one coherent thing. This project has no separately released package, so REP gives context here and no check [9].

Test the cuts with concrete changes, not with slogans. For each change, list the packages whose production code must change.

Change request	Packages that change	What the answer shows
Staff may reserve for 21 days	`domain` (`StaffReservationRules`)	The policy change stays in one package
The provider renames its status code `Y`	`infrastructure.directory`	Provider vocabulary stays outside the application
The confirmation wording changes	`application` (`ConfirmationFormatter`)	Its users, including the console sender, see the change without being edited
Reservations are stored in a file	`infrastructure.persistence` (a new class) and `bootstrap` (the construction line)	The coordinator is not edited, because it names only the port
A third borrower category is added	`domain` (the category and its policy), `infrastructure.directory` (the code mapping) and `bootstrap` (the policy map)	One business idea touches three packages

Most changes touch one or two packages, which is evidence that the cuts are reasonable. The last row is a legitimate cost to record, not a flaw to hide: a new category needs a policy, a way to recognise it and a place that registers it. The test is only as good as the changes you choose, so always add one you expect to be awkward.

Package coupling: what clients depend on

Coupling between packages is the number of references that cross a package boundary. Two habits keep it manageable. First, depend on a small contract where the other side may vary. The coordinator refers to `ReservationRepository`, not to the in-memory class, so changing the storage touches one package. Second, keep the package graph free of cycles. If `application` referred to `infrastructure.notification` while that package referred back to `application`, neither could be compiled, tested or replaced without the other. Martin calls this the Acyclic Dependencies Principle (ADP) [7][9].

The layer matrix in Section 10.2 rules out cycles between layers by construction, because each pair of layers allows references in one direction only. It does not do so inside one layer: the matrix lets `infrastructure.directory` and `infrastructure.notification` refer to each other freely, so a cycle between those two would pass the direction test. That is why `DependencyRulesTest` has the separate cycle check.

The `application.port` package shows the reuse side of the same idea. Its five types are small contracts and their failure types, so a class that needs one of them is not forced to depend on a large service. If a port package grew to hold many unrelated contracts, splitting it would be worth considering: a client that uses only the sender should not be affected when the directory contract changes.

Package stability: depend in the direction of stability

A package is **stable** when it is costly to change: many other packages depend on it, and it depends on little itself. A package is **unstable** when little depends on it, so changing it disturbs few others, and it depends on many. Both kinds are needed. The useful rule, Martin' s Stable Dependencies Principle (SDP), is that dependencies should point towards stability, so that code that changes often depends on code that changes rarely [7][9].

The idea can be counted. For a package P:

- **Afferent coupling, Ca**: the number of classes outside P that depend on classes inside P (incoming references).
- **Efferent coupling, Ce**: the number of classes inside P that depend on classes outside P (outgoing references).
- **Instability, I = Ce / (Ca + Ce)**, from 0 (maximally stable) to 1 (maximally unstable) [7][9].

Counting rules matter, because tools count differently. This book counts classes, treats a nested class as part of its outer class, counts only references between this project' s own packages and ignores the Java standard library. The supplied program `tools/PackageMetrics.java` applies exactly these rules to the compiled classes (the companion explains how to run it). For this version it reports the following, with package names shortened by removing the prefix `edu.se506.lending`.

Package	Classes	Ca	Ce	I
`domain`	8	16	0	0.00
`application.port`	5	14	3	0.18
`infrastructure.notification`	3	7	3	0.30
`application`	6	9	4	0.31
`infrastructure.directory`	2	2	1	0.33
`infrastructure.persistence`	1	1	1	0.50
`presentation`	1	1	1	0.50
`bootstrap`	7	0	7	1.00

Read the table as a set of questions. `domain` has I = 0: sixteen classes outside it depend on it and it depends on nothing else in the application. That is why it should stay small and free of mechanisms, since a change there can ripple everywhere. `bootstrap` has I = 1: nothing depends on it, so it can change freely, which suits the place where concrete choices are made. Most arrows in Figure 10.1 run from a higher I to a lower one: `application` (0.31) depends on `application.port` (0.18), which depends on `domain` (0.00), and `presentation`, `infrastructure.persistence` and `infrastructure.directory` each depend on a package with a lower I.

One arrow does not: `infrastructure.notification` (0.30) depends on `application` (0.31), because `ConsoleNotificationSender` uses the concrete `ConfirmationFormatter` (Section 10.3). The difference is one hundredth, and the counts come from a handful of classes, so it is no defect. It is the arrow to examine first in a review, because it is the one concrete dependency that runs from a mechanism to an application class.

An isolated package has no defined instability. The packages `patterns` and `examples.structure` have no references to or from the lending packages, so Ca + Ce = 0 and the formula would divide by zero. The tool prints *undefined*. Do not report such a package as 0, which would call it maximally stable, or as 1: it is simply isolated, as intended. (`behaviour` is referenced only by the Chapter 9 demonstration, so for it Ca = 1 and I = 0.)

Treat the numbers as questions, not grades. With one to eight classes per package, adding a single class moves a ratio by several hundredths. A package can have a good number and a muddled responsibility, and a poor number can be the right design (bootstrap). Compare numbers only when they come from the same tool and counting rules.

Package abstractness: stable packages should be easy to extend

The Stable Abstractions Principle (SAP) adds the last question [7][9]. A package that many others depend on cannot change freely, so it should be abstract enough to be extended without being edited. Martin measures abstractness, A, as the share of a package's classes that are abstract classes or interfaces. This book does not compute A for this version: it would need a second tool, and with one to eight classes per package the ratio moves with every class added. The principle still works as a reading question.

- `application.port` is made of contracts and their failure types, so it is mostly abstract, and it is stable (I = 0.18). SAP is satisfied.
- `bootstrap` is concrete and unstable (I = 1.00). SAP is satisfied: nothing depends on it.
- `domain` deserves the question. Sixteen classes depend on it, yet it is mostly concrete records and rules. It stays safe to depend on because it changes only when the borrowing vocabulary changes, and `BorrowingPolicy` gives the part that varies an abstract seam. A package that was stable, concrete and also changed often would be the warning sign.

A worked comparison: where should `NotificationException` live?

The notification implementations throw `NotificationException`, and the coordinator catches it. Two placements are plausible. In Option A, the choice made in this version, the exception sits in `application.port` with the `NotificationSender` contract. In Option B it sits in `infrastructure.notification`, beside the senders that throw it, on the argument that implementations should own their failures.

Option B was tried on a copy: the class was moved and the imports changed. The compiled direction check failed with `application.ReservationCoordinator → infrastructure.notification.NotificationException`, because the coordinator must name the exception in order to catch it, so the application would depend on infrastructure. The cycle check also rejects Option B, because `infrastructure.notification` already depends on `application`. The instability figures would not have shown the problem (`application` stays at 0.31 and `infrastructure.notification` would fall to 0.27, which looks unremarkable). The rejection came from the direction rule and the cycle check, not from the ratios.

The rule that follows: a type that callers must catch or inspect belongs with the contract they depend on, not with one implementation. Implementations may still define failures that no caller handles, and those can stay inside an adapter.

10.6 Entities and value objects: identity or value?

Identity is a domain decision

A reservation has an identifier such as R1. Two records with that identifier refer to the same intended reservation identity even if their other data differs. Whether such disagreement represents an update or an invalid duplicate depends on the storage contract. This version uses append-only storage and rejects duplicate identifiers.

`Reservation` remains immutable and does not override Java `equals` or `hashCode`. Consequently, two separately constructed objects with the same ID are not automatically equal through Java's inherited object equality. The repository uses the explicit identifier as its key rather than relying on object equality.

Entity identity and Java equality must therefore be explained separately. Calling a class an entity does not silently implement equality, persistence or a lifecycle. Nor does making it immutable remove its domain identity. Evans' s reference describes an entity as an object distinguished by its identity rather than its attributes, and a value object as one in which you care only about attributes and logic, treated as immutable [8, pp. 11–12].

Value meaning and Java value equality

A loan period is characterised by its start and exclusive end. Its meaning comes from those dates, not from a separate generated identifier. The existing `LoanPeriod` validates the interval and prevents mutation, making it useful for encapsulating the rule.

However, the class in the running project also retains inherited object equality. It is not a complete Java value-equality implementation: two periods containing equal dates are distinct according to `equals`. The chapter preserves that behaviour rather than silently changing it during a package move.

The separate `examples.structure.PeriodValue` record demonstrates component-based equality and hashing. Tests construct two equal-date records and observe one logical value in a set; a different end date remains distinct. Record semantics are useful here because the components are immutable dates and the constructor validates them.

Adopting that representation in the real model would be a separate, reviewed change. Replacing a class with a record can affect equality, hashing, textual representation and APIs even when its accessors look similar.

Immutability helps snapshots, but it is not the whole contract

An unmodifiable list prevents the caller from adding or removing elements through that list. It does not generally make the elements immutable. The repository' s snapshots are safe for this version because `Reservation` and its period are immutable as well.

A previously returned snapshot stays unchanged when another reservation is stored. The repository tests check both that independence and insertion order. Java' s `List.copyOf` supplies an unmodifiable list, but the application must still reason about the values placed inside it [5].

These distinctions matter when models evolve. If a later reservation becomes mutable, a shallow list copy no longer establishes a historical snapshot of its fields. Revisit that promise before claiming that an old snapshot represents past state.

10.7 Repositories and services: a storage contract

Extracting the smallest useful storage contract

The repository interface provides only `findAll()` and `add(reservation)`. It exposes insertion-ordered, stable, unmodifiable snapshots and an append operation that rejects null or duplicate IDs without replacing existing records.

```
public interface ReservationRepository {
    List<Reservation> findAll();
    void add(Reservation reservation);
}
```

The full contract comments and implementation appear in the code file. The in-memory adapter uses a `LinkedHashMap` keyed by reservation ID and returns `List.copyOf` snapshots. This gives the current use case collection-like access without exposing its storage fields. Repository literature provides the broader collection-oriented abstraction [6]; this is a deliberately smaller example, without a data mapper or query language.

The interface lives in `application.port` because the application use case consumes it. Domain-owned repository contracts are another possible design when the domain's needs justify that ownership. Location should follow actual responsibilities rather than a rule that all repository interfaces belong in one universal layer.

What the extraction leaves alone

The coordinator still calculates counts, checks conflicts and allocates size-based identifiers. Those are not moved into storage. The factory supplies a fresh repository for each coordinator, retaining the earlier session ownership and R1 starting point.

The new constructor is an explicit injection point, not a promise to support arbitrary preloaded or shared stores. The size-based identifier scheme assumes the current append-only, sequential ownership arrangement. A persistent store, deleted records or concurrent writers would require a separate identifier and consistency design.

No `delete`, generic `save`, transaction manager or repository for every class is introduced. Adding unused operations would imply lifecycle decisions that have not been made. The real project still does not cancel reservations or release their intervals after a loan return.

A port does not give you transactions

A test supplies a repository whose `add` throws before appending. The exception propagates, and notification is not reached. That establishes the order for this specific failure. It does not prove that a future database failure can never occur after a write succeeded.

Similarly, repository snapshots and duplicate-ID checks do not make count-check, conflict-check and append atomic across concurrent requests. The program remains sequential and in memory. Introducing a database would require reviewing those assumptions, not just replacing the constructor call.

An application service coordinates operations; a repository manages access to stored domain records; domain policies express rules. The boundaries are useful precisely because they expose what each responsibility does and does not promise.

10.8 Recording a small architecture decision

Writing down the concrete choice

The worked decision, ADR-010, proposes named packages, application-owned ports, one in-memory reservation adapter and explicit bootstrap wiring. Its status is **implemented teaching example; peer review pending**. That wording separates author verification from a review that has not occurred.

The decision identifies the problem: construction and storage ownership were insufficiently visible as the example grew. It compares keeping the unnamed package, introducing narrow boundaries, and adopting a larger framework or generic repository design. The selected approach keeps the build and domain behaviour small while making dependencies reviewable.

Its consequences include new imports, fully qualified launcher names, one additional coordinator constructor argument and a full-JDK dependency for the architecture test. These are real costs, not incidental details to hide under "clean architecture."

ADR-010 is the worked model for the book's project. On your team's project your decision will differ: a different boundary, different alternatives, different costs. Write your own record in the same form, and do not copy ADR-010's decision unless it is also yours. A record whose alternatives include "do nothing" and whose revisit trigger names an event you could observe is more useful than one that only justifies what was already done.

Keeping the change reviewable

The code file includes a migration map, complete final sources and the decision text. Older tests move into a regression package; direct constructor calls receive the new repository. Their existing expected behaviours are retained.

In a real repository, a useful review sequence separates package moves from the repository extraction and newly added checks. The supplied version is not evidence of particular Git commits or an opened pull request. Learners should record their actual sequence and review comments.

A decision record should state when to revisit the choice. Here those triggers include persistence, multiple writers, structured result objects and an agreed cancellation lifecycle. They do not automatically justify building those features now.

10.9 Peer review and preparing Milestone 3

Reviewing one scenario across the boundaries

Trace a valid staff request from presentation through directory lookup, policy evaluation, storage and notification. Then trace a missing borrower and an expected notification failure. Check that the layer diagram matches both source dependencies and the observed order of effects.

Ask who creates each concrete adapter, whether application code imports it, and whether domain code depends on application services. Inspect the repository's snapshot and duplicate-ID contract. Verify that presentation has not copied a policy rule to make the demo pass.

The dependency test and passing regression suite answer different questions. The first examines compiled relationships; the second examines selected behaviour. A semantically poor design can pass both, so a reasoned human review remains part of the evidence.

Preparing evidence without manufacturing it

The table below applies to your team's project. The companion's migrated version, ADR-010 and sample PR description are the worked model for the book's project, not evidence about your team. Each row corresponds to an item in the Milestone 3 checklist at the top of the chapter.

Checklist item	What to produce	What it should establish
1	Pattern note	A pattern used where it removes a real problem, with the reason, and one place you declined to use one
2	Compact package diagram	Actual responsibilities and the declared dependency direction, matching your code
3	Restructuring diff or PR	What moved and what logic changed, kept apart
4	Regression result and dependency report	Existing examples still execute with preserved expectations; compiled or reviewed references follow the chosen rules, and any package cycle is reported
5	ADR	Problem, alternatives (including doing nothing), decision, consequences and revisit triggers
6	Actual review record	A reviewer examined the change and the feedback was addressed
7	Demonstration and retrospective	What worked, what did not and one next action
8	Limits statement	What the structure does not provide: for example persistence, concurrency safety or a cancellation lifecycle

Use the approved Milestone 3 instructions to determine submission requirements. The table is preparation guidance, not a new rubric. Local author testing is not hosted CI evidence, and a sample PR description is not a completed peer review.

A useful reviewer question is not "Does it have four layers?" but "Where would this concrete change belong, and which existing tests protect its consequences?" The answer should identify files, contracts and trade-offs rather than only package names.

10.10 Checking an AI's dependency review

The goal here is to examine a proposed dependency review without treating its wording as evidence. Give the assistant the package map, the relevant Java files and the declared dependency policy. Limit the task to identifying references and responsibility concerns; do not ask it to invent a database or rewrite the module requirements.

For offline use, this is a **fictional AI answer written by the author**:

> The coordinator calls an in-memory repository at runtime, so it must import infrastructure and violates dependency inversion. Move the repository interface into infrastructure. Every entity needs a repository and a service. Making the reservation list unmodifiable makes every future element immutable. The bootstrap imports adapters, so remove it and let services construct their dependencies. Since all tests pass, the pull request has been peer reviewed.

For each claim, inspect actual imports or compiled dependencies and separate factual observations from design advice. Draw the source dependency and runtime call separately if needed. Identify the missing evidence for claims about equality, snapshots and review status.

Record the proposed claim, the reference inspected, your conclusion and any remaining limitation. A live AI service is optional; use synthetic code and preserve only an actual transcript if one is produced. Another generated opinion is not independent verification.

10.11 Structure habits that mislead

Mistake	Consequence	Better action
Treat package names as enforced architecture	Public types can still form forbidden dependencies	State and check a dependency policy
Confuse runtime calls with source imports	Misdiagnoses interface-based dispatch	Trace both relationships separately
Move provider codes into the domain	Couples rules to an external representation	Keep translation in the adapter
Duplicate eligibility in presentation	Creates inconsistent entry points	Delegate the use case
Call immutable objects automatically value-equal	Misreads `equals` and collection behaviour	Inspect and test equality semantics
Add generic CRUD to every class	Invents unsupported lifecycle operations	Add only needed storage capabilities
Construct adapters inside application services	Hides concrete dependencies	Assemble them explicitly at bootstrap
Claim passing tests prove peer review	Substitutes one type of evidence for another	Record actual review separately
Grade a package by its instability number	A number from a few classes is coarse, and a muddled package can score well	Use the number to pick what to examine, then reason about responsibility
Report an isolated package as I = 0 or I = 1	The formula is undefined when Ca + Ce = 0	Report it as undefined and explain why
Accept a package cycle because the layers look right	Two packages inside one layer can still depend on each other	Check the package graph for cycles, not only the layer directions

10.12 Try it yourself: restructure and check the dependencies

You start from the migrated version in the companion (Sections 3–5), the package responsibility table in Section 10.2, the package-metrics table in Section 10.5 and your team's repository. Steps B1 to B6 use the book's project; steps T1 to T5 use your team's project.

If your instructor demonstrates the task, they will trace the current coordinator's storage access, show the port and adapter after extraction, and compare the source dependency diagram with the runtime sequence. They will also run `tools/PackageMetrics.java` and `DependencyRulesTest`, and show one preserved demonstration using its new fully qualified launcher name. For guided practice, classify a domain policy, the

directory adapter, an application service and a bootstrap class, and explain why the same concrete import is appropriate in bootstrap but forbidden in application code.

What you must do: the book's project (read, trace and diagnose; nothing to rebuild)

- **B1. Classify.** For `StaffReservationRules`, `LegacyDirectoryAdapter`, `DirectoryReservationService`, `InMemoryReservationRepository` and `ReservationApplication`, name the package, say what it may refer to under the Section 10.2 rules, and name one thing it must not refer to.
- **B2. Check the references.** Run the build and the compiled dependency check. Find in the report one reference that Figure 10.1 permits and explain why it is permitted, then find the reference that runs from `infrastructure.notification` to `application` and say what it is for.
- **B3. Diagnose the package design.** Use the table in Section 10.5 as a model. (a) Write two change requests of your own and list the packages each would change. (b) Using the metrics table, identify the one arrow that points to a less stable package and say whether you would act on it. (c) Decide, with reasons, whether `ConfirmationFormatter` should stay in `application`, and name the test or number that would change your mind.

What you must do: your team's project (your Milestone 3 evidence)

- **T1. Diagram.** Draw a compact package diagram of your team's codebase in the style of Figure 10.1, with boxes for groups of packages and arrows for permitted references. Mark one reference you would forbid and say why. If your project has no packages yet, propose the groups.
- **T2. Decision.** Choose one architecture decision your team actually faces, such as where a boundary or port belongs, whether to extract a storage contract, or where construction happens. Write an ADR in the form of ADR-010: context, decision, alternatives including doing nothing, consequences and revisit triggers.
- **T3. Restructure.** Make one small restructuring change on a branch. Keep package moves separate from logic changes, keep your existing checks passing and say which behaviour you protected.
- **T4. Check the direction.** Run a compiled dependency check (the book's `DependencyRulesTest` or `jdeps`) if your build can. Otherwise review the imports by hand and record that this was a manual review. Look for cycles as well as for forbidden directions.
- **T5. Review.** Ask a teammate to review the change and the ADR. Record their feedback and what you changed. Then work through the Milestone 3 checklist at the top of the chapter and record which items are complete, which are gaps and who will close each one.

If you want more (the book's project)

- **B4. Break the rules on purpose.** In a scratch copy, construct `InMemoryReservationRepository` inside `ReservationCoordinator`. Run the dependency test, record the violation and restore the original.
- **B5. Reproduce the migration.** Starting from the Chapter 9 project, move the classes into the declared packages, extract the repository contract, supply it through construction, update imports and launchers, and preserve the earlier results.
- **B6. AI activity.** Complete the Section 10.10 activity.

What you should end up with is: written answers to B1 to B3; for your team's project, the T1 diagram, the T2 decision record, the T3 diff or PR description, the T4 dependency result, and the T5 review record with the checklist status. A pull request is a good vehicle for T3 and T5 where your course workflow permits it.

You are done when these things hold:

- your diagram matches your code;
- your decision record states a real alternative and an observable revisit trigger;
- the restructuring keeps moves apart from logic changes and keeps existing checks green;
- the dependency evidence covers direction and cycles and states whether it was compiled or manual;
- the review is real;
- for the book's analysis, the classification obeys the Section 10.2 rules, each change request names the packages it touches, and the formatter conclusion cites evidence.

Do not claim review or hosted CI that has not occurred.

If you want more, propose replacing size-based identifiers or adopting value equality for the running project's period class. Identify the observable changes and the tests before you implement anything, and keep that proposal separate from this structural change.

10.13 Questions to think with, and a transfer challenge

Concept questions

1. Why can the coordinator call an infrastructure object without depending on its concrete class?
2. What distinguishes an application service from a domain policy and a repository?
3. Why does an immutable reservation still have entity identity, and why does that not automatically define Java equality?
4. What can the `jdeps` check establish, and what does it leave for review?

Applied diagnosis

5. A console class checks the seven-day limit before calling the service. A new API endpoint does not. Explain the responsibility problem and a test that exposes inconsistent behaviour.
6. A developer adds a database repository and announces concurrent safety because `add` rejects duplicate IDs. Identify the missing consistency question across eligibility, conflicts and insertion.
7. In the metrics table `bootstrap` has I = 1.00 and `domain` has I = 0.00. Explain why neither is a defect and what a change to each would disturb.

8. A new class in `infrastructure.directory` refers to `infrastructure.notification`, and a class there already refers back to the directory package. Which check catches this, and which check does not?

Transfer challenge

A clinic appointment application has eligibility rules, an external patient directory, a console view and an in-memory appointment list. Propose responsibilities and dependency directions without inventing deployment services. Choose one storage port, explain which layer owns its contract and distinguish an appointment's identity from its scheduled time value. Name one behaviour that must be protected during restructuring, and one change request you would use to test whether your package cuts are cohesive.

What to take away, and what comes next

Application structure makes responsibilities and dependency decisions visible beyond individual classes. The worked version introduces named packages, a narrow storage port, explicit assembly and a thin presentation adapter, while preserving the tested use-case outcomes. Package design adds four questions: what changes together and is released together, what clients depend on, whether dependencies point towards stability, and whether a stable package is abstract enough to extend.

The remaining limitations are deliberate: in-memory sequential state, existing string results and unchanged lifecycle rules. Chapter 11 looks at refactoring and legacy code, building on the tests and boundaries established here.

Sources and further reading

Online sources 1–6 were checked on 12 October 2026, and sources 8 and 9 on 13 October 2026. The package policy, code and decision record are original teaching material.

1. Fowler, M., **Presentation Domain Data Layering**. Separation of responsibilities and the application-service variation.
2. Stafford, R., **Service Layer**, in Fowler's enterprise application architecture catalogue. Application operations and coordination.
3. Oracle, **The Java 17 jdeps Command**. Compiled dependency analysis.
4. Cockburn, A., **Hexagonal Architecture**. Original inside/outside and ports/adapters discussion.
5. Oracle, **Java SE 17 List**. Unmodifiable list factories and collection contracts.
6. Hieatt, E. and Mee, R., **Repository**, in Fowler's enterprise application architecture catalogue. Collection-like access to domain objects.
7. Martin, R. C., *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall, 2002. The module's required text; the Week 9 reading is its principles of package design. The definitions of afferent and efferent coupling and of instability, and the package principles, are also stated in Martin's own earlier paper [9]. This chapter treats package design in Section 10.5.
8. Evans, E., **Domain-Driven Design Reference: Definitions and Pattern Summaries**. Domain Language, 2015, licensed CC BY 4.0. Layered Architecture (p. 10), Entities (p. 11) and Value Objects (p. 12), with Modules and Repositories on pp. 15 and 17. A recommended companion to the Week 9 reading, which lists selected chapters of Evans's 2004 book.

9. Martin, R. C. (2000), **Design Principles and Design Patterns**. The author's own paper, hosted by the University of Turku. Its text was extracted and read on 14 October 2026: it names the reuse/release equivalence, common closure, common reuse, acyclic dependencies, stable dependencies and stable abstractions principles, and defines Ca, Ce, I and abstractness A. Its counting unit is the class, as in this chapter. It is the primary source for Section 10.5; [7] is the module's required treatment.

Chapter 11: Refactoring and Legacy Code

> *Characterisation tests, seams, code smells, and incremental refactoring steps.*

OO Development and Agile Practices
Designing, Testing and Evolving Software with AI Assistance

The records office has an old overdue report. It lists every unreturned loan that is past its due date and adds a fine. This week two things go wrong. A student points out that a laptop two days late shows a fine of 0.00 yet still counts as overdue. A colleague on another machine sees amounts printed as `0,50` where everyone else sees `0.50`. Then the office asks for a small change: staff should get seven days of grace instead of five.

You open the code. It is one method of about forty lines. The staff rule appears in one branch, the student rule in another, and a third branch handles anyone else. The grace number, five, appears twice in the staff branch, once in a comparison and once in a subtraction, and a change to only one of them would be a bug. The same literal 5 also stands for the staff cap, so a search-and-replace would change the wrong thing as well. The method reads today's date from the system clock, so its output changes every day. Nobody wrote tests, and the author has left.

There is a second reason for care. A colleague once tidied a report like this one, and the new version was shorter. It also listed fewer loans. Nobody could say whether a defect had been repaired or a promise had been broken, because nobody had recorded what the old report promised. What do you change first? Not the staff grace period. Before changing a rule you need to know what the code does today, so that you can tell afterwards whether you changed only what you meant to change. This chapter is about that situation: **changing code you do not yet understand, without tests that protect it**. The method is to observe, to pin what you observe, to reshape the structure in small checked steps, and to keep every correction of behaviour separate from every improvement of structure.

Along the way you will learn to diagnose a smell from a concrete difficulty of change instead of treating it as a verdict, to tell a refactoring from a behaviour change and say what must stay the same, and to write characterisation tests that record current behaviour, including behaviour that looks wrong. You will create a seam that removes a hidden dependency so that the code can be tested, apply small refactorings (Extract Method, Extract Class, Move Method, Rename) with a check after each step and a way back, and record uncertainties, suspected defects and technical debt separately, with evidence.

This is Week 10, supporting LO3 and LO4. The weekly reading is the refactoring chapter of Martin [13]; Feathers [14] is the recommended companion on legacy code. The Week 10 lab is refactoring untested legacy code with regression tests. The chapter creates no new deadlines, weights or marking rules. Quiz 2 is held in Week 10 and covers dependency injection, TDD and test doubles, design patterns and layered architecture (Weeks 5 to 9); the legacy lab in this chapter is separate from that scope, and Section 11.11 gives revision pointers for the quiz.

You start from Chapter 10's Java 17 project and its package structure. This chapter adds one isolated package, `legacy`, which stands for the old records system's overdue report. It is not part of the lending layers: production code must not refer to it, just as it must not refer to `patterns` or `behaviour`, and the dependency test from Chapter 10 now checks that. Only bootstrap launches it or feeds it rows. The companion holds the complete code, every refactoring stage, the tests, model answers and teaching notes.

Sections 11.1 to 11.6 are the core reading. Section 11.7 (technical debt and evidence) is working knowledge: learn to describe debt with evidence and to separate defects from preferences; you do not manage a debt register for a real project. What you must do is steps B1 to B3 and T1 to T4 of the practical activity in Section 11.10. Steps B1 to B3 run on the book's project, where the legacy report and its tests are supplied and you extend the tests and take the first refactoring steps; steps T1 to T4 run on your team's project, whose provided starter codebase contains intentional smells, so that you characterise one untested method and refactor it in small commits. If you want more, there are steps B4 to B7 (break the boundary on purpose, make the separate functional correction, do the AI activity, and connect the report to the live loans), the questions and the transfer challenge in Section 11.11, and the extension at the end of Section 11.10.

11.1 Working with unfamiliar code

What makes code "legacy"

Feathers gives a working definition: legacy code is code without tests [14]. The age of the code does not matter. Code written last month becomes legacy when you have to change it, you cannot tell what a change will break, and nothing will tell you. That is the sense used here. The report in this chapter is old, but the habits that produced it (global state, hidden dependencies, one long method) are not.

The dilemma, and the way out

Refactoring is a change to the internal structure that leaves external behaviour the same [1]. To know that behaviour stayed the same you need tests. But the code is hard to test because of its structure, so you want to refactor first. The way out is a short sequence, and Feathers' own account of why existing code resists testing is worth reading beside it [7]:

1. **Observe.** Run the code and read it with a specific question in mind. You do not need to understand all of it.
2. **Make the smallest edit that lets you test it.** This is a *seam* (Section 11.4). Keep the edit mechanical, so that the compiler and a quick read can check it.
3. **Pin.** Write characterisation tests that record what the code does now (Section 11.3).
4. **Reshape.** Refactor in small steps, running the tests after each one (Sections 11.5 and 11.6).
5. **Only then change behaviour**, in a separate step with its own tests.

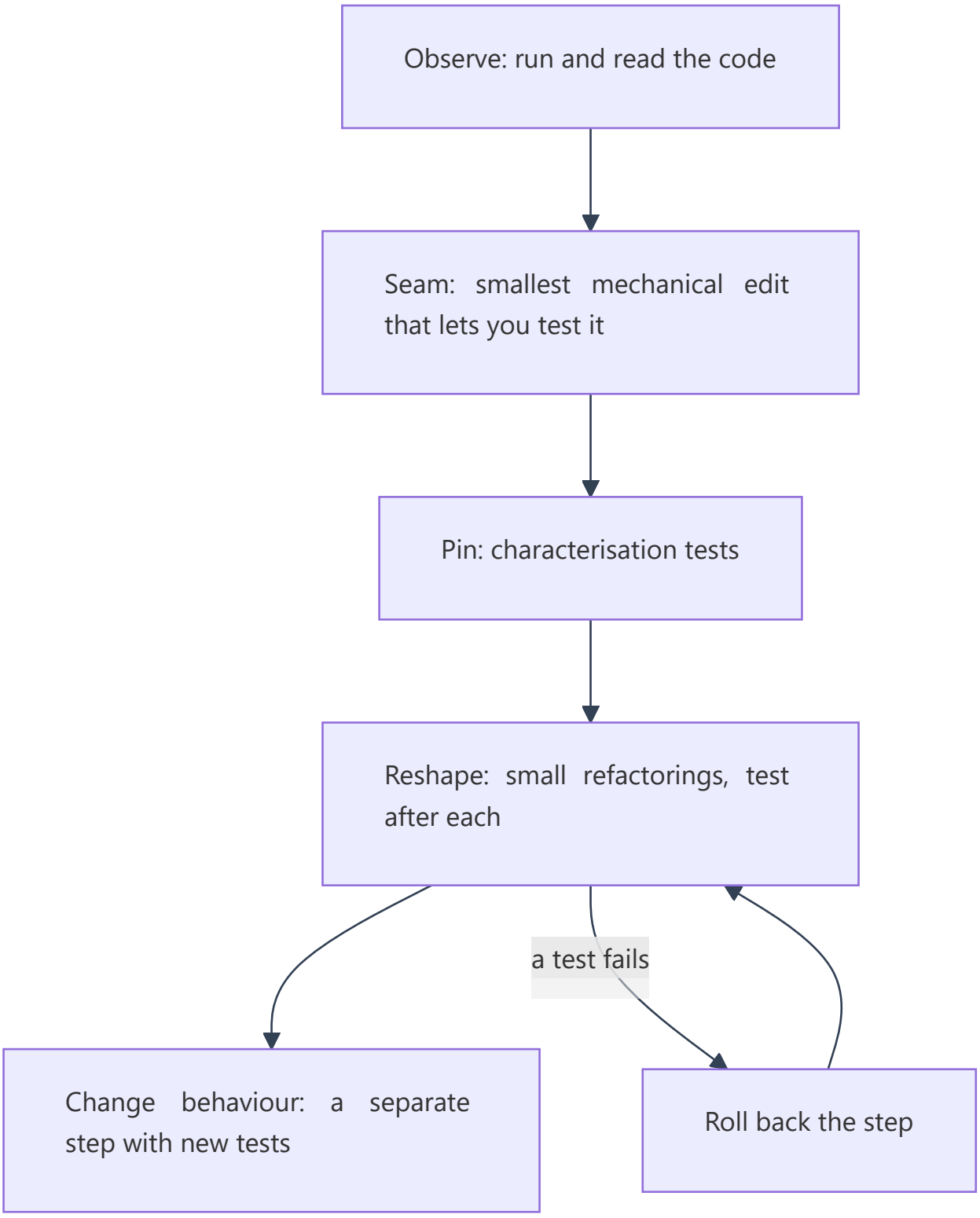


Figure 11.1. The order of work on code you do not understand. Boxes are activities and arrows mean "then". The loop at the end is the rule for a failing test: undo the last step and redo it, instead of debugging forward.

The code as found

Here is the report, exactly as it was found. Read it once and write down what you notice before reading on.

```
public class OverdueReport {

    public static String build(List<LoanRow> rows) {
        String out = "";
        double total = 0;
        int count = 0;
        LocalDate today = LocalDate.now();
        out += "OVERDUE REPORT " + today + "\n";
        for (int i = 0; i < rows.size(); i++) {
            LoanRow r = rows.get(i);
            if (r.returned) {
                continue;
            }
            long late = ChronoUnit.DAYS.between(r.due, today);
            if (late > 0) {
                double fine = 0;
                if (r.kind.equals("S")) {
                    if (late > 2) {
                        fine = (late - 2) * 0.50;
                        if (fine > 10) {
                            fine = 10;
                        }
                    }
                } else if (r.kind.equals("T")) {
                    if (late > 5) {
                        fine = (late - 5) * 0.25;
                        if (fine > 5) {
                            fine = 5;
                        }
                    }
                } else {
                    fine = 1.00 * late;
                }
                out += r.item + " " + r.borrower + " " + late + " day(s) late, fine "
                        + String.format("%.2f", fine) + "\n";
                total += fine;
                count++;
            }
        }
        out += count + " overdue, total fines " + String.format("%.2f", total) + "\n";
        return out;
    }

    public static void print(List<LoanRow> rows) {
        System.out.print(build(rows));
    }
}
```

`LoanRow` is the old system's export format: an item, a borrower, a kind (`"S"` for a student, `"T"` for staff, as in the old directory), a due date and a returned flag, all public mutable fields. Other programs read it, so it stays as it is.

Separate what you know from what you guess

Reading is evidence only when you record how sure you are. A reading note keeps the three apart.

Observation	How you know	Status
Returned loans are skipped	Read: `continue` on `returned`	Known
Fines start after a grace period and are capped	Read: two nested comparisons per kind	Known
Anyone who is neither `"S"` nor `"T"` pays 1.00 a day, with no grace and no cap	Read: the final `else`	Known what, unknown whether intended
The output depends on the date the program runs	Read: `LocalDate.now()`	Known
Amounts are printed with a comma in some locales	`String.format` without a locale; the documentation says the default locale is used and `%f` localises the decimal separator [8]	Likely; confirm by running
A loan two days late is listed with a fine of 0.00 and still counted	Read: the count is incremented inside `late > 0`	Likely; confirm by running

"Unknown whether intended" is an honest status. It is not your job at this stage to decide that the behaviour is wrong.

11.2 Smells: prompts to investigate, not verdicts

A code smell is, in Fowler's words, a surface indication that usually corresponds to a deeper problem, and it is not always a problem [5]. Chapter 5 introduced smells and the principles behind them, so this chapter does not repeat the definitions. What matters here is the use you make of a smell: it is a prompt to investigate, and it is only worth acting on when you can name a **concrete difficulty of change** that it causes.

Smell in the report	The difficulty it causes
Duplicated branches (the student and staff blocks have the same shape)	A change to the rule shape must be made in every branch; the grace number appears twice in each, so editing one of the two gives a wrong result
Long method	To change the fine you must read the loop, the formatting and the totals as well
Hidden dependency on the system clock	The output differs every day, so no test can compare it with a fixed answer
Hidden dependency on the default locale	The same data prints differently on two machines
Magic numbers (`2` , `0.50` , `10` , `5` , `0.25`)	A reader cannot tell which number is a grace period and which is a cap
A string kind compared with `equals`	An unexpected kind falls into the final branch without anyone deciding that
`String +=` in a loop	Not a difficulty of change here; a possible slowness for very long reports

The last row is a deliberate counter-example. Not every smell needs repair. A smell in code that nobody changes costs very little, and the value of repairing it depends on how often the code changes and how much the difficulty costs [4]. Decide where to spend effort by asking which changes are coming. Here the office has just asked for a change to a grace period, so the duplicated branches and the magic numbers are worth fixing now. The `String +=` loop is not.

11.3 Pinning what the code does: characterisation tests

A **regression test** protects behaviour that everyone agrees is correct. A **characterisation test** records what the code does today, whatever that is, so that you will notice if a change alters it [3]. The two look the same in the test framework, but they make different promises. A passing characterisation test means "unchanged". It does not mean "correct".

How to write one

Feathers describes a short loop [3]:

1. **Isolate the code** so that it can run in a test. This is often the hardest part (Section 11.4).
2. **Write a test with a placeholder expectation**, a value you know to be wrong.
3. **Run it and read the actual result** from the failure message.
4. **Record that result** as the expectation, after you have looked at it.
5. **Rename the test** to say what you now understand the code does.
6. **Keep exploring** with further inputs, especially the boundaries and the odd cases.

Step 4 contains the discipline: do not paste the result blindly. Look at each one and classify it.

Class	Meaning	What you do
Expected	It matches the rule you were told	Keep the test
Surprising	You did not expect it, but it may be intended	Keep the test; ask the owner
Suspected defect	It looks wrong	Keep the test, mark it clearly, and record the question; do **not** fix it here
Unknown	You cannot tell	Keep the test and record the question

What the report does

The observed fines for one loan, run on the supplied code with today fixed at 20 October 2026, are below. A comparison with the rule you were told is the first review.

Kind	Days late	Fine	Comment
S	1, 2	0.00	Listed and counted, with no fine (grace)
S	3	0.50	First fine
S	10	4.00	
S	22, 23, 100	10.00	The cap is reached at 22 days and holds
T	1–5	0.00	Listed and counted, with no fine (grace)
T	6	0.25	First fine
T	25, 26, 100	5.00	The cap is reached at 25 days and holds
X (unknown)	1	**1.00**	Suspected defect: no grace
X (unknown)	100	**100.00**	Suspected defect: no cap
s (lower case)	3	**3.00**	Suspected defect: a typo in the data costs six times the student fine
blank	2	**2.00**	Same branch

Four further observations belong in the suite: a loan due today or later is not listed; a returned loan is ignored; a null kind throws `NullPointerException` (and only for a loan that is actually overdue); and under a German locale the amounts print with a comma. The list shows one example of each class that is not simply expected: the grace listing and the comma are *surprising*, the unknown kinds are *suspected defects*, and the null-kind failure is *unknown*.

One test, in the style used here

```
// SUSPECTED DEFECT, pinned on purpose: any other kind pays 1.00 a day with no grace and no cap.
@ParameterizedTest
@CsvSource({"X,1,1.00", "X,100,100.00", "s,3,3.00"})
void otherKindsPayTheFullRateWithNoGraceOrCap(String kind, int daysLate, String fine) {
    assertEquals(oneLoan(daysLate, fine), OverdueReport.build(List.of(open(kind, daysLate)), TODA
}
```

The comment is part of the test. In six months someone will see this test fail while making an unrelated change, and the comment tells them whether to fix their change or to ask a question.

A second net

A table of examples covers only the inputs you thought of. A *differential* check adds a different kind of cover: keep a copy of the original code as an oracle (a trusted source of expected results), generate thousands of inputs with a fixed seed and compare the old and new output. The companion contains such a test. It is stronger than the examples for the kinds of input it generates, and it says nothing about inputs outside them.

Neither net proves that the new code is equivalent to the old. They show that a chosen set of behaviours is unchanged, and the right amount of testing depends on how risky the change is. Add more where you are about to change more.

11.4 Seams: getting hidden dependencies out of the way

A **seam**, in Feathers' s term, is a place where you can change what the program does without editing the code at that place, and Fowler describes it the same way [6]. To use a seam you need an **enabling point**: somewhere you can choose the other behaviour. The report has three hidden dependencies, and each needs a different answer.

Hidden dependency	Where it is	How to isolate it
The system clock	`LocalDate.now()` inside `build`	A parameter seam: add `build(rows, today)` and let the old method call it
The default locale	`String.format("%.2f", ...)`	Pin the locale in the test and restore it afterwards
Standard output	`print` writes to `System.out`	Capture `System.out` in one test and restore it afterwards

The clock

This is the only edit made before there are tests, so it must be as small and mechanical as possible. The method is split in two. The existing signature stays, and delegates to a new one that takes the date:

```
public static String build(List<LoanRow> rows) {
    return build(rows, LocalDate.now());
}

public static String build(List<LoanRow> rows, LocalDate today) {
    String out = "";
    double total = 0;
    int count = 0;
    out += "OVERDUE REPORT " + today + "\n";
    //  ...  the rest of the method is unchanged
```

The edit moves one expression from the body to a call, and deletes one line. The compiler checks that nothing else changed, and a reader can check the rest by eye. Existing callers see no difference. Fowler' s catalogue calls this move Change Function Declaration, and lists Add Parameter among its aliases [2]. This is the same idea as the injected `Clock` in Chapter 7, at the smallest possible scale: a parameter is enough here, because the report is a static method with no object to inject into.

Checking the enabling edit by running it

Compiling and reading are good checks, but this edit has no test yet, so run the evidence that is available. The companion supplies a small **probe**: it contains the original report with only its class name changed (`OriginalUnseamed`) and a program that calls the original and the seamed report through their public one-argument entry points. It compares their output for a handful of rows (a late student, a late staff member, a

row due today, an unknown kind, a returned loan and an empty list) and checks one literal result. It refuses to run if the date changes while it executes.

The probe is bounded. It examines a few rows, not every input, and it cannot tell you the report is correct. What it adds to the compiler and your eyes is a direct comparison of the code as found with the code after the only untested edit. Keep it small, run it once, and then put the real protection in the characterisation tests.

Who chooses the zone?

The report now takes a date, so the *caller* decides which calendar date "today" is. A `Clock` carries a time zone, and `LocalDate.now(clock)` converts the clock's instant into a date using it [9]. At 22:30 UTC on 21 October it is already 22 October in Baghdad. A loan due on 21 October is therefore not yet late in UTC and one day late in Baghdad. A test makes the difference concrete: it builds two dates from one instant with `Clock.fixed` and two zones, and records the two reports. The seam did not remove the question of which zone is right. It moved the question to the one place that has to answer it.

The locale and the output stream

The other two dependencies are left alone, because changing them would be a bigger edit than the tests justify. Instead the tests deal with them. One test setup step sets the default locale to `Locale.ROOT` before each test and restores it afterwards. One test sets it to `Locale.GERMANY` and records the comma. One test replaces `System.out` and restores it in a `finally` block. Both are global, process-wide state. Tests that change them cannot safely run in parallel, and a test that forgets to restore them will break an unrelated test far away, so the restoring is part of the technique.

A better seam would be a `PrintStream` parameter on `print`, and a locale argument to the formatting. They are worth adding when you have a reason to change those lines. Without a reason, the test setup is enough.

Other seams

Two other seams appear in legacy code and are worth knowing by name. In **subclass and override** you make a class non-final, and a test subclass replaces one method. In **extract interface** you put a dependency behind an interface, which is exactly the move of Chapter 6. Both change more code than a parameter does, so choose the smallest seam that gets the code under test.

11.5 Reshaping the report: Extract, Move and Rename

Refactorings have names so that a team can say what it did and a reviewer can check it. Fowler's catalogue calls the first one Extract Function and the second Move Function; the first edition called them Extract Method and Move Method, and the names are used interchangeably [2][15]. The companion lists the full file for every step. The steps below follow the order in which they were applied, and the characterisation tests and the differential test passed after each one.

Stage 2: Extract Method

The fine calculation and the formatting are lifted out of the loop, unchanged, into two private methods:

```
double fine = fineFor(r.kind, late);
out += r.item + " " + r.borrower + " " + late + " day(s) late, fine " + money(fine) + "\n";
...
private static double fineFor(String kind, long late) { /* the three branches, moved as they were
private static String money(double amount) { return String.format("%.2f", amount); }
```

The behaviour is the same because the moved code is the same. The loop now reads in one glance, and the formatting rule (and its locale dependency) lives in exactly one place.

Stage 3: Extract Class and Move Method

Look at the three branches again. They differ only in three numbers: free days, a daily rate and a cap. That is a missing concept, and it can be a small class.

```
record FinePolicy(int graceDays, double ratePerDay, double cap) {
    double fineFor(long daysLate) {
        if (daysLate ≤ graceDays) {
            return 0;
        }
        return Math.min((daysLate − graceDays) * ratePerDay, cap);
    }
}
```

`fineFor` moves into the new class (Move Method), and the report keeps three named policies:

```
private static final FinePolicy STUDENT = new FinePolicy(2, 0.50, 10.00);
private static final FinePolicy STAFF = new FinePolicy(5, 0.25, 5.00);
/** Suspected defect, preserved on purpose: any other kind pays 1.00 a day with no grace and no ca
private static final FinePolicy ANY_OTHER_KIND = new FinePolicy(0, 1.00, Double.MAX_VALUE);
```

The new code is not obviously equal to the old, so check it. For the student branch the old code had `if (late > 2) { fine = (late - 2) * 0.50; if (fine > 10) fine = 10; }`. The new code returns 0 when `late ≤ 2`, which is the same condition negated, and otherwise returns the smaller of `(late - 2) * 0.50` and 10, which is what the capped assignment computed. The staff branch is the same with 5, 0.25 and 5. For other kinds, grace 0, rate 1.00 and no cap give `1.00 * late`. The tests then confirm the algebra for the inputs they cover.

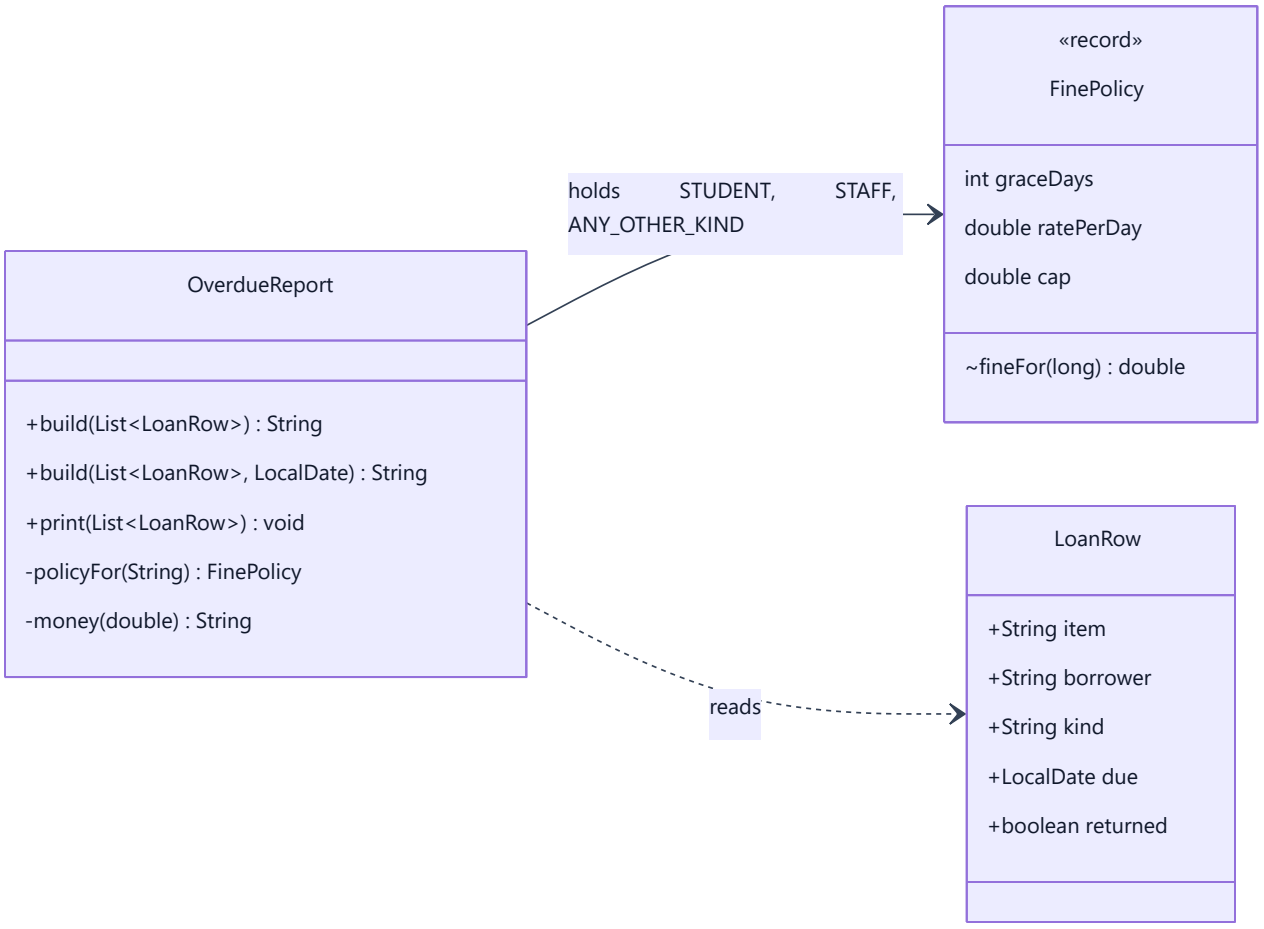


Figure 11.2. The report after stage 3. The solid arrow means that `OverdueReport` holds three `FinePolicy` values; the dashed arrow means that it only uses `LoanRow`, which it reads. A `+` marks a public member, a `-` a private one and a `~` a package-private one. `LoanRow` is drawn as the old system's format, with public fields, and has not changed.

Notice what the refactoring did not do. The odd third branch did not disappear, and its oddity is now visible: it has a name, a comment and a policy of its own. Hiding a suspected defect inside tidy code is the opposite of what refactoring is for. The edit also did not change `LoanRow`, and it did not change the signatures of `build` and `print`, because other programs call them.

Stage 4: Rename and tidy

The last step renames `r` to `row` and `late` to `daysLate`, uses a for-each loop, replaces `String +=` with a `StringBuilder` and renames `count` and `total` to `overdueCount` and `totalFines`. None of this changes the output. The method after the four stages is:

```
public static String build(List<LoanRow> rows, LocalDate today) {
    StringBuilder report = new StringBuilder("OVERDUE REPORT " + today + "\n");
    double totalFines = 0;
    int overdueCount = 0;
    for (LoanRow row : rows) {
        if (row.returned) {
            continue;
        }
        long daysLate = ChronoUnit.DAYS.between(row.due, today);
        if (daysLate > 0) {
            double fine = policyFor(row.kind).fineFor(daysLate);
            report.append(row.item).append(' ').append(row.borrower).append(' ')
                    .append(daysLate).append(" day(s) late, fine ").append(money(fine)).append('\
            totalFines += fine;
            overdueCount++;
        }
    }
    report.append(overdueCount).append(" overdue, total fines ").append(money(totalFines)).append(
    return report.toString();
}
```

The request that started the chapter is now a one-number change: `new FinePolicy(7, 0.25, 5.00)` for staff. It is a functional change, so it waits for its own commit and its own test change.

11.6 Small steps, and how to step back

One step, one check, one commit

The stages above were not applied at once. Each was a separate commit with the tests run before it was recorded. A sequence like this is easy to review and easy to undo.

Commit	Kind	What the reviewer checks
1. Add a date parameter to `build` (seam)	Mechanical edit before tests	The old method delegates and nothing else moved
2. Add characterisation tests and the differential test	Tests only	Every expectation was run and reviewed; surprises are commented
3. Extract `fineFor` and `money`	Refactoring	The moved code is identical; all tests pass
4. Extract `FinePolicy`	Refactoring	The algebra above; all tests pass; the odd policy is visible
5. Rename and tidy	Refactoring	No behavioural edits hide among the renames

Mixing a refactoring and a behaviour change in one commit is the most common way to lose control, because a failing test no longer says which of the two caused it. Chapter 5 made the same point with its two-stage change.

Deciding what a difference means

When a check reports a difference, ask one question before touching anything: was this behaviour change

agreed? The answer decides what you do next.

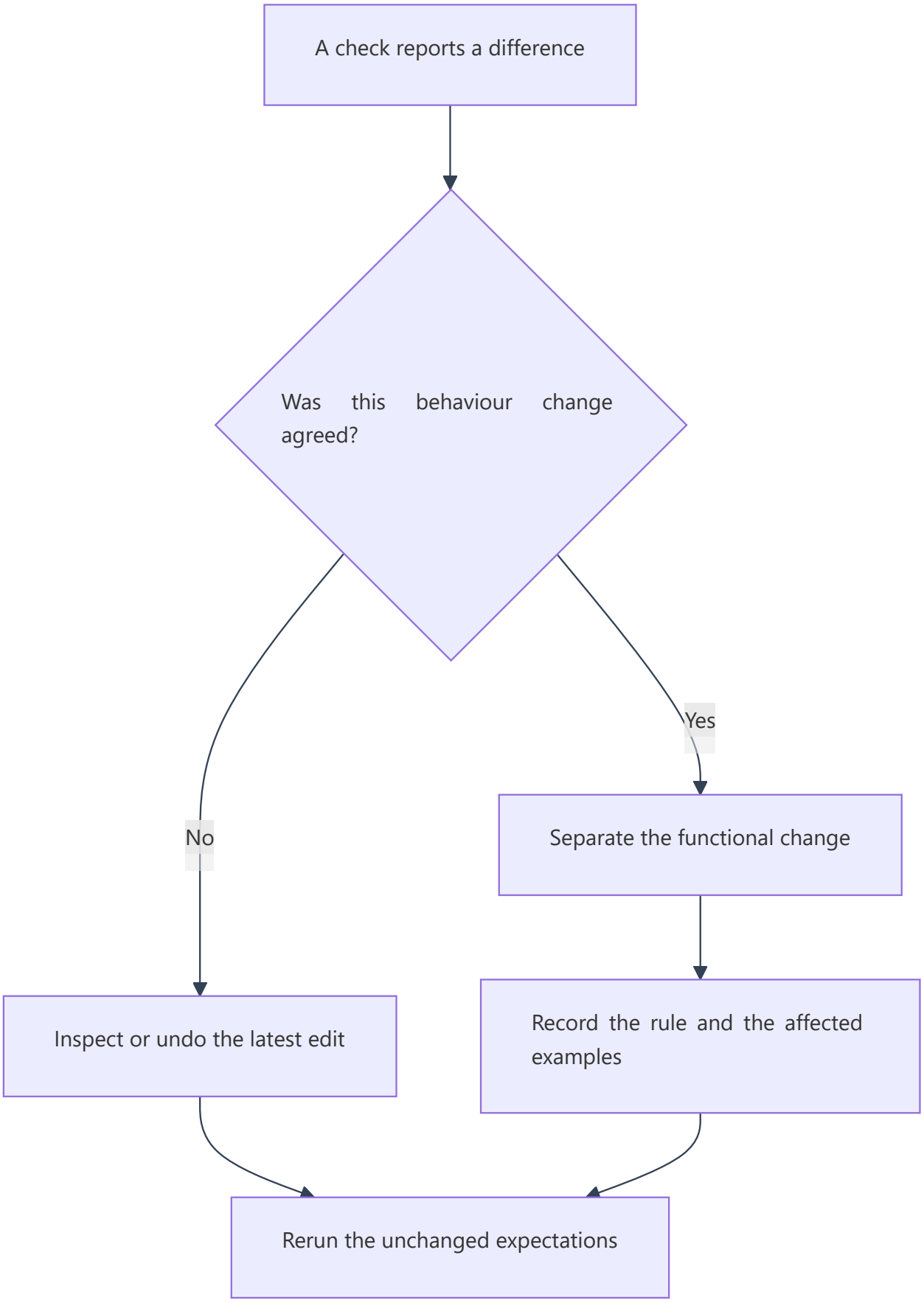


Figure 11.3. What to do when a check reports a difference. The diamond is a decision and the other boxes are actions. An agreement needs a specific rule: "clean up the report" is not one. Until the records office agrees a rule for unknown kinds or for loans inside their grace period, a difference at those boundaries is an unintended change.

Roll back, do not debug forward

If a test fails after a step, you have a precise failing change and a green state one step behind. Undo the step (`git restore` for uncommitted work, `git revert` for a commit) and redo it more carefully. Debugging forward from a failing refactoring, with the structure half-changed, is slower and riskier. The failing test is evidence about what the step got wrong: keep its name and message in your notes.

Making the functional correction separately

When the structure is clean, the questions that were parked can be answered. Suppose the records office confirms that only the kinds `"S"` and `"T"` exist and that anything else is a data error. That is a functional change, so it gets its own branch and commit. The code change is small: `policyFor` throws `IllegalArgumentException("Unknown borrower kind: …")` instead of returning the odd policy, and the `ANY_OTHER_KIND` constant disappears. The tests that pinned the suspected defect are **replaced deliberately**, in the same commit, with tests of the new rule, and the commit message says so. The differential test can no longer compare the unknown kinds with the original, so it is limited to the kinds that did not change. The companion contains this stage as a separate version of the code. The rule itself is illustrative: in a real project the records office decides it, not the developer.

11.7 Technical debt: say it with evidence

"Technical debt" is a metaphor, attributed to Ward Cunningham, for the accumulated cruft that makes a system harder to change [4]. The extra effort of every later change is the interest, and cleaning up is paying down the principal. The metaphor helps in conversations with people who do not read code. It misleads when it is used to excuse neglect, and Fowler also notes that the cost cannot be measured objectively, only estimated [4].

Treat a debt claim like any other claim: attach evidence. "This code is messy" is an opinion. "A change to the staff grace period needs two edits in one branch and breaks silently if one is missed, and there is no test" is a statement a colleague can check. After the refactoring the same change is one number, protected by 27 characterisation tests.

The companion contains two model records for this report: **ADR-011**, a decision record for pinning, isolating and refactoring it, and a **suspected-defect register** that lists each pinned oddity with its evidence and the question for the owner. They show the form. Your own records should describe your own code and decisions.

A short register entry

Field	Entry for the report
Item	Overdue report: fine rules duplicated, hidden clock and locale
Evidence	Grace number appears twice per branch; output varies by date and locale; no tests
Cost (interest)	Each fine change needs careful edits in several places and a manual check
Decision	Paid down: seam, tests and refactoring in five commits
Remaining	Three suspected defects, one unknown (Section 11.3)

Defects, doubts and preferences

Keep four kinds of statement apart in your notes: a **known defect** (confirmed wrong), a **suspected defect** (looks wrong, unconfirmed), **accepted behaviour** (confirmed as intended, however odd) and a **question** (unknown). Only the owner of the requirement can move an item between them. Mixing them is how a refactoring turns into an unreviewed behaviour change, and how a developer's taste turns into a defect report.

11.8 Reviewing an AI refactoring patch

Chapter 15 gathers the AI workflows that recur in this book. This activity is complete on its own, and the ready-made patch below is all you need.

	For this activity
Your goal	Decide whether a proposed refactoring patch preserves behaviour, using evidence rather than the patch's own claims
You start with	The refactored `OverdueReport` from Section 11.5, the characterisation suite and the differential test
What is in bounds	Evaluate and, if worthwhile, repair the patch; do not change the tests to make it pass
How to check	Trace the boundary case by hand; apply each claim to a scratch copy; run the suite; read each failing test
How to decide	A verdict on each claim, with the failing tests as evidence
No AI account?	The ready-made patch below is enough

Use this **fictional AI answer written by the author** for offline comparison:

> I tidied `build`. The overdue test is now `!row.due.isAfter(today)`, which reads better and is equivalent to `daysLate > 0`. I also choose the fine policy before the check, so the loop has one path. Unknown kinds fall back to the student policy, since students are the only borrowers who should ever be fined. All formatting is unchanged, so behaviour is preserved.

The patch changes the loop to:

```
long daysLate = ChronoUnit.DAYS.between(row.due, today);
FinePolicy policy = row.kind.equals("T") ? STAFF : STUDENT;
if (!row.due.isAfter(today)) {
    double fine = policy.fineFor(daysLate);
    // ... append the line, add to the total and the count, as before
}
```

There are three claims. **Trace the first one before running anything.** On the due date, `daysLate` is 0, so the original skips the row. What does `!row.due.isAfter(today)` do for a row due today? Then apply each claim to its own copy of the code and run the suite, so that you can see which tests belong to which claim. Finally, apply the whole patch. Answer: is "equivalent" true? What does choosing the policy first do to a loan that is returned, or not yet due, when its kind is `null`? Is treating unknown kinds as students a refactoring?

A repair is possible for the first two claims: keep the neater structure but restore the original test and the original lookup. The third is a functional change, so it needs a decision from the records office and its own commit with replaced tests. The patch's own summary was confident and wrong in all three places. Passing tests and your own trace are the evidence here. A live AI service is optional: give it synthetic data, keep its actual output, apply any patch to a scratch copy and treat its explanation as a claim to check.

11.9 Refactoring habits that mislead

Mistake	Consequence	Better action
Fix a suspected defect during a refactoring	One commit cannot be reviewed as "no behaviour change"	Pin it, record the question, fix it later in its own commit
Paste actual output into tests without reading it	The suite blesses bugs invisibly	Classify each result and comment the odd ones
Test only the happy path of a legacy method	The odd branches change unnoticed	Probe boundaries, unknown inputs, nulls and hidden dependencies
Make a large edit to create a seam	The seam edit is itself untested	Make the smallest mechanical edit and check it by eye
Leave a test changing the locale or `System.out` unrestored	Unrelated tests fail far away	Restore global state in `finally` or `@AfterEach`
Refactor several things per commit	Failures cannot be attributed; rollback loses good work	One refactoring per commit, tests between
Read "all tests pass" as "behaviour identical"	Untested behaviour was never protected	State what the tests cover and what they do not
Refactor everything that smells	Time spent where nothing will change	Pick smells that block a change you have to make
Change the shape of a public type other programs use	Hidden callers break	Keep signatures and formats that others depend on
Trust the seam edit because it compiled	The one untested edit could still have changed behaviour	Run the original beside the seamed version on a few rows, then pin
Accept "equivalent" for a boundary rewrite	`isAfter` and a day count differ exactly on the due date	Trace the equality case by hand, then run the tests

11.10 Try it yourself: pin, seam and refactor the report

You start from the Chapter 10 code plus the `legacy` package from the companion (`LoanRow`, the stage 1 `OverdueReport`, the characterisation suite, the differential test and the probe), and your team's repository. Steps B1 to B7 use the book's project; steps T1 to T4 use your team's project.

If your instructor demonstrates the task, they will run the legacy report on one day and then on the next (or with two dates through the seam) to show why the clock must be isolated, run the probe, and pin one surprising result live, with its comment. For guided practice, classify five pinned results from Section 11.3 as expected, surprising, suspected defect or unknown, and explain what you would ask the records office about each one.

What you must do: the book's project

- **B1. Observe and characterise.** Run the probe to check the enabling edit, then run the supplied suite against stage 1 (the original code with the date seam). Add three characterisation tests of your own for inputs the suite does not cover (for example, a borrower with many loans, two loans for the same item, or a due date far in the past). For each, record the actual result, classify it, and add a comment if it is

surprising.

- **B2. Refactor in small steps.** Starting from stage 1, apply the Extract Method and Extract Class refactorings of Section 11.5 (stages 2 and 3), one commit each, running the suite after each. If a test fails, roll back the step and redo it.
- **B3. Report your findings.** Write a short before-and-after rationale: what changed structurally, what you checked after each step, what did not change on purpose (and why), and which questions remain for the records office. Add one entry to a suspected-defect register in the form of the companion's model.

What you must do: your team's project

- **T1. Choose.** In your team's provided starter codebase, choose one method that has a smell and no tests, and that you have a reason to change. Write down what you observed and how sure you are, in the three-column form of Section 11.1.
- **T2. Pin.** Make the smallest edit that lets you test it, if it has hidden dependencies, and run the original beside the edited version on a few inputs. Write characterisation tests for at least six inputs, including a boundary and an odd case. Classify each result.
- **T3. Refactor.** Apply at least three small refactorings from this chapter, each in its own commit, with the tests run before each commit. Keep any behaviour change out of these commits.
- **T4. Rationale.** Write the before-and-after rationale: which structural change you made and why, what you protected, and which suspected defects and questions you found but did not fix. Ask a teammate to review the commits and record their feedback.

If you want more (the book's project)

- **B4. Break the boundary on purpose.** In a scratch copy, change the overdue test from `daysLate > 0` to `daysLate ≥ 0`. Predict which tests fail before you run them, read the failure messages, and restore the original.
- **B5. A separate functional change.** Make the correction of Section 11.6 on its own branch: unknown kinds are rejected. Replace the pinned tests deliberately, restrict the differential test to known kinds, and write the commit message to say what behaviour changed and why.
- **B6. AI activity.** Complete the Section 11.8 activity.
- **B7. Connect the report to the live loans.** The companion supplies a small bridge in bootstrap: `LoanSnapshotRows` joins the loan and reservation snapshots into report rows, and `OverdueReportService` reads one date from a `Clock` and one snapshot per call. Trace why the last permitted day is the day *before* a reservation's exclusive end, why a failure to read must not become "0 overdue", and why a loan without a matching reservation is an error. Run its tests and add one of your own.

What you should end up with is: B1 to B3 as described, with the commits or stage files; for your team's project, the observation notes, the characterisation tests, at least three refactoring commits, the rationale and the review record. A pull request is a good vehicle where your course workflow permits it.

You are done when these things hold:

- every characterisation test was run before its expectation was recorded;
- surprising and suspect results are commented;

- no refactoring commit changes behaviour;
- the tests pass after every commit;
- the seam edit is mechanical and has been checked by running the original beside it;
- questions are recorded and not silently fixed;
- signatures and formats that others depend on are unchanged.

Do not claim a review or CI run that has not happened. A passing suite is evidence for the behaviours it covers, not a proof of equivalence.

If you want more, extend the differential test to generate null items and borrowers, and say what a difference would mean. Keep this separate from the refactoring steps.

11.11 Questions to think with, transfer challenge and Quiz 2 revision

Concept questions

1. Why is a characterisation test not the same as a regression test, and what does a passing characterisation test tell you?
2. What is a seam, and why is the date parameter a better first edit than rewriting the method?
3. Why should a suspected defect be pinned rather than fixed during a refactoring?
4. What can a differential test show that a table of examples cannot, and what can it not show?

Applied diagnosis

5. A teammate refactors the report and says “all tests pass, so nothing changed”. Name two kinds of behaviour the suite does not cover, and say what you would do about each.
6. A developer extracts `FinePolicy` and, in the same commit, changes the staff cap from 5.00 to 6.00 “because it looked low”. Explain what is wrong with the commit and what the test run would and would not show.

7. The report runs at 22:30 UTC on a server whose clock zone is Baghdad. A loan was due on 21 October. Which date does the report use, what does it print for that loan, and which one object chooses the zone?

Transfer challenge

An analysis script in an ML model-serving tool formats prediction scores for a daily summary. It reads the current time, prints numbers in the default locale, and has three near-identical branches for different model types. It has no tests. Describe how you would find its seams, which six or seven inputs you would pin first, and which refactorings you would apply. Name one behaviour you would record as a question instead of fixing.

Quiz 2 revision pointers

Quiz 2 is held in Week 10, and the module descriptor says that it covers **dependency injection, TDD and test doubles, design patterns and layered architecture (Weeks 5–9)**. Revise Chapters 6–10 rather than treating this chapter's legacy lab as quiz content.

Chapter	Revision focus
6	Interfaces and ISP, dependency direction, constructor injection, composition root, what a container automates
7	Red–green–refactor, test anatomy, stubs, fakes, spies and mocks, controlled time, weak tests
8	Factory Method, Abstract Factory, Builder, Adapter, Decorator, and choosing the simpler alternative
9	Strategy, State, Observer and Command, partial failure, and choosing a pattern by the variation it handles
10	Layers and dependency direction, ports and adapters, repositories, entities and value objects, package cohesion, coupling and stability

Use the module leader's confirmed quiz guidance if it provides more detail. These pointers do not add assessment rules or predict particular questions.

What to take away, and what comes next

Changing code you do not understand is a sequence, not a leap: observe it, make the smallest edit that lets you test it, pin its behaviour with tests that record what it does, reshape the structure in small steps with a check and a way back after each, and keep every correction of behaviour apart from every improvement of structure. Smells say where to look, debt claims need evidence, and a green suite is evidence for the behaviours it covers.

The overdue report now has a seam that was checked against the original, a pinned suite, a small policy class and four recorded oddities waiting for the records office. Chapter 12 turns to measuring progress and flow, and introduces the visiting-researcher change request that the rest of the project must absorb.

Sources and further reading

Online sources 1–12 were opened on 13 October 2026. The report, its tests and the stages are original teaching material.

1. Fowler, M., **Definition of Refactoring**. Refactoring as an internal change that leaves external behaviour the same.

2. Fowler, M., **Refactoring catalogue**. Entries for Extract Function (alias Extract Method), Move Function (alias Move Method), Extract Class, Rename Variable and Change Function Declaration (alias Add Parameter); the first-edition names appear as aliases.

3. Feathers, M., **Characterization Testing**. Tests that describe what the code does rather than what it should do; the write, run, record loop.

4. Fowler, M., **Technical Debt**. The metaphor attributed to Ward Cunningham, interest and principal, and the point that the cost cannot be measured objectively.

5. Fowler, M., **Code Smell**. A smell as a surface indication that usually corresponds to a deeper problem; smells are not always problems.

6. Fowler, M., **Legacy Seam**, 2024. Defines a legacy seam as a place where behaviour can be altered without editing in that place, and an enabling point as the place where the choice is made; credits Feathers.

7. Feathers, M., **Testing Effectively With Legacy Code**, InformIT, 2005. A chapter from *Working Effectively with Legacy Code* on why existing code is hard to test.

8. Oracle, **Formatter (Java SE 17)**. The default locale when none is given and the localised decimal separator for `%f`.

9. Oracle, **Clock (Java SE 17)**. `Clock.fixed` and `Clock.system(zone)`: the instant-to-date conversion uses the clock's zone. `LocalDate.now(Clock)` is documented in **LocalDate** as querying the supplied clock; the zone behaviour is also shown by the tests in this chapter.

10. Oracle, **ChronoUnit (Java SE 17)**. `between` returns whole days, with the end exclusive and a negative result when the end is earlier.

11. Oracle, **Period (Java SE 17)**. `getDays` returns the days field, stored separately from years and months.

12. JUnit, **User Guide 5.13.4**. Parameterized tests and `@CsvSource`.

13. Martin, R. C., *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall, 2002. The module's required text; the Week 10 reading is its refactoring chapter. This chapter treats refactoring in Sections 11.5 and 11.6.

14. Feathers, M., *Working Effectively with Legacy Code.* Prentice Hall, 2004. The module's recommended reading for Week 10 (selected chapters). The working definition of legacy code and of a seam are Feathers's, and the characterisation-testing description was checked against source 3. This chapter treats legacy code, characterisation tests and seams in Sections 11.1, 11.3 and 11.4.

15. Fowler, M. (2018), *Refactoring: Improving the Design of Existing Code*, 2nd ed., Addison-Wesley. A recommended text, also cited in Chapter 5; its catalogue is online at source 2. Not reopened for this chapter.

Chapter 12: Measuring Progress, Flow and Technical Health

> *Empirical delivery metrics (lead time, cycle time, throughput), repository analysis, and planning a changed requirement.*

OO Development and Agile Practices
Designing, Testing and Evolving Software with AI Assistance

The lending team has completed several changes, and yet the line that shows remaining work has gone up. One item spent two days waiting for a directory decision. Another was removed from scope. One team member says that more commits prove the team is getting faster; an AI summary says the blocked item proves the architecture is poor. Who is right?

Before anyone can answer, a new request arrives. The equipment officer would like visiting researchers to reserve selected equipment after sponsor approval, with a shorter loan limit. Can the team finish that next iteration? Which existing rules will it touch? And what evidence would justify starting at once instead of first resolving an uncertainty?

A metric cannot make these decisions for us. What it can do is make part of the situation visible, provided its unit, its boundary and its limits are clear. This chapter follows a small issue history, checks the arithmetic behind it and uses it to plan the next change; the history is **synthetic teaching data**, not recovered history from the earlier chapters and not evidence about any real student team. You will learn to define a measurement boundary and to calculate lead time, cycle time, throughput, work in progress and work-item age; to tell velocity from item throughput and to see why neither ranks anyone's contribution; and to reconcile burn-down and burn-up charts with additions, removals and completed work. You will frame a forecast as a conditional statement from a small body of evidence and say what is missing, and you will assess a claim about technical health or debt without confusing activity with quality or with cause. Finally you will prepare acceptance examples, an impact analysis and an ordered response to a changed requirement.

This is Week 11, supporting LO5 and LO6. The required reading is the descriptor's Martin (2002), planning review; the recommended reading is selected chapters of *Accelerate* [7–8]. The required practical activity includes repository-metrics analysis. The requirement change is introduced in this chapter, implemented and verified in Chapter 13 for Milestone 4, and evaluated during handover in Chapter 14. The module leader releases your team's assessed variant, and the book's worked change does not replace it.

You need the backlog and Definition of Done from Chapters 3 and 4, the tested boundaries and structure of Chapter 10, and Chapter 11's distinction between refactoring, suspected defects and changed behaviour. The running project stays at Chapter 11's verified Java version: this chapter adds an isolated metrics exercise and planning records, and it does not implement the visiting-researcher feature. The companion contains all the calculator code, data, commands, worked answers and templates, so there is no need to copy another full application for this process chapter.

Sections 12.1 to 12.7 are the core, with the measurement contract and the worked change specification as the parts to keep. What you must do is to reproduce the synthetic calculations, inspect actual repository evidence, challenge the AI summary, and keep the version from before the change together with a change-impact note. The supplied Java calculator and the repository extraction method are working knowledge. Richer visualisation, larger cohorts and probabilistic forecasting, once you have suitable data, are for those who want more.

12.1 What decision needs a measure?

Start with a question that could change what you do

"Collect more metrics" is a weak objective. "Should we finish the two open items before pulling the sponsor-approval story?" points to a decision. We need to know which items remain, how long they have been open, whether they are blocked and how comparable they are with past work. A count of commits cannot answer all of that.

For each measure, state the question, population, unit, time window and intended response. Keep a note of the source and extraction date. If a number cannot influence a decision or test an assumption, consider whether collecting it is useful. For a small student project, a clear table often teaches more than a dashboard full of unexplained indicators.

The flow vocabulary below follows the Kanban Guide's emphasis on explicit started and finished points [1]. Our lead-time boundary is an additional local choice. Other teams may choose different boundaries, so compare definitions before comparing numbers.

Contract field	Definition for the book's synthetic history
Unit	One uniquely identified backlog item; IDs A–H
Accepted	Item admitted to the tracked scope
Started	First transition into implementation
Done	Accepted against the team's Definition of Done, including its required checks and review
Removed	Explicitly taken out of scope without completion
Reporting interval	`[2026-10-01T00:00:00Z, 2026-10-11T00:00:00Z)`
Duration	Elapsed 24-hour days; waiting and weekends remain included
Snapshot	State immediately after events at the stated timestamp
Completion cohort	Items whose Done event falls within the reporting interval
Exclusions	No reopened, split, merged or re-estimated items in this deliberately small dataset

The half-open interval includes completions at the starting instant and excludes completions at the ending instant. Adjacent windows then cannot count the same completion twice. A snapshot is a different question: a completion exactly at its timestamp has already left work in progress. The companion tests both conventions. These are reporting choices, not new reservation-period rules.

Keeping data and interpretation apart

Here is the complete issue history in day-of-month form. Every date is October 2026 at 00:00 UTC. A dash means that the event has not occurred in this dataset, not a duration of zero.

ID	Illustrative work	Points	Accepted	Started	Done	Removed
A	Reservation boundary examples	3	01	01	03	-
B	Directory mapping clarification	5	01	02	06	-
C	Confirmation wording	2	01	03	04	-
D	Command replay protection examples	3	01	05	08	-
E	Report failure investigation	5	01	07	-	-
F	Optional export format	2	01	-	-	08
G	Snapshot consistency checks	3	04	06	10	-
H	Review-notes improvement	2	06	08	-	-

B is additionally recorded as blocked from 03 October to 05 October while a mapping question is answered. This event is supplied as synthetic context. It is not inferred from inactivity in Git. G and H are scope additions. F is a removal. The labels describe an imagined iteration; they do not assert that earlier manuscript files were actually created on these dates.

A sound observation is “B spent four elapsed days between Started and Done.” A possible explanation is “waiting for the mapping decision contributed to B’s delay.” A stronger claim, such as “the directory adapter caused all delays,” needs evidence this table does not contain.

12.2 Estimates revisited: what they can and cannot tell us

An estimate is a planning judgement made with incomplete knowledge. It is useful when it exposes uncertainty or supports a choice between smaller and larger slices. It becomes misleading when the team treats it as a measured physical property of the work.

The points in this dataset are a local relative scale. They are neither hours nor a universal exchange rate between teams. B’s five points and C’s two points express the imagined team’s relative judgement before delivery. B taking four days does not establish that one point equals 0.8 days; that duration includes waiting, and another item may have different risks.

For the approval change, start with a concrete example: a recognised visitor requests CAM1 for two days with matching approval. Compare that slice with the failure case where the approval register is unavailable. They share code, but the second raises a distinct delivery risk. Splitting work by useful behaviour makes uncertainty easier to discuss than dividing it mechanically into “write classes,” “write tests” and “write documentation.”

When estimates differ, ask what each person assumed. One may expect an existing approval file; another may expect a new sponsor-facing website. The solution is to clarify scope, not average incompatible assumptions. Our worked change uses a small local approval register and excludes an approval-management interface.

Capacity also matters. A week with a midterm, review duties or an absent teammate is not interchangeable with an earlier week. Record the constraint without translating people into interchangeable point-producing units. A small time-bounded investigation can reduce uncertainty, but it should have an output: for example, a sample approval record and a checked parsing/failure contract.

Do not rewrite the old point estimates after seeing the duration merely to make the history look consistent. If a team genuinely revises an unfinished estimate, preserve the earlier value and record the event so that a chart can distinguish re-estimation from newly added scope. Our calculator intentionally omits re-estimation; that limitation is explicit rather than silently assigning new values to old dates.

12.3 Velocity and its limits

Many teams that use points quote a number called velocity, so let us define it and then see how far it can be trusted. **Velocity** here means the sum of estimates attached to items completed within a defined iteration. The five finished items A, B, C, D and G total:

`3 + 5 + 2 + 3 + 3 = 16 points`.

Throughput counts completed items over a stated period: five items in this ten-day interval. Sixteen points and five items describe the same completions with different units. Calling 16 "throughput" without specifying a point-based measure would confuse the comparison. The calculator therefore prints both count and completed points with explicit labels.

E and H contribute no completed points yet. Calling E "80% done" does not add four points to this velocity calculation. It may be useful to discuss progress on E, but that conversation is not a reason to redefine Done. F also contributes nothing: removing an optional item can be a sensible scope decision without becoming delivery.

Velocity can help one team inspect its own planning history when its estimation scale, Definition of Done and work mix are sufficiently consistent. It cannot fairly rank teams that estimate differently. It also cannot reveal an individual's contribution: reviewing a colleague's difficult change may increase team completion without producing a separate story or commit under the reviewer's name. Scrum allows contextual forecasting practices; it does not turn a burn chart into a substitute for inspecting actual results [2].

Making forecasts conditional

There are two remaining items totalling seven points at the snapshot. Dividing seven by the observed 16 points gives 0.4375 iterations. That arithmetic does **not** establish a delivery date. It assumes comparable work, capacity, dependencies, scope and completion behaviour. One observed iteration offers little evidence about variation, and unfinished work may be precisely the work with unresolved risk.

A defensible planning statement is: "We have completed five items in ten elapsed days in this synthetic example. E and H remain open. We will inspect their blockers and size the approval slice before committing to a date." If a later comparable history supports two to three items per five-day window, two remaining items might fit one such window under unchanged conditions. That is a scenario, not a calibrated probability or guarantee.

Preserve forecasts as dated predictions and compare them with later outcomes. If a forecast misses, inspect assumptions and changed scope before blaming effort. Do not fit a confident percentile or statistical model to this tiny classroom sample. Advanced forecasting is useful only when its inputs and interpretation justify the additional machinery.

12.4 Burn-down and burn-up: reading the lines

A burn-down displays remaining work. A burn-up displays completed work, normally alongside the current scope total. In this dataset, points are held fixed and removed items were never Done, so at each snapshot:

`remaining points = current scope points − completed points`.

Current scope includes completed items and accepted unfinished items; it excludes F after removal. The initial scope is 20 points. Adding G and H raises it to 25; removing F reduces it to 23. At the final snapshot, 16 points are complete and seven remain. The scope history must stay visible if we want to explain the line.

Snapshot day	Scope points	Completed points	Remaining points	WIP items
01	20	0	20	1
02	20	0	20	2
03	20	3	17	2
04	23	5	18	1
05	23	5	18	2
06	25	10	15	2
07	25	10	15	3
08	23	13	10	3
09	23	13	10	3
10	23	16	7	2
11	23	16	7	2

On day 04, the remaining line rises from 17 to 18 despite C’ s two-point completion: three points arrived as G. On day 08, the line falls by five points, but only three came from D’ s completion; two came from removing F. Without the scope events, both changes are easy to misread.

Figure 12.1 plots the eleven daily snapshots from the table. The upper line is current scope; the lower line is cumulative completion. It is a plot of the observed snapshots, not a smoothed model of unobserved activity.

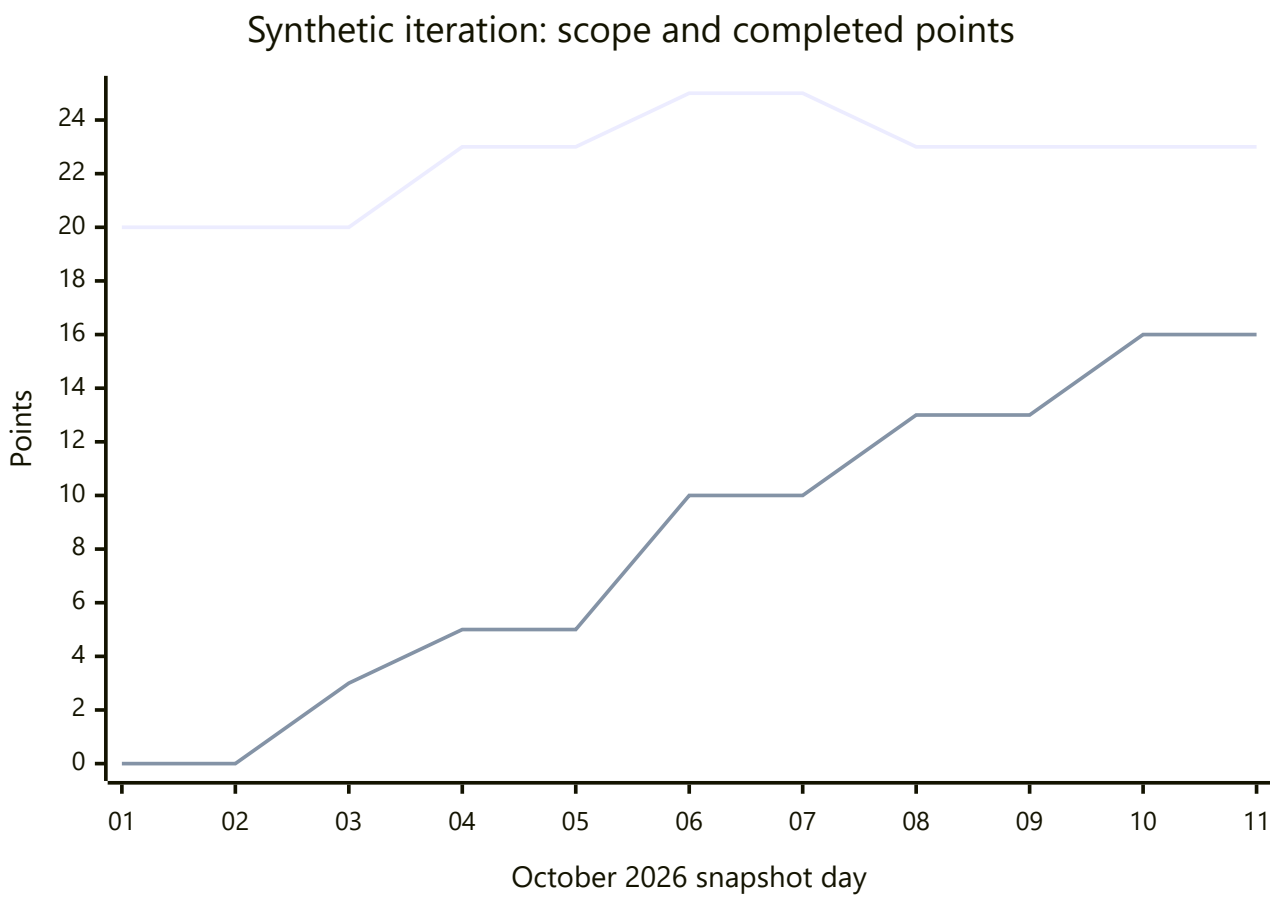


Figure 12.1. Synthetic iteration, one point per daily snapshot. Upper line: current scope (20, then 23 when G is added on day 04, 25 when H is added on day 06, 23 when F is removed on day 08). Lower line: cumulative completed points (3 on day 03, 5 on day 04, 10 on day 06, 13 on day 08, 16 on day 10). The gap between the lines is the remaining work, seven points at the end. The chart type has no legend, so the lines are identified here.

An ideal diagonal burn-down can be a reference for a fixed initial scope. It is not evidence that real work should finish in equal daily increments. Review, integration and acceptance can produce uneven completions. Do not move Done timestamps to make a curve attractive. Ask whether the team can explain the change and whether the remaining work still serves the goal.

12.5 How long do things take? Lead time, cycle time, throughput, WIP

For this history, lead time runs from Accepted to Done; cycle time runs from Started to Done. The difference is the recorded wait before starting. Neither measures hands-on effort. The following calculations use the completion cohort, not every row in the backlog.

Finished item	Lead days	Cycle days	Wait before start
A	2	2	0
B	5	4	1
C	3	1	2
D	7	3	4
G	6	4	2
Mean	4.6	2.8	1.8

The cycle-time median is three days: sort `1, 2, 3, 4, 4` and take the middle value. B's two blocked days remain within its four-day cycle time. Subtracting them would answer a different question. Even the remaining two days would not establish two days of active coding, because other unrecorded waiting could remain.

At the final snapshot, E and H are started but unfinished, so WIP is two. Their ages are four and three elapsed days. They have no completed cycle time yet. Excluding them from the mean is correct for a completed-item statistic, but forgetting them in the interpretation would create a misleadingly favourable account. Report the completed cohort alongside open-item age.

Throughput for `[01,06)` is two items, A and C. For `[06,11)` it is three, B, D and G. These are five-day windows, not claims about official five-day Sprints. B belongs only to the second window because it completes exactly at that window's starting instant.

Equal throughput can conceal different delay. Imagine two separate five-day periods, each finishing two items. In one, both items started one day before completion; in the other, both started seven days before completion and carried in from earlier work. Both periods have throughput two, while their completed cycle times are one and seven days. Counting completions does not reveal how long the recipients waited.

Using real repository events carefully

The required lab supplements this dataset with real issue or pull-request history. The companion pins **Repository Metrics Method RM-12 v1.0**, using exact event timestamps and the supplied Java 17 calculator. In the PR route, Started means PR opened, Done means merged, and the unit is a PR. Call the resulting duration **PR open-to-merge elapsed time**. It is not issue cycle time, developer effort, deployment lead time or time since coding began.

Choose the population before looking for an attractive result. For throughput, enumerate every qualifying merge in the chosen repository, target branch and interval, including all result pages. A convenient sample of five PRs supports a sample-duration description, not the repository's throughput. Include PRs already open at the start, and record open PRs at the final snapshot if reporting WIP. Avoid mixing code changes with automated dependency updates without stating the chosen inclusion rule.

Keep URLs or IDs, original timestamps and the extraction date in the evidence record. Local Git can cross-check commit identity, but author and committer dates are not reliable substitutes for PR-open or server merge events. Squashing, rebasing and workflow choices change the relationship. If only Git history is available, report what it actually measures and obtain the required issue/PR timestamps from the team or an instructor-provided real export. Synthetic rehearsal alone does not complete this part of the lab.

12.6 Technical health, and how metrics get misused

A useful technical-health discussion combines a concrete risk with evidence and a proposed response. "There are 159 passing tests in the Chapter 11 version" records a build result. It does not say every rule is correct, that every important boundary has a test or that more tests would automatically produce a better design.

Chapter 11 supplies stronger evidence for one narrow claim: the selected tests detect a deliberately changed equality boundary. That is relevant when deciding whether a report refactoring is protected. It says nothing about sponsor approval, which does not exist yet. Similarly, a high line-coverage percentage can identify unexecuted areas, but it cannot tell us whether assertions distinguish correct and incorrect outcomes.

For debt, name the expected future cost. The report's duplicated selection rule had a plausible maintenance cost; the refactoring removed that duplication. We have not measured hours saved or production defects avoided. The sequential snapshot join may become a problem if concurrency enters scope, but a feature deliberately excluded from a teaching application is not automatically debt. Fowler's debt discussion is useful for examining why a design decision was made rather than labelling all untidy code alike [3].

Claim	Evidence to seek	Cautious response
Approval changes will touch many unrelated classes	Trace actual category, directory, wiring and reservation references	Review the boundary before estimating the implementation
Reports are unreliable	Reproduce a specific wrong or unavailable report	Distinguish SD-11 policy uncertainty from an implementation failure
Builds are slowing feedback	Timed comparable builds, environment and wait causes	Try one targeted improvement and observe later runs
A package is difficult to change	Dependency evidence and affected responsibilities	Refer to Chapter 10's package-design treatment; do not use instability as a quality score

Goodhart's Law is a useful caution: pressure to maximise a proxy can encourage behaviour that weakens its connection to the goal. Here, a target for commit counts could encourage unnecessary splitting; a target for velocity could encourage inflated estimates. Neither improves the borrower's experience by itself. Research on developer productivity likewise warns against reducing it to a single activity measure [4].

Use measures for team learning and system improvement, not individual rankings. A review, a careful deletion or help given to another person may be valuable without increasing the chosen count. Pair quantitative observations with a short account of what happened. Investigate suspicious numbers without assuming misconduct, and avoid presenting a tiny correlation as a causal explanation.

12.7 A changed requirement, and what it touches

The first six sections taught how to read evidence about a team' s work. This section uses the same habits on a new request: before estimating a change, say what it must do, what it touches and what evidence will show it works. The request is the visiting-researcher change from the opening problem, and it is the change that Chapters 13 and 14 implement and review.

CR-12: visiting-researcher reservations

The following is a **precise book exercise contract**, chosen for teaching. It is not an approved institutional borrowing policy. Teams apply their own Week 11 released requirement variant.

> A recognised visiting researcher may request CAM1 or CAM2 for one to three calendar days when a trusted sponsor-approval record matches that borrower, item and exact loan period and is still valid at request time. Missing, rejected, mismatched, expired or unavailable approval must not create a reservation or send a confirmation. Student and staff behaviour must remain unchanged.

The full contract and acceptance matrix are in the companion. For this small change, approval is supplied through a local read-only data register, not a new sponsor website. It will be a genuine file-reading boundary in Chapter 13, checked with temporary files as well as controlled test doubles. Training a model, predicting trust and calling a live approval service are outside this exercise.

This data-dependent component is distinct from asking an AI assistant to explain metrics. It gives the running project another concrete contract and failure boundary; it does not claim to cover every aspect of external AI or data integration.

Tracing the change through the current design

`BorrowerCategory` currently contains STUDENT and STAFF. The legacy directory recognises `S` and `T`. The coordinator requires a policy for every enum value. Adding a category therefore affects more than one enum: directory mapping, composition, exhaustive test examples and the new visitor path must agree.

`BorrowingPolicy.mayRequest(count, days)` deliberately answers only a count/duration question. It has no borrower identity, item, approval record or clock. Passing approval data through hidden global state to avoid changing the application boundary would defeat that contract. Keep the simple duration decision separate from obtaining and validating approval for a particular request.

The current coordinator counts every retained reservation for a borrower, despite the local variable name `activeCount`. Returning a loan does not remove its reservation. This change preserves that existing count convention; the companion records it as a separate policy question. Do not silently replace it with an active-loan count while adding visitors.

Area	Planned impact	Evidence needed in Chapter 13
Borrower and directory	Add a visitor category and explicit external mapping	Known visitor, unknown code and existing category cases
Request orchestration	Validate a matching approval before a visitor reservation	Success; denied, expired, mismatched and unavailable outcomes
Data boundary	Parse the read-only approval register	Real temporary-file parsing and failure checks
Policies and wiring	Add the three-day limit and all required collaborators	Boundary tests and composition that includes every category
Existing reservation/loan/report behaviour	Retain old results and SD-11 status	Existing regression suite plus targeted old-path examples
Delivery configuration	One visitor-enablement flag	Both settings, failure behaviour and removal plan

A missing approval is an expected rejection; an unreadable or malformed register is an operational failure. Both prevent a new visitor reservation, but the response should not falsely say that a sponsor rejected something when no valid decision could be read. Keep these outcomes visible in the application contract.

Planning the smallest checked response

First keep the passing Chapter 11 version safe. Next record the requirement, accepted examples, assumptions and affected files. Then order a thin valid visitor path, the genuine register boundary and failure paths, old-path regression, and delivery/demo evidence. Each item should have an observable exit condition.

Do not copy the synthetic history’s points onto these new tasks. Estimate with the team’s own knowledge and capacity. Resolve the approval-record format early because it affects both orchestration and integration tests. Plan a short review after the first vertical slice; use its evidence to revise the rest of the forecast.

Chapter 12 finishes with a plan and a comparison point you have kept. Chapter 13 must still implement the feature, test the actual boundary and demonstrate a pipeline run. A design note is not a delivered capability, and a green build from before the change is not evidence that the new rule works.

12.8 Checking an AI’s explanation, not its confidence

This activity is for the book’s project. The text below is **fictional AI output written by the author** for offline review. It is not a recorded service response.

You start with the eight-row history, the blocked interval, the measurement contract and the chart table. Your task is to verify every arithmetic or causal claim independently: explain the data and identify questions, but do not change timestamps, estimates or requirements to fit the explanation.

> The team delivered six items and 18 points because F left the backlog. Mean cycle time is two days after subtracting B’s blocked days. The rise on 4 October proves productivity declined, while the fall on 8 October proves five points were delivered. The legacy architecture caused the waiting. At 16 points per iteration, the remaining seven points will definitely finish in 0.44 iterations. Rank team members by commits to remove the bottleneck.

Mark each claim as supported, contradicted or not established. Recalculate the completed cohort and show why even subtracting two blocked days would not produce the claimed mean. Explain what additional evidence would be needed for a causal conclusion. Produce a corrected paragraph with one actionable next step and at least one limitation. The companion supplies the reasoning path; another AI opinion is not verification.

If you like, you can ask a live assistant for the same bounded analysis, using synthetic or appropriately authorised project data, as long as you follow the module leader's assessment rules. A live service is unnecessary for the lab.

12.9 Measurement habits that mislead

Mistake	Consequence	Correction
Treat removed work as Done	Inflates delivery and hides scope decisions	Retain a separate removal event
Divide by all accepted items when averaging completed duration	Treats unfinished cases as if their duration were known	Use the completed cohort and report open ages separately
Drop waiting from cycle time without saying so	Changes the question and makes delay disappear	Keep elapsed time; define any additional measure separately
Infer work-item completion from commits	Counts workflow traces as delivered value	Match the unit and event to the evidence
Use an incomplete PR sample as repository throughput	Understates or selectively counts completions	Enumerate the full qualifying interval or label a sample
Convert one velocity observation into a deadline	Hides variation, new scope and uncertainty	State conditional assumptions and revisit with new evidence
Add visitors by defaulting unknown categories to students	Bypasses the explicit new rule	Preserve rejection of unknown categories and test mapping

12.10 Try it yourself: measure, interpret and prepare the change

You start from the synthetic issue history and the blocked-event note; the Java calculator and its checks; the RM-12 v1.0 repository method; the CR-12 contract and acceptance examples; the backlog and impact templates; and the Chapter 11 version of the code.

If your instructor demonstrates the task, they will calculate B's lead and cycle time, explain day 08's five-point drop, and trace the count/duration policy boundary in the current code. For guided practice, pairs reconcile the final scope and trace the completion that falls exactly on an interval edge. Then complete both the book's project tasks and your team's project tasks below.

1. **The book's project: calculate.** Reproduce the duration table, completed count/points, final WIP and ages. Run the Java exercise and explain one boundary check.
2. **The book's project: interpret.** Explain the scope additions and removal. Write a cautious forecast paragraph and reject the unsupported AI claims.

3. **Your team’ s project: the required repository lab.** Apply RM-12 to real PR or issue timestamps from the team’ s repository, or an instructor-supplied real repository export. Produce an auditable duration result and, if the population is complete, throughput for a fixed interval. State missing evidence honestly. The synthetic table is rehearsal, not a replacement.
4. **The book’ s project: keep and plan.** Keep the version from before the change; review CR-12’ s acceptance matrix; identify code/data/test boundaries; order a small backlog for Chapter 13. Do not implement the feature as part of this required work.
5. **Your team’ s project: transfer.** Apply the same impact method to the released team variant. Record one dependency or policy question and how its answer would change the plan. Another domain need not copy lending classes.

What you should end up with is: your calculations; the source and measurement contract; a cautious interpretation; the reviewed AI claims; a reference to the version you kept from before the change; the updated backlog; and the change-impact note. Keep actual repository evidence separate from the synthetic sample. Obtain real peer feedback where possible and label a solo review accurately.

You are done when the arithmetic reconciles, the event boundaries and population are explicit, open items and removals stay visible, repository results trace back to real sources, no individual is ranked, new behaviour has acceptance examples, and existing behaviour and unresolved defects have a plan for being preserved. These are learning checks, not new official assessment weights.

If you want more, compare two complete repository intervals using the same inclusion rule, inspect the longest open item, or add support for reopened items with a clearly revised data model. Do not silently extend the supplied calculator’ s assumptions.

12.11 Questions to think with, and a transfer challenge

Concept questions

1. How do lead time and cycle time differ under this chapter’ s contract? Why must the boundaries travel with the reported number?
2. How can a burn-down fall without an equivalent amount of work being delivered?
3. Why can equal throughput coexist with different cycle times and different customer experiences?
4. Why does a passing regression suite support preserving old behaviour without proving the new approval requirement?

Applied diagnosis

5. A dashboard calculates average cycle time only for items both started and finished this week, then claims it represents every item finished this week. What is wrong with the cohort, and which carried-in work could disappear?

6. A developer adds VISITOR to the enum and passes sponsor approval through a static Boolean read by `BorrowingPolicy`. Identify the contract, test isolation and request-matching risks; propose a clearer boundary without implementing it.

Transfer challenge

An ML model-serving team receives a change: only approved model versions may be served to external callers. Its dashboard counts successful builds and reports that "all requested releases are complete." Define a useful unit and Done event; identify the missing approval and serving evidence; plan a narrow change with an unavailable-registry case. Explain why model accuracy, build success and release approval are different observations. Use synthetic model identifiers; no model training is required.

What to take away, and what comes next

A measurement becomes useful when its definition supports a concrete decision. The synthetic history shows why scope, elapsed time, unfinished work and completion counts must be read together. Repository evidence adds traceability but does not reveal every cause of delay or every valuable contribution.

The application remains at the version from before the change. CR-12 and the ordered backlog now give Chapter 13 a precise basis for its implementation, genuine boundary tests and delivery evidence.

Sources and further reading

Sources checked 13 October 2026. Tables, numerical examples, scenarios, code, contracts and model answers are original teaching material. No student or institutional performance data is represented here.

1. Kanban Guides. **The Kanban Guide, May 2025**. Started/finished boundaries and the four flow measures. The worked lead-time definition is local to this chapter.
2. Schwaber, K. and Sutherland, J. **The Scrum Guide, November 2020**. Empirical planning, Done and contextual forecasting practices.
3. Fowler, M. **Technical Debt Quadrant**, 2009. Context for reasoning about debt and design decisions.
4. Forsgren, N., Storey, M.-A., Maddila, C., Zimmermann, T., Houck, B. and Butler, J. **The SPACE of Developer Productivity: There' s more to it than you think**, *ACM Queue*, 19(1), 2021. Developer productivity cannot be reduced to one activity measure. The Goodhart caution is used here as a general interpretive warning, not a numerical law or a claim that gaming occurred.
5. Git Project. **git-log documentation, version 2.51.1**. Commit traversal and formatting used for the optional local provenance cross-check.
6. Oracle. **Duration: Java SE 17**. The calculator uses elapsed seconds and nanoseconds; its day unit is 86,400 seconds, not a business-day calendar.
7. Martin, R. C. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002. Required Week 11 planning review, as assigned by the supplied module descriptor; consult Moodle for the precise reading.
8. Forsgren, N., Humble, J. and Kim, G. *Accelerate*. IT Revolution, 2018. Descriptor-recommended selected reading. This chapter does not claim access to the full book or attribute causal findings to its small synthetic dataset.

Chapter 13: Integration, Delivery and Feedback

> *Milestone 4, contract testing, integration suites, and feature flags (VisitorGate).*

OO Development and Agile Practices
Designing, Testing and Evolving Software with AI Assistance

The visiting-researcher rule passes every unit test, and the demonstration works on the developer's laptop. Then comes the morning of the Milestone 4 demonstration, and three things go wrong. The sponsor register exported by the office has a stray column, so a visitor request ends in an exception. The borrower directory takes eleven seconds to answer, and the officer's screen freezes. And the new flag, switched on for the demonstration, is off in the build: the build is green and the demonstration fails.

Not one of these is a mistake in the rule. Each lives at a **boundary** between our code and something we do not control: a file, a network service, a setting. A unit test that builds its own objects by hand cannot see a boundary, because the boundary is exactly what the test has replaced. This chapter is about checking the places where our code meets something else, putting those checks into a pipeline that runs them in the right order, and releasing a change in a way that can be undone.

The worked change is CR-12 from Chapter 12: visiting researchers, with sponsor approval, behind one flag. It is a **teaching contract**, not an institutional policy, and your team applies its own released variant. You will start by choosing unit, integration and end-to-end checks for different risks, saying what each of them cannot show, and by writing a contract test that several implementations must pass. You will implement the changed requirement with regression protection and a clear separation between expected rejections and operational failures, then configure a modest delivery pipeline, run it and learn to read a failure in it. You will distinguish continuous delivery from continuous deployment and label a simulated deployment honestly. And you will evaluate a feature flag and a rollback plan, including what rollback does not undo.

This is Week 12, supporting LO3 and LO6 and Milestone 4 (Sprint 3: the change absorbed, with integration tests and CI/CD). The required reading is Martin (2002), *Testing* (review) [11]; the recommended reading is selected chapters of Humble and Farley, *Continuous Delivery* [12]. The Week 12 lab is an automated build, test and integration pipeline. The chapter creates no new deadlines, weights or marking rules, and the approved Milestone 4 instructions decide what you submit.

You start from the Chapter 11 project and Chapter 12's change specification CR-12, its acceptance matrix (examples AC1 to AC18) and its ordered backlog. This chapter implements the change. The Chapter 11 listings are not repeated, and the companion lists only the files that are new or changed, along with the tests, the scripts, the model answers and teaching notes.

Sections 13.1 to 13.5, 13.7 and 13.8 are the core reading. Section 13.6 (feature flags, configuration and rollback) is working knowledge: learn to run one flag safely; you do not build a flag platform. What you must do is steps B1 to B4 and T1 to T5 of the practical activity in Section 13.11. Steps B1 to B4 run on the book's project and are running, tracing and diagnosis, because the finished version is supplied and there is nothing to rebuild; steps T1 to T5 run on your team's project and produce your Milestone 4 evidence. If you want more, there are steps B5 to B7 (deliver and roll back a simulated release, do the AI activity, break a boundary on purpose), the questions and the transfer challenge in Section 13.12, and the extension at the end of Section 13.11.

13.1 What can go wrong at a boundary? Risks and test scope

Start from the risk, then pick the check

"Which kind of test should I write?" is the wrong first question. Ask what could go wrong in this change, where it would go wrong, and which check would notice. Then choose the cheapest check that can notice it. The table below does that for CR-12.

What could go wrong	Where	The cheapest check that can notice	What that check cannot show
The validity window is off by one at its edges	Approval record	Unit test with a controlled instant	That the register file is read correctly
A visitor is served without a matching approval	Visitor gate and service	Acceptance examples with a controlled approval source	That the real register has the shape the source expects
The register has a stray column or a bad date	File adapter	Integration test with a real temporary file	That the office's real export looks like the test file
The directory is slow, down or answers nonsense	HTTP client	Integration test against a local stub service	That the real provider behaves like the stub
A flag is read in the demonstration but not in the build	Whole program	End-to-end journey in a separate process	Behaviour at scale, or with real users
Old behaviour changed by accident	Existing code	The existing regression suite	Anything the old suite never covered

Notice that three different kinds of check appear. A **unit test** runs a small part of the system in memory, with everything it collaborates with controlled. An **integration test** runs our adapter against the real mechanism it adapts: a real file, a real HTTP client talking to a service under our control. An **end-to-end journey** starts the whole program the way a user would, in a separate process. The chapter uses these three names throughout.

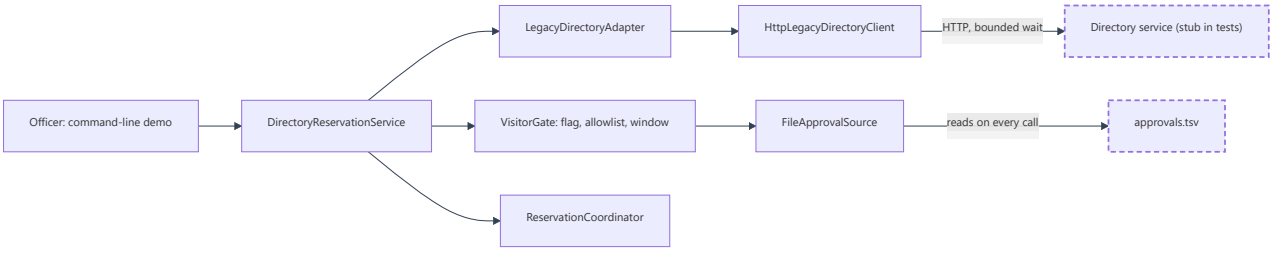


Figure 13.1. The two boundaries added by CR-12. Boxes with a solid border are our code; the two boxes with a dashed border (the directory service and the register file) are what we do not control. Unit tests stop at the adapters, integration tests cross the labelled arrows with the real mechanism, and an end-to-end journey runs the whole picture in a separate process.

Narrow checks first

Fowler distinguishes *narrow* integration tests, which cover only the code in our service that talks to another service and replace that service with a double, from *broad* ones, which need live versions of everything [1]. Narrow checks are small, fast and resilient, so most of the boundary evidence in this chapter is narrow. A few broad checks remain worth having, because only they can show that the pieces are joined correctly. The rule of thumb: many narrow checks, a few broad ones, and a reason for each.

A boundary is where we stop controlling the answer

Our own classes answer the same way every time. A file may be missing today. A network call may not return. A setting may differ between the machine that built the program and the machine that runs it. Each boundary needs two kinds of evidence: what happens when the other side behaves, and what happens when it does not. In this chapter the second kind is not an afterthought. The most important design decision in the change is how a failure at a boundary is reported.

13.2 Where our code meets the outside: integration and contract tests

The file boundary: use a real file

CR-12 puts the sponsor approvals in a small tab-separated register with eight columns. The adapter `FileApprovalSource` reads it, rejects it as a whole if any row is invalid or any request key repeats, and returns the record for one exact request. Its tests write real files into a temporary directory and read them back through the real adapter:

```
@Test void oneBadRowRejectsTheWholeRegisterEvenWhenTheRequestMatchesAGoodRow() throws IOException
    failureOf(register(HEADER, GOOD,
            "V9\tCAM2\tnot-a-date\t2026-10-22\tS1\tAPPROVED\t2026-10-18T00:00:00Z\t2026-10-20T00:(
}
```

Nothing replaces the file system or the parser, because those are the boundary under test. The suite also tests a missing file, a wrong header, a wrong column count, an unknown status, a bad date or instant, a blank sponsor, an invalid interval, a duplicate key, text that is not UTF-8, and a file that changes between two calls. A missing record is an expected result (the source returns empty). An unreadable or untrustworthy file is a failure (the source throws). The two must never be confused, because the first means “no sponsor approved this” and the second means “we could not find out”.

The network boundary: a controlled local service

A few words of HTTP for readers who have not used it directly. A client sends a request to an address. The server answers with a status code (200 means success, 404 means the thing asked about does not exist, 500 means the server itself failed), then headers (information about the answer) and then a body (the data). A timeout can in principle apply to each of those stages, which matters below. A loopback address is the local machine only, and an ephemeral port is one that the operating system picks because it is free, so a test can start a server without clashing with anything else.

The directory used to be an in-memory stand-in. It is now reachable over HTTP. `HttpLegacyDirectoryClient` asks `GET /borrowers/<key>` and understands one line, the provider's kind and status separated by a tab. To test it we need a service that we control: the JDK includes a simple HTTP server API for building embedded servers [9], which is enough for a stub that answers known keys, returns an error, never answers, or sends nonsense. A real service on the network would make the tests slow and dependent on someone else's availability.

The stub is a stand-in, not the provider. Every test that uses it shows how *our client* behaves when a service behaves in these ways. It does not show that the real service behaves in these ways.

An explicit policy for the failure case

A client that can wait for ever, retry silently or answer "not found" when it cannot reach the service has made an unreviewed policy decision. The decision is written down instead:

What the client sees	What the client does
200 with one line, kind and status	Returns the row
404	Returns no row: the borrower is absent
500 or any other status	Fails: the directory could not answer
No complete answer within the bounded wait, even if the headers arrived	Fails with a timeout
Connection refused	Fails
A body that is not exactly two fields	Fails: a guess is worse than a failure

Policy: one attempt, a bounded wait, no retry and no fallback answer. The wait bounds the whole lookup, including the time to receive the body. A failure becomes a `DirectoryException` in the adapter and stops the request, so no reservation is made and nobody is notified. Two consequences matter. An outage must never look like a missing borrower, because that would tell a real borrower that they do not exist. And no borrower is given a category that the provider did not state: an unknown kind code is also a failure, and it is never defaulted to student or staff. A retry would repeat a slow call while a person waits, and a cached answer would grant a category the provider is not currently stating. Both could be justified later by measured outages, and neither is assumed now. Appendix F applies the same questions to a model rather than a directory: a contract the domain owns, an enforced time limit, a cautious fallback where one is safe, and a staged rollout.

A timeout that covers less than you think

The JDK's HTTP client lets a request carry a timeout. The documentation says that if the response is not received within it, the call throws `HttpTimeoutException`, that without a timeout the call can wait for ever, and nothing more about what "received" includes [8]. It does not say whether a response whose headers have arrived and whose body has not is covered.

A client that relies on that timeout alone, tested with a stub that waits **before** sending anything, passes its test. That is the only case the test imagines. Try another: a server that sends the headers and the first byte of the body at once, and then holds the rest for 1,600 milliseconds, with the client's timeout set to 300. The lookup **succeeds after about 1,700 milliseconds**. The policy said "a bounded wait" and the client did not keep it, because the timeout bounded the wait for the headers and not the whole answer.

The fix is a deadline on the whole lookup. The client sends the request asynchronously and waits for the result for at most the configured time; if the time runs out it asks the client to drop the exchange and fails the lookup. Asking is all it can do: the documentation says that cancelling the future attempts to cancel the exchange and that the cancellation is not guaranteed to be prompt or to take effect at all [13][14]. The caller does not depend on it, because the caller has already stopped waiting. The header timeout stays as well. Removing the overall deadline is caught by the new test. Removing the header timeout alone cannot be detected by any test, because the overall deadline still bounds the call: the fault experiment records it as an equivalent change.

A new stub scenario, `SLOWBODY`, sends the headers and the first byte and then holds the rest until the stub is closed. The new test asserts the **outcome**, a failure and not a result, and runs it with a generous outer deadline so that a defect shows as a failed test and not as a hang. Run against a client that relies on the request timeout alone, that test fails with `execution timed out after 5000 ms`. Against the client of this chapter the lookup fails after about 330 milliseconds. The lesson is wider than HTTP: a test of a failure path covers the failure you imagined, so try a different one. A second reader, or a fault experiment that applies a different fault, can find what you could not.

Contract tests: two meanings, one limit

A **contract test** states what callers may rely on and checks that an implementation honours it. The chapter uses the term in two ways, and they have different scopes.

First, an *interface contract*: `DirectoryClientContract` is a test class that is written once and run against every implementation of `LegacyDirectoryClient`. It says that a known key returns its row, an unknown key returns nothing and is not a failure, and a provider that cannot answer fails and never returns an empty answer. It runs against the in-memory stand-in that earlier chapters used, and against the real HTTP client talking to the stub. Two implementations pass the same test, so a caller can swap one for the other.

Second, a *provider contract test*, as Fowler describes it: a separate suite, run periodically against the real external service, that checks that your test doubles still return what the real service returns. It checks the structure of the response rather than exact data, and it follows the provider's rate of change, so it is not part of every pipeline run [2]. This book has no real provider, so no such suite exists here. That is a limit to record, not hide: the stub's fidelity to the real directory is an **assumption**, written in the decision record and tested only when someone has a real service to test against.

13.3 A few end-to-end journeys

Finding the real boundary of a command-line program

A web application has a browser and a server. This program has none of them. Its real boundary is the **process**: how it is started, which settings it reads, what it prints, how it exits, and the network and files it touches. An end-to-end journey therefore starts `Chapter13Demo` as a separate Java process, exactly as a user would, with a directory service on the loopback address and a real approval file, and checks what the user sees.

The program reads its settings from system properties: the directory address, the approval file, the flag (`lending.visitors`, default false), a fixed time for the test, and the directory timeout. The journeys check what no in-process test can: that the flag is actually read, that the pieces are wired together, and that the classpath is complete. The first journey runs with the flag off:

```
Visitors enabled: false
Student B1: Confirmed: R1 for B1 on CAM1 [2026-10-20, 2026-10-22).
Visitor V1: Rejected: visitor reservations are not enabled.
Visitor V2: Rejected: visitor reservations are not enabled.
Unknown U9: Rejected: borrower not found or inactive.
Outage BROKEN: Unavailable: Borrower directory unavailable
Reservations: 1
Notices: 1
```

The second runs with the flag on and shows an approved visitor served and an unapproved one refused. The third corrupts the register and shows that only the visitor requests fail, with the failure reported as an unavailable register and not as a sponsor’ s rejection, while the student is still served.

Keep journeys few and focused

End-to-end journeys are slow, because they start a process, and brittle, because they check output text. Keep the number small. Check the lines that carry the meaning of the journey, fix the clock and the ports so that a run is repeatable, and let the narrower tests carry the detail. A journey that duplicates every unit test is cost without evidence.

13.4 Pipeline stages and quality gates

A pipeline orders the checks by cost

A deployment pipeline splits an automated build into stages. Early stages give fast feedback, and later stages add slower, broader checks [3]. If a stage fails, later stages should not run: there is no point in starting a process when the code does not compile.

Stage	What runs	Why here	If it fails
1 Compile	`mvn compile`	Nothing else can run without it	The code does not build
2 Unit tests and architecture checks	Everything not tagged `integration` or `e2e`	Fast, controlled, no outside resources	A rule or a boundary in our own code is wrong
3 Integration tests	Tests tagged `integration`	Real file system, real HTTP client, local stub	An adapter does not meet its boundary
4 End-to-end journeys	Tests tagged `e2e`	Needs compiled classes and a free port	The pieces are not joined correctly
5 Package	`mvn package`, tests skipped because stages 2–4 passed	Produces the release candidate	The package cannot be built

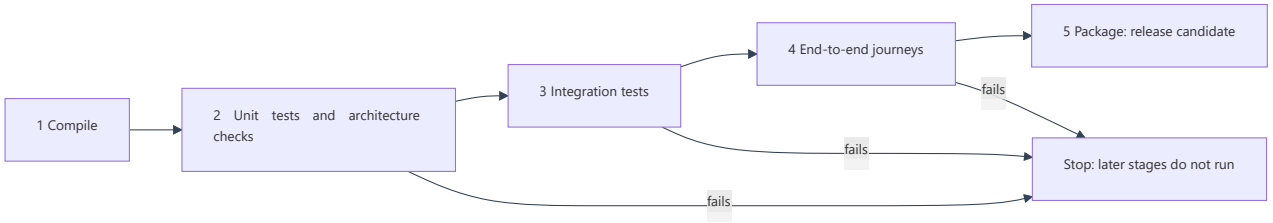


Figure 13.2. The pipeline of this chapter. Each arrow means "only if the previous stage passed". The cheap stages come first so that most failures are reported within seconds, and the process is started only once the pieces it needs are known to work.

The stages are selected with JUnit tags: `@Tag("integration")` and `@Tag("e2e")` on the test classes, and Surefire's `groups` and `excludedGroups` parameters [7], given on the command line as `-Dgroups` and `-DexcludedGroups`. The Surefire page documents the parameters in the POM and not the command-line form, so the pipeline run in the companion is the evidence for that form. The script `scripts/pipeline.sh` runs the five stages in order and stops at the first failure:

```
stage "2 unit tests and architecture checks" "$MVN" "${ARGS[@]}" test -DexcludedGroups=integratio
stage "3 integration tests (real file, real HTTP client, local stub service)" "$MVN" "${ARGS[@]}"
```

The same chain is written as a GitHub Actions workflow, where `needs` makes jobs run one after another [4]. The workflow file is in the companion. It has not been run, and no hosted run is claimed.

Quality gates are decisions about what blocks a release

A **gate** is a rule that stops the pipeline. Here the gates are: every selected test passes, the architecture checks of Chapter 10 pass, and a stage that selects no tests fails instead of passing silently. The last rule matters: a mistyped tag would otherwise give a green stage that checked nothing.

A pipeline is evidence about the checks it contains. It does not establish that the program is secure, that every rule is right or that real users will be served. Say what each gate checks, and what it does not.

Reading a failure

A failing pipeline is information, so learn to read one. A deliberate fault was introduced: the register parser's header check was removed, so that a file with a wrong header is accepted. Stage 1 passed. Stage 2 passed, because no unit test reads the real file. Stage 3 failed at the test named for the wrong or missing header, with the message of the assertion that did not throw. The companion shows the real output. Three habits help. Find the first failing stage and ignore everything after it. Read the name of the first failing test before the stack trace, because the name says what the test promised. And ask whether the earlier stages *should* have caught it: here they could not, which is the reason stage 3 exists.

13.5 Delivery is not deployment

Continuous delivery is a discipline in which software is built so that it can be released to production at any time; **continuous deployment** goes further, and puts every change that passes the pipeline into production automatically [5]. You can do delivery without deployment, and you must do delivery to do deployment. Our pipeline ends with a release candidate, a jar that has passed every stage. That is delivery. Nothing here deploys it to production.

The course needs an understanding of delivery, not a public deployment, so the companion supplies a **simulation**. `scripts/simulate-release.sh` builds the jar, copies it into a local `build/staging` folder as the current release, and runs the end-to-end journeys against that copied jar as a smoke test. A second delivery moves the old release to "previous", and `rollback` makes it current again and smoke-tests it. Two limits matter. The script packages with the tests skipped and then runs only the end-to-end journeys, so it assumes that the pipeline has already passed for the same candidate and does not enforce the other gates. And rollback assumes the same version-derived jar name in both releases, so the exercise's two releases are two labels for one build and not a tested rollback between different versions. This is a **simulation**: there is no server, no network deployment and no production. A real deployment would add environment-specific settings, secrets, a health check and monitoring, and those are outside the scope of the module.

13.6 Feature flags, configuration and rollback

One flag, two paths

A **feature flag** lets you ship code without turning the behaviour on. CR-12 has exactly one: `visitingResearchersEnabled`, default **off**. Off means the old behaviour: a visitor request is rejected explicitly, and the approval register is not read. On means the new path. Student and staff requests behave the same under both settings and never depend on the flag. The flag is a *release toggle*: it hides unfinished or unproven behaviour, and release toggles should be short-lived [6].

Both paths are tested, in the acceptance examples, in the end-to-end journeys and in the service tests. Testing both settings is cheap with one flag and becomes expensive with many, because each toggle state may go live. Test the configuration you expect in production and the fallback with the new behaviour off [6]. An "off" that means legacy behaviour and an "on" that means the new behaviour keeps that simple.

A removal plan is part of the flag

A flag that nobody removes becomes a permanent branch. The decision record sets the conditions: the released variant is accepted, both settings have passed in the pipeline for the release candidate, and a decision to keep visitors enabled is recorded. Then the flag, the disabled path and its tests are deleted together, in one commit. Review the flag at the handover in Chapter 14. Flags are inventory, and inventory has a carrying cost [6].

Configuration is not a flag

The directory address, the timeout and the register path are *configuration*: they differ between machines and are not decisions about releasing behaviour. A flag chooses between behaviours; configuration chooses between places and values. Mixing them gives a program that nobody can reason about. One more caution: the demonstration reads the flag with `Boolean.parseBoolean`, which treats any text other than "true" as false, so a mistyped value silently means off. That is the safe direction, and it is also a reason to print the effective setting, which the program does on its first line.

Rollback, and what it does not undo

To stop new visitor reservations, turn the flag off. To go back to an earlier build, redeliver the previous release. Neither removes a reservation that already exists, undoes a notice already sent, or restores a decision a person already made on the strength of it. AC16 pins one consequence: a visitor reservation made before the flag was switched off can still be checked out and returned. A rollback plan is only a plan when it says what

stays behind.

13.7 Did the change work? Verifying the changed behaviour

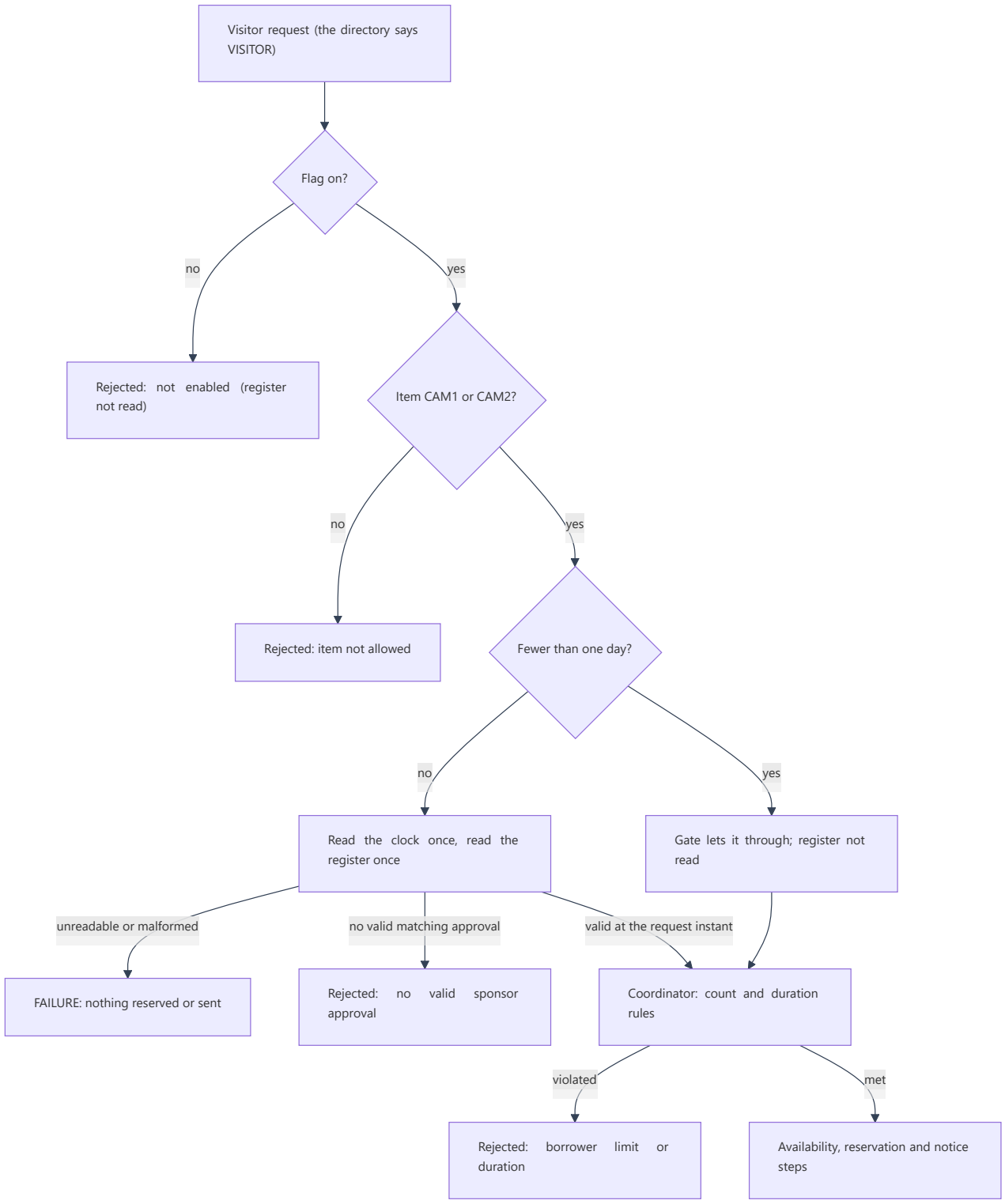


Figure 13.3. The order of the visitor decision, as the code does it. Rectangles are results; the one marked FAILURE is an exception that nothing on the request path catches. Every rejection is an ordinary return value. The gate runs first and the coordinator’ s count and duration rules run last.

The order is observable when more than one thing is wrong. A four-day visitor request with an unreadable register does **not** get the duration rejection: it fails with the register's error, because the gate reads the register before the coordinator applies the duration rule. A zero-day request is the opposite: the gate leaves it alone, the register is not read and the coordinator rejects it. A test pins both. If your team needs a different precedence, such as duration before register, that is a change to the contract, the implementation and the tests together, and not a change to the picture.

Mapping every acceptance example to evidence

Chapter 12 specified eighteen acceptance examples. Each has a test, and the companion lists them. Every example builds its own setup, so one failure cannot hide another. A short excerpt:

```
@Test void ac9_theRequestInstantAtValidUntilOrBeforeValidFromIsRejected() {
    var atExpiry = new World(true, UNTIL).approve("V1", "CAM1", START, 2);
    assertEquals(VisitorGate.NOT_APPROVED, atExpiry.service.reserve("V1", "CAM1", START, 2));
    var early = new World(true, FROM.minusSeconds(1)).approve("V1", "CAM1", START, 2);
    assertEquals(VisitorGate.NOT_APPROVED, early.service.reserve("V1", "CAM1", START, 2));
    assertNothingChanged(atExpiry);
    assertNothingChanged(early);
}
```

Group	Examples	What they establish
Success path	AC1	One matching, valid approval gives one reservation in the existing confirmation format and one notice
Duration and item	AC2–AC5	The upper boundary of three days, no zero or negative days, and the two-item allowlist
Approval matching	AC6–AC9	Missing, denied, mismatched and expired approvals, and the equality at both ends of the window
Failure and flag	AC10, AC12	An unreadable register is a failure, not a rejection; a disabled flag never reads the register
Existing behaviour	AC11, AC13–AC18	Count rule, student and staff unchanged under both settings, availability, notification failure, lifecycle, directory semantics and the report

What the regression suite shows

Adding `VISITOR` to the category enum has a deliberate consequence: the coordinator requires a policy for every category, so every test that builds a coordinator with a complete policy map needed one more entry. That was four constructions in three test files. **No assertion changed.** The inherited Chapter 11 tests passed unchanged after that edit. A regression suite that passes shows that the old behaviours it covers are intact. It says nothing about behaviour the old suite never covered, and nothing about whether the new rule is right.

Unresolved items keep their names

AC18 deserves attention. A visitor who returns a loan late appears in the overdue report, and the report knows only student and staff kinds, so it applies its "any other kind" rule: one unit a day with no grace and no cap. That is the suspected defect SD-11-1 from Chapter 11, and it now has a real consequence. The change does not fix it: only the records office can say what a visitor's late fine should be. The count rule SD-12 also

stays as it is. Naming these items, and showing in a test what each does today, is more useful than silently resolving them.

13.8 Preparing the Milestone 4 evidence

Milestone 4 falls in this week. By your demonstration you should be able to show, on your team's project:

1. your released requirement change, implemented from the smallest vertical slice outwards and behind one flag, with acceptance examples;
2. regression evidence from your existing suite, with any necessary changes to test setups disclosed;
3. integration evidence for at least one genuine boundary, including its failure cases;
4. an observed pipeline run, and a failing run that you have read;
5. a delivery step, simulated or real, with a smoke test and a rollback;
6. flag evidence: both settings tested and a removal plan;
7. a demonstration and a retrospective with a next action; and
8. a statement of what your evidence does not show.

The approved Milestone 4 instructions decide what is submitted, and this list is preparation, not a rubric.

Item	What to produce	What it should establish
1	Change record and acceptance examples	The requirement and each example are traceable to a test
2	Regression result	Old behaviour is intact; changed test setups are listed
3	Boundary tests	The real file, service or store was used, and both outcomes were checked
4	Pipeline output	Which stage ran which checks; one real failure read correctly
5	Delivery record	A release candidate was smoke-tested, and a rollback was tried
6	Flag evidence	Both paths tested; a named condition for removal
7	Retrospective	What worked, what did not and one next action
8	Limits statement	For example: hosted CI not run, real provider not tested, concurrency not addressed

Do not claim a hosted pipeline run, a public deployment or a peer review that did not happen. A local pipeline run is evidence of a local run.

13.9 An integration test that mocks its boundary: judging an AI proposal

Chapter 15 gathers the AI workflows that recur in this book. This activity is complete on its own, and the ready-made proposal below is all you need.

	For this activity
Your goal	Decide what a proposed integration test actually verifies, and repair it so that it verifies what its name claims
You start with	The approval adapter, the visitor gate and the integration tests of Section 13.2
What is in bounds	Judge and repair the proposed test; do not weaken the adapter or its other tests
How to check	Introduce one fault into the parser in a scratch copy and see which tests notice it
How to decide	A verdict on the test, a repaired test and an explanation of what each verifies
No AI account?	The ready-made proposal below is enough

Use this **fictional AI answer written by the author** for offline comparison. An assistant was asked for an integration test of the approval register and produced this:

> I added an integration test for the approval register. It builds a visitor approval, makes the register return it, and checks that the gate accepts the visitor and that the register was consulted.

```
@Tag("integration")
class ProposedRegisterIntegrationTest {
    @Test void anApprovedVisitorIsAcceptedFromTheRegister() {
        ApprovalSource register = mock(ApprovalSource.class);
        ApprovalRecord record = new ApprovalRecord("V1", "CAM1", start, end, "S1", APPROVED, from
        when(register.find("V1", "CAM1", start, end)).thenReturn(Optional.of(record));

        VisitorGate gate = new VisitorGate(register, clock, true);

        assertEquals(Optional.empty(), gate.rejection("V1", "CAM1", start, 2));
        verify(register).find("V1", "CAM1", start, end);
    }
}
```

Run it. It passes. Now introduce a fault into `FileApprovalSource` in a scratch copy, for example reading the two instant columns the wrong way round, or skipping a broken row instead of rejecting the file, and run it again. Then answer: which part of the system does this test touch? Is there a file anywhere in it? What would it still say after you delete the parser? What does the name `@Tag("integration")` promise that the body does not deliver?

Repair it. Write a real register file in a temporary directory, read it with the real `FileApprovalSource`, and pass that source to the gate. Add the failure case: append a broken row and check that the gate throws the register's exception instead of rejecting politely. Then run both tests against the same fault and compare. (Removing the header check would not separate them, because the repaired test does not exercise the header. Say why that is acceptable, and which test does cover it.) Finally, say what the *original* test is still good for: it is a fine unit test of the gate's decision logic with a controlled source, and it belongs in the unit stage under an honest name.

A secondary activity: take the real failure output in the companion's pipeline run and say, for each stage, what passed and why the earlier stages could not have caught the fault. A live AI service is optional. If you use one, give it synthetic data, keep its actual output, and treat its description of its own test as a claim to check.

13.10 Delivery habits that mislead

Mistake	Consequence	Better action
Mock the boundary and call the test an integration test	The test passes whatever the real adapter does	Use the real file or a controlled local service, and fault-test it
Test only the successful response of a boundary	An outage or bad data is found in the demonstration	Test missing, slow, malformed and refused answers
Report an unreachable service as “not found”	A real borrower is told that they do not exist	Fail the request and say that the service could not answer
Add a timeout without deciding what happens next	The program waits less long and still behaves unpredictably	Write the policy: one attempt, no retry, no fallback, or whatever is chosen
Default an unknown category to the nearest known one	A visitor becomes a student without anyone deciding it	Treat an unknown code as a failure
Let an end-to-end journey check everything	Slow, brittle, and it duplicates the other tests	Check the lines that carry the journey’s meaning
Use real clocks, ports or paths in tests	Runs differ by day, machine or luck	Inject the clock, take a free port and use a temporary directory
Assume a timeout setting covers the whole call	A slow body or a trickling response holds the caller for as long as the server likes	Bound the whole operation, and test the stall you did not think of
Make a stage that can select no tests	A mistyped tag gives a green stage that checked nothing	Fail a stage that runs nothing
Treat a green pipeline as proof	Security and unseen behaviour are not examined	State what the stages check and what they do not
Switch a flag on by default or never remove it	The old path rots and the code keeps both	Default off, test both paths, name a removal condition

13.11 Try it yourself: run the pipeline and deliver a flagged change

You start from the Chapter 11 code with the Chapter 13 files from the companion, Chapter 12’s CR-12 specification, and your team’s repository. Steps B1 to B7 use the book’s project; steps T1 to T5 use your team’s project.

If your instructor demonstrates the task, they will run `scripts/pipeline.sh` and read each stage’s output, break the register parser on purpose and run it again to show which stage notices, and run the flag-off and flag-on journeys. For guided practice, classify four risks of your choice as unit, integration or end-to-end checks and say what each check cannot show, then read the directory policy table in Section 13.2 and predict the outcome for a response the table does not list.

What you must do: the book’s project (run, trace and diagnose; nothing to rebuild)

- **B1. Run the pipeline.** Run `scripts/pipeline.sh`, or the five Maven commands in it. Record each stage, the number of tests it ran and its time. Say what a failure in each stage would mean.

- **B2. Trace three acceptance examples.** Choose AC8 or AC9 (the window boundary), AC10 (the operational failure) and AC12 (the flag). For each, name the test, the production class that decides the outcome and the observable result, and say which stage runs it.
- **B3. Make a test bite.** In a scratch copy, introduce one fault into `FileApprovalSource` or `HttpLegacyDirectoryClient`. Predict which stage fails first, run the pipeline, and compare. Restore the original.
- **B4. Complete the evidence table.** Fill in a table for the whole change in the form of Section 13.1: the risk, the check that covers it, the test's name and what that check cannot show. Mark at least one risk that nothing here covers.

What you must do: your team's project (your Milestone 4 evidence)

- **T1. Implement your released change.** Start with the thinnest vertical slice that behaves correctly, put it behind one flag that defaults to off, and record the acceptance examples. Keep expected rejections apart from operational failures.
- **T2. Test one genuine boundary.** Choose a boundary in your project (a file, a service, a store) and test it without replacing it: a real temporary file, a local stub service or a throw-away database. Cover at least one failure case.
- **T3. Protect the old behaviour and both flag settings.** Run your existing suite, disclose any test setup you had to change, and add tests for the flag on and off.
- **T4. Build and read a pipeline.** Configure at least three stages (for example compile and unit, integration, and a smoke or end-to-end stage). Capture one observed run and one failing run, and read the failing one: first failing stage, first failing test, and why earlier stages could not have caught it.
- **T5. Prepare the Milestone 4 evidence.** Record a flag removal plan and a rollback step, run your demonstration, write the retrospective with one next action, and complete the eight items of Section 13.8, marking each as complete or a gap.

If you want more (the book's project)

- **B5. Deliver and roll back.** Run `scripts/simulate-release.sh` twice with different `RELEASE_LABEL` values, then `rollback`, and record which release is current after each step.
- **B6. AI activity.** Complete the Section 13.9 activity.
- **B7. Break a boundary on purpose.** Make the directory timeout twenty seconds in a scratch copy and run the slow-provider test, then explain what the test shows and why a pipeline that waits this long is itself a problem.

What you should end up with is: B1 to B4 as described; for your team's project, the implemented change with its acceptance examples, the boundary tests, the pipeline output (one passing and one failing), the delivery and rollback record, the flag evidence, and the Milestone 4 checklist. A pull request is a good vehicle where your course workflow permits it.

You are done when these things hold:

- the change is behind a flag that defaults to off;
- expected rejections and operational failures are distinguishable;

- one boundary is tested without being replaced, and its failure cases are tested;
- old behaviour is protected and any changed test setup is disclosed;
- a stage that selects no tests fails;
- the delivery step is labelled as a simulation unless it is real;
- the limits statement says what was not run.

Do not claim a hosted run, a public deployment or a review that did not happen.

If you want more, add a third flag setting, but only if you can say what "on", "off" and the new setting each mean and how each is tested. Otherwise remove a flag you no longer need.

13.12 Questions to think with, and a transfer challenge

Concept questions

1. Why can a unit test not show that a file adapter parses the real register? What does it show instead?
2. What does the directory contract test say, and what can it never show about the real provider?
3. Why is "not found" the wrong answer when the directory times out? Name two other defaults the policy refuses.
4. What is the difference between continuous delivery and continuous deployment, and which does `simulate-release.sh` imitate?

Applied diagnosis

5. A teammate replaces the file reader with a mock "so that the test is repeatable" and keeps the name `FileApprovalSourceIntegrationTest`. Which risk is no longer covered, and how would you keep the repeatability without replacing the reader?
6. The pipeline is green and the demonstration fails because the flag was off. Name two checks, one in the pipeline and one in the process of delivering, that would have noticed.

7. A client sets a request timeout of two seconds and the officer's screen still freezes for forty. The server sends its headers at once and then sends the body a few bytes at a time. What does the timeout cover, what does it not, and what would you change and test?

Transfer challenge

An ML model-serving team adds a rule: only approved model versions may be served to external callers. The approval list is a file produced by another team, and the model registry is queried over the network. Design the checks: which risks are unit, integration or end-to-end checks, how a registry timeout is handled and what it must not default to, what the one flag does, and what the rollback does not undo. Name one thing that your evidence would not show about the model's accuracy. Use synthetic model identifiers: no model needs to be trained.

What to take away, and what comes next

A change is only as safe as the checks at the places where our code meets something else. Test the real file and a controlled service without replacing them, write down what happens when they fail, order the checks in a pipeline from cheap to broad, release behind a flag that defaults to off, and say what a rollback cannot undo. A green pipeline is evidence about the checks it contains and nothing more.

The visiting-researcher change is implemented and verified, with two unresolved items still named: SD-11-1, which a visitor's late fine now exposes, and SD-12, the count rule. Chapter 14 reviews how the design accommodated the change, prepares the handover and decides the flag's removal.

Sources and further reading

Online sources 1–9, 13 and 14 were opened on 13 October 2026. The code, the stub, the policy, the scripts and the decision record are original teaching material.

1. Fowler, M., **Integration Test**. Narrow and broad integration tests.
2. Fowler, M., **Contract Test**. A suite run periodically against the real service to check that test doubles still match it; format rather than exact data.
3. Fowler, M., **Deployment Pipeline**. Stages that give fast feedback first and slower, broader checks later.
4. GitHub, **Workflow syntax for GitHub Actions**. Jobs run in parallel unless `needs` makes them sequential. The workflow in the companion has not been run.
5. Fowler, M., **Continuous Delivery**. Delivery as the ability to release at any time, and the contrast with continuous deployment.
6. Hodgson, P., **Feature Toggles (aka Feature Flags)**. Toggle categories, short-lived release toggles, removal tasks, and testing the configurations that may go live.
7. Maven, **Using JUnit 5 Platform in Surefire**. The `groups` and `excludedGroups` parameters. The command-line properties `-Dgroups` and `-DexcludedGroups` are not documented on that page; they were checked by running them.
8. Oracle, **HttpRequest.Builder (Java SE 17)**. The request timeout and the `HttpTimeoutException`. The page says the response must be received within the timeout and does not say whether a stalled body is covered.

9. Oracle, **com.sun.net.httpserver (Java SE 17)**. A simple high-level API for building embedded HTTP servers; the page does not call it a production server.

10. JUnit, **User Guide 5.13.4**. `@Tag` and `@TempDir`. This book treats JUnit tests in Chapter 7 and the use of test tags in Section 13.4.

11. Martin, R. C., *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall, 2002. The module's required text; the descriptor lists the Week 12 reading as "Martin (2002): Testing (review)". This book treats testing in Chapter 7 and integration and end-to-end testing in Sections 13.1 to 13.3.

12. Humble, J. and Farley, D., *Continuous Delivery.* Addison-Wesley, 2010. The module's recommended reading for Week 12 (selected chapters). This chapter treats pipelines, delivery and deployment in Sections 13.4 and 13.5.

13. Oracle, **HttpClient (Java SE 17)**. The futures returned by `sendAsync` are cancelable and cancelling attempts to cancel the exchange, but is not guaranteed to be prompt or to take effect.

14. Oracle, **CompletableFuture (Java SE 17)**. `get` with a timeout throws `TimeoutException` without cancelling the future; `cancel` completes it with a `CancellationException`.

Chapter 14: Responding to Change and Handing Over

> *Final system handover, traceability matrix, qualitative contribution, and release packaging.*

OO Development and Agile Practices
Designing, Testing and Evolving Software with AI Assistance

It is the morning of the final presentation. A classmate who was not on your team opens your repository link to see what you built. The README says to run one command, and the command fails, because the classmate's machine has no `mvn` on its path and the error says only "command not found". The README promises one number of passing tests; the build reports another. The requirements table points at a test that was renamed last week. A reviewer asks which of the three of you wrote the approval-file adapter, and the commit log says it was whoever has the most commits. It was not.

None of these is a defect in the software. Each is a defect in the **handover**: the material that lets someone else set the project up, trust what it says and understand who did what. This chapter finishes the project the way Chapter 13 started the change, with evidence. It is not a new feature sprint: the change began in Week 11 and was implemented in Chapter 13, so here we add no behaviour. The one fix in the worked project is to the setup and the demonstration, and everything else is a decision, a document or a check.

You will review how the design accommodated the change, using the affected responsibilities, the test failures, the revised assumptions and the review feedback, and see why a small diff is not automatically a better design. You will decide what remains to fix, defer or accept; write the documents a stranger needs and check them by following them in a fresh copy; and trace requirements to implementation and tests in a way that a check can keep true. You will support your retrospective with evidence and write an individual reflection that explains your own contribution, and you will prepare a final submission that does not claim evidence that does not exist.

This is Week 13, the final sprint and retrospective, and it touches all learning outcomes, LO1 to LO6. The required reading is Martin (2002), a review of the chapters relevant to your project [10]; the recommended reading is whatever is relevant to your own team's project. Week 13 has a final-sprint workshop with formative feedback, Week 14 has no classes and is for independent project work with optional team consultations, and the final demonstration, presentation and submission are in Weeks 15 and 16. The descriptor says teams submit a repository link, a technical report and an individual reflection, and give a demonstration and presentation, and that marks are moderated by individual contribution. This chapter creates no new deadlines, weights or marking rules: the approved final-project instructions decide what you submit.

You start from the Chapter 13 version (the visitor change, version 0.13.1) or, for your team's project, from a Milestone 4 outcome with its remaining defects named. You finish with version 0.14.0, and its production code is unchanged. The companion lists only the new and changed files, with the model answers, the experiments and teaching notes.

Sections 14.1 to 14.5 and 14.7 are the core reading. Section 14.6 on contribution evidence is working knowledge: you are not asked to measure anyone, you are asked to know what each kind of evidence can and cannot show. What you must do is steps B1 to B4 and T1 to T5 of the practical activity in Section 14.10. Steps B1 to B4 run on the book's project and are running, tracing and diagnosis, since the finished version is supplied; steps T1 to T5 run on your team's project and produce your handover bundle, retrospective and individual reflection. If you want more, there are steps B5 to B8 (rehearse removing the flag, validate an AI-drafted README, read a synthetic contribution table, and close the gap that the one-line experiment found), the questions and the transfer challenge in Section 14.11, and the extension at the end of Section 14.10.

14.1 Did the design absorb the change? Reviewing the outcome

Ask four questions, in this order

"Did the design absorb the change?" is too vague to answer. Break it into four questions, each with its own evidence.

Question	Evidence	Weakness
Which responsibilities were affected?	The list of files changed and added, and what each one does	Counts files and lines, not the importance of a line
What broke when the change started?	Test failures after the smallest edit that introduces the change	Shows what the tests cover, not everything that depends on the code
Which assumptions turned out to be wrong?	The assumptions written in earlier chapters, checked against the change	Only the assumptions someone wrote down
What did a reviewer say?	Actual review comments and the responses	Only as good as the review

Affected responsibilities

Between the Chapter 11 version and the Chapter 13 version, 23 files were added and 10 changed. Production code first. Nine new production files hold 399 lines: the visitor policy, the approval record and its status, the gate, the approval port and its exception, the file adapter, the HTTP client and the demonstration program. Five **existing** production files changed, by 21 added lines and 3 removed: the category enum, the directory entry point, the composition root, the directory adapter and the directory client contract. Ten new test files hold 992 lines, and three existing test files changed by 10 added lines and 4 removed. The rest is scripts, a workflow, a decision record and the README and POM.

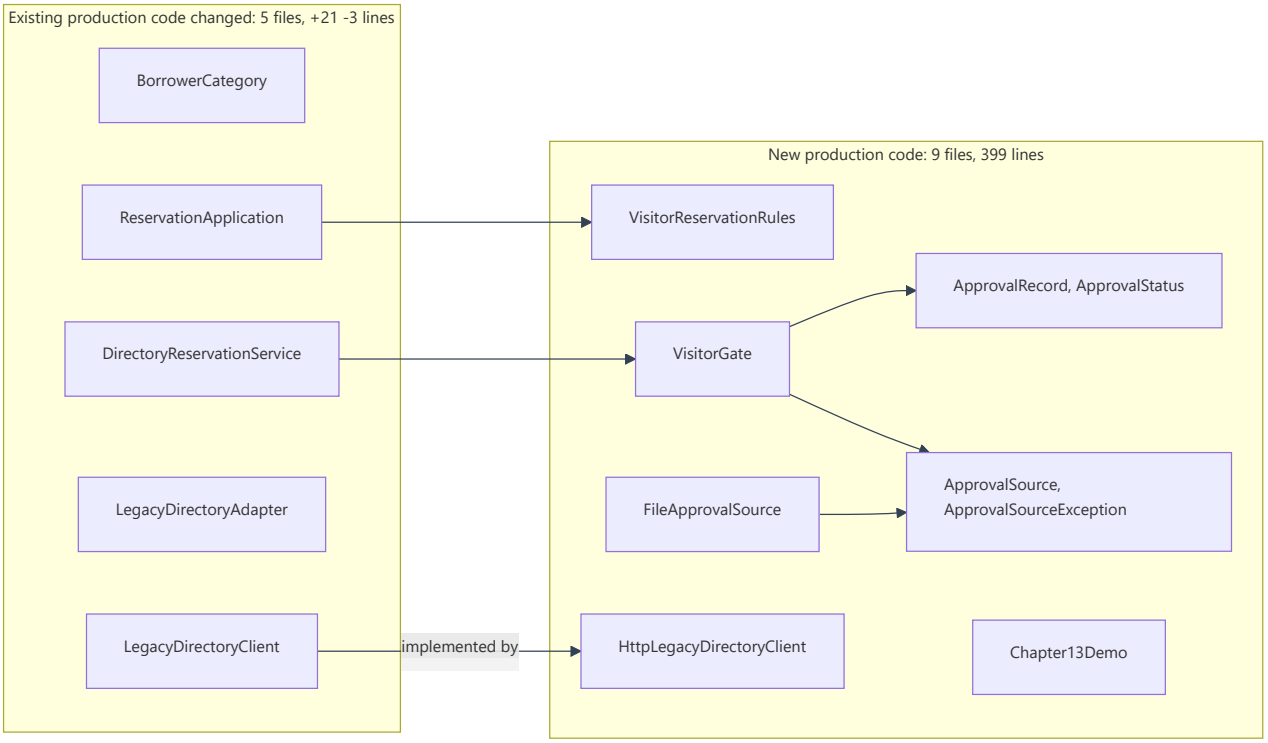


Figure 14.1. The footprint of the visitor change in production code. The change is mostly new files that sit beside the existing ones. The existing files that changed did so by a few lines each. Arrows show use: an existing file using a new one, or a new file using another.

What does this show? Mostly that the existing structure took the change by extension, which is what the abstractions of Chapters 5, 6 and 10 were for. A new borrowing rule was one new class behind the policy contract. A new source of approval was one new port with an adapter. Two things do **not** show in a line count: the existing files that changed are the ones every request passes through, and nine new files are nine more things to read.

What broke first

Start the change in the smallest possible way and see what the tests say. On the Chapter 11 version, add `VISITOR` to the category enum and change nothing else. The build fails: more than a quarter of the suite fails (one failure and a long list of errors), and nearly all of them say `Missing policy: VISITOR`. The coordinator requires a policy for every category and says so when it is constructed. The exact figures are in the companion (Section 5.2).

Now add the visitor policy to the composition root, `ReservationApplication`, as well. The failures drop to five tests, in exactly the three test files that build their own policy map: `LegacyRegressionTest.chapter06` (through `Chapter06Checks`), `BehaviourStrategyTest` and three tests in `RepositoryBoundaryTest`. Add the policy to those four maps and the whole Chapter 11 suite passes, with no assertion changed.

Is that good or bad? Both. It is the design telling you, at construction time and with a precise message, everywhere that lists every category. That is a safety net: the alternative would be a request that fails in production with a null policy. It is also a cost: every test that spells out the full policy map must be edited when a category is added. You can reproduce this in Section 14.10 (B1) by going the other way: remove the visitor policy from the composition root of the finished version, and roughly three in ten tests fail, in fourteen test classes.

Revised assumptions

Chapters 5 to 11 made statements that the change tested. Some held and some did not.

Assumption (where it was made)	What the change showed	Verdict
A policy answers a count and duration question and nothing else (Ch. 5, 6)	Approval needs the borrower, the item, the period and the time. Widening the policy would give it hidden state, so the request-specific decision went to a separate gate	Held, because the contract was not widened
Every borrower category has a policy in one place (Ch. 6)	True in production, but four tests also spelled out the map	Held, with a cost
The directory is a synthetic in-process provider with a failure contract but no transport (Ch. 8)	Chapter 8 already translated a provider failure into a directory exception and told an outage from an absence. Chapter 13 added a real HTTP client and observed the failures a transport brings, including a timeout that has to cover the whole answer and not only the headers	The contract held. The behaviour behind it was extended
The count of retained reservations is the borrowing count (Ch. 7)	CR-12 preserved it on purpose and AC11 pins it	Still open (SD-12)
The overdue report knows two borrower kinds (Ch. 11)	A late visitor reaches the “any other kind” rule and is fined at a rate nobody chose	Still open (SD-11-1)
Notifications are one attempt, no retry (Ch. 6)	The directory client follows the same rule	Held and extended

A small diff is not automatically a better design

Here is the cheapest possible "implementation" of visitors. The directory adapter maps the kind code `V` to the existing staff category, in one line, and no gate, no approval register and no new policy exist. The diff is one line.

Run it against the finished suite (experiment E3 in the companion, Section 5.2). **Every unit test passes.** What fails is the tests that cross the real boundary: the three end-to-end journeys that use visitors and the integration test that serves a visitor through the real directory. In the journey with the flag off, visitor V1 is confirmed, so the flag is ignored; an unapproved visitor is rejected only because the item is already taken, not because it is unapproved; and a malformed register no longer stops anyone. It is a smaller diff, and it does not implement the requirement.

Be careful about what this shows. It shows that this suite has a **gap**: no unit test checks the adapter's mapping of the visitor code, so a one-line change to it passes the whole unit stage. A focused unit test of the adapter would catch it, and so would the integration tests that did. The lesson is not that unit tests cannot see such a change. It is that a passing unit stage is evidence only about what the unit tests cover, and that a suite with a hole in it is found by trying a change that goes through the hole. Neither the size of a diff nor a green unit stage is the measure of a design. The acceptance examples are, and the gap itself is a finding: Section 14.10 (B8) asks you to close it.

The same reasoning protects the opposite case. A change that is larger than you expected, if each added file has a purpose that the requirement needs, is not a failure. It is a failure when the added files exist to support code that nobody asked for, or when existing files change in ways the change did not require. The footprint above has no such file: the three changes to existing test files add an entry to a map, and nothing else.

Where the design bent

A fair review also names the place that did not absorb the change cleanly. `DirectoryReservationService` now contains this:

```
if (borrower.category() == BorrowerCategory.VISITOR) {
    Optional<String> rejection = visitorGate.rejection(borrower.id(), itemId, start, days);
    if (rejection.isPresent()) return rejection.get();
}
```

That is a category-specific branch in the entry point. For one special category it is fine and easier to read than an abstraction. At a second special category it becomes a smell, and the next step would be a per-category pre-check chosen from a map, introduced as a refactoring with its own tests. Record this as a known issue with its trigger (the second category) instead of fixing it now. The known-issues list in Section 14.2 does exactly that (KI-14).

Review feedback

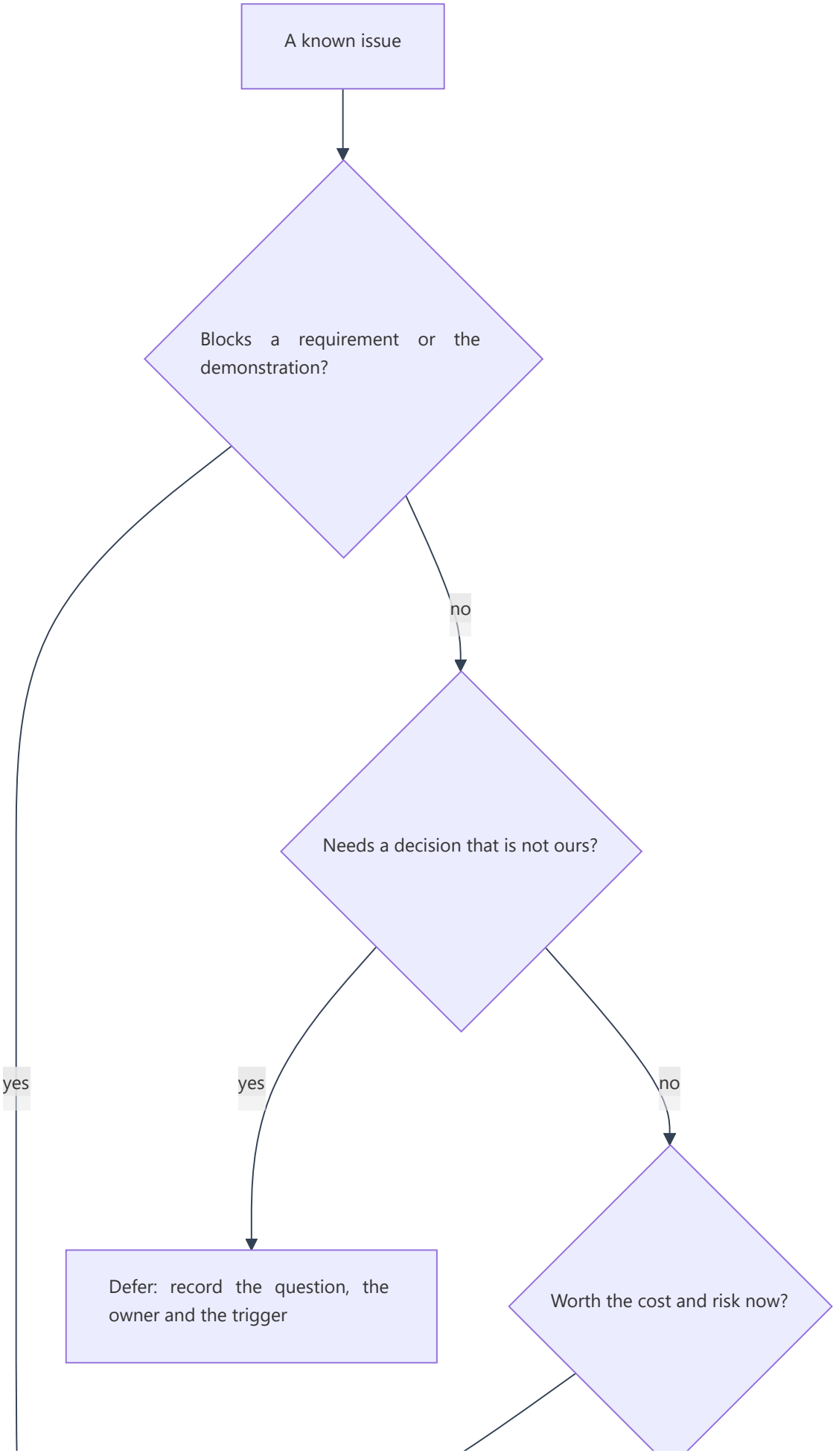
Real review feedback is evidence, and **constructed** feedback is not. If your team has review comments on its change, use those and the responses. The book’ s project has no team, so the companion supplies a short, author-written, fictional review of the visitor change, labelled as such, and asks you to check each comment against the code instead of accepting or rejecting it by tone. No real peer review of this book’ s project has taken place.

14.2 Fix, defer or accept: scope control in the last week

The temptation of the last week

In the final sprint every team sees things it could improve. Some are real defects. Some are polish. Many are new features in disguise. If you act on all of them, you finish nothing and introduce faults close to the deadline. Three questions sort them.

1. **Does it block a requirement or the demonstration?** If yes, it is a blocker: fix it first or change the plan.
2. **Does fixing it need a decision that is not ours?** If yes, it is *deferred*: record the question, the owner and what would trigger the decision.
3. **Is the cost and risk of changing it now worth the benefit?** If not, it is *accepted*: say so in writing, as a limit, not as a secret.


A known issue
Blocks a requirement or the demonstration?
yes
no
Needs a decision that is not ours?
yes
no
Defer: record the question, the owner and the trigger
Worth the cost and risk now?

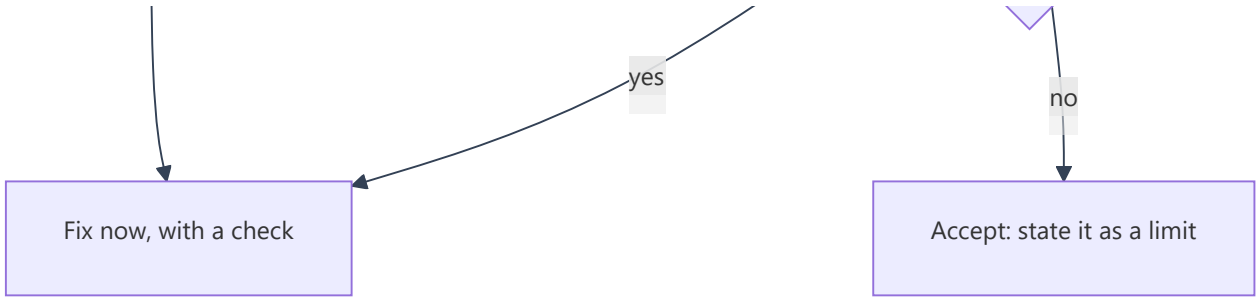


Figure 14.2. Triage of a known issue. Every path ends in something written down. "Fix now" ends with a check, so the fix is visible later.

What the project did with its own list

The visitor project has fourteen known issues (`docs/KNOWN-ISSUES.md`). One was fixed, seven are deferred (five wait for an owner's decision, one for a JDK 17 and a hosted runner, one for the flag decision) and six are accepted limits. The fixed one is a good example of polish worth doing: the demonstration program needed a directory service that only the tests could start, so a new reader could not run it. That is a blocker for a handover, because a reader who cannot run the system cannot check anything else about it. The fix added no behaviour. It added a small launcher that starts the same test stub and runs the system twice, and a script to run it.

The deferred ones are different. SD-11-1 says that a borrower kind other than student or staff is fined at 1.00 a day with no grace and no cap. A late visitor now reaches that rule, and acceptance example AC18 pins it. It is tempting to "fix" it by giving visitors a sensible rate. But nobody on the team knows the right rate. Only the records office does. A guess would be a new rule disguised as a bug fix, and it would hide the question. So it is deferred: the question, the owner, and the test that will change when the answer arrives are all written down.

The accepted ones (a one-second directory wait that was not measured; no retry after an outage) are limits that the documents state. "Accepted" is not "ignored". It means that the team decided, with a reason, and that a reader can find the reason.

Resist the plausible additions

Tempting addition	Why not now
Retry the directory call after a timeout	Changes the one-attempt contract of earlier chapters, and nobody has measured an outage
Replace the approval file with JSON or a database	Nothing in the requirement needs it, and it adds a new boundary to test
Remove the flag to "clean up"	The conditions for removal do not hold (Section 14.3)
Give a visitor a late-fine rate	A decision for the records office, not the team
Add a second category of visitor	A new requirement. Raise it as a change request, not as polish

Fowler names four costs of building a feature before it is needed: the cost of building it, the cost of the delay it causes, the cost of carrying its complexity, and the cost of repairing it if its assumptions were wrong [5]. In the last week the delay and the repair cost are the ones that hurt.

A release is "done" by a definition, not a feeling

The Scrum Guide defines a Definition of Done as the state an increment must reach to meet its quality measures, and says that work that does not meet it cannot be part of the increment [3]. Write yours for the handover version, as a short list that a check can confirm: the pipeline passes; the handover check passes; no test is skipped; every number in a document matches a run; every decision record has a true status; every known issue has a decision. The next section shows which of these a script can confirm.

14.3 Writing it down: documents, decision records and release notes

One document per question

A new reader asks different questions at different times. One long document answers none of them well. The project has six, and each is kept true in a different way. Appendix C has forms for the requirements table and the known-issues list (C.10), for decision records (C.5) and for release notes (C.11).

Document	Question it answers	How it is kept true
`README.md`	What is this, and where do I start?	Short; points to the others
`docs/HANDOVER.md`	How do I set it up, run it and understand it in an hour?	`scripts/handover-check.sh` follows it in a fresh copy
`docs/REQUIREMENTS-TO-EVIDENCE.md`	Which test checks each requirement?	`TraceabilityTest` fails if a named test or class is gone
`docs/KNOWN-ISSUES.md`	What is wrong or missing, and what was decided?	A reviewer; every row has a decision
`docs/ADR-0NN-*.md`	Why is it built this way?	Never edited after acceptance except its status; a new record supersedes it
`docs/RELEASE-NOTES-*.md`	What changed in this version?	Written at release; one file per version

The common failure is documentation that was true once. The cheapest document to maintain is one that a check keeps true, so wherever a statement can be checked by a script, write the script.

Decision records

A decision record captures a decision, not a description of the code. Nygard proposes short records kept in the repository with five parts: a title, the context (the forces at play), the decision, its status (proposed, accepted, or later superseded) and its consequences, numbered in order and never reused; if a decision is reversed, the old record stays and is marked as superseded so that a reader can see what was decided and why it changed [1]. The records of this project add three more parts that earn their place: the alternatives considered, how the decision was validated, and the triggers for revisiting it.

ADR-013 recorded the visitor change and set conditions for removing the flag. ADR-014 is the handover decision about that flag, and it shows the shape of a good one. It states the three conditions from ADR-013 and checks each: the pipeline has run both settings (true); the visitor contract has not been accepted by an owner (not yet); nobody has recorded a decision to keep visitors enabled (not yet). It decides to keep the flag, default off, and **rehearses** the removal so that the change is ready and its cost is known. It does not remove it, and it does not edit ADR-013.

Release notes

A changelog is for people, so write it for them. Keep a Changelog groups changes as Added, Changed, Deprecated, Removed, Fixed and Security, gives each version its own dated entry and puts the newest first [2]. The notes for version 0.14.0 use three of those headings and add two of their own that a handover needs: “Behaviour you should know about” and “Not verified”. The first line of the notes says that no production behaviour changed. That is a statement a reader can check, and it was checked: every file under `src/main` is byte for byte as in 0.13.1.

14.4 Keeping the evidence true: rehearsal and traceability

Section 14.3 described the documents. This section shows two ways to keep what they say true: rehearse a risky change before deciding it, and give the requirements table a check that fails when a name in it disappears.

Rehearse a risky removal instead of arguing about it

The removal of a flag looks like deleting an `if`. Rehearsed in a scratch copy, it touched eleven files: three production classes, five test classes, the requirements table, the expected numbers in the handover document and the handover check, with 87 lines added and 131 removed. The reason is that the flag was not one `if`. It was a constructor parameter, a result text, a factory for the old two-argument constructor, a system property in the demonstration, the acceptance example AC12 that exists only for the flag, two other acceptance examples that mention it, a journey that tests the flag off, and two documents that explain it. After the rehearsal the full build had no failure, the pipeline passed and the handover check passed. The patch is stored in `rehearsal/` and not applied.

Notice what the rehearsal also taught. Retiring AC12 means that a requirement in the specification (a disabled flag rejects explicitly) disappears when the flag does. That is a change to the specification, and it needs a decision, not just a patch. After removal, the rollback is a redelivery of the previous release, because there is no longer a switch.

Requirements to evidence

Traceability is the chain from a requirement to the code that decides it and the test that checks it. In a small project a table is enough:

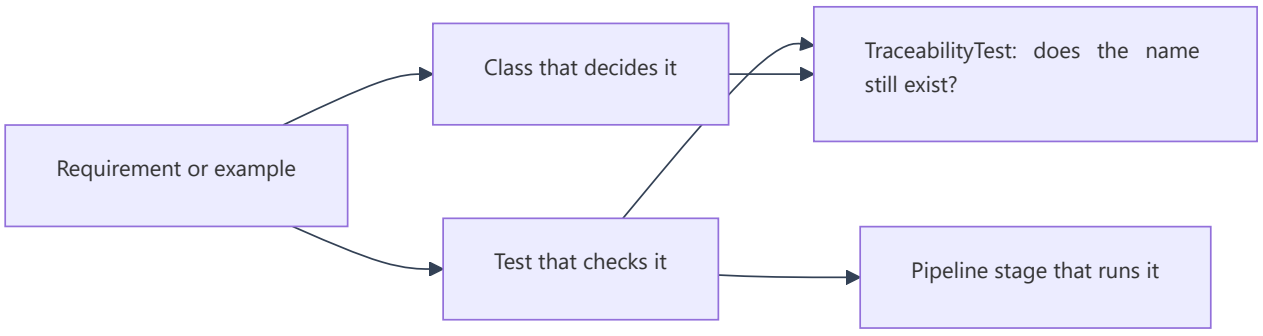


Figure 14.3. The chain that `docs/REQUIREMENTS-TO-EVIDENCE.md` records. The last step is the one that stops the table rotting.

`TraceabilityTest` checks three things: every test named in the table still exists as a method, every class named as deciding still exists, and each of the eighteen acceptance examples has exactly one row and every acceptance test appears in the table. In an experiment with six faults (Companion, Section 10.2) it caught the four that make a name disappear or an example lose its row. It did **not** catch two others, and it cannot: a row that points at a test that exists but is the wrong test for that requirement, and a row that lists an extra test. A script can check that a name exists. Only a reviewer can check that a name is the right evidence. Say so in the document, as the project does.

14.5 Setup and a demonstration that anyone can reproduce

Write for someone who is not you

Your own machine has things installed that you have forgotten about. The only way to find them is to try the instructions somewhere else. The handover document for the project begins with what you need (a full JDK 17 or later with the `jdk.jdeps` module, Maven 3.9.11, a POSIX shell, network access for the first build), then gives three commands with their expected output, then shows how to see the system working.

A script can follow the document for you. `scripts/handover-check.sh` copies the project, without build output, into a new folder, runs the pipeline and the demonstration there, and compares what they print with what the document promises: the pipeline passes with the expected number of tests in each stage, the document itself states those numbers, and the demonstration shows the expected lines. If a test is added and nobody updates the document, the script fails, and so it does if the document is edited to a number the run does not give. In an experiment, adding one test made it report that a stage had run a different number of invocations from the one promised, and renaming one word in the demonstration's output made it report the missing line.

What the dry run found

Following the handover in a fresh folder found four things before any reader did.

Finding	Effect on a new reader	Change
The setup assumed `mvn` was on the PATH. On the test machine it was not, and the failure said only `command not found`	The reader stops at step 1	The document starts with a tool check, and the scripts print a clear message and stop if Maven or Java is missing
On a recent JDK, Maven 3.9.11 prints `WARNING` lines about `sun.misc.Unsafe` (these come from the JDK version, not from your code), and the mocking library prints a warning that an agent was loaded	A reader may think the build is broken	The document lists "things you may see that are not errors" and says to read the last lines
The demonstration program needed a directory service that only the tests could start	The reader could not see the system work	A launcher that runs the stub and the program in one process, and `scripts/demo.sh`
The first version of the clear message for a missing Maven was wrong: it rejected a valid Maven path on Windows (a `.cmd` file that is not executable in the shell sense)	The check meant to help would have blocked a correct setup	Found by running the handover check on a machine with a full path in `CHAPTER_MVN`; fixed by also accepting a file that exists

The last row matters. It was a fault in a fix written during this very exercise, and it was found only because the fix was run in the setting it was meant to protect. A document you have not followed is a guess about what it says.

A folder name with spaces in it was also tried, and worked. That is a result and not a finding: say what you tried that did not break.

The demonstration

A demonstration is a short, repeatable story with a fixed clock and fixed data. The project's is `scripts/demo.sh`: it compiles, starts the local directory stub inside one process, writes a small approval register, and runs the system twice, with visitors disabled and then enabled. A viewer sees the same requests produce different results, and an unreachable directory reported as unavailable and not as a missing borrower. The handover also suggests an outline of about ten minutes: the problem and what changed, the pipeline, the demonstration, one row of the requirements table followed to its test, and what is not shown. The timings are a suggestion. Your instructions decide the length and format of your own demonstration.

What the dry run cannot show

It ran on one operating system with one JDK and a local Maven repository that already held the dependencies. It cannot show a first build over a slow network, a Java 17 runtime, a different shell, or a reader who misreads a sentence you thought was clear. To cover the last, ask a real person from outside your team to follow the document while you watch and say nothing. Write down every place they hesitate. That is the test.

14.6 Retrospective and individual reflection

A retrospective is evidence-based or it is a mood

The Scrum Guide gives the purpose of the Sprint Retrospective as planning ways to increase quality and effectiveness: the team inspects the last sprint's individuals, interactions, processes, tools and Definition of Done, discusses what went well and what went wrong, and identifies the most helpful changes, which may be added to the next Sprint Backlog [3]. Kerth's Prime Directive asks the team to assume that everyone did the best they could, given what they knew and the resources they had [4]. Together they say: look at the work and the process, not at blame.

Make it evidence-based by bringing the sprint's records: the board, the pipeline runs (passing and failing), the review comments, the defect list, the time between starting and finishing items from Chapter 12. For each thing the team wants to say, ask what record shows it.

Observation	Evidence	Action	Owner and date
"Reviews were slow"	Two pull requests waited more than two days (board timestamps)	Review before starting a new item	Named person, next sprint
"The pipeline was flaky"	One of ten runs failed with no code change (run list)	Fix or quarantine the unstable test	Named person
"We rushed the end"	Seven items moved to Done on the last day	Plan a smaller scope and a freeze	Whole team

Keep the number of actions small. Two actions with owners that are done beat ten that are not.

The process note, revisited

Chapter 3 asked each team to say which parts of Scrum it used as defined and which it adapted for the course. Now say what the evidence shows about each adaptation. For example, the course's sprints end at milestone weeks and not at a fixed calendar interval; a student team is not a full-time team; the Product Owner is usually a teaching role. For each adaptation, say whether it helped, whether it hurt and what you would change. This is the closing of an earlier commitment, and it does not need new terms.

Individual contribution is evidence of three kinds

The descriptor assesses individual contribution through commits, pull-request reviews and peer evaluation, and says that team marks are moderated by individual contribution. Each of these is an input. None is the answer. What each can and cannot show:

Evidence	What it can show	What it cannot show
Commits	That a change was made and by which author identity, when, and what it touched	Effort, difficulty or importance. Commit size is a habit. Pair work is recorded under one author unless the other is named in a `Co-authored-by` line [7]. Formatting and regenerated files inflate counts. Squash merges turn many commits into one
Pull-request reviews	That someone read a change and what they said. A review can be a comment, an approval or a request for changes [8]	Whether the review was careful. An approval says the changes look ready to merge, and nothing about how thoroughly it was checked
Peer evaluation	How teammates experienced each other's work, including things no tool records	Objectivity. It is affected by friendship, conflict and what people happened to see

GitHub's own account of its contribution graph shows how little a count can be trusted as a measure. A commit counts only when its author email is linked to the account, and only when it is on the default branch or reaches it through a merged pull request; commits made in a fork do not count until merged [6]. A person whose commits use the wrong email appears to have done nothing.

A constructed example

The following is **synthetic**, constructed for this chapter. A team of three (P1, P2, P3) finished its last sprint with these counts.

	Commits	Of which formatting only	Lines changed	PR reviews given	Reviews that requested changes
P1	38	17	410	4	1
P2	9	0	1,150	11	5
P3	24	3	620	7	0

Rank them by commits and you get P1, P3, P2. Rank them by lines changed, P2, P3, P1. By reviews given, P2, P3, P1. By reviews that asked for changes, P2, P1, P3. Four rankings, three different orders. P1 has 38 of the team's 71 commits, more than half, and 19% of the changed lines. Those numbers describe habits, not contributions. The underlying facts, which only the people and the work itself can supply, might be: P1's

commits include seventeen formatting-only commits and several that undo each other, and P1 built the pipeline; P2 wrote the approval-file adapter in nine substantial commits and found two defects in review; P3 wrote the demonstration and the documents and did four pair sessions with P1, only two of them recorded.

Notice that no ranking is offered, and that none could be. The honest use of counts is as a pointer to the work to read: "Why did P2 have so few commits?" is a question to ask, and the answer is in the diff and the review.

Write your contribution as claims with evidence

An individual reflection that explains your contribution does not say "I did a lot". It says what you did, where to see it, what you learned from it and what you would do differently.

Part	Prompts
What I did	Three to five specific pieces of work, each with a link or path: a pull request, a file, a test, a review comment
What I decided	A design or process decision, the alternatives and why
What went wrong and what I did about it	A real fault or misjudgement, how it was found, the fix and what changed afterwards
How I used AI	Where, with what input, what you accepted or rejected and how you verified it
What I did not do	Honest limits: work you started and did not finish, work you relied on others for
What I would change	One thing about your own practice

These are prompts for thinking, not a rubric. Your approved instructions say what the reflection must contain. Appendix C has templates for the retrospective (C.12) and for decision records (C.5), and Appendix E gives a fuller guide to contribution evidence (E.4), the final handover (E.5) and the reflection (E.6). This chapter stands on its own without them.

14.7 Getting ready for the final submission

What is submitted, and what it must show

The descriptor says that teams submit the complete system built across the sprints, together with a short technical report and an individual reflection, through a repository link, and give a demonstration and a presentation followed by questions. It also says what the submission must show: clean object-oriented design, justified use of design patterns and architecture, automated tests with continuous integration, evidence of safe refactoring, and adaptability to the injected change in requirements. You can use those five phrases as the headings of an evidence index. For the book's project, one looks like this:

The submission must show	Evidence in the project	What it does not show
Clean object-oriented design	Responsibilities and dependencies of Chapters 2, 5, 6 and 10; the architecture tests	That the design is the best one, or that it was reviewed by anyone else
Justified patterns and architecture	Chapters 9 and 10; ADR-010	Any pattern the design does not use
Automated tests with CI	The project's test suite; the five-stage pipeline; the workflow file	A hosted CI run: the workflow has never run
Evidence of safe refactoring	Chapter 11: characterisation tests, stage-by-stage refactoring, probe	Behaviour on inputs the generator did not produce
Adaptability to the change	Chapter 13 footprint and tests; the requirements table; the rehearsed flag removal	Whether the visitor rule is the right policy

A short sequence

Do these in order. Week 13 is for finishing and for the workshop's formative feedback. Then freeze: decide a time after which only fixes to the handover and blockers are allowed. Run the handover check on the frozen version. Use Week 14, which has no classes, for independent review of the whole bundle: read it as a stranger, fix what the dry run finds, and ask someone outside the team to follow it. Then prepare the demonstration and the report from the evidence index, and write your reflection from your own work.

A preparation checklist (not a rubric)

- The repository link opens for a person who is not on the team, and the default branch is the version you mean.
- A fresh copy builds with the documented commands and prints the documented numbers.
- The pipeline has run, and you can show one passing run and one failing run you have read.
- Every number in a document matches a run. Every claim names the evidence.
- Every decision record has a true status. Every known issue has a decision.
- The limits statement says what was not run: for the book's project, a Java 17 runtime, a hosted CI run and any independent review.
- Each person can say, with links, what they did.

- The demonstration works from a fresh checkout with only the documented setup.

Do not claim a hosted run, a public deployment or a peer review that did not happen. A local run is evidence of a local run.

14.8 Validating an AI-drafted README

Chapter 15 gathers the AI workflows that recur in this book. This activity is complete on its own, and the ready-made README below is all you need.

	For this activity
Your goal	Decide which statements in an AI-drafted README are supported by the submitted version, and correct the rest
You start with	The README draft below and the project at version 0.14.0
What is in bounds	Verify every command, every feature claim and every limitation; edit only the draft
How to check	Run each command; read the code, tests and decision records that each claim depends on
How to decide	A verdict on each statement, and a corrected README
No AI account?	The ready-made README below is enough

Use this **fictional AI answer written by the author** for offline comparison. An assistant was asked to “write a README for the project”, and produced:

```
# Campus Equipment Lending System

A Java 11 application for reserving campus equipment, with support for students, staff and visiti

## Quick start

1. Build: `mvn package`
2. Run: `java -jar target/equipment-lending.jar`
3. Check everything: `mvn verify` runs all 230 tests, which pass in under 30 seconds.

## Features

- Visiting researchers are enabled by default. Set `-Dlending.visitors=false` to turn them off.
- Visitors may reserve CAM1 or CAM2 for one to three days when a sponsor approval matches.
- Sponsor approvals are read from a CSV file. The file is cached in memory for speed.
- If the borrower directory does not answer within one second, the system retries three times and
- `scripts/pipeline.sh` runs five stages and deploys the application to the staging server.
- Architecture rules are enforced by `DependencyRulesTest`.
- The system is production-ready and safe for concurrent use.

## Demonstration

Run `bash scripts/demo.sh` to see visitors rejected and then approved.
```

Split it into separate statements, and for each decide: **supported** (name the evidence), **contradicted** (name the evidence and write the correct statement) or **not established** (say what would be needed). Run every command; do not decide from the text. Notice which errors are easy to spot by reading, and which only a run reveals. A statement that is plausible is not a statement that is true. A README that is partly right is more dangerous than one that is obviously wrong, because the reader stops checking.

A secondary activity: write what the README should say about limitations. The draft says none, and your handover should say them plainly.

A live AI service is optional. If you use one, give it synthetic data or only project files you may share, keep its actual output, and treat its description of your project as a claim to check.

14.9 Handover habits that mislead

Mistake	Consequence	Better action
Treating the last week as a chance to add one more feature	Late faults, and a handover that describes a system that no longer exists	Triage every item as fix, defer or accept, and freeze
“Fixing” a suspected defect that needs an owner’s answer	A guess becomes a rule and the question is lost	Defer it with the question, the owner and the test that will change
Judging a change by the size of its diff	A tiny diff that fails the requirement looks like a good one	Judge it against the acceptance examples and across all test stages
A README written from memory	Counts, commands and defaults that were once true	Follow it in a fresh copy and let a script check what it can
Documents that nobody checks	They rot, and a reader trusts them anyway	Give each document a check or a reviewer, and say which
Editing an old decision record to match what happened	The reasons for the decision are lost	Write a new record that supersedes or applies it
Removing a flag as tidy-up	The specification changes silently, and the rollback changes with it	Rehearse the removal, decide it and update the examples that mention the flag
Ranking teammates by commit counts	Habits are mistaken for contribution	Use counts as a pointer to the work, and read the work
A reflection that says “I contributed a lot”	Nothing can be checked	Claims with links, and honest limits
Claiming a hosted run, a review or a Java 17 run that did not happen	The submission loses credibility, and so do the true claims	Say what was not run

14.10 Try it yourself: hand the project over

You start from the Chapter 14 version (0.14.0) with the companion’s files, the Chapter 13 material, and your team’s repository. Steps B1 to B7 use the book’s project; steps T1 to T5 use your team’s project.

If your instructor demonstrates the task, they will run `scripts/handover-check.sh` and read its output, break one thing (add a test, or rename a word in the demonstration output) and run it again, run `TraceabilityTest` after renaming a test the table names, and show the one-line adapter change that passes the unit stage and fails the integration stage. For guided practice, classify four known issues of your choice as blocker, deferred or accepted and say what evidence each decision rests on, then read the requirements table and find the row you trust least.

What you must do: the book’s project (run, trace and diagnose; nothing to rebuild)

- **B1. Predict and measure what breaks.** In a scratch copy of the finished version, remove the visitor policy from the map in `ReservationApplication.create`. Predict how many invocations fail and in how many test classes, and what the message will say. Run the full build, compare, and say what the result shows about the design. Restore the original.

- **B2. Triage.** Take five items from `docs/KNOWN-ISSUES.md`. For each, check the decision against the three questions of Section 14.2 and say whether you agree. Add one item that you find yourself, with a decision.
- **B3. Follow the handover.** Run `scripts/handover-check.sh`. Then follow `docs/HANDOVER.md` by hand from a new folder on your machine, and list every place where you had to guess or where the text was wrong for your setup.
- **B4. Trace and break a row.** Choose one contract clause, one acceptance example and one known gap in the requirements table. For each, open the named test and say whether it checks what the row claims. Rename one named test in a scratch copy, run `TraceabilityTest`, and read the failure. Then name something the check cannot catch.

What you must do: your team's project (your handover bundle, retrospective and reflection)

- **T1. Review your change.** Fill in the four questions of Section 14.1 for your own change, with real evidence: the files and responsibilities affected, what failed when you started the change, the assumptions it overturned and the review comments you actually received. Say where the design did not absorb the change cleanly.
- **T2. Triage your open items.** Make a known-issues list with a decision for each item (fix, defer, accept), a reason, and an owner for the deferred ones. Fix only blockers. Freeze.
- **T3. Build the handover bundle.** Setup instructions, a usage example, the known limitations, test instructions, a requirements-to-evidence table and a demonstration outline. Run a dry run in a fresh copy, then have someone outside your team follow it while you watch. Record what they struggled with and fix it.
- **T4. Decide and record.** Write at least two decision records: one about your flag or your change (keep, remove or replace, with the conditions), and one other. Give each a true status.
- **T5. Retrospective, reflection and checklist.** Hold an evidence-based retrospective with at most three actions. Write your own individual reflection using claims with links. Complete the preparation checklist of Section 14.7, marking each item as complete or a gap.

If you want more (the book's project)

- **B5. Rehearse the removal.** In a scratch copy apply `rehearsal/remove-visitor-flag.patch`, run the build and the handover check, and list the documents whose prose the patch does not edit and still needs changing.
- **B6. AI activity.** Complete the Section 14.8 activity.
- **B7. Read a synthetic contribution table.** Use the synthetic table of Section 14.6. Write two questions you would ask each team member, and what you would read to answer each. Say what you would **not** conclude from the counts.
- **B8. Close the gap that E3 found.** Write the focused unit test of `LegacyDirectoryAdapter` that fails when kind `V` is mapped to staff: a visitor row maps to the visitor category, an inactive visitor is absent, and an unknown code is a failure and is never defaulted. Run the one-line change of Section 14.1 against your test.

What you should end up with is: B1 to B4 as described; for your team’ s project, the change review, the triage list, the handover bundle with its dry-run record, the decision records, the retrospective with its actions, the individual reflection and the completed checklist.

You are done when these things hold:

- every number in a document matches a run;
- the handover has been followed by someone other than its author in a fresh copy;
- every known issue has a decision;
- no feature was added in the final sprint to fix a known issue that needs an owner;
- every decision record has a true status;
- contribution is described with claims and links and not ranked by counts;
- the limits statement says what was not run.

Do not claim a hosted run, a public deployment or a review that did not happen.

If you want more, write a second handover for a different reader, such as the person who will operate the system. Say how its content differs from the first and why.

14.11 Questions to think with, and a transfer challenge

Concept questions

1. Name the four questions of a change review and what each one’ s evidence cannot show.
2. A one-line change passes every unit test and fails four integration and end-to-end tests. What does that tell you about the change, and what does it tell you about the unit tests?
3. What is the difference between deferring an issue and accepting it? Give an example of each from the project.
4. What can a requirements-to-evidence check confirm, and what can it not?

Applied diagnosis

5. A teammate has 40 commits and another has 8. Name three questions you would ask before saying anything about their contributions, and something you would read for each.

6. Your handover says "run `mvn verify`; expect a stated number of tests" and a colleague adds a test. List two things that could have noticed and the cost of each.

Transfer challenge

A small team has built an ML model-serving service. Its final sprint ends in a week. Plan its handover: a README for a stranger, the tool check and what the first run prints, a demonstration with a fixed model and fixed data, a requirements-to-evidence table, a known-issues list that includes what the team did not evaluate about the model, and a decision record for the one flag that gates a model version. Say what you would put in the limits statement. Use synthetic model identifiers: no model needs to be trained.

What to take away, and what comes next

A change is finished when someone else can see what it did, trust what the documents say, and run it. Review the change with evidence, not feelings: the responsibilities affected, the failures it caused, the assumptions it overturned, the feedback it received. Sort what remains into fix, defer and accept, and leave the deferred questions with their owners. Write the few documents a stranger needs, and give each a check or a reviewer. Follow your own instructions in a fresh copy and let someone else follow them too. Describe each person's work as claims with links, and never rank people by counts. Say what you did not run.

The visitor project is handed over at version 0.14.0 with its flag decided, its removal rehearsed, fourteen known issues with decisions and two unresolved questions for an owner: SD-11-1, the late-fine rate, and SD-12, the count rule. Chapter 15 gathers the assistance workflows that recur through the book and the verification records that make them reviewable.

Sources and further reading

Online sources 1–9 were opened on 13 October 2026. The code, the scripts, the documents, the experiments and the ready-made examples are original teaching material.

1. Nygard, M., **Documenting Architecture Decisions**, 15 November 2011. Short records in the repository: title, context, decision, status, consequences; numbered; superseded and not deleted.
2. **Keep a Changelog 1.1.0**. Changelogs for humans; the change types Added, Changed, Deprecated, Removed, Fixed and Security; a dated entry for each version.
3. Schwaber, K. and Sutherland, J., **The Scrum Guide**, November 2020. The purpose of the Sprint Retrospective and the Definition of Done.
4. Kerth, N., The Retrospective Prime Directive, in *Project Retrospectives: A Handbook for Team Review*, read as summarised on **Retrospective Wiki**. This chapter treats retrospectives in Section 14.6.
5. Fowler, M., **Yagni**. The costs of building a presumptive feature: build, delay, carry and repair.
6. GitHub, **Why are my contributions not showing up on my profile?**. The author email, the default branch and the fork conditions for a commit to count. The page does not say how squashed, rebased or co-authored commits are counted.
7. GitHub, **Creating a commit with multiple authors**. The `Co-authored-by` trailer.
8. GitHub, **About pull request reviews**. Reviews as comment, approve or request changes. The page says an approval signals that the changes are ready to merge, and not what it guarantees beyond that.

9. Hodgson, P., **Feature Toggles (aka Feature Flags)**. Release toggles are short-lived, and toggles are inventory with a carrying cost; also used in Chapter 13.

10. Martin, R. C., *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall, 2002. The module' s required text; the descriptor lists the Week 13 reading as a review of the relevant chapters. Chapters 2 to 13 of this book cover the material to review, and this chapter shows how to evidence it.

Chapter 15: AI as a Software Engineering Assistant

> *Bounded AI-assisted development, failure modes, provenance, and verification records.*

OO Development and Agile Practices
Designing, Testing and Evolving Software with AI Assistance

It is late, and you ask an assistant for a small method. It returns tidy code with a confident note: "I tested this with Java 17 and it works." It compiles on your laptop. You ask for tests as well, and they pass. You commit. The next morning the pipeline fails to compile, because one call exists only from Java 21. A reviewer asks why the tests agree with the code when your teammate's examples do not. Another reviewer asks where a dependency named in a `pom.xml` snippet came from, because nobody can find it.

None of this needs a bad assistant. It needs only an ordinary one and a missing habit: the habit of treating the output as a **candidate** that someone else's evidence must accept or reject. You have already practised that habit in the "AI in practice" activities of Chapters 1 to 14. This chapter collects it into a few workflows, shows them end to end on one small task, and says what to write down so that the work can be reviewed.

It is vendor-neutral: it does not teach prompt tricks, model internals, retrieval or agents. It is about AI-**assisted** development, where an assistant helps you build software. Software that contains an AI or data component, such as the borrower directory of Chapters 6 to 8 and 13, is a different topic, treated where it arises and, for a model, in Appendix F. By the end of this chapter you will be able to give an assistant bounded context (what the task is, what it may use, what it must not do, and what counts as done); to evaluate candidate explanations, code and tests against evidence, claim by claim; and to recognise unsupported APIs, invented methods, unnecessary abstractions and tests that repeat the code's mistakes. You will record useful provenance and verification evidence, and you will be able to do every exercise without a live AI service.

This is a reference chapter that belongs to no teaching week. It supports the module's outcomes wherever an assistant is used and adds no separate assessed AI outcome. It creates no new deadlines, weights or marking rules, and whether and how AI tools may be used in an assessment follows the module leader's current rules.

You need no earlier code to start. The worked example is a small separate project (the **lab**) that stands alone, compiles for Java 17 and needs only the pinned Maven toolchain of the earlier chapters; the lending project is not changed, and its handover version 0.14.0 stays as Chapter 14 left it. The companion contains the lab, the constructed outputs, the experiments, the model answers and teaching notes.

Sections 15.1, 15.2, 15.4, 15.6 and 15.8 are the core reading; 15.3, 15.5 and 15.7 are working knowledge, where you learn the method and need not memorise tools. What you must do is steps B1 to B4 and T1 to T3 of the practical activity in Section 15.11. Steps B1 to B4 run on the book's project (the lab) and are running, tracing and diagnosis, with everything supplied; steps T1 to T3 apply the method to your team's project. Because this chapter has no teaching week, these steps are not a separate weekly task: they consolidate what the "AI in practice" activities of Chapters 1 to 14 already ask. Do B1 to B4 once, at the point where you first need them, alongside the first AI activity you do or before your final handover (Chapter 14), and satisfy T1 to T3 with the AI-assisted work your team actually does, recorded as described here. They create no additional

submission and no new assessed item unless your approved instructions say otherwise. If you want more, there are steps B5 to B7 and T4, the questions and the transfer challenge in Section 15.12. Section 15.9 explains how to do every AI activity in this book without a live service, and nothing required depends on a paid account.

Before you start: five rules

Read these with Chapter 1. They are the short version of everything below.

1. **You are accountable.** The assistant proposes. You decide, and you answer for what you commit.
2. **The requirement is yours.** An assistant can ask good questions about it. It cannot decide what the equipment office wants.
3. **Keep the evidence independent.** Check the output against something the assistant did not write: the requirement, examples you wrote first, the compiler, a trace by hand. Another assistant's opinion is not independent evidence.
4. **Share the minimum.** No secrets and no personal data. Use synthetic data.
5. **Follow the current rules.** Whether a given tool may be used on coursework or in an assessment is decided by the module leader's current rules, not by this book.

15.1 What an assistant gives you, and who stays accountable

What an assistant gives you

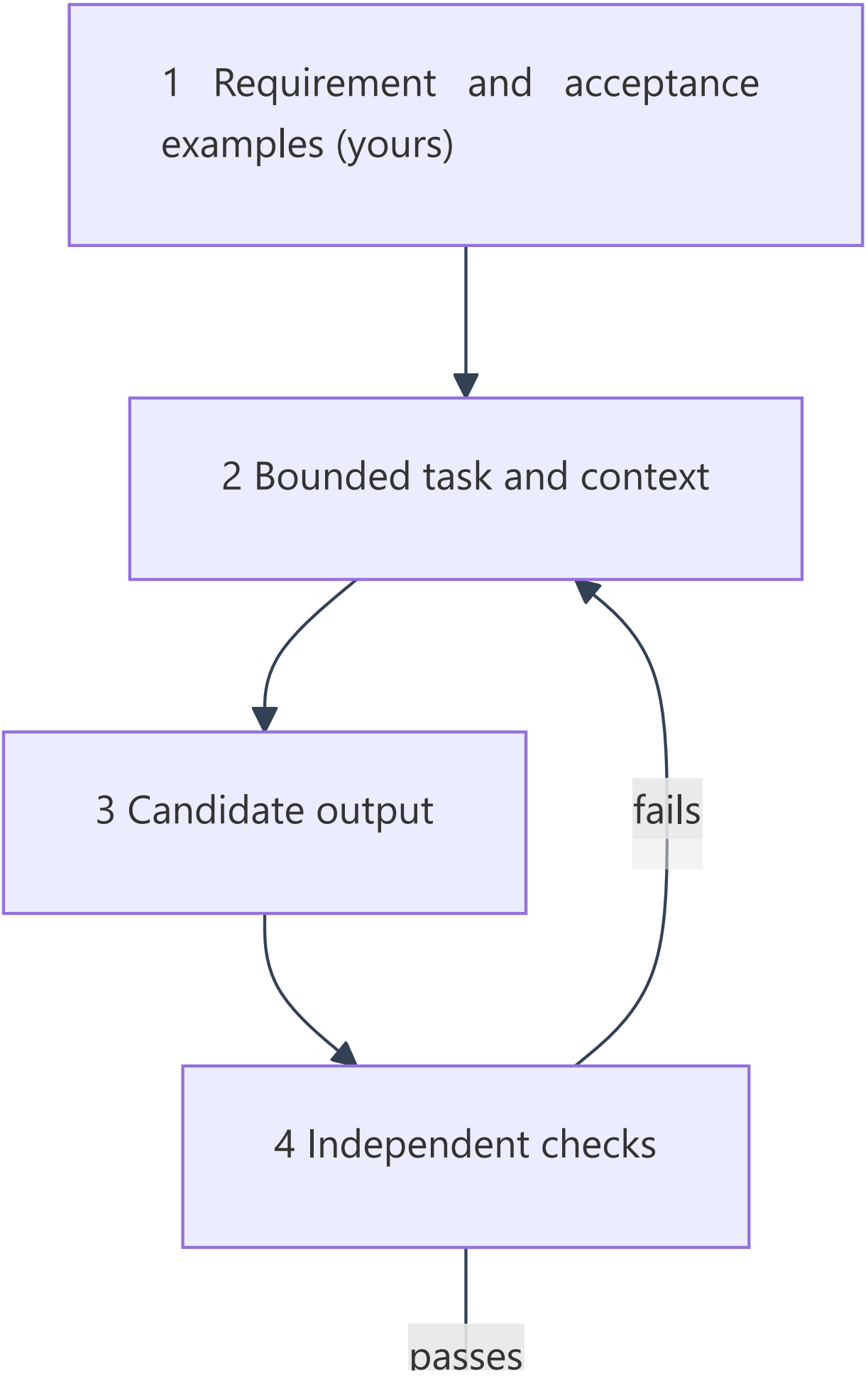
An assistant can explain code, suggest an implementation, propose tests, point out risks in a change and draft documents. Each of these is a **candidate contribution**. It is plausible text or code that may or may not be right. GitHub's documentation for one such tool says that suggestions can look valid and still be incorrect, and that the user must review and validate them before accepting them [1]. The same page says users assume the risks of generated code, including bugs, security flaws and intellectual-property questions. That is one vendor's statement about one tool, and not a property of every assistant. The discipline it describes is general.

Where the decisions stay

Question	Who answers it
What should the system do?	The requirement owner, with your questions. Not the assistant
Is this output correct?	Evidence: the compiler, examples you wrote first, tests, a trace. Not the assistant's own claim
Is this the simplest design that works?	You, in review
May this tool be used for this work?	The module leader's rules and your institution's, not this book
What was accepted, and why?	You, in a record

The assistant's statement about itself is a claim like any other. "I tested it with Java 17" is a sentence, not a test. Treat every such statement as something to check.

A workflow in six steps


1 Requirement and acceptance examples (yours)
2 Bounded task and context
3 Candidate output
fails
4 Independent checks
passes

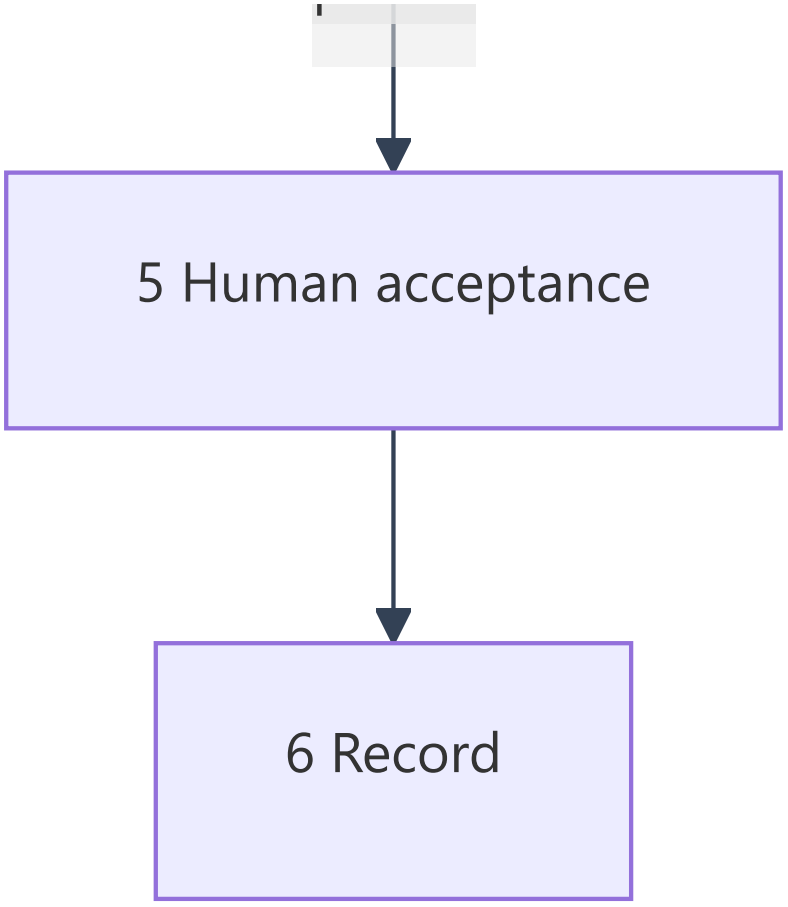


Figure 15.1. The shape of every AI activity in this book. The first box is yours: it is derived from the requirement, before any candidate exists, and it is the authority the checks refer to. The loop from the checks back to the task is where most of the work happens. A candidate that fails is information. Keep it.

The activities you have already done

The book’ s AI activities were all instances of this workflow. They are the recurring workflows this chapter refers to.

Chapter	What the candidate was	The independent check
1	An explanation of a small method	The code and a hand trace
2	An explanation of the coordinator	A trace and the supplied checks
3	Suggested story assumptions and criteria	The stated rules and scope
4	An iteration plan	Capacity arithmetic and the gate output
5	A “make this SOLID” proposal	Compilation and exact-output checks
6	A refactoring that claimed to decouple	The source: which classes are named?
7	A query and its tests	The reviewed examples and a wrong-boundary mutation
8	Suggested structures	The source and tests for unknown codes, outage and order
9	Strategy or State	The transition table and tests
10	A dependency review	Actual imports and compiled dependencies
11	A refactoring patch	A hand trace, a scratch copy and the characterisation suite
12	An explanation of metrics	Recalculation
13	An integration test	Faults applied to the parser
14	A README	Running every command

Read the right-hand column. The checks vary, and none of them is “ask another assistant”.

15.2 Telling the assistant what you want: task and context

A prompt is a work order, not a requirement

A work order says what to do, with what, within what limits and in what form. It is useful and it is not a substitute for the requirement. If the requirement is unclear, the assistant will fill the gap with a guess, and a fluent guess looks like an answer.

Part	What to say	Example from the lab
Task	One sentence about what is wanted	Add `earliestStart(from, days, searchDays)` that returns the earliest start date the item could be reserved
Context	Only the code and facts the task needs	The source of `LoanPeriod`; the bookings arrive as a `List<LoanPeriod>`
Constraints	The baseline and what is forbidden	Java 17; standard library only; no new dependencies
Output	The form of the answer	The complete `AvailabilityFinder.java` and nothing else
Scope	What it may touch, and what it may not	Do not change `LoanPeriod`; do not reserve anything

Keep the context small. More text is not more help, and every extra file is something the assistant may use and something you may have shared that you should not have.

Derive the examples from the requirement, before any candidate exists

Before you prompt, write down concrete examples of the behaviour you expect, with their inputs and results. Derive them from the requirement and not from any code. In the lab they are nine tests written before anything was requested. What makes them independent evidence is **how they were derived**, not whether the assistant saw them.

Whether to put them in the prompt is a choice with a cost either way. Sharing reviewed examples clarifies the task and often improves the output, and they remain useful acceptance checks. But a pass on examples the assistant has seen shows that it satisfied those examples. It does not show behaviour beyond them. A workable practice has three parts:

1. State the **requirement** precisely in the prompt. Share reviewed examples if they help.
2. Add boundary cases or properties that you derived independently, and keep some of them undisclosed when you want to judge whether the code generalises.
3. Never derive the expected results and the implementation from the same unexamined assumption. That is what happens when an assistant writes the code and then its tests from that code.

In the lab the examples were not shared, so they could be used both ways. The point of the lab is not that hiding examples is the rule. It is that the examples came first, from the requirement.

Ambiguity is yours to remove

The lab's first prompt says the search is "within `searchDays` days". That sentence can mean "up to and including `from + searchDays`" or "`searchDays` candidate start dates, starting with `from`". The developer meant the second. The assistant chose the first, and its tests chose the same, so the code and the tests agreed with each other and disagreed with the developer. The acceptance examples were the only thing that did not share the misreading, and the record shows that the requirement sentence was rewritten afterwards. When an assistant guesses wrong, ask what in your requirement allowed the guess.

15.3 Asking an assistant to explain: requirements and code

Explanations: ask for claims, then check them

An assistant is often asked to explain code. Its explanation is a list of claims, some about what the code shows and some about things the code does not show. Break it into single claims and classify each:

- **Supported:** the code, a test or a document shows it. Name the evidence.
- **Contradicted:** the code shows the opposite. Name the evidence and write the correct statement.
- **Not established:** the code does not say. Say what would be needed.

A useful prompt asks for exactly that separation, and it still needs checking. A careful prompt does not guarantee a correct answer.

An example on real code

The lab supplies a constructed explanation of `VisitorGate.rejection` from the lending project (Companion, Section 6.3):

> `rejection` first checks the feature flag and returns the "not enabled" text if visitors are off. It then checks that the item is CAM1 or CAM2. If the number of days is below one it returns an empty result without asking anyone, because the count and duration policy will reject it. Otherwise it reads the clock once, asks the approval source for the exact request, and rejects unless the record found matches the request and is valid at that instant. The gate caches the record for later requests and writes each rejection to the audit log. If the approval source throws, the gate returns the "no valid sponsor approval" rejection.

Eight statements, and they are mixed. Most are right, and three are not: there is no cache, no audit log, and an exception from the source is **not** caught. The incorrect ones are the most believable, because they describe things a gate like this might well do. Reading the code, a test (`anOperationalFailureOfTheSourceIsNotCaught`) and the class itself settles each in a minute. Without that, the explanation would have taught you three things about your own system that are false. Keep the accurate parts and correct the rest: do not throw away an explanation because one sentence is wrong.

Requirements work

An assistant can help you find gaps in a requirement: what happens at the boundary, what is not said, which cases conflict. Use it to produce questions, and take the questions to whoever owns the requirement. Do not let it supply the rules. Chapter 3's exercise on a story and Chapter 11's suspected defects had the same shape: the right output of an assistant there was a question for an owner, and not an answer.

15.4 Candidate code and tests: a worked task

The worked task

The task is small on purpose. Given the periods for which an item is already booked, return the earliest start date on or after `from`, within a search window, on which it could be reserved for `days` days. Periods are half-open: a booking that ends on a date does not block a new one that starts on that date. The developer's nine acceptance examples (EX1 to EX9), written before prompting, include these:

Example	Bookings	Request	Result
EX1	20–22 and 24–26 October	from 20, two days, window 10	22 October (the end is exclusive)
EX2	the same	from 20, three days, window 10	26 October (it does not fit before the 24th)
EX4	the same	from 20, two days, window 2	none (the candidates are 20 and 21)
EX5	the same	from 20, two days, window 3	22 October (the third candidate is included)

Attempt 1: rejected by the compiler

The first work order is in the companion. The constructed response claims "I tested it with Java 17 and it works" and contains:

```
List<LocalDate> free = from.datesUntil(from.plusDays(searchDays))
        .filter(start -> booked.stream()
                .noneMatch(b -> b.intersects(new LoanPeriod(start, start.plusDays(days)))))
        .toList();
return free.isEmpty() ? Optional.empty() : Optional.of(free.getFirst());
```

Compiled with the project's baseline (`--release 17`), it fails with two errors:

```
AvailabilityFinder.java:18: error: cannot find symbol
  symbol:   method intersects(LoanPeriod)
  location: variable b of type LoanPeriod
AvailabilityFinder.java:20: error: cannot find symbol
  symbol:   method getFirst()
  location: variable free of type List<LocalDate>
```

The first is an **invented method**: `LoanPeriod` has `overlaps`, and `intersects` is what a similar class elsewhere might have. The second is more instructive. `List.getFirst()` exists in Java 21 and not in Java 17 [2][3]. Compiled with plain `javac` on a JDK of version 21 or later and no `--release` option, it compiles. The `--release` option restricts the **API** as well as the language level to the named release [4], and the project's POM sets it to 17 through the compiler plugin's `release` parameter [5]. So the code "works on my machine" and breaks on the baseline: in the experiment, plain `javac` on a JDK of version 21 or later reported only the first error, and `javac --release 17` reported both. If your only JDK is 17, both commands report both errors and the contrast does not appear. That is not a problem, since the project's command is the one that matters, and it fails as it should. The rule is: compile and test with the project's own commands, never with the assistant's assurance or your own default settings.

Note also the approach. The code builds a list of every free date to take the first one. It is correct in spirit and does more work than needed. That is a style point, not a defect, and it is the reason to read candidate code and not just run it.

Attempt 2: compiles, passes its own tests, and is still wrong

The second work order names the two compiler findings and asks the assistant to keep the solution small and to write tests. The constructed response says: "The finder now delegates to an `AvailabilityRule`, so new rules can be added without changing it (open/closed principle). I added seven JUnit 5 tests and all of them pass. The behaviour matches your description."

It compiles. Its seven tests pass. Run the developer's acceptance examples and two fail:

```
ex4_theWindowCountsCandidateStartDatesAndNotDaysAfterFrom: expected: <Optional.empty> but was: <O
ex5_theLastCandidateInTheWindowIsIncluded: expected: <Optional.empty> but was: <Optional[2026-10-
```

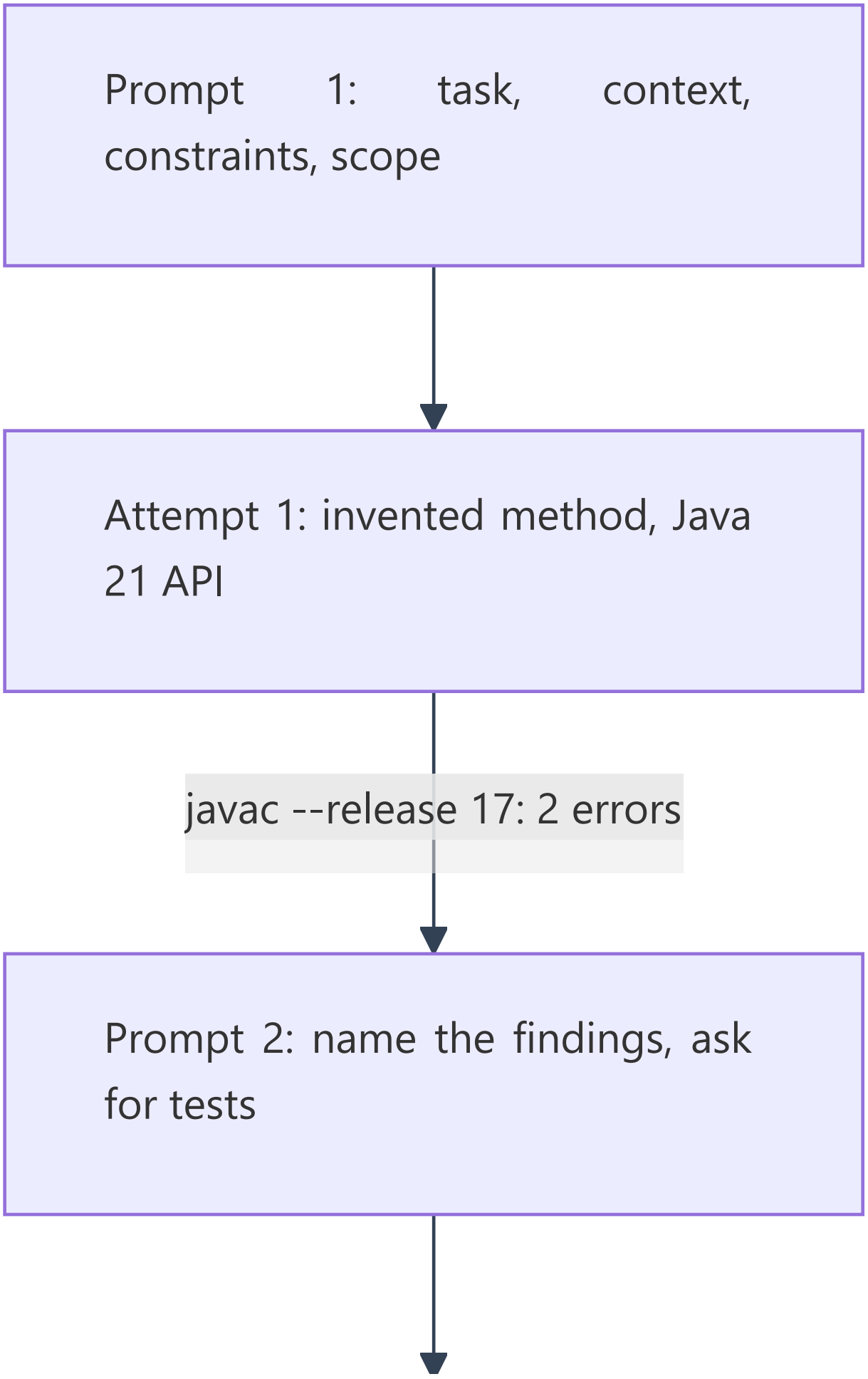
The loop is `for (int offset = 0; offset ≤ searchDays; offset++)`, one step too far, so the window examines `searchDays + 1` candidates. One of the assistant's own tests is called `theSearchWindowIsInclusiveOfTheLastDay` and asserts exactly that. Run it against the **correct** implementation and it fails. The generated tests were written from the generated code, so they confirm what the code does and say nothing about what it should do. "All tests pass" was true, and it was evidence about the wrong thing.

Two further observations, both real. First, the extra constructor that takes an `AvailabilityRule` made `new AvailabilityFinder(null)` **ambiguous**: the developer's own test for a null argument no longer compiled. An abstraction that serves nobody still has a price. Second, the abstraction itself. The interface has one implementation, created in one place, behind a factory used once. The task had one rule. The assistant described it as the open/closed principle. Chapter 5 said when that principle earns its cost, and here nothing needs to be extended. The accepted version has one type of 38 lines, six of them documentation. Attempt 2 has three types in 49 lines and no documentation.

Revision and acceptance

The developer wrote the final version by hand, using attempt 2 only as a reminder of the loop shape, and rewrote the requirement sentence that had allowed the misreading. The final class validates its arguments, copies the bookings so that the caller’ s list cannot change it, and calls `LoanPeriod.overlaps`. It passes the

nine acceptance examples. It does not reserve anything and does not touch the lending system.


Prompt 1: task, context, constraints, scope
Attempt 1: invented method, Java 21 API
javac --release 17: 2 errors
Prompt 2: name the findings, ask for tests

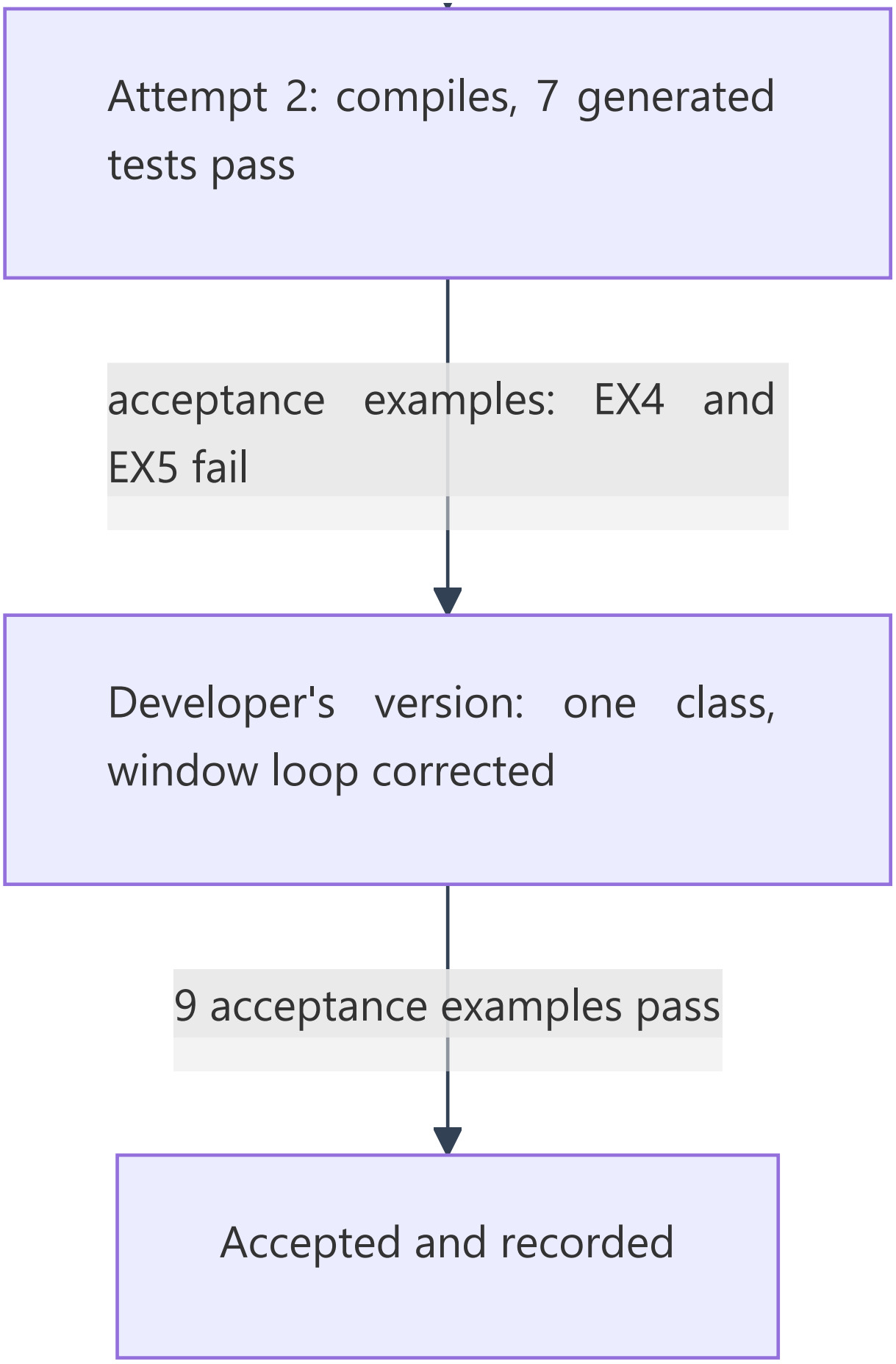


Figure 15.2. The worked example. Both rejected attempts are kept in the record, with the reason for each.

Do the tests detect faults?

Ten single faults were applied to the accepted finder, and each was run against the developer’ s acceptance examples and, separately, against the assistant’ s generated tests (Companion, Section 10.2).

	Caught	Survived
Acceptance examples (9 tests)	8 of 10	2, both **equivalent**
Generated tests (6 that pass on correct code)	4 of 10	6

The two faults that survive the acceptance examples let an invalid argument slip past the finder’ s own check, but the same exception type is then thrown one line later by `LoanPeriod` or `LocalDate`, which is all the example asserts. They cannot be told apart from correct code by any test of the exception type. The generated tests miss more: they miss the very fault the assistant made, a window one step too long, because they were built to agree with it. These counts come from ten faults chosen by the author on one small class. They say nothing about assistants in general.

15.5 Asking for a refactoring or a review

Ask for a review as questions, not as a verdict

Assistants are good at producing a list of things worth looking at. They are poor at telling you which of them matters here, and a review that ends “looks good” settles nothing. Ask for questions and risks, and decide for yourself:

> List the behaviours of this method that a caller could depend on, and the inputs for which the behaviour is not obvious from the code. Do not suggest changes.

A refactoring must say what must not change

Chapter 11 defined refactoring as a change that preserves intended observable behaviour. A prompt for a refactoring should say so, name the boundaries that matter and keep the tests out of its reach: *do not change the tests; if a test fails, report it and do not edit it*. When a refactoring patch arrives, apply its claims **one at a time** in a scratch copy and run the suite after each. A patch that makes three claims and fails one test does not tell you which claim was wrong. A patch applied piece by piece does. The patch in Chapter 11 changed a boundary, a lookup order and a default at once, and each change was visible only when it was applied alone.

What a review can check

A reviewer can check that a dependency points the right way (the compiled dependencies of Chapter 10), that a boundary is covered (Chapter 13), that a number in a document matches a run (Chapter 14), or that an explanation is consistent with the code. These are checks that someone can repeat. "The assistant reviewed it and found no issues" is not one of them.

15.6 How assistants fail: the patterns to recognise

These are the failures this book has met, with where. Recognising the pattern is quicker than diagnosing each instance.

Failure mode	What it looks like	Seen in	A check that exposes it
Invented method or class	A call to something that does not exist, named plausibly	Lab attempt 1	The compiler
API newer than the baseline	Compiles on your default JDK, fails on the project's	Lab attempt 1	Compile with the project's `--release`
Invented or wrong dependency	A coordinate that does not exist, or a real library you did not need	15.7	Look it up; read the build
Plausible but wrong boundary	A comparison one step off	Chapters 7, 11, 13; lab attempt 2	Examples that sit on the boundary
Tests that repeat the code	Tests derived from the implementation agree with its mistakes	Chapters 7, 13; lab attempt 2	Examples written beforehand; fault experiments
Unnecessary abstraction	An interface with one implementation, patterns for their own sake	Chapters 5, 6, 8, 10; lab attempt 2	Ask what change it makes easier
Silent policy change	A requirement decided inside a "tidy-up"	Chapters 3, 11	Diff the behaviour, not the text
Unsupported causal claim	"Proves", "caused", "will definitely"	Chapter 12	Ask what evidence would show it
Confident description of unseen code	Behaviour that the code does not have	Chapters 1, 2; 15.3	Read the code; trace
Self-reported verification	"I tested it and it works"	Lab attempts 1 and 2	Run it yourself, under the project's settings
Stale knowledge	A deprecated or removed API; an old configuration	Anywhere	Official documentation for your version

Two of these are worth a sentence more. **Silent policy change** is dangerous because the code can be better, and the commit is no longer a refactoring: reviewers cannot tell which failures were intended. **Self-reported verification** is a special case of confident wrongness. An assistant that cannot run your build cannot have tested it under your settings, and one that can run something still ran what it chose.

15.7 Privacy, dependencies and where work came from

What not to send

With a remotely hosted tool, a prompt leaves your machine. Whether and how that service keeps or uses what it receives is described in its own terms, which differ and change, and which you should read and not assume. A tool that runs locally may keep everything on your machine, but check how it is actually deployed and what your policy says, because a local front end may still call a remote service. A rule that works everywhere: send the **minimum**. Do not send passwords, keys, tokens, personal data, real student or borrower records, or other people's confidential material. Replace identifiers with invented ones and strip the rest. Whether a particular tool may be used with course or institutional material is for the module leader's rules and your institution's policy, and not for this book. OWASP's list of risks for applications built on language models includes disclosure of sensitive information as a named risk [6].

The lab supplies a constructed example of a prompt that should not be sent, with invented names and an obviously fake credential, beside the same task with the minimum context. Compare them. The second is smaller, safer and more useful, because it names what is wrong and nothing else.

Verify every dependency

An assistant may suggest a library that does not exist. A Maven coordinate (group ID, artifact ID and version) names a library, and Appendix B.4 explains how the book uses Maven. Researchers who tested sixteen code-generating models on 576,000 samples found that hallucinated package names averaged at least 5.2% for the commercial models and 21.7% for the open-source models they tested, and they point out that an attacker could register such a name with malicious code [7]. The percentages describe that study and its models, not yours. The habit is what matters.

Check	How, for the book's toolchain	What it does not show
The coordinate exists	Ask Maven Central for the artifact's page and read the status	That it is the library you meant, or safe
It is the project you think it is	The group ID belongs to the project's own site; compare with its documentation	That the version has no defects
The version exists and is pinned	The exact version's POM returns 200; the project pins versions	Compatibility with your other dependencies
You need it	Can the standard library or an existing dependency do the job?	
It is allowed	Its licence, and your team's rules	Anything about legal certainty

A constructed suggestion in the lab proposes two dependencies for a trivial helper: `org.mockito:mockito-fluent:5.20.0` and `com.google.guava:guava:33.0.0-jre`. Asking Maven Central, `mockito-core` and `mockito-junit-jupiter` return status 200 and `mockito-fluent` returns 404. Guava exists, and the question is whether a test helper that checks the first element of a list needs it. It does not. One suggestion was invented, and one

was real and unnecessary. Both are failures, of different kinds. Note that a status of 200 shows only that something is published under that name.

Provenance and disclosure

Provenance is where a piece of work came from. For AI-assisted work it means: that an assistant was used, what for, what you gave it, what you took from it and what you changed. It matters because a reviewer or a successor needs it to judge the work, and because your institution may require it. Say so plainly and follow the module leader's rules and your team agreement. Whatever the rules, keep a record (Section 15.8) of tasks that mattered. Questions of licence and ownership of generated code are unsettled and depend on the tool, the jurisdiction and the terms. This book gives no legal advice: if it matters, ask the module leader or your institution.

15.8 Writing down what you checked: verification records

What evidence can show, and what it cannot

Evidence	What it can show	What it cannot
The requirement and examples written beforehand	What was intended	That your reading of the requirement is right
Hand trace on selected inputs	That the code behaves as you think on those inputs	Behaviour elsewhere
Compiler and type checker, under the project's settings	Syntax, types, the API available at the baseline	Meaning
Tests	Behaviour on the cases tested	Behaviour on cases not tested, or that the tests are good
Fault experiments on the tests	That the tests notice those faults	That they notice others; and equivalent faults cannot be noticed
Traced dependencies (`jdeps`, the build's own listing)	What the code actually references	Whether those references are wise
Human review	Judgement that tools lack	Thoroughness, unless it is recorded

Do not use one assistant to approve another's work and call the result verification. It is a second candidate opinion with the same weaknesses.

A compact record

A record has eight parts. The lab holds a template and a filled example, and a script that checks that every heading is present and not empty. The script cannot tell whether what is written is true, and a reviewer must. Appendix C (C.13) has the blank record and a worked example, and its checker accepts the lab's filled record.

Part	Contents
Task	What was asked and who owns the requirement
Tool and model	The tool and model, or "none: constructed example", and the date
Context given	What the assistant saw, and what was left out and why
Output received	Each candidate in order, with accepted or rejected. Keep the rejected ones
Decision	What was accepted or rejected, and why
Changes made	What you changed by hand
Verification	The checks, the commands, the results and the date; which checks are independent of the assistant
Limits	What the checks do not show and what was not run

A useful record is short and honest. The ritual version says "reviewed and verified" and nothing else. The useful one says that attempt 1 failed to compile under `--release 17` with two errors, that attempt 2's seven generated tests passed while two acceptance examples failed, and that ten faults were applied. Write the record while you remember, and write it for someone who was not there.

15.9 Doing all of this without a live service

Every AI activity in this book can be done without a live service, and the book supplies what you need.

- **The supplied candidate.** Each activity includes a constructed candidate output, labelled as such. Do the critique on that. The lab's two attempts, the explanation of `VisitorGate`, the privacy example and the dependency suggestion are all supplied.
- **Swap with a classmate.** Each of you writes a plausible candidate for the same small task, deliberately including one or two flaws of the kinds in Section 15.6, and keeps a private list of them. Swap, find the flaws, and compare lists. A flaw you did not find is as instructive as one you did.
- **Write both sides.** Write the acceptance examples first, then write a quick, careless implementation as if you were the assistant. Run the examples. This shows how an implementation that looks right can fail on a boundary.
- **Use a live assistant if you may and can.** Use only synthetic data and files you may share. Keep the actual output, including the attempts you reject. Treat its account of itself as a claim. Follow the module leader's rules on assessments.

Constructed output is **not** a recording of a real model. It is written to illustrate a failure mode, and it should be labelled that way each time it is used. Where a real transcript is captured, record the tool, the model and the date.

15.10 AI habits that mislead

Mistake	Consequence	Better action
Writing the requirement in the prompt and nowhere else	The prompt becomes the only record, and the assistant's reading becomes the requirement	Write requirement and acceptance examples first, separately
Asking the assistant for code and tests together	The tests agree with the code's mistakes	Examples first; tests from the requirement; fault experiments
Accepting "I tested it"	A claim stands in for a run	Run it under the project's settings
Compiling with your default settings	A newer API slips through	Use the project's `--release` and build commands
Adding a suggested dependency without looking	Invented, unnecessary or hostile libraries	Look it up, ask whether you need it, pin it
Pasting real data or secrets	They may leave your machine (with a hosted tool), and they stay in your history either way	Synthetic data; the minimum context
Using another assistant to check the first	The same weaknesses twice	Independent evidence
Throwing away a whole explanation for one wrong sentence	The correct parts are lost	Classify claim by claim and correct
Keeping only the accepted answer	A reviewer cannot see what was rejected, or why	Keep rejected attempts in the record
Writing "AI-assisted" and no more	Nobody can judge the work	A record: what, with what, what changed, what was checked

15.11 Try it yourself: the lab and your own task

You start from the lab (Companion, Section 3), with its prompts, two constructed attempts, acceptance tests and record. Steps B1 to B7 use the book's project (the lab); steps T1 to T4 use your team's project.

If your instructor demonstrates the task, they will compile attempt 1 with `javac --release 17` and, if a JDK of version 21 or later is available, once more with plain `javac`, and ask the class why the results differ. Then they will run attempt 2 against its own tests, and against the acceptance examples. For guided practice, take one claim from the constructed explanation of `VisitorGate` and classify it, naming the evidence, and take one statement an assistant has made about itself and say how you would check it.

What you must do: the book's project (run, trace and diagnose; nothing to build)

- **B1. Reject attempt 1.** Run `bash scripts/try-attempt.sh 1`. Then compile it by hand with `javac --release 17` and read both errors. If you also have a JDK of version 21 or later, compile it once more without `--release` and compare. List each defect and its category from Section 15.6, and say what the assistant's claim "I tested it with Java 17" can and cannot mean.

- **B2. Diagnose attempt 2.** Run its own tests (`try-attempt.sh 2 generated`), then the acceptance examples (`try-attempt.sh 2 adapted`). Find the failing line, name the shared misunderstanding between the code and its tests, and explain why the acceptance examples were independent evidence (what made them so: how they were derived). Run the generated tests against the accepted finder and say what the one failure shows.
- **B3. Make the tests bite.** Apply three faults of your choice to the accepted finder in a scratch copy. Predict which acceptance examples fail, run them, and compare. Find one fault that survives and say why.
- **B4. Fill in a record.** Complete a record for a different, very small task of your own (a method you ask an assistant, or a constructed candidate from a classmate), using the template, and run the record checker on it.

What you must do: your team's project

- **T1. Define one AI-assisted task before you prompt.** Pick a small, real task in your project. Write the requirement, the acceptance examples (derived from the requirement, before any candidate), the constraints and the scope. Decide which examples you will share and which you will keep undisclosed, and why.
- **T2. Run it and keep everything.** Use a live assistant if the rules allow, or an offline route (Section 15.9). Keep each attempt, including the ones you reject. Compile and test with the project's own commands.
- **T3. Record it.** Fill in the record, including the independent checks and the limits, and put it with the change. Run the record checker.

If you want more (the book's project and your team's project)

- **B5. Repeat the lab task with a live assistant**, if you may. Compare its output with the two constructed attempts: which failure modes did it show?
- **B6. Verify two dependencies.** Look up the coordinates of the constructed suggestion, report the status of each and decide what to do about each.
- **B7. Swap and find.** Do the classmate exercise of Section 15.9 on a task of your choice.
- **T4. Agree your team's practice.** Write down, in a short note, what your team will do about: which tools may be used (following the module leader's rules), what may be shared with them, how work is disclosed and recorded, and who reviews AI-assisted changes.

What you should end up with is: B1 to B4 as described; for your team's project, one defined task with its acceptance examples, the attempts, a record and, if you did T4, a team note.

You are done when these things hold:

- the acceptance examples were written before the candidate was requested;
- every claim the assistant made about itself was checked or marked unchecked;
- the check used the project's own compiler settings;
- a rejected attempt is in the record with its reason;
- the record's limits say what was not run;
- no secrets or personal data were shared;

- no live tool was used where the rules forbid it.

If you want more, do the same task twice, once with the acceptance examples in the prompt and once without, and compare the outputs and what each tells you.

15.12 Questions to think with, and a transfer challenge

Concept questions

1. Why is "I tested it with Java 17" not evidence? Name two checks that would be.
2. An assistant writes a function and its tests, and they pass. Why does that not establish that the function is right, and what would?
3. What does the `--release` option change that `--source` and `--target` alone do not?
4. A dependency suggested by an assistant exists on Maven Central. What have you learned and what have you not?

Applied diagnosis

5. An explanation of a method says "the method caches the result." The code has no field. What are your options for settling it, and what would you write down?
6. A refactoring patch makes four claims and one test fails. How do you find which claim caused it, and why is the patch's own summary not the way?

Transfer challenge

A team building an ML model-serving service wants to use an assistant to write a data-validation function and its tests. Plan the task: the requirement and acceptance examples (with synthetic records), the context to share and not to share, what the independent checks are, how you would check any suggested dependency, and what the record would say about what the checks do not show about the model. Say what is different about AI **in** the product, such as the model itself, and why this chapter does not cover it.

What to take away, and where to go next

An assistant gives you candidates. Your job is to define the task, keep the requirement and the examples yours, check the output against evidence the assistant did not write, and record what you accepted and why. The failures are familiar: invented methods, APIs newer than the baseline, dependencies that do not exist, boundaries one step off, tests that repeat the code, abstractions nobody needs, policy changed in passing, claims of testing that nobody ran. Each has an inexpensive check. None of them requires a live service to practise.

This is the last chapter of the book's main sequence. The appendices that follow collect material the chapters point to: notation and a programming refresher, tools and reproducible setup, engineering templates, the project versions and a glossary, and a guide to critical reading and project evidence.

Sources and further reading

Online sources 1–8 were opened on 13 October 2026. Tool and service behaviour changes quickly, so each source is dated by that opening. The lab, the constructed outputs, the experiments and the ready-made examples are original teaching material.

1. GitHub Docs, **Application card: GitHub Copilot inline suggestions**. One vendor's account of one tool: suggestions can look valid and be incorrect; the user must review and validate them; users assume the risks of generated code, including security, bugs and intellectual property. Used as an example of tool-specific guidance, not as an endorsement.
2. Oracle, `List` **(Java SE 21)**. `getFirst()` and `getLast()`, since 21.
3. Oracle, `List` **(Java SE 17)**. No `getFirst()`.
4. Oracle, **javac (Java SE 17)**. `--release` restricts the API as well as the language level to the given release.
5. Apache Maven, **maven-compiler-plugin: compile mojo**. The `release` parameter (`maven.compiler.release`) passes `-release` to the compiler; available since plugin version 3.6.
6. OWASP, **Top 10 for LLM Applications**, 2025 edition: lists sensitive information disclosure, supply chain and misinformation among its entries. About applications built on language models; used here for the named risk categories only.
7. Spracklen, J., Wijewickrama, R., Sakib, A. H. M. N., Maiti, A., Viswanath, B. and Jadliwala, M., **We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs**, arXiv 2406.10279 (version 3, 2 March 2025). Sixteen models and 576,000 code samples; hallucinated packages averaged at least 5.2% for commercial and 21.7% for open-source models in that study. This chapter treats dependencies in Section 15.7.
8. Maven Central, `org/mockito/`. The coordinates used in the dependency example were checked by status code on 13 October 2026. A name's absence then does not show that it can never be published.

Appendix A: Programming Readiness and Notation

> *Java 17 baseline, exception idioms, collections, and UML class diagram notation.*

OO Development and Agile Practices
Designing, Testing and Evolving Software with AI Assistance

How to use this appendix

You can understand a design decision only if you can follow the code that expresses it. This appendix supplies a short diagnostic and a reference for the Java and diagram notation used in the book. It assumes previous programming experience; it is not a first programming course.

Start with A.1. Revise the sections that address your gaps, then return to the chapter. Use A.8 beside Chapter 2's longer class-diagram exercises, and return to it when Chapters 8 to 10 introduce sequence, state and dependency diagrams. Records, streams and `Optional` can wait until a chapter uses them.

The language and API baseline is **Java 17 without preview features**. The syntax here describes the book's examples, not permissions for an examination. Your approved assessment instructions decide what tools and notation you may use. Appendix B covers installation and the wider toolchain; Appendix D covers project versions and domain terminology.

The Appendix A companion (in the Code, Explanations and Solutions companion) contains complete runnable files, diagnostic answers and a diagram solution. Its small lab is separate from the running lending project. It neither changes the project nor depends on any of its versions.

A.1 Readiness diagnostic

Try these before opening the answers. Explain your reasoning as well as giving a result. Java fragments in this diagnostic are reading exercises; the companion supplies their executable context.

D1. Trace a boundary

```
boolean mayRequest(int count, int days) {
    return count ≥ 0 && count < 3 && days ≥ 1 && days ≤ 7;
}
```

What does it return for `(2, 7)`, `(3, 7)`, `(0, 0)` and `(-1, 2)`? What changes if `< 3` becomes `≤ 3`? Does a result of `true` mean a camera has been reserved?

D2. Follow two references

```
List<String> first = new ArrayList<>();
List<String> second = first;
second.add("CAM1");
second = new ArrayList<>();
```

What does `first` contain at the end? Which statement changes an object and which changes a variable?

D3. Distinguish identity from value

```
String left = new String("CAM1");
String right = new String("CAM1");
```

Predict `left == right` and `left.equals(right)`. Why would two equal borrower identifiers not necessarily mean two variables refer to the same object?

D4. Read `final` precisely

```
final List<String> items = new ArrayList<>();
items.add("CAM1");
```

Does this compile? Would `items = new ArrayList<>();` compile afterwards? Does `final` make the list immutable?

D5. Trace a snapshot

```
List<String> source = new ArrayList<>(List.of("CAM1"));
List<String> snapshot = List.copyOf(source);
source.add("CAM2");
```

What does each list contain? What happens if you call `snapshot.clear()`? Would the copy protect fields inside a mutable element?

D6. Protect an invariant

A loan period has a start date and an exclusive end date. Which inputs should its constructor reject? Do `[20 October, 22 October)` and `[22 October, 24 October)` overlap?

D7. Identify the method that runs

```
BorrowingPolicy policy = new StudentRules();
boolean allowed = policy.mayRequest(2, 7);
```

Which type describes the variable? Which class supplies the implementation? Does declaring an interface automatically establish that the implementation obeys the borrowing rule?

D8. Classify three failures

Classify each as a compile-time problem, a run-time failure or a wrong result:

- Calling `period.intersects(other)` when `LoanPeriod` declares only `overlaps`.
- Calling a method through a null reference.
- Changing the reservation limit comparison from `< 3` to `≤ 3`.

D9. Preserve the meaning of failure

A directory lookup can find no borrower or fail because the service is unavailable. Why is catching every exception and returning "borrower not found" misleading? What does `throws IOException` require a caller to consider?

D10. Review a diagram after Chapter 2

A department keeps zero or more equipment items. Each item belongs to exactly one department at a time. One item may have several reservations; each reservation concerns exactly one item. Reservations have identifiers and periods. Draw those three classes and their multiplicities. Does this description justify a filled composition diamond?

Choose a revision route

If you struggled with…	Revisit	Evidence that the gap is closing
D1	A.2	Trace both sides of a boundary and state what the method does not do.
D2–D5	A.3–A.4	Explain the difference between changing a reference and changing its object.
D6	A.3 and A.6	Reject invalid periods and explain adjacency.
D7	A.5	Follow a call through an interface to an implementation.
D8–D9	A.7	Separate an ordinary result from invalid input and operational failure.
D10	A.8 and Chapter 2 §2.6	Read each multiplicity in both directions and justify the relationship.

There is no pass mark. Record the topic to revisit and the example you can now explain. D10 is revision after the notation lesson, not a prerequisite for opening Chapter 1.

A.2 Reading a small Java method

In `public boolean mayRequest(int count, int days)`, `public` describes access, `boolean` is the result type, `mayRequest` is the name, and the two named `int` parameters receive the supplied values. `return` ends the method and supplies its result. A `void` method supplies no result value.

Syntax	Meaning here	Common mistake
`int count = 2;`	Declare and initialise an integer variable.	Confusing assignment `=` with comparison `==`.
`count < 3`	Compare two numbers.	Using the limit after addition when the input count is before addition.
`a && b`	Require both; evaluate `b` only if `a` is true.	Treating `&` as an interchangeable spelling.
`a			b`		
`!allowed`	Negate a Boolean value.	Negating the wrong part of a longer condition.
`if (...) { ... } else { ... }`	Select a branch.	Believing indentation controls Java's blocks.
`for (String id : ids)`	Visit each element of the iterable in its encounter order.	Assuming every collection has the same ordering guarantee.

Use parentheses when grouping is difficult to read. For example, `id ≠ null && !id.isBlank()` avoids invoking `isBlank()` when `id` is null because `&&` short-circuits. This is a specific guard, not a general solution to missing values. Java's evaluation and operator rules are defined in [1].

A variable declared within a method is local to its scope. A field belongs to a class or an object and may outlive one call. When tracing code, write down both the local values and the object state that a later call can observe.

A.3 Classes, constructors and references

An instance method operates on a particular object. `this` refers to that object. A constructor establishes its initial state and has the class's name with no return type. A `static` method belongs to the class and has no instance `this`. These mechanisms say how code runs; they do not decide whether a responsibility belongs in that class [2].

The companion's `LoanPeriod` checks its invariant in the constructor. This is a **constructor excerpt**, with fields and accessors supplied in the complete file:

```
public LoanPeriod(LocalDate start, LocalDate endExclusive) {
    this.start = Objects.requireNonNull(start, "start");
    this.endExclusive = Objects.requireNonNull(endExclusive, "endExclusive");
    if (!start.isBefore(endExclusive)) {
        throw new IllegalArgumentException("End must be after start.");
    }
}
```

Passing a reference copies the reference value; Java does not pass the caller's variable itself. A method can use that reference to mutate a mutable object. Reassigning the method's parameter does not reassign the caller's variable. D2 is the same distinction within one method.

For references, `==` compares identity. `equals` follows the class's equality implementation. The companion's ordinary `LoanPeriod` does not override `equals`; two separate periods with identical dates therefore do not become equal automatically. Compare their values deliberately, or design an equality contract when the domain needs one. Never assume a domain identifier and Java object identity are the same concept.

Reading access and modifiers

Form	Reading
`public`	Accessible where the enclosing type and module rules permit.
`private`	Restricted to the declaring top-level class and its nested classes.
No access modifier	Package access. It does not mean public.
`protected`	Package access plus subclass access, subject to Java's subclass rules.
`final` variable	May be assigned only once; a referenced object can still be mutable.
`final` class	Cannot be subclassed.
`final` method	Cannot be overridden.
`abstract` class	Cannot be instantiated directly; may supply both implemented and abstract operations.

Chapter 2 develops encapsulation and information hiding. Keep those design ideas separate from memorising keywords.

A.4 Collections, copying and small functional expressions

`List<Reservation>` describes a list whose element type is `Reservation`; angle brackets express a generic type argument. `new ArrayList<>()` supplies a mutable list implementation. Declaring against `List` gives callers a different commitment from exposing `ArrayList` specifically.

Expression	Useful interpretation
`List.of("CAM1", "CAM2")`	An unmodifiable list containing those values.
`new ArrayList<>(items)`	A mutable copy of the list structure.
`List.copyOf(items)`	An unmodifiable snapshot of the element references.
`items.size()`	Number of elements, not a count of completed loans or eligible borrowers.
`items.get(0)`	First element; fails if the list is empty.

`List.of` and `List.copyOf` reject null elements. A snapshot does not follow later structural changes to its source. It also does not deep-copy elements. These distinctions matter when a class returns a collection to its callers [3].

Later chapters use lambdas and streams. Read the following **equivalent counting excerpts** before deciding which is clearer for the task:

```
long cameras = 0;
for (String id : ids) {
    if (id.startsWith("CAM")) {
        cameras++;
    }
}
```

```
long cameras = ids.stream()
        .filter(id -> id.startsWith("CAM"))
        .count();
```

`id -> ...` supplies a small function. Here, `filter` selects matching elements and `count` performs the terminal operation. This expression does not remove nonmatching entries from `ids`. A method reference such as `wanted::overlaps` supplies a compatible operation on an existing object. Prefer a loop when it makes a state change or failure path easier to follow.

A.5 Interfaces, overriding and the later Java idioms

The companion defines `BorrowingPolicy` and a `StudentRules` implementation. The declaration `BorrowingPolicy policy = new StudentRules();` lets the caller use the interface's operation while the object's class supplies its implementation. `@Override` asks the compiler to check that a declaration overrides or implements an inherited operation. It does not test the operation's meaning [4].

Overloading provides methods with the same name and different parameter lists. **Overriding** supplies an implementation of an inherited instance method. A call's declared types help select an overload; the receiving object's run-time class matters for an overridden instance method. Chapter 5 examines the behavioural obligations that syntax cannot enforce.

The following is the appendix's reading scope, not an examination rule:

Idiom	Depth	What to recognise
Classes, interfaces, constructors, enums and generics	Core	Read declarations and follow ordinary calls.
Lambdas, method references and simple streams	Working knowledge	Explain a short expression; expand it to ordinary steps.
Records	Working knowledge when encountered	A compact data-carrier class with component accessors and generated value methods.
`Optional<T>`	Working knowledge when encountered	A result explicitly containing a value or no value.
`var`	Reading knowledge	Local type inference; the type remains fixed at compile time.
Arrow-form `switch` expressions	Reading knowledge	Select a result without the traditional case fall-through.
Sealed types and advanced type patterns	Awareness only here	No dependency on them in this appendix.
Preview syntax and APIs newer than Java 17	Outside this appendix	Do not introduce them into its runnable files.

A record such as `record Request(String borrowerId, int days) {}` receives accessors named `borrowerId()` and `days()`, together with generated `equals`, `hashCode` and `toString`. Records are final; their component fields are final. A component can still refer to a mutable object, so a record is not automatically deeply immutable. A compact constructor can validate its arguments [5].

`Optional.empty()` should mean the absence specified by the operation' s contract. Calling `get()` on an empty optional throws. Inspect it, transform it or provide an appropriate alternative; do not use "empty" to conceal a failed lookup. For a present value, `Optional.of` requires non-null; `ofNullable` can convert a nullable value into present or empty [6].

`var` is useful where the initializer makes the type clear; it is not a dynamically typed variable. The companion demonstrates an enum, a switch expression and a record in executable context. You need not rewrite earlier ordinary classes into records to follow the book.

A.6 Dates and the interval convention

`LocalDate` represents a date without a time or time zone and is immutable. `start.plusDays(2)` returns a date; it does not modify `start` [7]. The book' s reservation periods are **start inclusive, end exclusive**, written `[start, end)`.

First period	Second period	Overlap?
`[20 Oct, 22 Oct)`	`[21 Oct, 23 Oct)`	Yes
`[20 Oct, 22 Oct)`	`[22 Oct, 24 Oct)`	No
`[20 Oct, 22 Oct)`	`[20 Oct, 22 Oct)`	Yes

For valid nonempty periods, overlap is expressed by both strict comparisons:

```
// Method-body excerpt from LoanPeriod.overlaps.
return start.isBefore(other.endExclusive)
        && other.start.isBefore(endExclusive);
```

Use `Instant` for a point on the time line and an injected `Clock` when tests need a controlled "now", as later chapters do. A date and an instant answer different questions. Do not turn a calendar-day loan into an assumed fixed number of elapsed hours without an explicit time-zone policy.

A.7 Exceptions and evidence of a failure

An exception transfers control away from the ordinary return path. `throw` raises one; `throws` declares possible propagation from a method. Checked exceptions must be caught or declared where Java's rules require it. `RuntimeException` subclasses and `Error` subclasses are unchecked [8].

Situation	Example	What the caller needs to know
Ordinary negative result	No matching borrower found	The lookup completed and found no match.
Invalid argument	End date is not after start	The caller supplied an invalid period.
Operational failure	Approval file could not be read	The decision could not be completed.
Assertion failure	A check expected rejection but saw approval	The observed result disagrees with the check.

The choice between a result and an exception belongs to the operation's contract. Chapter 1 deliberately returns `false` for a negative count; the period constructor rejects invalid dates. Read the particular contract instead of assuming every invalid input has the same treatment.

The companion includes this **complete method**, with its imports and containing class provided there:

```
static String readFirstLine(Path path) throws IOException {
    try (var reader = Files.newBufferedReader(path, StandardCharsets.UTF_8)) {
        return reader.readLine();
    }
}
```

The resource closes when the block exits, including exceptional exit. A file-reading failure propagates as an `IOException`; this method does not substitute "no borrower". An empty file returns null because that is `readLine`'s contract. Interpret that result at the appropriate boundary.

Read an error in context: identify the operation, the exception type, the first relevant frame in your code and any cause. Catch an exception where you can handle it meaningfully or translate it into the declared boundary contract. Catching everything simply to make the program continue can destroy the information needed to diagnose it.

A.8 UML and Mermaid notation reference

UML describes modelling concepts; Mermaid is the text notation used to render diagrams in this book. Mermaid supports a useful subset and does not enforce every UML semantic rule. A successfully rendered diagram can still describe the wrong design [9, 10].

Class compartments and member notation

Meaning	Conventional UML form	Book's Mermaid form
Private attribute	`- start : LocalDate`	`-LocalDate start`
Public operation	`+ overlaps(other : LoanPeriod) : Boolean`	`+overlaps(LoanPeriod other) boolean`
Interface	Interface annotation	`<<interface>>` inside the class declaration
Generic type	A parameterised type	`List~Reservation~` in a member label
Visibility	Public, private, protected, package	`+`, `-`, `#`, `~`, respectively

Keep one convention within a diagram. Omit implementation details that are irrelevant to its question. Missing members in a selective diagram are not evidence that the members do not exist.

Relationships and arrow direction

The strings in the following table are **Mermaid source syntax**, not diagrams made from text characters. The renderer supplies the actual symbols; the last column says how to draw the same relationship by hand.

Relationship	Mermaid declaration	Meaning and review question	Drawn by hand
Association	`A ⟶ B : holds`	A structural link, navigable from A to B. Does the design retain that link?	A solid line, with an open arrowhead at B to show navigation.
Association with navigation unspecified	`A -- B`	A link without a stated navigation direction; do not infer that both Java classes have fields.	A solid line with no arrowhead.
Dependency	`A ..> B : uses`	A uses B. A method-only use often needs no stronger relation.	A dashed line with an open arrowhead at B.
Shared aggregation	`Whole o-- Part`	A grouping claim, with a hollow diamond at the whole. Its precise meaning needs the model's explanation.	A solid line with a hollow diamond at the whole.
Composite aggregation	`Whole *-- Part`	Strong whole/part ownership, with a filled diamond at the whole. Is exclusive composite ownership and lifecycle intent actually specified?	A solid line with a filled diamond at the whole.
Generalisation / inheritance	`Base ⊲-- Derived`	The hollow triangle points to the general type. Does Derived meet Base's behavioural obligations?	A solid line with a hollow triangle at the general type.
Interface realisation	`Contract ⊲.. Implementation`	The hollow triangle points to the interface; the line is dashed. Does the class implement that contract?	A dashed line with a hollow triangle at the interface.

A class that holds another type in a field depends on that type too, but the diagram normally shows only the more informative association and not a second dependency line for the same pair. Use a dependency arrow for a weaker link, such as a method that only uses the type as a parameter or a local variable. Navigation expresses a modelling decision and can be implemented in more than one way.

Neither `private`, `final` nor construction with `new` proves UML composition. Garbage collection is not a domain lifecycle rule. A shared immutable period needs no filled diamond merely because a reservation refers to it. Use plain association when the stronger claim is unsupported.

Multiplicity: look at the far end

Label at an end	Number of objects at that end for one object at the other end
`1`	Exactly one
`0..1`	None or one
`0..*` or `*`	Zero or more
`1..*`	One or more

A Java `List<T>` does not by itself enforce a minimum size, uniqueness or exclusive ownership. A reference type does not by itself exclude null. The constructor, operations and domain rules supply those obligations.

Figure A.1 is a selective reading of Chapter 2’ s reservation structure. It specifies the multiplicities needed in the indicated direction; omitted opposite-end numbers are not claims about exclusive ownership.

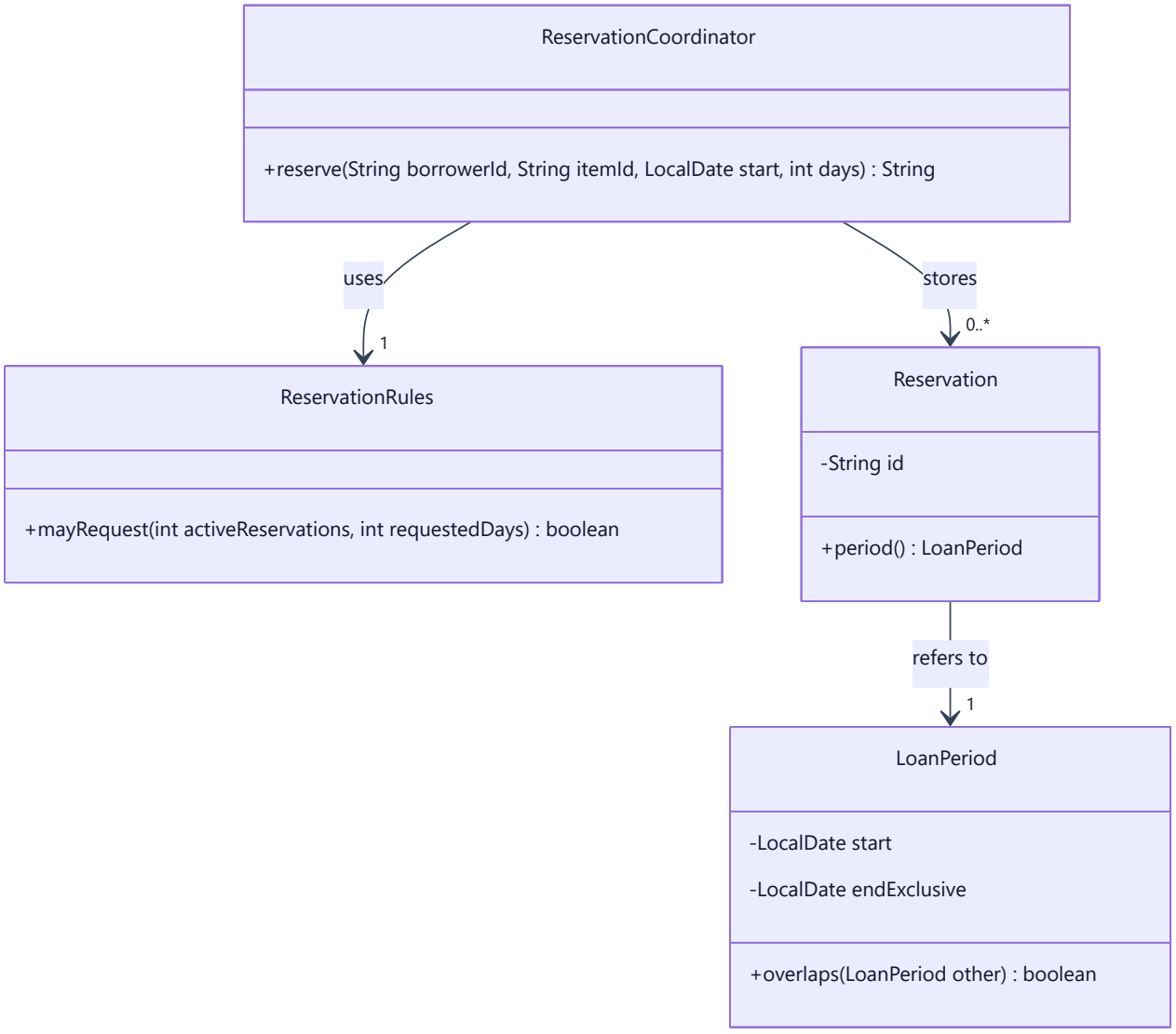


Figure A.1. One coordinator uses one rules object and stores zero or more reservations; each reservation refers to one period. This is a diagram reference to Chapter 2, not another running source-code version. The appendix lab uses `StudentRules` to demonstrate an interface separately.

Read different diagram kinds differently

Diagram	What its edges mean in this book	What not to infer
Class diagram	Structural relationships or type dependencies	Execution order.
Sequence diagram	Messages in a scenario, read down the page; `alt` separates alternatives	Every possible path or real network traffic unless stated.
State diagram	Permitted transitions between states	A Java subclass for every state.
Flowchart	Steps and decisions in a process	Automatically enforced source-code dependencies.
Package/dependency view	Which part depends on which other part, as defined by the caption	The opposite direction merely because a callback returns control.

The three kinds below are the ones, besides class diagrams, that the book draws. Each has a symbol table (the Mermaid source, and how the same symbol is drawn by hand) and one small example. The examples are notation exercises, not part of the lending project's design. Treat each diagram's caption as part of its meaning: it says what an arrow stands for in that figure.

Sequence diagrams

A sequence diagram shows **one scenario in time order**, from top to bottom. It answers "who asks whom, and in what order?" and not "what classes exist?". Chapter 8 introduces it; Chapters 9 and 10 use it again [11].

Element	Mermaid source	Drawn by hand	Reading
Participant	`participant R as ReservationRules`	A box at the top of the page, with a name.	An object or role taking part. `as` gives it a short label for the lines below. The renderer repeats the box at the bottom; by hand the top box is enough.
Lifeline	(the renderer draws it)	A dashed vertical line under the box.	That participant's time line. Time runs downwards.
Call	`U->>R: mayRequest(2, 7)`	A solid line with a filled arrowhead, labelled with the call.	The sender asks the receiver to do something.
Return	`R-->>U: true`	A dashed line with an open arrowhead.	The answer. In this book a return can also carry an exception.
Note	`Note over U,R: Nothing is stored`	A note box across the participants it concerns.	A comment on the scenario, not a message.
Alternatives	`alt` … `else` … `end`	A frame labelled `alt`, with a dashed line between the alternatives.	Exactly one alternative happens in a given run.
Optional, repeated	`opt` … `end`, `loop` … `end`	A frame labelled `opt` or `loop`.	A block that may happen once or not at all, and a block that repeats. The book's figures do not use them.

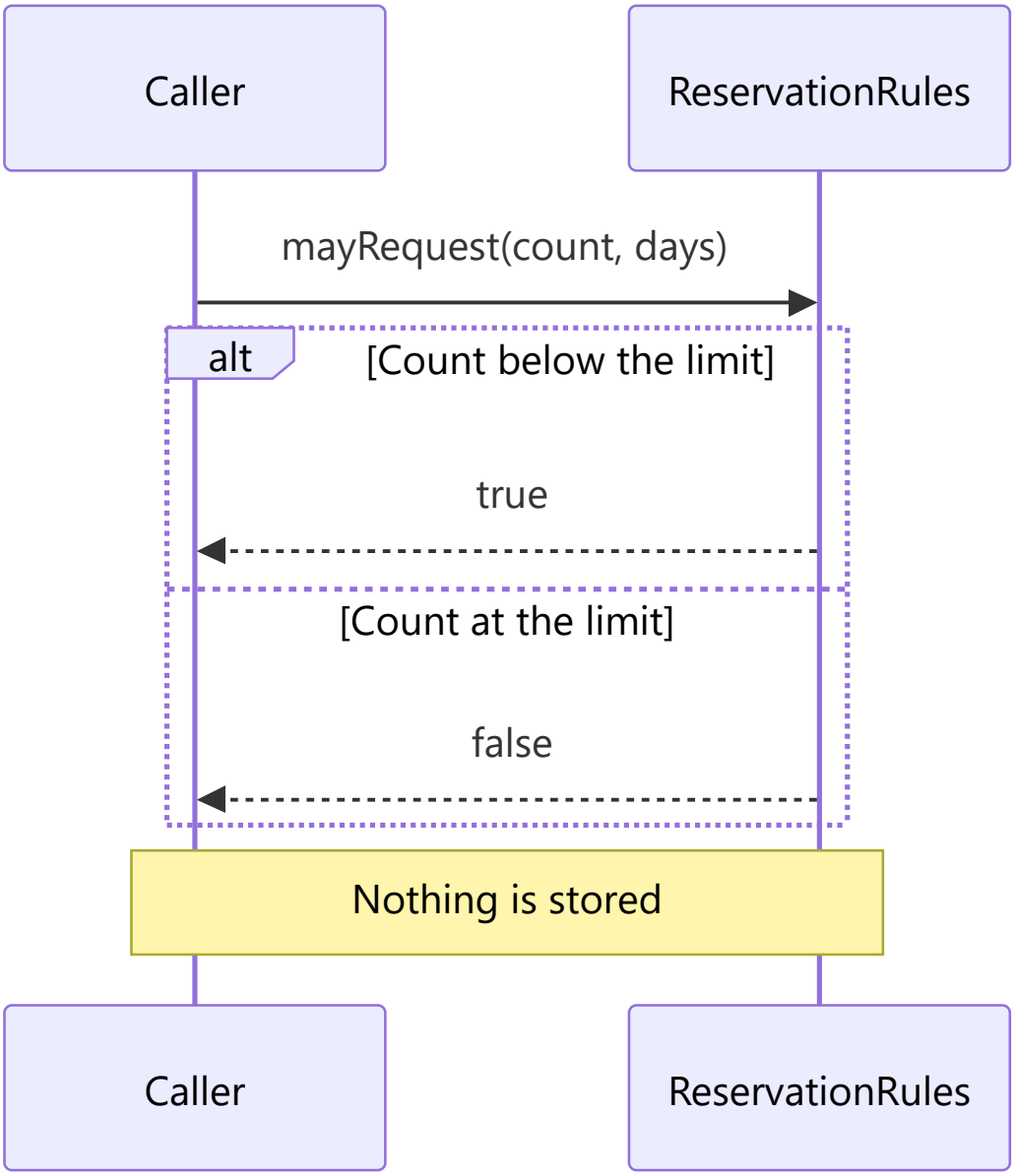


Figure A.2. A sequence-diagram notation example. The caller asks the rules object one question and receives one of two answers. The note says what does *not* happen.

Do not read a sequence diagram as a complete specification. It shows the scenario its author chose, and a path that is missing is not a claim that the path cannot happen (Figure 8.2 leaves out malformed provider data to stay readable). The vertical order is time order. It is not the order of the classes in the source code.

State diagrams

A state diagram shows the **positions an object can be in and the events that move it between them**. It says nothing about how the states are implemented. Chapter 9 introduces it [12].

Element	Mermaid source	Drawn by hand	Reading
State	`AVAILABLE`	A rounded rectangle with a name.	A situation the object can be in.
Start	`[*] ⟶ AVAILABLE`	A filled circle with an arrow to the first state.	Where a new object begins.
Transition	`AVAILABLE ⟶ RESERVED: reserve`	An arrow labelled with the event.	When the event happens in the first state, the object moves to the second.
Guard	`AVAILABLE ⟶ RETIRED: retire [no open reservation]`	A label of the form `event [condition]`.	The transition fires only if the condition holds. The reservation lifecycle of Chapter 9 has no guards.
End	`RETIRED ⟶ [*]`	A circle with a filled centre.	The object's life ends here.

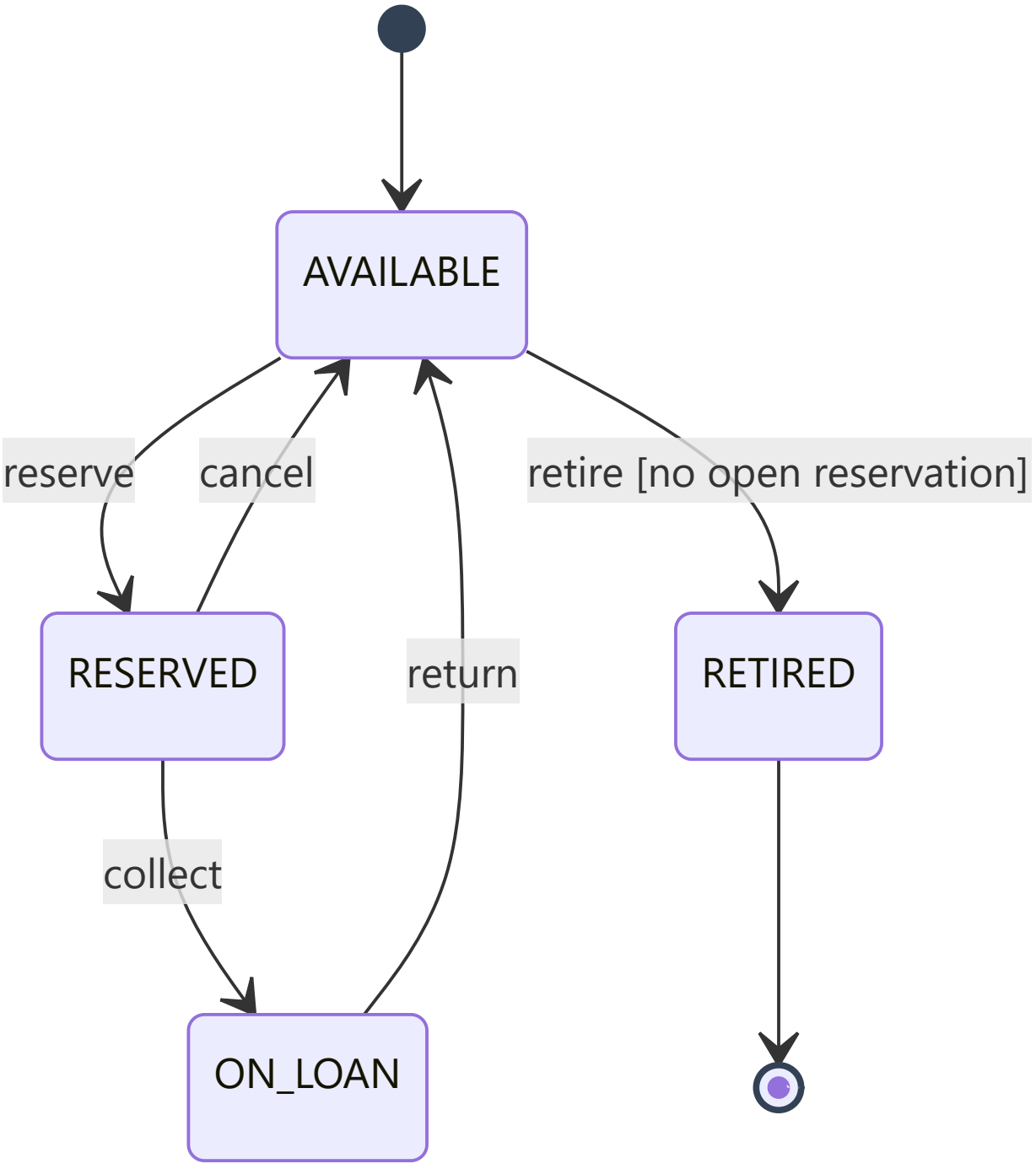


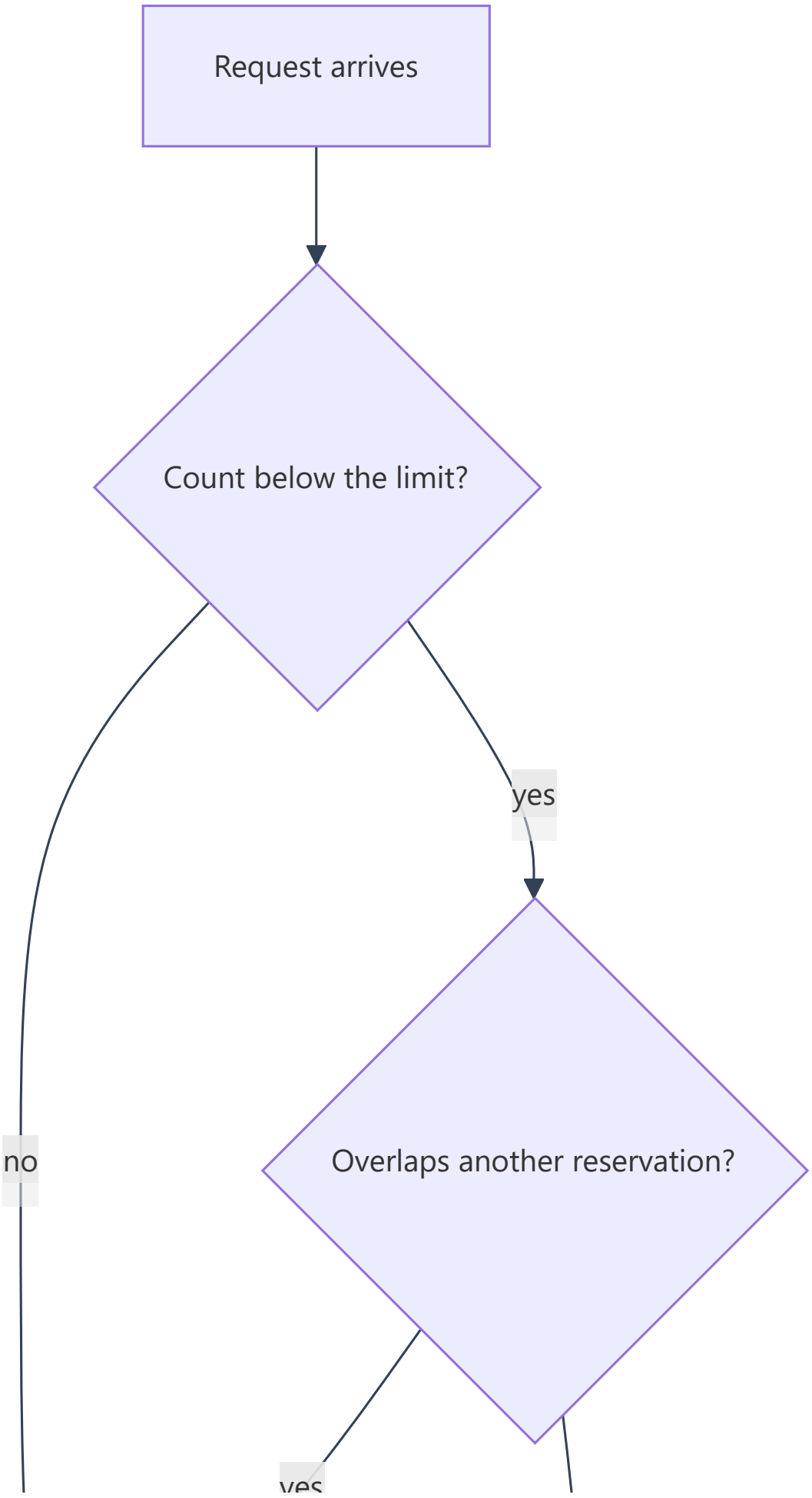
Figure A.3. A state-diagram notation example for an equipment item. It is not the reservation lifecycle of Chapter 9 and not a design for the project.

To read one, take each state and each event in turn and ask whether an arrow exists. A missing arrow means the model does not allow that event in that state. The diagram does not say what the program does when the event arrives anyway: it might throw, ignore it or report an error. That is a separate decision. A state diagram also does not require one Java subclass per state.

Flowcharts and dependency views

A flowchart is one Mermaid kind with several uses in this book. In a **process flow** the arrows mean "then" or "only if the previous step passed" (Figures 11.1 and 13.2). In a **dependency view** an arrow means that the code at its start may refer to the code at its end (Figure 10.1). In an **object-assembly view** a dotted arrow means "creates" (Figure 6.2). The caption always says which; never assume it.

Element	Mermaid source	Drawn by hand	Reading
Step or part	`A["Check for overlap"]`	A rectangle with a label.	An action, or a part of the system.
Decision	`B{"Count below the limit?"}`	A diamond with a question.	A choice. Each exit carries a label.
Arrow	`A ⟶ B`	A solid arrow.	What the caption says: "then", "may refer to" or "passed to".
Labelled arrow	`B -- "yes" ⟶ C`	An arrow with a short label.	The condition or reason for that exit.
Dotted arrow	`A -. creates .-> C`	A dashed arrow with a label.	A different relationship from the solid arrows; the label says which.
Direction	`flowchart TD` or `flowchart LR`	(the page layout)	Top-down or left-to-right. It is layout only and carries no meaning.


Request arrives
Count below the limit?
yes
no
Overlaps another reservation?
yes

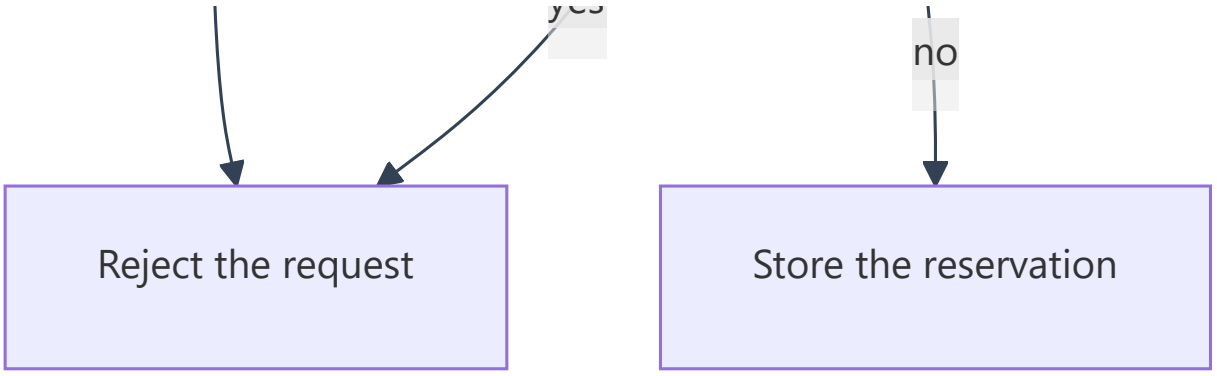


Figure A.4. A process-flow notation example. Boxes are steps, diamonds are decisions and every exit of a diamond is labelled.

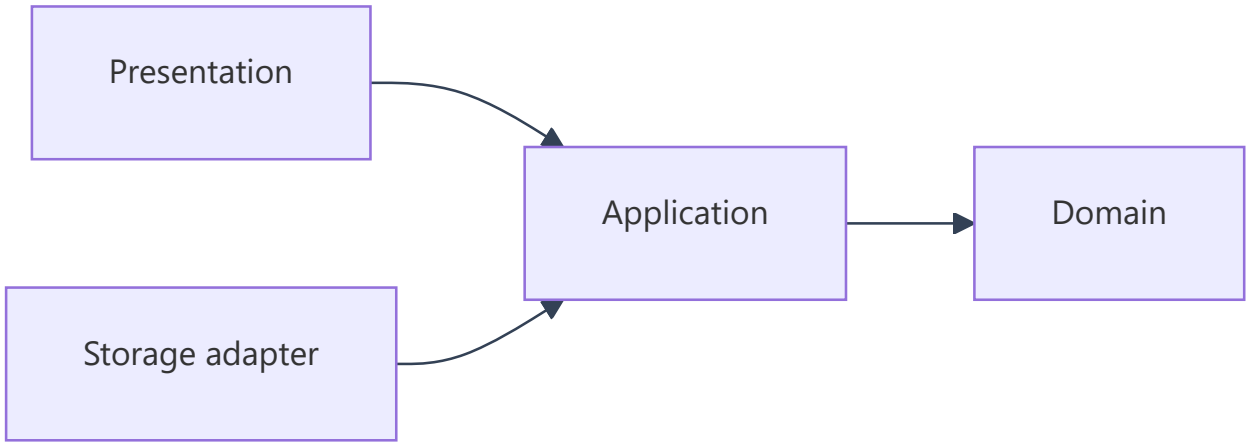


Figure A.5. A dependency-view notation example. An arrow means that code in the first box may refer to code in the second. A missing arrow means the reference is not permitted. The figure says nothing about the order of calls.

Formal UML has its own notation for the second use. A **package diagram** draws each package as a folder-shaped box, and a dependency is a dashed arrow with an open head from the package that uses to the package that is used. A **component diagram** draws each component as a box labelled `«component»` and shows its interfaces as a small ball (provided) and a socket (required). The book does not draw either; it draws the dependency view as a flowchart, because the only question is “who may refer to whom”. The reading rule is the same in both: the arrow starts at the code that refers and points to the code referred to. In a dependency view the arrow is **not** the direction of runtime calls. In Figure 10.1 the storage adapter refers to an application-owned contract, so the arrow points from the adapter to the application, although the coordinator calls the adapter at run time [9, 13].

To check a diagram: name its question, read every relationship aloud, justify each multiplicity, compare shown operations with their source, and trace any claimed order. Use top-down layouts for wider structures. A diagram is finished when it explains the intended relationship clearly, not when it contains every class.

A.9 A short verification activity

Complete D1–D9, then run the companion lab or trace it on paper. Select one mistaken assumption from your answers and explain which check exposes it. After Chapter 2, attempt D10 and compare your diagram with the solution. After Chapters 8 to 10, read the sequence, state and flowchart fragments in the companion, Section 4.

Optional AI activity, with an identical offline route. The following is an author-constructed candidate explanation:

> "The list is final, so callers cannot clear it. Copying a list makes all its elements immutable. Every object created in a constructor should have a composition diamond."

Classify its three claims and correct each using the code or the stated modelling rule. A live assistant may produce a different candidate if permitted, but is not required. Record the claim, your check and your conclusion. Do not use another unsupported explanation as the final evidence.

What you hand in at the end is a short personal readiness note: topics checked, one corrected explanation and any remaining revision. This is formative study guidance and adds no assessment requirement.

Sources and further reading

Official references checked on 13 October 2026. The explanations and exercises above are original; source examples have not been reproduced.

1. Oracle, **Java Language Specification, Java SE 17, Chapter 15: Expressions**. Evaluation, reference equality, lambdas and switch expressions.
2. Oracle, **JLS 17, Chapter 8: Classes**. Fields, constructors, modifiers and methods.
3. Oracle, **Java SE 17: List**. List operations and unmodifiable copies.
4. Oracle, **JLS 17, Chapter 9: Interfaces**. Interface declarations and implementation contracts at the language level.
5. Oracle, **Java SE 17: Record Classes**. Generated members and compact constructors.
6. Oracle, **Java SE 17: Optional**.
7. Oracle, **Java SE 17: LocalDate**.
8. Oracle, **JLS 17, Chapter 11: Exceptions**.
9. Object Management Group, **Unified Modeling Language 2.5.1**. Standard terminology and semantics.
10. Mermaid, **Class diagrams**. Renderer syntax, members, relationships and multiplicities.
11. Mermaid, **Sequence diagrams**. Message and alternative-block syntax.
12. Mermaid, **State diagrams**. States, transitions and the start and end markers.
13. Mermaid, **Flowcharts**. Nodes, decisions, labelled and dotted links.

Appendix B: Tools and Reproducible Setup

> *Pinned toolchain, Maven build commands, PMD setup, and Git pull request workflows.*

OO Development and Agile Practices
Designing, Testing and Evolving Software with AI Assistance

How to use this appendix

A design discussion is only useful if you can run the code it describes. This appendix records, in one place, the tools the book uses, the exact versions it pins, the commands that build and test the examples, and the Git and pull-request routine your team will follow. Chapters explain design and cite this appendix for setup, so that a change of tool does not force a rewrite of an explanation.

Start with B.1 and B.2: they tell you what to install and how to check it. Chapters 1 to 6 need only a JDK and a Bash shell. From Chapter 7 the examples use Maven. Use B.8 beside Chapter 2′ s Git exercise and again when your team starts its pull-request routine. The remaining sections are reference: look up a tool when a chapter first uses it.

This appendix records what works for the book′ s examples. It is not institutional policy. Your approved assessment instructions decide which tools you may use in an assessment, and a section here never grants permission. The Appendix B companion (in the Code, Explanations and Solutions companion) contains a complete small starter project (every file), a scripted Git walk-through and the answers to the activity in B.12.

B.1 What you need, and when

Tool	Needed from	Used for	What the book pins or requires
JDK (a full Java Development Kit, not only a runtime)	Chapter 1	Compile and run	Version 17 or later. The examples are compiled with `--release 17`.
Bash	Chapter 4	The check scripts	Git Bash on Windows; a terminal on macOS or Linux.
Git	Chapter 2	Version control and review	Version 2.28 or later, because the commands use `git init -b`, `git switch` and `git restore`.
Apache Maven	Chapter 7	Build, dependencies and tests	Version 3.9.11.
JUnit Jupiter	Chapter 7	Unit tests	5.13.4 (the `junit-jupiter` aggregate artifact).
Mockito	Chapter 7	One interface-only test double	`mockito-core` 5.20.0.
Maven Compiler Plugin, Surefire	Chapter 7	Compile with `--release 17`; run the tests	3.14.1 and 3.5.4.
PMD	Chapter 5 only	The static-analysis lab	7.28.0.
GitHub Actions	Chapters 4 and 13	A hosted check	`actions/checkout@v7`, `actions/setup-java@v6`, Temurin 17.
An editor or IDE	Any	Writing code	Not pinned. Use what you know.

What "pinned" means. A pinned version is named exactly, so that everyone who follows the book gets the same behaviour. The Maven versions live in each project's `pom.xml`; the others are named in this table. When a tool changes, this table and the POMs change and the design explanations stay as they are. Maven plugins that the POM does not name take the versions that Maven 3.9.11 itself binds; the two that appear in the examples are the Resources plugin (3.3.1) and the Jar plugin (3.4.1).

Newer releases exist. The Maven site lists a newer stable release than 3.9.11. The book's output was produced with 3.9.11, so use that version when you want to reproduce what the chapters show, and treat a newer one as your own experiment.

Why the book compiles with `--release 17`

The examples use only the Java 17 language and library. If you compile with a newer JDK and no option, the compiler checks your code against that JDK's library, so you can call a method that does not exist in Java 17 and nothing complains until the program meets a Java 17 runtime. The option `--release 17` asks the compiler to check against the Java 17 language and library instead. The following program uses `getFirst()`, a method added to lists after Java 17:

```
import java.util.List;

public class NewApi {
    public static void main(String[] args) {
        System.out.println(List.of("a", "b").getFirst());
    }
}
```

Compiled with plain `javac` on a recent JDK it succeeds and prints `a`. Compiled with `javac --release 17` it fails with `cannot find symbol` for `getFirst()`. The option restricts what the compiler accepts. It does not make a newer runtime behave like Java 17, so run `java -version` separately to see which runtime you have [1].

B.2 Check your setup

The companion lists `scripts/check-setup.sh`. It changes nothing and downloads nothing; it reports whether each tool is on the path and whether the compiler is recent enough:

```
bash scripts/check-setup.sh
```

On a machine where everything is in place it prints one `OK` line per tool and ends with `SETUP OK`:

```
OK       bash: GNU bash, version 5.3.9(1)-release (x86_64-pc-cygwin)
OK       git: git version 2.54.0.windows.1
OK       java: java version "24.0.2" 2025-07-15
OK       maven: Apache Maven 3.9.11 (3e54c93a704957b63ee3494413a2b544fd3d825b)
OK       javac: compiler version 24 can compile with --release 17
SETUP OK
```

The version text on your machine will differ. What matters is that no line starts with `MISSING` or `TOO-OLD`. A `MISSING` line names the command that is not on the path. A `TOO-OLD` line means the compiler cannot target Java 17; install a newer JDK. A runtime without `javac` is not enough, which is why the script checks the compiler separately.

Installing. Obtain a JDK of version 17 or later from the vendor your institution recommends. The book's hosted-check example uses the Eclipse Temurin build of Java 17 [2]. For Maven, the project's installation page says to extract the binary archive, add its `bin` directory to the `PATH`, make sure a JDK is found either through the `JAVA_HOME` variable or through `java` on the path, and confirm with `mvn -v` in a new shell [3]. To reproduce the book, take version 3.9.11 from the Apache archive, where each archive is published with a checksum file [4]:

```
apache-maven-3.9.11-bin.zip
SHA-512  03e2d65d4483a3396980629f260e25cac0d8b6f7f2791e4dc20bc83f9514db8d0f05b0479e699a5f34679250
```

Compare it with the digest of your download (`sha512sum apache-maven-3.9.11-bin.zip`) before you unpack it. If the two differ, delete the file and download it again.

Windows notes. Use Git Bash for the scripts. Windows can have more than one program called `bash` on the path, among them the one from Git for Windows and the Windows Subsystem for Linux launcher, and the first one found runs your script. Scripts that assume Windows-style paths fail under the wrong one. The book's scripts take the Maven command from an optional variable, `CHAPTER_MVN`, for the case where `mvn` is not on the path or you call it by its full path; on Windows the launcher is `mvn.cmd`.

B.3 Compile and run without a build tool (Chapters 1 to 6)

The early chapters use plain Java. A typical compile-and-check pair is:

```
javac --release 17 -Xlint:all -encoding UTF-8 -d build/classes src/main/java/*.java src/test/java
java -cp build/classes Chapter04Demo
```

`-Xlint:all` turns on every compiler warning category, `-encoding UTF-8` makes the source encoding explicit, `-d` names the output folder and `-cp` (class path) tells `java` where to find the compiled classes. The shell expands `*.java` into the files that are directly inside the two folders; it does not enter sub-folders, so a project that grows nested packages needs a build tool. Chapter 4's `scripts/check.sh` wraps these commands, stops at the first failure and ends with an explicit success message that appears only if everything before it passed. Appendix A's lab uses the same commands.

B.4 Build and test with Maven (from Chapter 7)

Maven builds from a `pom.xml` that names the project, the Java release, the dependencies and the plugin versions. By convention, application code is in `src/main/java`, tests in `src/test/java` and test resources in `src/test/resources`. Generated files go to `target`, which Git ignores.

A Maven build moves through **phases** in a fixed order, and asking for a phase runs everything before it. The ones the book uses are `compile`, `test`, `package` and `verify` [5]. The command you will run most is:

```
mvn --batch-mode verify
```

`--batch-mode` removes interactive prompts and colour codes, so the output is the same in a terminal, in a script and in a hosted check. A successful run ends with `BUILD SUCCESS` and exit status 0; a failure prints `BUILD FAILURE` and returns a non-zero status, which is what scripts and hosted checks look at.

Command	What it does
`mvn --batch-mode verify`	Compile, run all tests, package, verify. The normal check.
`mvn --batch-mode test`	Compile and run the tests only.
`mvn --batch-mode -Dtest=ToolchainReportTest test`	Run one test class.
`mvn --batch-mode "-Dtest=ToolchainReportTest#aJavaVersionBelowTheTargetCannotRunTheCode" test`	Run one test method. Quote the argument so the shell does not treat `#` as a comment [6].
`-Dgroups=integration`, `-DexcludedGroups=integration,e2e`	Run or leave out tests that carry a JUnit tag (Chapter 13) [7].
`--offline`	Do not contact any remote repository.
`-Dmaven.repo.local=PATH`	Use another local repository folder.

What the first build downloads. Maven fetches the plugins and dependencies named in the POM, and the plugins' own dependencies, from Maven Central, and keeps them in a local repository, by default the `.m2/repository` folder in your home directory. Later builds reuse that folder. `--offline` works only when the folder already holds everything. On an empty one it fails with a message such as:

```
Cannot access central (https://repo.maven.apache.org/maven2) in offline mode and the artifact
org.apache.maven.plugins:maven-resources-plugin:jar:3.3.1 has not been downloaded from it before.
```

Run one online build first, or copy a populated repository folder to the machine. Behind a proxy, Maven reads a `<proxies>` section from `settings.xml`, either in the Maven installation's `conf` folder or in `.m2` in your home directory [8]. Ask your institution for the proxy details; the book does not supply any.

Counting tests. Surefire reports **invocations**: a parameterised test with three values counts as three. The starter in the companion has five test methods and runs seven invocations. When you report a count, say what you counted and do not compare counts from different projects.

Warnings that are not failures. On a recent JDK a passing build can still print warnings. Maven's own libraries may print `WARNING: A terminally deprecated method in sun.misc.Unsafe has been called`; they come from Maven itself and do not mean your project is wrong. Mockito may print a notice that it is attaching itself as a Java agent (see B.5). Read the last lines: `BUILD SUCCESS` and a zero exit status mean the build passed.

B.5 JUnit and Mockito

JUnit Jupiter is the test framework. The single dependency `junit-jupiter` brings the API for `@Test`, the parameterised-test support (`@ParameterizedTest`) and the engine that runs them, and Surefire runs it through the JUnit Platform [9]. The POM gives it the scope `test`, so it is on the class path for tests and not for the application.

Mockito creates test doubles. The book uses it in one place, to replace the notification interface in Chapter 7, and configures it for **interface-only** doubles. A one-line file, `src/test/resources/mockito-extensions/org.mockito.plugins.MockMaker`, contains `mock-maker-proxy`, which selects the mock maker that Mockito's documentation describes as avoiding code generation and able to mock only interfaces [10]. With it you cannot mock a final class or a `Clock`; use a real value or a small hand-written fake instead. Chapter 7 shows hand-written fakes and explains where each kind of double fits.

Whether you may use a mocking framework in an assessment is decided in your assessment instructions. This appendix only records how the book configures it.

The agent warning. On a recent JDK, Mockito can still load its helper as a Java agent when it starts, and the JVM prints a warning that a Java agent has been loaded dynamically. The tests pass either way. The JDK's own documentation says that dynamic loading will be disallowed by default in a future release, and names the option `-XX:+EnableDynamicAgentLoading` that permits it and hides the warning [11]. To run the starter without the warning:

```
mvn --batch-mode -DargLine=-XX:+EnableDynamicAgentLoading verify
```

B.6 Static analysis with PMD (Chapter 5)

Chapter 5's lab uses PMD 7.28.0. It is not a Maven dependency: you download it once and run it from the chapter's script.

1. Download `pmd-dist-7.28.0-bin.zip` from the project's GitHub releases page (tag `pmd_releases/7.28.0`) [12].
2. Compare its SHA-256 digest with the one the release page shows. The digest recorded for this file is:

```
f974ba571f7bc01c73fffe11bade25fbc1f438698f9e9c00e416e8d65e9fbac2
```

3. Unzip it into a new folder and set `PMD_HOME` to the `pmd-bin-7.28.0` folder inside it. PMD needs a Java runtime of version 8 or above.

4. From each Chapter 5 version, run `bash scripts/pmd.sh`. Confirm the installation with `"$PMD_HOME/bin/pmd" --version` (`pmd.bat` on Windows), whose output includes `PMD 7.28.0`.

A static analyser reports patterns, not defects. Its output is specific to its version and ruleset, and a different version may report different findings. Treat a finding as a prompt for review.

B.7 Dependency-injection containers (Chapter 6)

The book uses no library container. Chapter 6 walks through a small educational container written in plain Java, so there is nothing to install for that section. It also compares the walk-through with what a framework container does, for reading only. If your team uses a framework container in its own project, record its name and exact version in your team's setup notes, as the pinned tools above are recorded here.

B.8 Git and the pull-request routine

One-time setup

Tell Git who you are, once per machine. Use your own name and address; do not use another person's identity.

```
git config --global user.name "Your Name"
git config --global user.email "you@example.org"
```

On Windows, Git may convert line endings to CRLF when it checks files out and print a warning such as `LF will be replaced by CRLF the next time Git touches it`. The warning is harmless for Java files. It matters for shell scripts, because Linux Bash and WSL do not accept a CRLF script (`set -euo pipefail` fails with `invalid option name`). Every starter in this book therefore contains a `.gitattributes` file with one line, `*.sh text eol=lf`, which keeps shell scripts in LF form [13].

The commands you will use

The companion's `git-walkthrough.sh` runs all of these in a throwaway folder, with a bare repository standing in for the remote so that no account or network is needed. Its output matches what is shown here.

Task	Command	What to look for
Start a repository	`git init -b main`	An empty history on branch `main`.
See what changed	`git status --short`	`M` modified, `A` staged new file, `??` untracked.
See the change	`git diff`	Unstaged changes only.
See what will be committed	`git diff --staged`	Staged changes. An empty `git diff` after staging is normal.
Stage a file	`git add path/to/file`	Name files; avoid `git add .` until you have read `git status`.
Record a commit	`git commit -m "Say what changed"`	One logical change per commit.
Start a branch	`git switch -c feature/short-name`	`Switched to a new branch`.
Return to a branch	`git switch main`	
Discard an edit to a tracked file	`git restore path/to/file`	The edit is gone: this cannot be undone [14].
Unstage a file	`git restore --staged path/to/file`	The file stays changed but leaves the next commit.
Reverse a commit safely	`git revert --no-edit HEAD`	A new commit that undoes the old one; history is kept [15].
Combine branches	`git merge feature/short-name`	`Fast-forward` when `main` has not moved, otherwise a merge commit.
Look at history	`git log --oneline --graph --decorate`	One line per commit, with the branches.
Share	`git push -u origin feature/short-name`	The branch exists on the remote and is tracked.
Copy a repository	`git clone URL`	A full copy with its history.

Two cautions. First, `git restore` on a tracked file and `git reset --hard` throw away uncommitted work. Run `git status` first. Second, never rewrite the history of a branch that other people have pulled, for example with a forced push, unless your team has agreed to it.

From change to merge

A pull request (a merge request on some platforms) is a proposal, on a hosting platform, to bring one branch into another so that others can review it. It is not the same as a commit [16]. A routine that works for a small team:

1. Update your copy of the shared branch.

2. Create a short branch for one change.
3. Make small commits, each with a message that says what changed. A team may adopt a written convention such as Conventional Commits; use whichever it agrees and apply it consistently (Chapter 2).
4. Run the project's check script before you push, and keep its output.
5. Push the branch and open the pull request. Describe the problem, the limited change and the checks you ran, with their results.
6. Reviewers read the diff and comment on what they inspected and why it matters. Answer by adding commits, not by rewriting the ones they have read.
7. Merge when the agreed review is complete and the required check has passed. Platforms offer a merge commit, a squash or a rebase; your repository's setting decides which one you get.
8. Delete the merged branch and update your copy of the shared branch.

Steps 5 to 8 happen in the hosting platform's interface, which differs between platforms and institutions. The book supplies no remote and no account. Record real pull-request addresses and real review comments in your team's evidence notes, and never invent review attribution. Commit and pull-request counts show activity, not contribution or quality; Chapter 14 and Appendix E explain what such evidence can and cannot show.

B.9 A starter hosted check

The companion lists `.github/workflows/verify.yml`. It asks GitHub Actions to run the project's check script on every push and pull request, and when started by hand:

```yaml
name: Verify

on:
  push:
  pull_request:
  workflow_dispatch:

permissions:
  contents: read

jobs:
  verify:
    runs-on: ubuntu-latest
    timeout-minutes: 10
    steps:
      - uses: actions/checkout@v7
      - uses: actions/setup-java@v6
        with:
          distribution: temurin
          java-version: '17'
          cache: maven
      - name: Build and test
        run: bash scripts/check.sh
```

`permissions: contents: read` gives the job read access only. `actions/checkout` fetches your repository, and `actions/setup-java` installs a Java 17 JDK from the Temurin distribution and, with `cache: maven`, keeps Maven's downloads between runs [17][18]. The job then runs the same script you run locally, so a local pass and a hosted pass mean the same thing. A push that belongs to an open pull request can start two runs; the template accepts that.

A workflow file in the repository is only a request. It becomes evidence when it has actually run on your repository, and you can quote its run address. A green run shows that the script passed on a clean machine of one kind. It does not show that the tests are good, that the design is sound or that a reviewer agreed. To make a pull request wait for the check, configure the shared branch to require it where the platform supports that, or write the manual rule down. If no hosted platform is available, run the script locally and keep its output; that is a weaker but honest substitute.

Major action versions and runner images change. Check the current versions before you reuse the file, and record the versions that a run actually resolved from its log.

B.10 Repository metrics

Chapter 12 measures flow from real data, and its repository lab pins one method, RM-12, with a supplied calculator and a blank evidence record. This appendix adds only the data-collection facts that every method shares.

Git alone can list commits with their times, for example:

```
git log --first-parent -20 --format='%H%x09%cI'
```

`%H` is the commit hash and `%cI` the committer time in strict ISO form. Following only the first parent changes which history you see. A list of commits does not give a pull request's opening time, its review time or any deployment time, and commit counts do not measure work. Those times come from the hosting platform's interface or an authorised export with exact timestamps. If you can see only relative dates such as "3 days ago", you cannot calculate a precise duration; say so, and do not guess a time.

The module's agreed repository-metrics method is not settled by this appendix. Until your module states one, follow Chapter 12.

B.11 When something goes wrong

What you see	Likely cause	What to do
`mvn: command not found`	Maven's `bin` folder is not on the `PATH`.	Run `scripts/check-setup.sh`; fix the `PATH` and open a new shell.
`MISSING` or `TOO-OLD` from `check-setup.sh`	A tool is absent, or the JDK is older than 17 or has no compiler.	Install a full JDK of version 17 or later.
`Cannot access central ... in offline mode`	`--offline` with an empty or incomplete local repository.	Run one online build, or supply a populated repository folder.
`BUILD FAILURE`, `Tests run: ..., Failures: 1`	A test failed.	Open the named test, read the expected and actual values, and decide whether the code or the test is wrong.
`No tests matching pattern ... were executed`	`-Dtest=` named a class or method that does not exist.	Check the spelling; the message names the pattern it used.
`WARNING: ... sun.misc.Unsafe ...`	Maven's own libraries on a recent JDK.	Ignore if the build ends in `BUILD SUCCESS`.
`WARNING: A Java agent has been loaded dynamically`	Mockito attaching its helper (B.5).	Ignore, or add `-DargLine=-XX:+EnableDynamicAgentLoading`.
`set: pipefail: invalid option name`	A shell script was checked out with CRLF line endings and run by Linux Bash or WSL.	Add `.gitattributes` with `*.sh text eol=lf` and check the file out again (Chapter 4).
`LF will be replaced by CRLF` from Git	Windows line-ending conversion.	A warning only; see B.8.
A script behaves differently in two terminals	Different programs called `bash` on the `PATH`.	Run it from Git Bash and check with `command -v bash`.

B.12 A short verification activity

Do these in a copy of the companion's starter project, in a throwaway folder.

1. Run `bash scripts/check-setup.sh`. Resolve any `MISSING` or `TOO-OLD` line.
2. Run `bash scripts/check.sh`. Note the number of tests Maven reports and say what it counted.
3. In a scratch copy, change `≥` to `>` in `ToolchainReport.canRunOn`, and run `bash scripts/check.sh` again. Read the failing test's message. Restore the original.
4. Initialise a repository, commit the starter on `main` and make the one-line change from step 3 on a branch. Run `git diff`, stage the file, run `git diff --staged`, and commit.
5. Write a short setup note: the versions your machine reported, the commands you ran and anything that differed from this appendix.

Optional AI activity, with an identical offline route. The following is an author-constructed candidate answer, not a captured model response:

> "Any Java installation will do because the code is plain Java. To confirm a project works, run `mvn install -DskipTests`. If tests fail unpredictably, add `-Dmaven.test.failure.ignore=true` so the build always passes. Share your work with `git add .` and `git push --force`."

Check each sentence against the starter and this appendix, and say what evidence you used. A live assistant may produce a different candidate if your course allows one, but it is not required. Record the claim, your check and your conclusion; do not accept an unsupported explanation as the final evidence.

What you hand in at the end is the short setup note. This is formative practice and adds no assessment requirement.

Sources and further reading

Official references opened on 13 October 2026. The explanations and commands are original; source examples have not been reproduced.

1. Oracle, **The javac Command, Java SE 17**. The `--release` option and compilation target.
2. Eclipse Adoptium, **Temurin**. A source of JDK builds; the build named in the hosted-check example.
3. Apache Maven, **Installation**. Prerequisites, extracting the archive, adding `bin` to the path and `mvn -v`.
4. Apache, **Maven 3.9.11 binary archive and checksum**. The archive and its `.sha512` file.
5. Apache Maven, **Introduction to the build lifecycle**. Phases and their order.
6. Apache Maven, **Surefire: running a single test**. The `-Dtest` parameter.
7. Apache Maven, **Surefire and the JUnit Platform**. Tag selection with `groups` and `excludedGroups`.
8. Apache Maven, **Settings reference**. The local repository, offline mode and proxies.
9. JUnit, **User Guide 5.13.4**. Dependency artifacts and parameterised tests.
10. Mockito, `MockMakers`. The proxy mock maker, which can mock only interfaces.
11. OpenJDK, **JEP 451: Prepare to Disallow the Dynamic Loading of Agents**. The warning and the `-XX:+EnableDynamicAgentLoading` option.
12. PMD, **Installation** and the **7.28.0 release page**.
13. Git, **gitattributes**. The `eol` and `text` attributes.
14. Git, **git-restore** and **git-switch**.
15. Git, **git-revert**. A revert records a new commit.
16. GitHub Docs, **About pull requests**. Also Chacon and Straub, **Pro Git: branching**.
17. GitHub, **actions/checkout**. Usage and versions.
18. GitHub, **actions/setup-java**. Distributions, `java-version` and `cache: maven`.

Appendix C: Engineering Templates

> *Thirteen templates that follow the chapters' own records, each with a purpose, a blank form, a worked example and the usual mistakes, and a script that checks their structure.*

OO Development and Agile Practices
Designing, Testing and Evolving Software with AI Assistance

OO Development and Agile Practices
Designing, Testing and Evolving Software with AI Assistance

How to use this appendix

A template is a set of questions that a team agrees to answer every time. It saves you from forgetting the question that matters at the moment you are tired. It is not evidence. A story card with every field filled in shows that someone wrote down a story, a retrospective form shows that a meeting was held, and neither shows that the story was understood or that anything changed afterwards. The Chapter 3 companion says it plainly: a template is not evidence that a review occurred.

This appendix collects the thirteen templates that the chapters use, in the order you will meet them. Each section gives the purpose of the template, when to use it, the blank template, a small worked example and the mistakes to avoid. The examples are **constructed to illustrate**. They belong to an invented team and are labelled as such in the first line of the file. They are not real evidence, and you should not copy their contents into your own record.

Your approved instructions say what your module requires. The templates add no assessment requirement. If your instructions name a different form, use that form; the questions in these templates will still help you fill it in.

The Appendix C companion lists the same 26 files, each under its path, together with a small script that checks the structure of a filled-in template, the experiments that show what it catches, the answers to the activity in C.15 and notes for instructors. Each template and each example in this appendix is introduced by its path. Saved at those paths, the 26 blocks and the script form the kit that the script checks.

Each template uses the headings of the record that the chapters themselves use, so a document made from it follows the chapter that introduced it. A decision record has the headings of the project's ADR-010, ADR-013 and ADR-014, and the AI assistance record has the eight headings that the Chapter 15 lab checks. The script accepts those records unchanged.

C.1 Using a template well

1. **Start from the question, not from the form.** Decide what the document must let a stranger do (re-run a check, understand a choice, find the evidence), and write for that person.
2. **Replace every `{{placeholder}}`.** A placeholder left in a document is a promise you did not keep. The script in the companion looks for them.

3. **Write "none yet" rather than invent.** An empty evidence field is honest. A made-up run address or review comment is not. Say what is missing and what the next action is.
4. **Keep the headings.** Other people and the script find things by heading. Add a heading if you need one; do not rename or delete one.
5. **Say what is constructed.** An example, a sample or a model answer is labelled as one, as the examples here are.
6. **Keep it with the work.** A template filled in and left in a chat message is lost. Commit it beside the code it describes, so that a change to one prompts a change to the other.

A script can check the shape of a filled-in template. It cannot check that the content is true.

The script can check	Only a reviewer can check
Every required heading is present	That the answer under it is correct
No heading is empty	That "reviewed and verified" describes a real review
No `{{placeholder}}` is left	That the evidence named is the right evidence
An example is labelled as constructed	That a hosted run, a reviewer or a date exists

C.2 Story card

Purpose. A story card records one piece of value for one kind of user, with its boundaries, so that two people reading it expect the same thing. **Use it** when you refine a backlog item (Chapter 3), and again when the item is finished, to add the evidence.

Save as `templates/story-card.md`:

```
# {{ID}}: {{short outcome}}

## Actor and value
{{Who needs this, and why. One or two sentences.}}

## Scope
{{What this item includes, and the boundary of the demonstration.}}

## Rules and assumptions
{{Confirmed decisions first. Then assumptions, each marked as temporary.}}

## Acceptance examples
{{Inputs, action, expected result and state effects. Link each to its acceptance-example file.}}

## Current support
{{Existing code or evidence, or "none".}}

## Dependencies and questions
{{Blocking decisions, related items, and who follows them up.}}

## Estimate
{{Optional: a relative size, its assumptions and the date it was discussed.}}

## Completion evidence
{{Links or notes for checks, review, integration and demonstration. Say "not started" until someth
```

Save as `examples/story-card.md`:

```
<!— Example: constructed to illustrate. Not real evidence. —>
# US-02: Inspect availability without reserving

## Actor and value
As an equipment officer, I want to see whether one item is free for a stated period, so that I ca

## Scope
One item and one period per enquiry. No reservation is created or changed. Searching several item

## Rules and assumptions
Confirmed: a period includes its start day and excludes its end day (Chapter 2). Assumption, to b

## Acceptance examples
AQ-01 an empty store reports the item available. AQ-02 an overlapping reservation reports it unav

## Current support
None. The only operation today is `reserve`, which stores a record.

## Dependencies and questions
How an invalid period is reported (an exception or a result value) must be decided before buildin

## Estimate
Medium, assuming no catalogue lookup. Discussed at the planning meeting.

## Completion evidence
Not started. The evidence will be the acceptance checks, a review note and the passing run on the
```

Common mistakes. Writing the story as a feature name with no actor and no value. Listing rules as if all were confirmed when some are guesses (separate them, as the example does). Filling in "Completion evidence" before anything exists. Marking a story done because the code exists, without the evidence the team agreed.

C.3 Acceptance example

Purpose. An acceptance example turns a criterion into one concrete case that two people can check the same way. It names the observation and the state that must remain unchanged. **Use it** for each important criterion of a story (Chapter 3), including rejection paths.

Save as `templates/acceptance-example.md`:

```
# {{AC-ID}}: {{the criterion in one sentence}}

## Given
{{The starting state, with concrete values.}}

## When
{{The one action, with concrete values.}}

## Then
{{What the caller observes: a result, a message or an error.}}

## State afterwards
{{What stored data must be unchanged or changed. Write "unchanged" when it is.}}

## Evidence
{{The check that runs this example, and the run that executed it.}}
```

Save as `examples/acceptance-example.md`:

```
<!— Example: constructed to illustrate. Not real evidence. —>
# AC-02: An overlapping request is rejected and changes nothing

## Given
CAM1 is reserved for B1 from 20 October to 22 October (start included, end excluded).

## When
B2 requests CAM1 from 21 October for one day.

## Then
The request is rejected as unavailable.

## State afterwards
The original reservation is the only stored record. Nothing was added or changed.

## Evidence
The AC-02 check in `Chapter03AcceptanceExamples`, and the run of the check script that printed it
```

Common mistakes. Describing only the visible result and not the state afterwards: a program that prints the right rejection and still stores a record has not met the criterion. Using vague values (“a long period”) instead of concrete dates. Putting two actions in “When” . Pointing “Evidence” at a check that does not run this example.

C.4 Definition of Done

Purpose. A Definition of Done is the quality standard that every increment must meet, whatever the story. It differs from acceptance criteria, which belong to one story, and from a Ready policy, which says when work may start. The Scrum Guide makes the Definition of Done a commitment attached to the increment [1]. **Use it** from the first iteration (Chapters 3 and 4), and extend it when the team gains a check, such as an automated gate.

Save as `templates/definition-of-done.md`:

```
# Definition of Done: {{team or project}}

## Applies to
{{What counts as an increment, and the version or date of this definition.}}

## Conditions
1. {{A condition that another person could check.}}
2. {{Another condition.}}

## Not covered
{{What meeting this definition does not promise.}}

## Ready policy
{{Optional: what the team needs to know before it starts an item.}}

## Agreement and review
{{Who agreed it, when, and what would make the team revisit it.}}
```

Save as `examples/definition-of-done.md`:

```
<!— Example: constructed to illustrate. Not real evidence. —>
# Definition of Done: equipment lending team

## Applies to
Each increment of the controlled teaching baseline, from the first iteration.

## Conditions
1. The relevant acceptance examples have been checked, including the important rejection paths an
2. The code compiles and runs with the documented commands in the agreed environment.
3. Another team member has reviewed the change, and findings that block acceptance are resolved.
4. The work is integrated into the shared codebase, and the existing checks still pass.
5. Usage notes, domain decisions and known limitations are current.
6. Another team member can repeat the demonstration within its stated scope.
7. The agreed automated check has passed on the integrated revision, and the team can find its re

## Not covered
Approval by a product owner, performance, and any grading rule of the module. Condition 7 applies

## Ready policy
An item is ready to start when its scope, its example inputs and any blocking question are writte

## Agreement and review
Proposed by the team for the controlled increment. Revisit it when the automated check or the rev
```

Common mistakes. Listing conditions nobody could check (“good quality”). Claiming a condition that does not exist yet, such as a passing pipeline before there is one; the example says when condition 7 starts to apply. Weakening the definition to move a card across the board. Treating a team’ s definition as a grading policy.

C.5 Decision record

Purpose. A decision record captures a decision and the reasons for it, not a description of the code. Nygard proposes short numbered records kept in the repository, with a title, context, decision, status and consequences, and says that a reversed decision is kept and marked superseded, not deleted [2]. The project’ s records add alternatives considered, how the decision was validated and what would reopen it. **Use it** whenever a choice will be hard to explain later: the storage choice at kickoff (Chapter 4), the application structure (Chapter 10), the visitor flag (Chapters 13 and 14).

Save as `templates/decision-record.md` :

```
# ADR-{{NNN}}: {{the question and the choice}}

Date: {{date}}
Status: {{proposed, accepted, or superseded by ADR-NNN}}

## Context
{{The forces at play: the problem, the constraints, and what must not change.}}

## Decision
{{What was decided, in full sentences.}}

## Alternatives considered
- {{An alternative, and why it was not chosen.}}

## Consequences
{{What becomes easier, what becomes harder, and what is still unsolved.}}

## Validation and review
{{How the decision was checked, and by whom. Do not claim a review that did not happen.}}

## Revisit triggers
{{The conditions that would reopen this decision.}}
```

Save as `examples/decision-record.md`:

```
<!-- Example: constructed to illustrate. Not real evidence. -->
# ADR-002: Keep reservations in memory for the kickoff baseline

Date: 13 October 2026
Status: Accepted for the kickoff baseline.

## Context
The kickoff demonstration judges reservation decisions in one session. The team has no storage re

## Decision
Keep one in-memory coordinator. A restart loses every record, and identifiers start again from th

## Alternatives considered
- A file or database store: it would survive a restart, but no selected outcome needs that, and i
- Do nothing explicit: the behaviour would be a surprise to the first user who restarts the progr

## Consequences
The demonstration is simple and fast to run. The officer must be told that records do not persist

## Validation and review
Discussed at the team meeting. No independent review has taken place.

## Revisit triggers
A selected outcome requires a session to survive a restart, or two users need the same data at on
```

The numbers in the examples belong to the example team. They are not the book's project's ADR-010, ADR-013 and ADR-014. Once a record is accepted, change only its status. A new decision gets a new record that says which one it supersedes.

A smaller form. For a small choice, such as accepting or rejecting a proposed criterion, a one-row table can be enough:

Proposal	Accept, reject or clarify	Reason and evidence	Follow-up or related item
Treat an enquiry as read-only	Accept	An enquiry must not create a record (AC for US-02)	Add AQ-05 for equal dates

The check script covers the full record and not this table.

Common mistakes. Recording a decision after the fact without saying so. Editing an old record to match what happened. Listing only the option that was chosen. Writing "reviewed" under validation when nobody reviewed it.

C.6 Iteration plan

Purpose. An iteration plan makes capacity visible next to the work, so that the team plans what it can verify and not what it hopes. It shows the goal, the hours available, the forecast and what would make the team replan. **Use it** at the start of an iteration (Chapter 4).

Save as `templates/iteration-plan.md`:

```
# Iteration plan: {{team}}, {{dates}}

## Iteration goal
{{One sentence naming an outcome a user could observe.}}

## Capacity
| Person | Hours available |
|---|---:|
| {{name or initials}} | {{hours}} |

Reserved for coordination, integration and administration: {{hours}}.
Planning budget: {{hours}}.

## Forecast work
| Work | Person-hours |
|---|---:|
| {{task}} | {{hours}} |

Selected work: {{total}}. Unallocated: {{budget minus selected work}}.

## Calendar and skill constraints
{{Who is available when, who knows what, and when the review can happen.}}

## When to replan
{{The evidence that would make the team change the forecast, and who decides.}}
```

Save as `examples/iteration-plan.md`:

```
<!— Example: constructed to illustrate. Not real evidence. —>
# Iteration plan: example team, next iteration

## Iteration goal
Let the officer inspect availability for one item and period without changing reservations.

## Capacity
| Person | Hours available |
|---|---:|
| P1 | 6 |
| P2 | 6 |
| P3 | 5 |
| P4 | 5 |

Reserved for coordination, integration and administration: 4.
Planning budget: 18 (22 available, less 4).

## Forecast work
| Work | Person-hours |
|---|---:|
| Resolve the query contract and update the examples | 2 |
| Implement the read-only enquiry | 4 |
| Add the thin interaction and the invalid-input response | 3 |
| Add and run the query checks beside the regression checks | 3 |
| Review, integrate, update notes and rehearse the demonstration | 2 |

Selected work: 14. Unallocated: 4, kept as room for uncertainty and not as a promise to add a fea

## Calendar and skill constraints
Only P1 can explain the coordinator, and the reviewer is free only at the start. Hold an early pa

## When to replan
If an estimate rises by more than two hours, discuss the scope at the next meeting. Reduce an opt
```

The figures are invented for reasoning. They are not official contact hours and not measured productivity. Review and correction consume capacity, so they appear in the forecast; calling them "after development" does not make them free. A total that fits does not make the plan feasible, which is why the example names the constraints on people and time.

Common mistakes. Allocating every available hour to features. Converting story points into hours as if they were the same thing. Hiding an overrun by postponing checks. Treating the unallocated hours as a promise to add scope.

C.7 Team agreement

Purpose. A team agreement answers the practical questions that otherwise become arguments: where requirements live, how work moves, who reviews, which check runs and what happens when evidence is unavailable. **Use it** when the team forms and revisit it at each retrospective (Chapter 4).

Save as `templates/team-agreement.md`:

```
# Team agreement: {{team}}

## Where things live
{{Where requirements, questions, decisions and evidence are kept.}}

## How work moves
{{The board columns, the limit on work in progress, and what blocked work does.}}

## Branches and review
{{How big a branch may be, who reviews, and what a reviewer looks at.}}

## Checks
{{Which check the author runs, when, and where its result is recorded.}}

## Gaps and unavailable evidence
{{What the team does when a review or a hosted run is not available.}}

## Review of this agreement
{{When the team rereads this, and how a change is recorded.}}
```

Save as `examples/team-agreement.md`:

```
<!— Example: constructed to illustrate. Not real evidence. —>
# Team agreement: example team

## Where things live
Stories and acceptance examples are in the shared project notes, with their identifiers kept when

## How work moves
Backlog, Ready, Building, Review, Done. Building plus Review holds at most two items, and a block

## Branches and review
A branch is small enough that a reviewer can explain its purpose and effects. Reviewers inspect b

## Checks
The author runs the check script before review and again after corrections. The result on the int

## Gaps and unavailable evidence
If a review or a hosted run is not available, the team writes the gap and a next action in the ev

## Review of this agreement
Reread at each retrospective. A change is recorded with its reason in the project notes.
```

A repository does not enforce a policy because a document requests it. Where the platform allows, configure the shared branch to require the agreed check and review; where it does not, say that the team follows a manual policy and keep the review evidence. Do not describe a requested policy as enforced unless it is.

Common mistakes. Copying another team's agreement without adapting it. Leaving the gaps section empty and then inventing a result when evidence is missing. Never rereading it.

C.8 Pull-request description

Purpose. A pull request, or merge request, asks a hosting platform to bring one branch into another so that others can review it. It is not a commit. Its description tells the reviewer what to look at before they open the diff. **Use it** for every change you ask someone to review (Chapters 2 and 4, and B.8).

Save as `templates/pull-request.md`:

```
# {{short title of the change}}

## Problem
{{What is wrong or missing, and for whom.}}

## Change
{{What this change does, and what it deliberately does not do.}}

## Checks run
{{The commands, the results, and where the output is kept.}}

## Review focus
{{What the reviewer should look at first, and why it is the risk.}}

## Limits
{{What the checks do not show, and what was not run.}}
```

Save as `examples/pull-request.md`:

```
<!— Example: constructed to illustrate. Not real evidence. —>
# Extract reservation confirmation formatting

## Problem
The request method mixes storing a reservation with assembling the confirmation text, so a change

## Change
Moves the text assembly into a private helper. The messages and the stored records are unchanged.

## Checks run
The check script before and after the change printed the same results. The output is attached to t

## Review focus
Confirm that the helper changes no state, and that the record is still added before the confirmat

## Limits
Only the supplied checks were run. No hosted check has run on this branch yet.
```

On GitHub, a file named `pull_request_template.md` in the repository fills the description of every new pull request with your headings [3]. Other platforms have their own equivalents.

For the reviewer. Start from the claimed behaviour, then trace the code or text that is supposed to deliver it, then ask whether the evidence supports the claim. “Looks good” does not say which risk was examined. “The helper changes no state, and the record is still stored before the confirmation is returned” does. Write what you inspected.

Common mistakes. A description that restates the diff. A change that mixes a functional change with a workflow change, which is hard to review. Quoting a check result that was not run on this revision. Inventing a review exchange to complete a milestone template: keep the real comments and the real response, or write that none exist.

C.9 Evidence note

Purpose. An evidence note collects what the team can show, with the gaps visible, so that a reader can tell a result from a hope. **Use it** at each milestone (Chapter 4); the repository-metrics lab of Chapter 12 has its own record with different fields.

Save as `templates/evidence-note.md`:

```
# Evidence note: {{team}}, {{date}}

## Repository and baseline revision
{{The repository and the exact revision this note describes.}}

## Implemented behaviour
{{Only the behaviour present at that revision.}}

## Local environment and command
{{The Java version, the shell, and the command that was run.}}

## Local check result
{{The observed exit status and the relevant output.}}

## Hosted passing run
{{The run address or retained log, the revision it tested, and the date. Or "none yet" and the ne

## Deliberate failure
{{The practice revision, the failed assertion, and the failed run. Or "not done yet".}}

## Restored pass
{{The restored revision and its passing run. Or "not done yet".}}

## Independent review
{{The actual reviewer, the finding, the correction and the follow-up. Or "none yet".}}

## Unresolved questions
{{Each with an owner, its impact and the next decision point.}}

## Known limitations
{{Scope boundaries that matter for reading the demonstration.}}
```

Save as `examples/evidence-note.md`:

```
<!— Example: constructed to illustrate. Not real evidence. —>
# Evidence note: example team, week 2

## Repository and baseline revision
The team repository, at the revision tagged for the first iteration. The revision identifier is c

## Implemented behaviour
US-01 only: an eligible request for an available item creates one stored reservation. No availabi

## Local environment and command
A full JDK of version 17 or later, Git Bash, and `bash scripts/check.sh`.

## Local check result
Exit status 0, ending with the success line of the script. Output saved with this note.

## Hosted passing run
None yet. The workflow file is in the repository but has not run. Next action: push the branch, o

## Deliberate failure
Not done yet. Planned for the next meeting on a practice branch.

## Restored pass
Not done yet.

## Independent review
None yet. A self-review was written for practice and is labelled as a self-review.

## Unresolved questions
How an invalid period is reported (owner: P2; decide before US-02 is built). Who may cancel a res

## Known limitations
Records live in memory and vanish on restart. Requests are handled one at a time.
```

Look at the example's hosted-run field. It says "none yet" and gives the next action. That is the right way to record a gap. A passing workflow file in the repository is not a passing run, and a hosted run you did not do cannot be written down.

Common mistakes. Writing the planned behaviour as if it were implemented. Citing the check result of one revision for another. Copying an example's "restored pass" into a note when no failure was ever introduced. Leaving the template's guidance text in place of an answer.

C.10 Requirements to evidence, and known issues

Purpose. Two tables carry a project through a handover (Chapter 14). The **requirements-to-evidence** table follows each requirement to the class that decides it and the test that checks it, so that a reviewer can find the evidence. The **known-issues** list says what is wrong or missing and, for each row, what was decided. **Use them** when you prepare a release and when you hand a project to someone else.

Save as `templates/requirements-to-evidence.md`:

```
# Requirements to evidence: {{project}}

## Table
| Requirement or example | Class that decides it | Test that checks it | Stage that runs it |
|---|---|---|---|
| {{ID and one line}} | {{class}} | {{test class and method}} | {{script or pipeline stage}} |

## How this table is kept true
{{The script or test that fails when a name in the table no longer exists.}}

## What this table cannot show
{{For example: that a named test is the right test for its requirement.}}
```

Save as `examples/requirements-to-evidence.md`:

```
<!— Example: constructed to illustrate. Not real evidence. —>
# Requirements to evidence: example project

## Table
| Requirement or example | Class that decides it | Test that checks it | Stage that runs it |
|---|---|---|---|
| AC-01 an eligible request creates one reservation | `ReservationCoordinator` | `Chapter03Accepta
| AC-02 an overlapping request is rejected and changes nothing | `ReservationCoordinator`, `LoanP

## How this table is kept true
A reviewer reads it at each release. A script that fails when a named class or test disappears is

## What this table cannot show
That each named test is the right evidence for its requirement. A name can exist and still check
```

Save as `templates/known-issues.md`:

```
# Known issues: {{project}}

## Issues
| ID | What is wrong or missing | Effect | Decision | Owner | Review trigger |
|---|---|---|---|---|---|
| {{KI-n}} | {{description}} | {{who is affected, how}} | {{fix, accept or defer, and why}} | {{na

## How this list is kept true
{{Who reads it, and when. Every row needs a decision, not only a description.}}
```

Save as `examples/known-issues.md`:

```
<!— Example: constructed to illustrate. Not real evidence. —>
# Known issues: example project

## Issues
| ID | What is wrong or missing | Effect | Decision | Owner | Review trigger |
|---|---|---|---|---|---|
| KI-1 | Reservations are kept in memory | A restart loses every record | Accept for the kickoff |
| KI-2 | The hosted workflow has never run | No hosted result exists for any revision | Defer unt

## How this list is kept true
Read by the team at each retrospective. A row without a decision is not finished.
```

A table that was true once rots. The cheapest way to keep a statement true is a script that fails when it stops being true, as the book's project's traceability test does. Even then, a script can check that a named test exists; only a reviewer can check that it is the right test. The example says so.

Common mistakes. A known-issues row with a description and no decision. Listing a test that exists but checks something else. Deleting a known issue to make the list shorter instead of resolving it.

C.11 Release notes

Purpose. Release notes tell a person what changed in a version. Keep a Changelog groups changes as Added, Changed, Deprecated, Removed, Fixed and Security, gives each version its own dated entry and puts the newest first [4]. The template keeps three of those headings and adds two that a handover needs: behaviour a reader could misread, and what was not verified. **Use it** at each release (Chapter 14).

Save as `templates/release-notes.md`:

```
# Release notes {{version}}

Date: {{date}}

## Summary
{{One or two sentences. Say whether production behaviour changed.}}

## Added
{{New behaviour, or "nothing".}}

## Changed
{{Changed behaviour, or "nothing".}}

## Fixed
{{Defects fixed, or "nothing".}}

## Behaviour you should know about
{{Behaviour a user or a maintainer could misread.}}

## Not verified
{{What was not run or not checked for this version.}}
```

Save as `examples/release-notes.md`:

```
<!-- Example: constructed to illustrate. Not real evidence. -->
# Release notes 0.2.0

Date: 20 October 2026

## Summary
Adds a read-only availability enquiry. The existing reservation behaviour is unchanged.

## Added
An enquiry that reports whether one item is free for a stated period, without creating a reservat

## Changed
Nothing.

## Fixed
Nothing.

## Behaviour you should know about
An item can be reported available even when a borrower cannot reserve it, because that borrower ha

## Not verified
No hosted run of the workflow exists for this version. Behaviour on a Java 17 runtime was not run
```

Add a heading for Deprecated, Removed or Security when a release has such a change. One file per version, written at the release.

Common mistakes. A summary that says "various improvements". Saying production behaviour did not change without checking. Leaving "Not verified" empty because nothing seemed missing.

C.12 Retrospective

Purpose. A retrospective plans ways to increase quality and effectiveness. The Scrum Guide describes the team inspecting how the last sprint went with regard to individuals, interactions, processes, tools and its Definition of Done, and identifying the most helpful changes [1]. Make it about evidence and not mood: for each thing the team wants to say, ask which artefact shows it. **Use it** at the end of each iteration (Chapter 14).

Save as `templates/retrospective.md`:

```
# Retrospective: {{team}}, {{period}}

## Period and evidence brought
{{The dates, and the artefacts the team brought: board, runs, reviews, defects.}}

## What the evidence shows
| Observation | Evidence | Action | Owner and date |
|---|---|---|---|
| {{what the team says}} | {{the artefact that shows it}} | {{one change}} | {{name, next iterati

## Process note
{{Which adaptations of the process helped, which hurt, and what to change.}}

## Actions agreed
{{At most three, each with an owner and a date.}}

## Last retrospective's actions
{{What happened to each action agreed last time.}}
```

Save as `examples/retrospective.md`:

```
<!— Example: constructed to illustrate. Not real evidence. —>
# Retrospective: example team, first iteration

## Period and evidence brought
The first iteration. The team brought the board with its timestamps, the list of check runs and t

## What the evidence shows
| Observation | Evidence | Action | Owner and date |
|---|---|---|---|
| Reviews were slow | Two requests waited more than two days (board timestamps) | Review before s
| The checks were unreliable | One of ten runs failed with no code change (run list) | Find and f
| We rushed the end | Seven items moved to Done on the last day | Plan a smaller scope and freeze

## Process note
Our iterations end at the module's milestone weeks, not at a fixed interval. That helped us plan

## Actions agreed
1. Review before starting a new item (P2, next iteration). 2. Fix or isolate the unstable check (

## Last retrospective's actions
None. This was the first retrospective.
```

Keep the number of actions small. Two actions with owners that get done are worth more than ten that do not. The process note closes the commitment from Chapter 3: say which adaptations of the process helped, which hurt and what you would change. The last field makes the team look at what became of last time’ s actions, which is where most retrospectives fail.

Common mistakes. An observation with no evidence. Actions with no owner or no date. Naming a person as the cause of a problem; look at the work and the process, not at blame.

C.13 AI assistance record

Purpose. An AI assistance record says what was asked, what was received, what was accepted or rejected, and what checks were run, so that a reader can judge the result without trusting it. **Use it** whenever an assistant contributes something you will rely on (Chapter 15). The lab in Chapter 15 supplies a script that checks the same eight headings.

Save as `templates/ai-assistance-record.md`:

```
# AI assistance record: {{task}}

## Task
{{What was asked, in one or two sentences, and who owns the requirement.}}

## Tool and model
{{The tool and model, or "none: constructed example", and the date. If no tool was available, how

## Context given
{{What the assistant was shown, and what was left out and why. No secrets or personal data.}}

## Output received
{{Each candidate in order, marked accepted or rejected. Keep the rejected ones.}}

## Decision
{{What was accepted or rejected, and the reason for each.}}

## Changes made
{{What you changed by hand, and why.}}

## Verification
{{The checks, the commands, the results and the date. Say which checks are independent of the ass

## Limits
{{What the verification does not show, and what was not run.}}
```

Save as `examples/ai-assistance-record.md`:

```
<!-- Example: constructed to illustrate. Not real evidence. -->
# AI assistance record: explain the overlap rule

## Task
Ask for an explanation of `LoanPeriod.overlaps`, so that a teammate can review it. The developer

## Tool and model
None: constructed example. The candidate below was written by the author of this example and stan

## Context given
The source of `LoanPeriod` and one sentence about the half-open convention. The existing checks w

## Output received
1. Candidate explanation: **rejected**. It said two periods overlap when the first one's end is o

## Decision
Rejected. The sentence contradicts the convention that an end day is excluded.

## Changes made
Wrote the explanation by hand from the code: two periods overlap only when each starts before the

## Verification
Traced the periods 20 to 22 October and 22 to 24 October by hand: they do not overlap. Ran the Ap

## Limits
Only the overlap rule was checked, on a few dates. Nothing here shows that the explanation covers
```

Keep the rejected output. Say which checks are independent of the assistant, such as the compiler, a hand trace and examples written beforehand. A second assistant's opinion is not verification. A record that says only "reviewed and verified" is the ritual version; the useful one says what was run, with what result, and what remains unchecked.

Common mistakes. Writing the record after the details are forgotten. Leaving out the context, so that nobody can tell what the assistant saw. Including secrets or personal data. Treating a passing check as proof of more than it covers.

C.14 What is not here

The individual reflection and the guide to contribution evidence belong to Appendix E (E.4 to E.6). The record for the repository-metrics lab and the change specification used in Chapter 13 are in Chapter 12, where their fields are explained. The handover documents (the README and the setup document) are specific to a project and are described in Chapter 14.

C.15 A short verification activity

Do these in a copy of the kit, in a throwaway folder.

1. In a copy of the kit from the companion, run `bash scripts/check-templates.sh`. It should end with a pass for all 26 files.
2. Copy `templates/pull-request.md` to a new file and fill it in for a real change you made. Run `bash scripts/check-templates.sh --file pull-request YOUR-FILE`. Fix whatever it reports.

3. Break your file three ways, one at a time: delete a heading, empty a section, leave a `{{placeholder}}`. Note what the script says each time, then restore it.
4. Choose one template from this appendix that your team does not yet use and say, in two sentences, what question it would force you to answer.
5. Fill in an evidence note for your own team, writing "none yet" where something does not exist.

Optional AI activity, with an identical offline route. The companion lists a complete evidence note that an assistant might produce when asked to "fill in the template". It is author-constructed, not a captured model response. Its key entries read:

> Hosted passing run: run 1432, passed on revision a3f9c21 on 12 October. Independent review: reviewed and approved by the team. Deliberate failure: introduced and fixed, all checks green. Known limitations: none.

Run the script on it. Does it pass? Which entries could a reviewer verify, and how? Which entries look like inventions? A live assistant may produce a different candidate if your course allows one, but it is not required. Record the claim, your check and your conclusion. What you hand in at the end is your filled note; it adds no assessment requirement.

Sources and further reading

Opened on 13 October 2026. The explanations, templates and examples are original; source text has not been reproduced.

1. Schwaber, K. and Sutherland, J., **The Scrum Guide**, November 2020. The Definition of Done as a commitment of the increment; the purpose of the Sprint Retrospective.
2. Nygard, M., **Documenting Architecture Decisions**, 15 November 2011. Short numbered records: title, context, decision, status, consequences; superseded and not deleted.
3. GitHub Docs, **Creating a pull request template for your repository**. The `pull_request_template.md` file.
4. **Keep a Changelog 1.1.0**. Changelogs for humans, the change types and the dated entry for each version.

Appendix D: Project Versions and Glossary

> *Where each chapter's project stands, the counts to expect, the rules of the lending domain as the chapters state them, and a glossary of the book's terms.*

OO Development and Agile Practices
Designing, Testing and Evolving Software with AI Assistance

How to use this appendix

The chapters build one project, the Campus Equipment Lending System, and use it to teach design, testing and change. A reader who joins at Chapter 9, or who is checking a build against the text, needs three things in one place: where the project stands at each chapter, what the build should report, and what the rules of the domain are. This appendix gathers them. It also lists the terms the book uses and says where each is explained.

Everything here is taken from the chapters and their companions. If this appendix and a chapter ever differ, the chapter's companion is the authority, because that is where the code and its checks are listed.

- **D.1** shows how the running project and the isolated examples fit together.
- **D.2** is the table of versions: version, location, count and command for each chapter.
- **D.3** explains what the counts mean and lists the open questions the chapters name.
- **D.4** records the rules of the domain, with the chapter that states each.
- **D.5** suggests how to carry the same questions to another domain.
- **D.6** is the glossary.

D.1 The running project and the isolated examples

The book keeps two kinds of code apart. The **running project** grows from chapter to chapter and is the code a build must keep working. **Isolated examples** teach one idea in a corner of the repository and are never

used by the project.

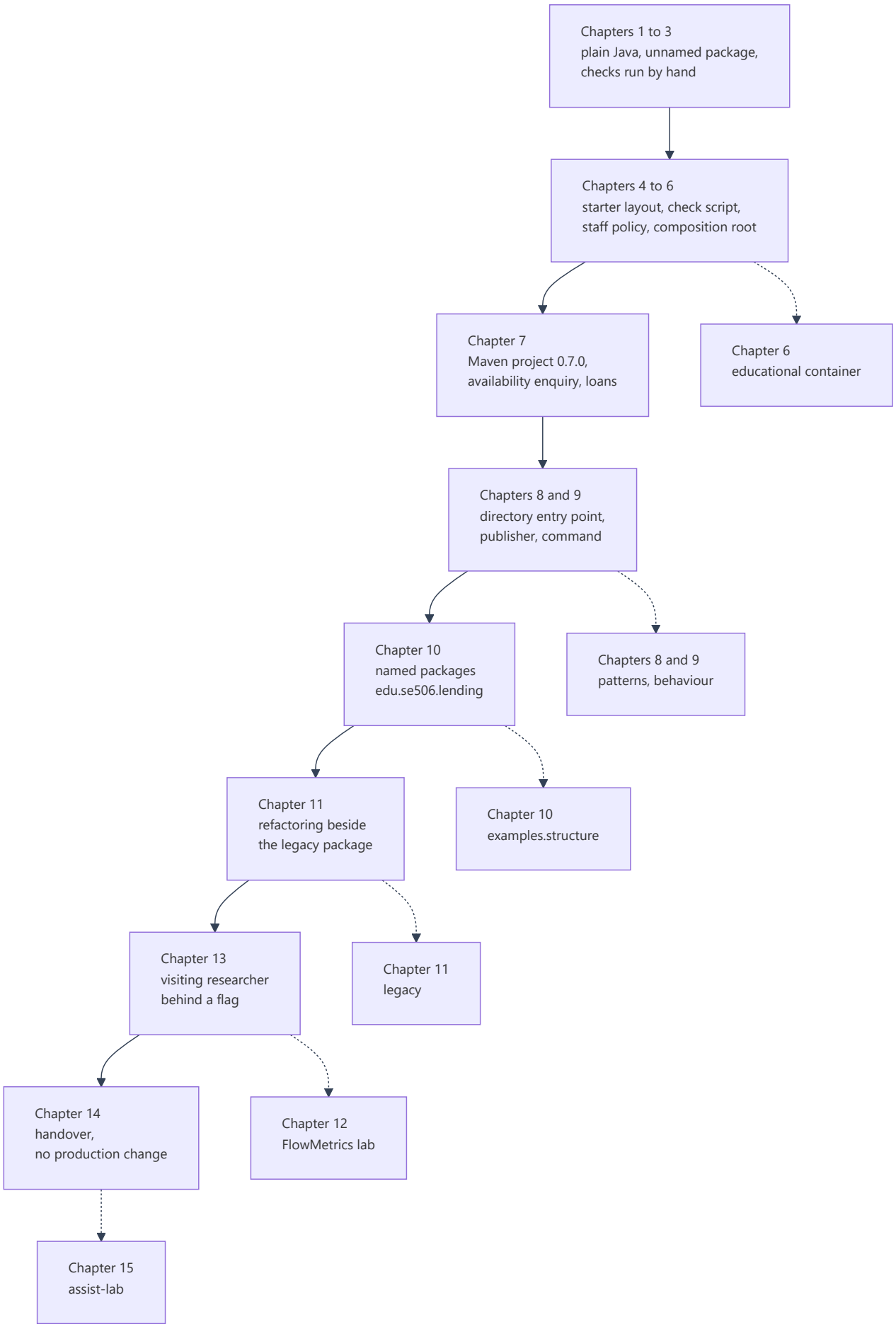

Chapters 1 to 3
plain Java, unnamed package, checks run by hand
Chapters 4 to 6
starter layout, check script, staff policy, composition root
Chapter 7
Maven project 0.7.0, availability enquiry, loans
Chapter 6
educational container
Chapters 8 and 9
directory entry point, publisher, command
Chapter 10
named packages edu.se506.lending
Chapters 8 and 9
patterns, behaviour
Chapter 11
refactoring beside the legacy package
Chapter 10
examples.structure
Chapter 13
visiting researcher behind a flag
Chapter 11
legacy
Chapter 14
handover, no production change
Chapter 12
FlowMetrics lab
Chapter 15
assist-lab

Figure D.1. Solid arrows are the running project. Dashed arrows lead to examples that are kept apart from it.

The isolated examples are:

1. `ContainerWalkthrough` (Chapter 6), a small educational container, outside the normal gate.
2. `patterns` (Chapter 8, extended in Chapter 9) and `behaviour` (Chapter 9), the pattern and lifecycle exercises. The lifecycle exercise does not modify the real application.
3. `examples.structure` (Chapter 10), deliberately flawed packages for the dependency exercises.
4. `legacy` (Chapter 11), the overdue-report routine that the chapter characterises and refactors. Production code under `edu.se506.lending` does not use it, and from Chapter 11 a compiled dependency test checks this. Only the bootstrap code launches it or feeds it rows.
5. `FlowMetrics` (Chapter 12), the calculator for the metrics lab. The application stays where Chapter 11 left it.
6. `assist-lab` (Chapter 15), a separate Maven project with its own copy of `LoanPeriod`. It does not depend on the lending project and does not change it.
7. `risk-lab` (Appendix F), a separate Maven project, package `risk`, with 67 tests (56 without the `integration` tag and 11 with it). It puts a stub model behind one domain question and does not touch the lending project.

Until Chapter 10 the production code is in the unnamed package. From Chapter 10 it is under `edu.se506.lending`, in the packages `domain`, `application`, `application.port`, `infrastructure`, `presentation` and `bootstrap`.

D.2 Versions

The table gives, for each chapter, the project version, where the code lives, the count that the chapter’ s own build reports, what the chapter adds and the command that runs it. Chapters 1 to 6 report **checks**: the lines a plain-Java checking program prints. From Chapter 7 the build is Maven and the count is of JUnit **invocations**

(see D.3). A count from a later chapter includes the earlier ones that it inherits.

Chapter	Project version	Where the code lives	Count that the chapter reports	What the chapter adds	Command
1	none	Plain Java, unnamed package	12 checks	`ReservationRules`: at most three active reservations, one to seven days	`javac ReservationRules.java Chapter01Demo.java Chapter01Checks.java`, then `java Chapter01Checks`
2	none	Plain Java, unnamed package	34 checks	`LoanPeriod`, `Reservation`, `ReservationCoordinator`; extracting the confirmation text	`javac --release 17 -d out ...`, then `java -cp out Chapter02Checks`
3	none	Plain Java, unnamed package	5 acceptanc	Backlog, acceptance criteria, Definition of Done; `Chapter03AcceptanceExamples`	`java -cp out Chapter03AcceptanceExamples`

			e e x a m pl e g r o u p s; n o p r o d u ct io n c h a n g e		
4	none	Plain Java, `src/main/java` and `src/test/java`	3 4 c h e c k s a n d t h e 5 g r o u p s	The starter layout, `scripts/check.sh`, a hosted-check workflow file, `Chapter04Demo`	`bash scripts/check.sh`
5	none	Plain Java, folders `before`, `refactored`, `after`	4 4 C	Staff policy, `ConfirmationFormatter`, the PMD 7.28.0 lab	`bash after/scripts/check.sh`, `bash scripts/pmd.sh`

			hapter5checks,withthe34checksand5groupsstillrun		
6	none	Plain Java, unnamed package	28Chapter6chec	Notification sender, composition root `ReservationApplication`, policy map	`bash scripts/check.sh`

			ks, with the earlier ones still run		
7	0.7.0	Maven project, unnamed package	35 JUnit invocations, four of which wrap the ear	Availability enquiry (US-02), a checkout and return flow, one Mockito interface double	`mvn --batch-mode verify`

			liers runners		
8	0.8.0	Maven; `patterns` isolated	63 invocations: 35 inherited, 17 directory-bound, 11 pattern	A directory-backed entry point that translates synthetic provider records	`mvn --batch-mode verify`
9	0.9.0	Maven; `behaviour` isolated	10	Strategy selection, `NotificationPublisher`,	`mvn --batch-mode verify`

			7 invocations: 63 inherited and 44 new	`ReserveEquipmentCommand`, an isolated lifecycle exercise	
10	0.10.0	`edu.se506.lending.*`; `examples.structure` isolated	121 invocations	Named packages, `ReservationRepository`, a console adapter, compiled dependency tests	`mvn --batch-mode verify`
11	0.11.0	As Chapter 10, plus `legacy` isolated	159 invocations: 121 in he	The legacy overdue report: characterisation, seams, small refactorings	`mvn --batch-mode verify`

			rited, 28 for the legacy report and 10 for the bridge		
12	no new version	No code change; `FlowMetrics` in a separate `lab` folder	None tests; the version sta	Flow metrics, the repository lab, the specified change (CR-12)	Plain JDK, as the companion lists

			ysat159		
13	0.13.1	As Chapter 11, plus the visitor change	232invocations:203unit,26integration,3end-to-end	The visiting-researcher rules behind a flag, an approval-file adapter, an HTTP directory client, the pipeline	`mvn --batch-mode verify`, `bash scripts/pipeline.sh`
14	0.14.0	As Chapter 13; no production change	235invocation	`TraceabilityTest`, the handover documents, the handover check, a rehearsed flag removal	`mvn --batch-mode verify`, `bash scripts/handover-check.sh`

			s: 206 unit, 26 integration, 3 end-to-end		
15	1.0.0 (`assist-lab`)	Separate project, package `assist`	9 acceptance tests	The AI-assistance lab: two constructed attempts, a record and a record check	`mvn --batch-mode test` in the lab folder

Chapters 11, 13 and 14 list only the files that are new or changed. To reconstruct a version, start from the Chapter 10 companion and apply each later companion in order, as each says.

D.3 What the counts mean, and the questions left open

Invocations are not methods

From Chapter 7 the build runs Maven’ s Surefire plugin with JUnit Jupiter. Surefire reports **invocations**. A plain test method runs once and counts once. A parameterised test with three values runs three times and counts three. A class with ten methods can therefore report more than ten. Report the number the build prints, and say what it counts. Do not compare it with a count from a different project.

The count grows as each chapter inherits the one before:

Step	Arithmetic the chapters give
Chapter 8	35 inherited + 17 directory-boundary + 11 pattern = 63
Chapter 9	63 inherited + 9 Strategy + 21 lifecycle + 10 publisher + 4 Command = 107
Chapter 10	107 migrated + 7 repository + 2 value-semantics + 3 console + 2 dependency = 121
Chapter 11	121 inherited + 28 for the legacy report (27 characterisation and one differential) + 10 for the bridge = 159
Chapter 13	159 inherited + 73 new = 232
Chapter 14	232 + the three `TraceabilityTest` tests = 235

Chapter 12 changes no code. If a text quotes a different count for the Chapter 11 suite, the Chapter 11 companion is the authority.

The stages of the pipeline (Chapter 13)

The pipeline runs five stages in order and stops at the first failure. Tests are selected by JUnit tags, and **only two tags exist**: `integration` and `e2e`. Unit tests carry no tag; the unit stage is “everything not tagged”.

Stage	Command	Chapter 13 count	Chapter 14 count
1 compile	`mvn compile`		
2 unit tests and architecture checks	`mvn test -DexcludedGroups=integration,e2e`	203	206
3 integration	`mvn test -Dgroups=integration`	26	26
4 end-to-end	`mvn test -Dgroups=e2e`	3	3
5 package	`mvn package -DskipTests`		

A stage that selects no tests fails, so a mistyped tag cannot give a green stage that checked nothing.

Questions the chapters leave open

These are written down in the chapters and not settled by the code. They belong to the person who owns the requirement.

Reference	What is open	Chapter
SD-11-1	Any other borrower kind pays 1.00 a day with no grace and no cap. Do other kinds exist, and should they be rejected? A late visitor now reaches this rule	11, 13
SD-11-2	A loan within the grace period is listed with fine 0.00 and counted as overdue. Should it appear at all?	11
SD-11-3	A null kind throws `NullPointerException`, only for a loan that is overdue and unreturned. Can the export contain one?	11
SD-11-4	Amounts use the default locale (`0,50` under `Locale.GERMANY`). Should the separator follow the machine or the office?	11
SD-12	The coordinator counts every retained reservation, although a variable is called `activeCount`. Should a future rule count active loans, future reservations or all retained reservations?	12, 13

D.4 The rules of the domain

Time and periods

A period includes its start day and excludes its end day, written `[start, end)`. Dates are `java.time.LocalDate` values. Two periods overlap exactly when each starts before the other ends:

```
A.start < B.endExclusive  and  B.start < A.endExclusive
```

Periods that meet at a boundary do not overlap. A reservation ending on 22 October and one starting on 22 October can both hold the same item. Where an instant matters (the approval window of Chapter 13) the code reads one `Clock` and decides on that reading, so tests can fix the time.

Identifiers and equality

Identifiers in the examples are `B1` for a borrower, `CAM1` for an item and `R1` for a reservation. The first versions check only that an identifier is not blank; they do not look it up in a directory or catalogue. The coordinator allocates reservation identifiers in sequence, and a rejected request does not use one.

`LoanPeriod` and `Reservation` do not override `equals` or `hashCode`. Two separate objects with the same data are not equal through Java's inherited equality. The repository (Chapter 10) uses the reservation's identifier as its key and rejects a duplicate. Domain identity and Java object identity are different ideas (Chapter 2, Section 2.1, and Chapter 10, Section 10.6).

Borrower categories and limits

The count is the number of reservations the borrower already holds before the new one. All stored reservations count; the code does not remove a reservation when a loan is returned.

Category	Existing reservations allowed before a successful request	Duration	Other conditions
Student	0, 1 or 2	1 to 7 days	None
Staff	0 to 4	1 to 14 days	The staff limits are an illustrative teaching requirement (Chapter 5)
Visitor	Fewer than three	1 to 3 days	Only CAM1 or CAM2; a matching sponsor approval; the flag must be on (Chapter 13)

Both the student and staff policies reject a negative count or a non-positive duration. Chapter 1’ s first rule returns `false` for a negative count.

A **visitor** needs one approval record, read from a tab-separated register, that matches the exact borrower, item and period, carries a non-blank sponsor identifier and the status APPROVED, and is valid at the request instant. Approval for another borrower, item or period does not count. The gate and the coordinator decide in this order:

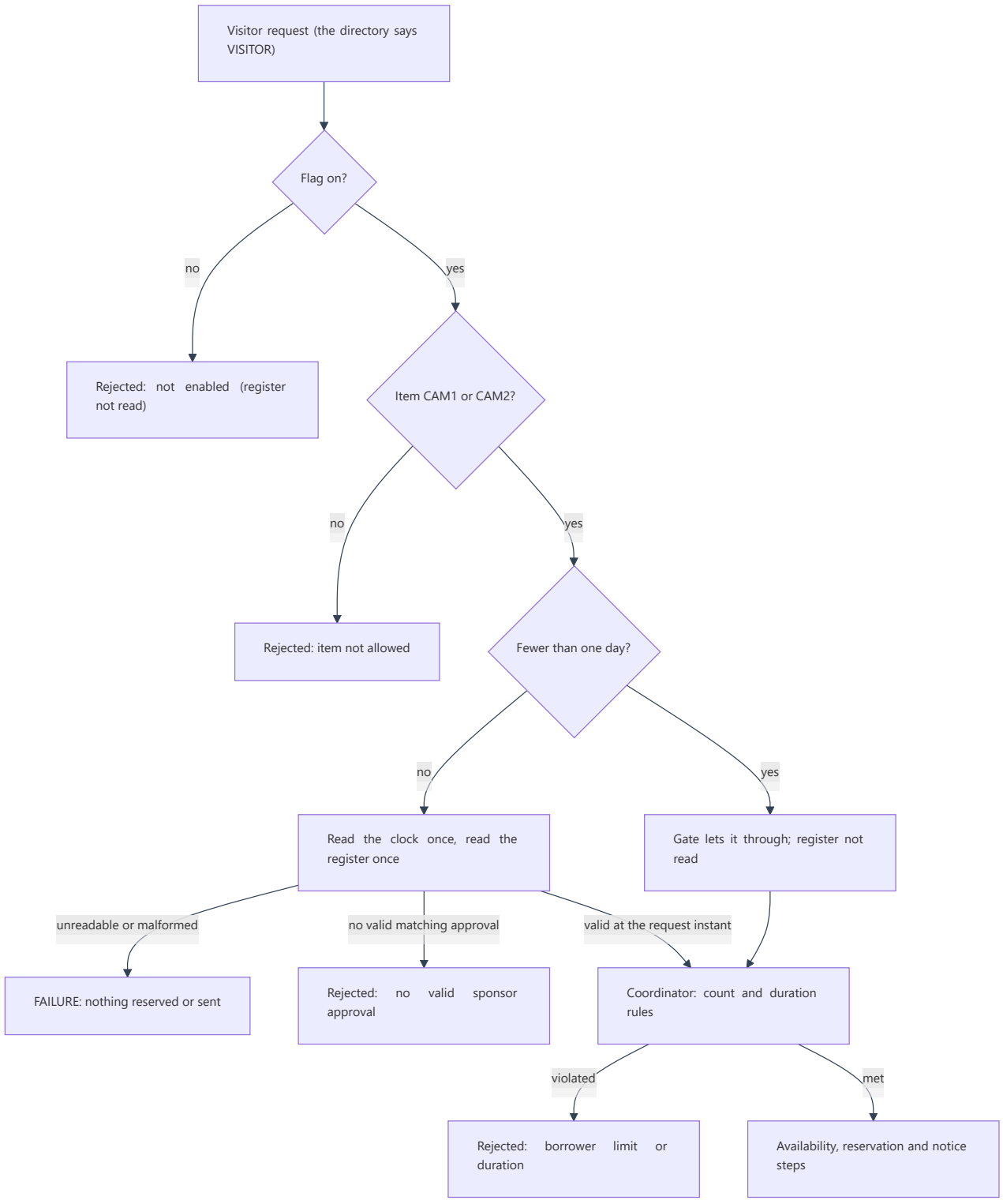


Figure D.2. The visitor decision, as Chapter 13’ s code makes it. Rectangles are results. Every rejection is an ordinary return value; only an unreadable register is a failure.

The directory

Borrower categories come from the directory in Chapters 8 and later. The adapter translates the provider’s records exactly; it does not trim, normalise or guess.

Provider outcome	Application outcome
Kind `S`, status `Y`, matching identifier	A student borrower
Kind `T`, status `Y`, matching identifier	A staff borrower
Kind `V` (added by the visitor change)	A visitor
No row	An empty result: the borrower is absent
A recognised kind with status `N`	An empty result: the borrower is inactive
An unknown or null kind or status, or a mismatched identifier	A `DirectoryException`: invalid provider data
The provider’s declared failure	A `DirectoryException` that keeps the cause
An unexpected programming exception	It propagates and is never turned into absence

An outage is never “not found”, and an unknown code never defaults to a category, least of all to student or staff. From Chapter 13 the HTTP client makes one attempt, bounds the whole wait, and neither retries nor falls back.

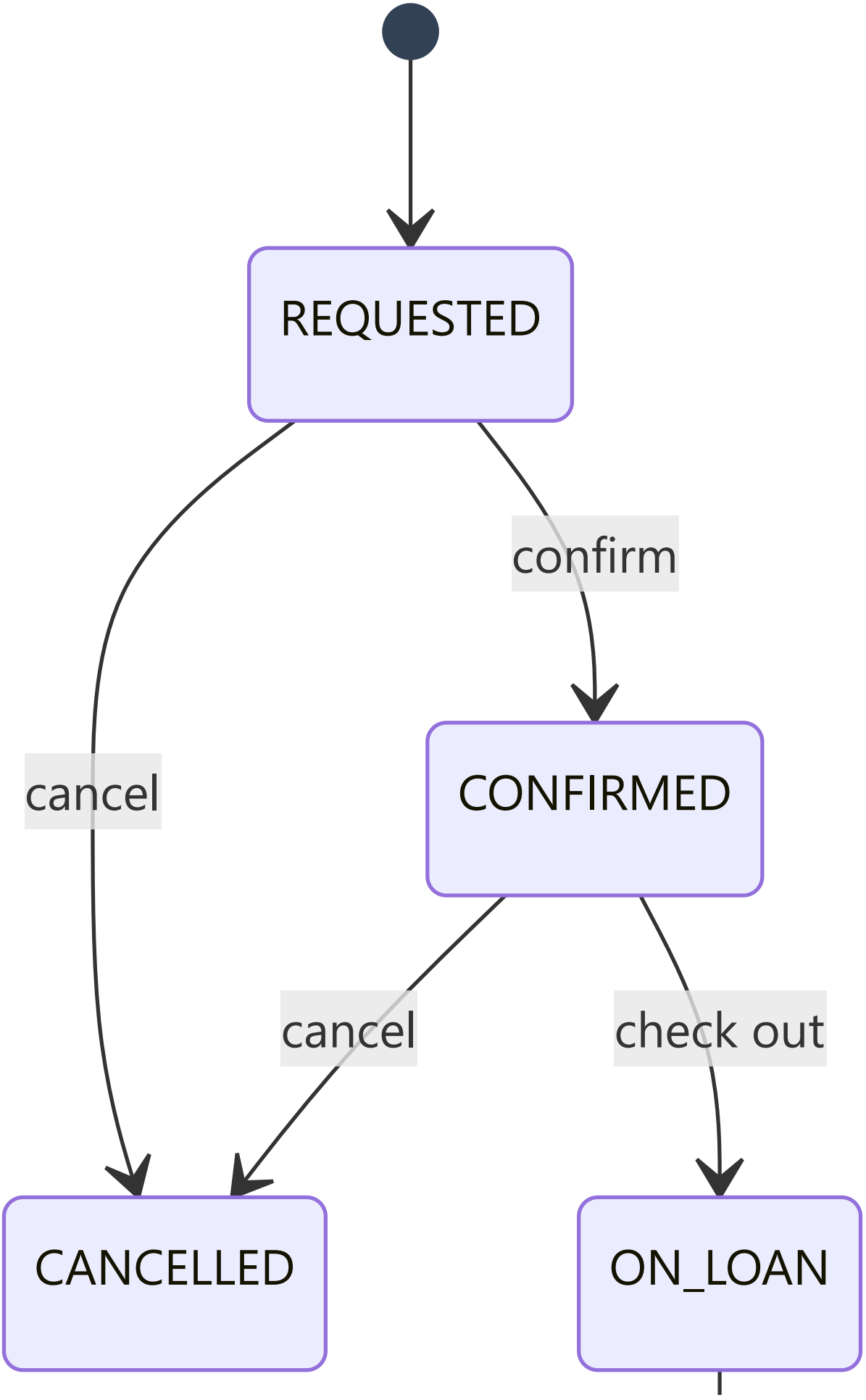
Failure and what stays stored

When a reservation is accepted, the coordinator stores it **before** it notifies. If notification fails with the expected exception, the reservation stays and the result carries a failure suffix (Chapter 9). If the approval register is unreadable, nothing is reserved and nothing is sent (Chapter 13).

The reservation lifecycle (an isolated exercise)

Chapter 9′ s lifecycle exercise models a reservation as REQUESTED, CONFIRMED, ON_LOAN, RETURNED or CANCELLED. The running system does not use it: it confirms accepted reservations immediately, tracks loans

separately and has no cancellation operation.


REQUESTED
confirm
cancel
CONFIRMED
cancel
check out
CANCELLED
ON_LOAN

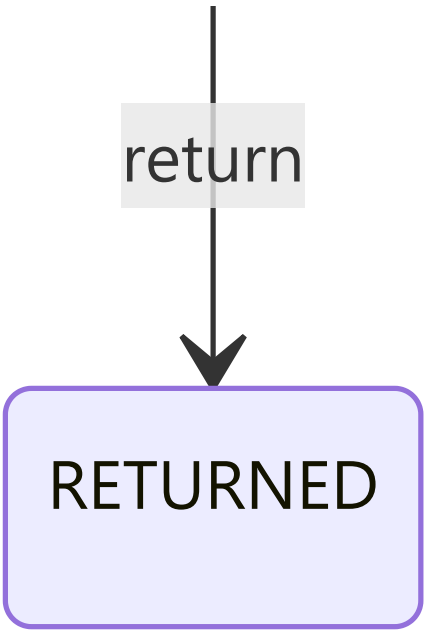


Figure D.3. The proposed lifecycle. RETURNED and CANCELLED allow no further action. Every other combination is a rejection that throws `IllegalStateException` and leaves the status unchanged.

Packages and the direction of dependencies

From Chapter 10 the packages are grouped as follows. An arrow means that classes in the first group may refer to classes in the second. A missing arrow means the reference is not permitted: `Domain` refers to nothing else in the application. A compiled test checks the direction and the absence of package cycles.

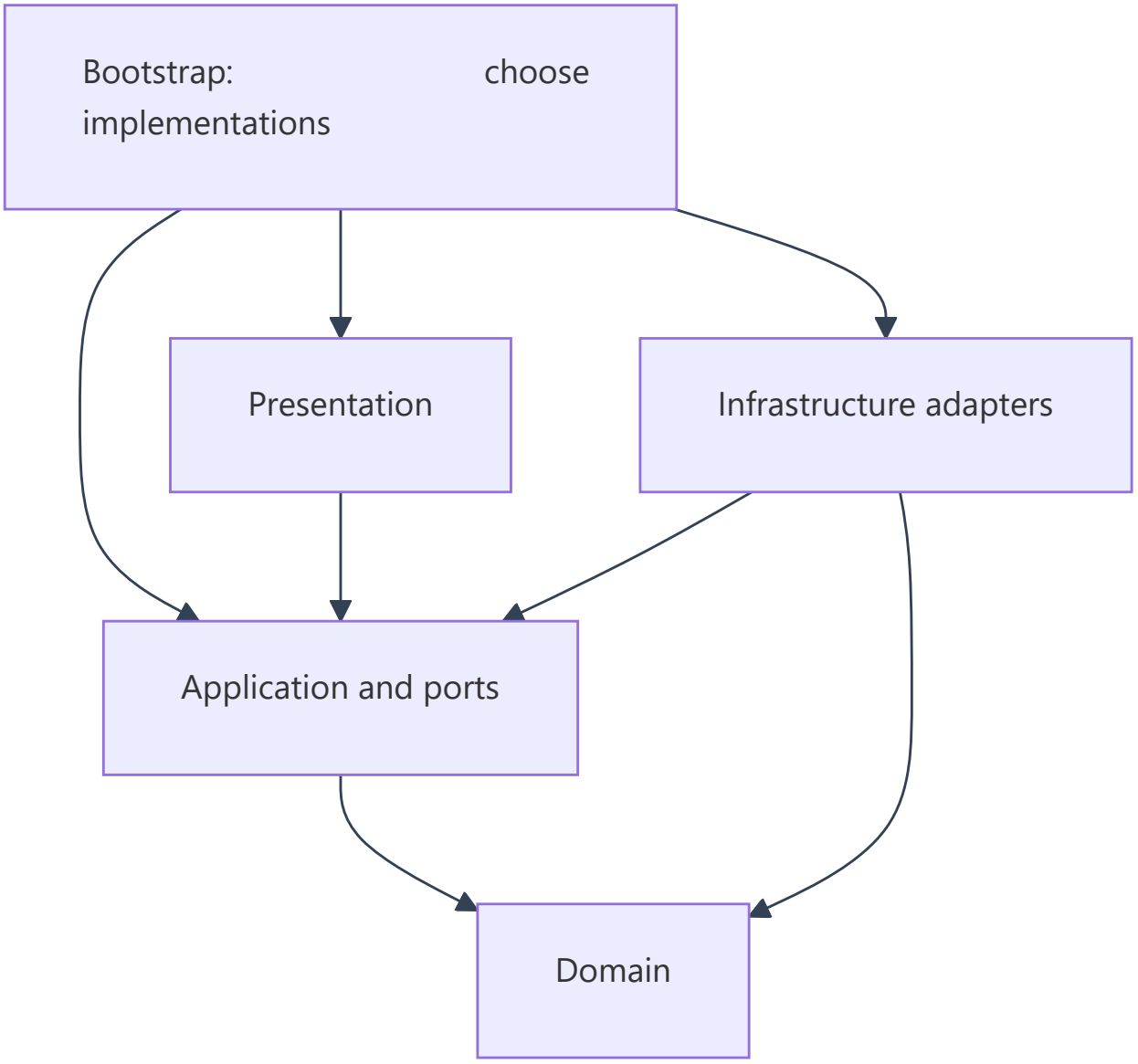


Figure D.4. Permitted source dependencies. The figure shows references in the source, not the order of calls.

Snapshots

A method that returns a collection returns an unmodifiable snapshot, `List.copyOf(...)`, so that a caller cannot change the coordinator's state through it. A snapshot does not freeze the objects inside it.

D.5 Carrying the questions to another domain

The book builds on equipment lending and supplies no starter version for another domain. The descriptor names an ML model-serving tool as another possible starter domain. If your team works on one, the same questions apply. The table is a suggestion for reading the book, not a provided version.

In the lending project	A possible counterpart in a model-serving service	The question to ask
Borrower and category	A client and its tier	Who is asking, and who decides the tier? Never default an unknown client to a higher tier
Item such as `CAM1`	A model deployment or a worker	What is the unit that is reserved?
`LoanPeriod`	A reserved capacity window	Is the window half-open, and when do two windows overlap?
`ReservationRules`	A quota policy	What does the rule count, and does the count include everything retained?
The directory and its outage rule	A model registry or an identity service	What happens on an outage? Say "unavailable", not "not found"
`VisitorGate`	A flag for an external client	Is the flag off by default, and are both settings tested?
Notification after storing	An alert after accepting a job	If the alert fails, does the accepted job stay?

D.6 Glossary

The entries follow the chapters. The reference after each says where the term is explained.

Acceptance criteria. The behaviour expected of one backlog item, specific enough that a second team can tell whether an implementation meets it (Chapter 3, Section 3.7). An acceptance example is one concrete case of a criterion.

ADR (architecture decision record). A short record in the repository of a decision, its context, the alternatives, the consequences, how it was checked and what would reopen it (Chapter 10, Section 10.8; Chapter 14, Section 14.3; Appendix C).

Afferent coupling (Ca). For a package, the number of classes outside it that depend on classes inside it (Chapter 10, Section 10.5).

Arrange–Act–Assert. A test structure: establish the state, perform the operation being examined and inspect the outcome (Chapter 7, Section 7.2).

Association, aggregation, composition, dependency. UML relationships between classes: a structural link; a grouping claim; strong ownership with a lifecycle; and a use that is not held in a field (Chapter 2, Section 2.6; Appendix A, A.8).

Boundary. The place where our code meets something it does not control, such as a file, a network service or a setting (Chapter 13, Section 13.1).

Characterisation test. A test that records what code does today, whatever that is, so that a change which alters it is noticed (Chapter 11, Section 11.3).

Cohesion and coupling. How closely the responsibilities inside a component belong together, and how much components depend on each other (Chapter 2, Section 2.4).

Composition root. The small application-entry area where concrete collaborators are assembled (Chapter 6, Section 6.5).

Continuous delivery and continuous deployment. Building software so that it can be released at any time; and going further, releasing every change that passes the pipeline automatically (Chapter 13, Section 13.5).

Contract test. A test that states what callers may rely on and checks that an implementation honours it (Chapter 13, Section 13.2).

Cycle time and lead time. In Chapter 12' s history, lead time runs from Accepted to Done and cycle time from Started to Done. The difference is the recorded wait before starting. Neither measures hands-on effort (Chapter 12, Section 12.5).

Definition of Done. The shared quality standard that an increment must meet, whatever the story (Chapter 3, Section 3.7; Appendix C).

Dependency injection. Supplying collaborators to an object from outside it (Chapter 6, Sections 6.5 and 6.6).

Dependency Inversion Principle (DIP). High-level policy should not depend on low-level detail; both depend on an abstraction that the policy owns (Chapter 6, Section 6.4).

Efferent coupling (Ce). For a package, the number of classes inside it that depend on classes outside it (Chapter 10, Section 10.5).

Encapsulation and information hiding. Grouping state with the operations that govern it and controlling access to it; and deciding which implementation choices callers need not know (Chapter 2, Section 2.3).

End-to-end journey. A test that starts the whole program the way a user would, in a separate process (Chapter 13, Sections 13.1 and 13.3).

Fault experiment. Making one deliberate change at a time, in a scratch copy, to see whether the tests notice it. A change that cannot be noticed because it behaves the same is called equivalent (the companions of Chapters 5 to 15).

Feature flag. A switch that lets code ship without turning its behaviour on. The book' s flag is off by default and both settings are tested (Chapter 13, Section 13.6).

Gate. A rule that stops the pipeline (Chapter 13, Section 13.4).

Half-open interval. A period that includes its start and excludes its end (D.4).

Instability. For a package, I = Ce / (Ca + Ce), from 0 (maximally stable) to 1 (maximally unstable) (Chapter 10, Section 10.5).

Integration test. A test that runs an adapter against the real mechanism it adapts, such as a real file or a real HTTP client talking to a service under our control (Chapter 13, Sections 13.1 and 13.2).

Interface Segregation Principle (ISP). A client should not be forced to depend on operations it does not use (Chapter 6, Section 6.2).

Invariant. A condition that must hold for valid objects or valid system states (Chapter 2, Section 2.2).

Liskov Substitution Principle (LSP). An object of a subtype must be usable wherever its supertype is promised, without breaking the promise (Chapter 5, Section 5.5).

Open–Closed Principle (OCP). Where variation is anticipated, add behaviour by adding code and not by editing code that already works (Chapter 5, Section 5.3).

Precondition and postcondition. What must hold before an operation for its promise to apply, and what the operation guarantees afterwards (Chapter 5, Section 5.4).

Refactoring. Improving the internal organisation of code while preserving its observable behaviour (Chapter 1, Section 1.2; Chapter 11, Section 11.5).

Regression and regression test. An unintended loss or change of behaviour that used to work; and a test that protects behaviour everyone agrees is correct (Chapter 1, Section 1.4; Chapter 11, Section 11.3).

Seam and enabling point. A place where you can change what the program does without editing the code at that place, and the place where you choose the other behaviour (Chapter 11, Section 11.4).

Single Responsibility Principle (SRP). A unit should have one reason to change (Chapter 5, Section 5.2).

Suspected defect. Behaviour that looks wrong but is not confirmed. The book keeps it apart from a known defect, accepted behaviour and an open question (Chapter 11, Section 11.7).

Technical debt. A metaphor, attributed to Ward Cunningham, for accumulated cruft that makes a system harder to change. The book treats it as evidence to point at and not as a complaint (Chapter 11, Section 11.7).

Test double. A substitute for a collaborator in a test. The roles are dummy, stub, fake, spy and mock; the tool does not decide the role (Chapter 7, Section 7.5).

Test-driven development (red–green–refactor). Writing a failing test, making it pass with the simplest change, then improving the structure while the tests stay green (Chapter 7, Section 7.3).

Throughput, velocity and work in progress. Completed items over a stated period; the sum of estimates of the items completed in an iteration; and the items started but not finished at a stated moment (Chapter 12, Sections 12.3 and 12.5).

Traceability. The chain from a requirement to the code that decides it and the test that checks it (Chapter 14, Section 14.4).

Unit test. A test that runs a small part of the system in memory with everything it collaborates with under control (Chapter 13, Section 13.1).

User story, epic and vertical slice. A small user-centred need; a label for a large area of work that needs dividing; and a slice that passes through the necessary parts of the system to deliver one observable outcome (Chapter 3, Section 3.6; Chapter 4, Section 4.3).

Verification record. The record of what was asked of an AI assistant, what was received, what was accepted or rejected and what checks were run (Chapter 15, Section 15.8; Appendix C).

D.7 Sources

The definitions and rules above come from the chapters and their companions. Each chapter's own source list gives the references behind them. Three outside sources are used directly, all opened on 13 October 2026:

1. Schwaber, K. and Sutherland, J., **The Scrum Guide**, November 2020. The Definition of Done and the Increment (D.6).
2. Fowler, M., **Test Double**, 2006. A summary of Gerard Meszaros' s vocabulary of test doubles, as used in Chapter 7, Section 7.5.
3. Fowler, M., **Technical Debt**. The metaphor attributed to Ward Cunningham, as used in Chapter 11, Section 11.7.

Appendix E: Critical Reading and Project Evidence

> *A method for the individual critical book review, and guides to contribution evidence, the final handover and the individual reflection. It follows the module descriptor and adds no rubric or policy.*

OO Development and Agile Practices
Designing, Testing and Evolving Software with AI Assistance

How to use this appendix

Two pieces of work in this module ask you to show your own thinking and your own evidence. The **individual critical review** of a software-design book asks you to read carefully and judge what you read. The **final project** asks your team to hand over a system someone else can run, and asks each of you to explain, with evidence, what you contributed. This appendix gives a method for each. Sections E.1 to E.3 are about the review; E.4 to E.6 are about the project.

> **This appendix is guidance, not a rule.** The module descriptor and your module leader's assignment brief decide the deadlines, lengths, formats, approved reading list, rubrics and the rules on the use of AI tools. Where they differ from this appendix, they are right. Nothing here adds a requirement or changes how any work is marked.

Start the reading method in Week 1. Chapter 1 introduces the reading record and Chapter 9 gives a reading point to compare with your progress; this appendix develops both.

E.1 What the review asks

The descriptor describes the first assessment this way. Each student selects a software design or agile engineering book from a list approved by the module leader and writes an individual critical review of 1,500 to 2,000 words. The review summarises the book's purpose and audience, critically evaluates its principles and examples, discusses the trade-offs and limits of its advice and reflects on how it applies to engineering AI-enabled systems, supported by concrete code or project examples and at least three additional sources. It is due in Week 8 and is submitted as a PDF through Moodle, which checks it for similarity. The descriptor's rubric lists the criteria and their weights; read it before you start, and again before you submit.

The book comes from the approved list. The list is the module leader's. The descriptor gives examples of the kind of book that may be on it (Clean Code, Refactoring, Design Patterns, Working Effectively with Legacy Code and Accelerate). This appendix does not supply or change the list. Check the current one on Moodle and choose in the first two weeks, because a careful review cannot be written from a late start.

This textbook is not the book you review. Reading it does not satisfy the assignment. It can help in one way: the Campus Equipment Lending project gives you concrete code to test a book's advice against (see E.2).

E.2 A method for reading

Read for claims, then test them

Technical books recommend practices that suit particular circumstances. Treat each recommendation as a claim to understand and then evaluate. Take the strongest reasonable reading of the advice and not an exaggerated one. Keep a reading record as you go, with the six entries of Chapter 1, Section 1.7:

Entry	Question
Claim and location	What exactly does the author recommend, and where? (chapter and page, or a stable section)
Context	What problem and assumptions surround the advice?
Supporting example	How does the example demonstrate the claimed benefit?
Limitation	Under what conditions might the cost outweigh the benefit?
Project application	What would change if we applied it to our system?
Further evidence	Which independently checked source or experiment helps evaluate it?

Add one habit from Chapter 9: keep three things apart in your notes, and label each. **What the author claims. What you infer. What a tool suggested**, if you used one. A review that mixes them cannot be checked, and your reader cannot tell whose idea an argument is.

Ask a question for each part of the task

The prompts below follow the parts of the descriptor's task. They are questions to think with and are not a template for headings.

Part of the task	Questions to ask
Purpose and audience	What problem was the author answering, and for whom? What did the author assume the reader already knew? When was it written, and what has changed?
Principles and examples	Does the argument follow from its premises? Are the examples realistic, or small enough to hide the difficult cases (persistence, concurrency, failure)? Does the author claim a rule applies everywhere when it depends on context?
Trade-offs and limits	What does the advice cost: more files, more indirection, slower change in some other direction? When does it fail, and does the author say so? What simpler alternative did the author consider?
AI-enabled systems	Which assumptions break when part of the system is a model or a data pipeline: repeatable outputs, requirements that are exact, code that changes only when someone edits it? Which of the book's techniques still help (boundaries, test doubles, seams, flags), and what do they leave out? Appendix F works one example of software around a model
Concrete examples	Which piece of code or project decision shows the advice working or failing?
Additional sources	What independent source, a study, a competing view or a standard, supports or challenges the claim?

Test the advice on code you can run

The lending project is a small, checked system. Pick one claim from your book and apply it. For a claim about interfaces, look at `NotificationSender` (Chapter 6). For a claim about small classes and responsibilities, look at the coordinator before and after Chapter 5. For a claim about patterns, compare a Strategy with the plain conditional it replaces (Chapter 9). For a claim about legacy code, use the characterisation tests of Chapter 11. For a claim about delivery, use the pipeline of Chapter 13. Then write what you saw: where the advice simplified the code, and where it added weight. A concrete case, honestly examined, is worth more than a catalogue of definitions.

Check every source you cite

For each additional source, answer these before you rely on it:

1. Who wrote it, and what is their basis for the claim: a study, an experience report, an opinion?
2. Have you opened it yourself? Do not cite a source you have only seen quoted. If you are citing a summary, say so.
3. Does the source support the exact claim you make, and not a nearby one?
4. How old is it, and has the field moved?

5. Is it independent of the book, or does it repeat the author?
6. For a web page, have you recorded the address and the date you opened it?

Use short quotations only, in quotation marks, with the location. Put the rest in your own words.

A suggested plan

The descriptor sets the deadline and nothing else. This plan is a suggestion for a part-time reader:

Weeks	Do
1 and 2	Choose a book from the approved list. Read its introduction and decide what its central claim is. Start the reading record
3 to 5	Read and record. Choose one claim to test on the lending project
5 and 6	Run the test on the code; write down what you saw. Find and open your additional sources
7	Draft. Check every claim against your record. Check the word count and the references
8	Read the rubric once more, correct, and submit

AI tools and the review

An AI summary of a book you have not read is not evidence about that book, and a critique built from one will repeat what the summary left out. Whether and how you may use an AI tool for this assignment is set by your module' s rules, which apply here. If you are allowed to use one and do, Chapter 15 shows how to record what it was asked, what it returned and what you checked; the record format of Appendix C is one way to do that.

E.3 A reading record, filled in

The entries below are **constructed to illustrate**. The claim is invented for the exercise and is not a quotation from any book. The book, chapter and page are shown as fields for you to fill in with your own.

Book: *your chosen book* **Chapter and page:** *your chapter and page* **Date:** *date you read it*

Entry	Record
Claim and location	*(Constructed claim.)* "Every collaborator of a class should be reached through an interface." Location: *[chapter and page in your book]*
Context	The author is arguing about code that changes often and is replaced in tests. The assumption is that more than one implementation will exist
Supporting example	The author replaces a concrete sender with an interface so a test can supply a recorder, and the test becomes simple
Limitation	For a class with exactly one implementation and no test that needs a substitute, the interface adds a file and a name and removes nothing. The author does not say when to stop
Project application	`BorrowingPolicy` has a student and a staff implementation and `NotificationSender` has test substitutes, so the interface earns its place in both (Chapters 5 and 6). `LoanPeriod` has one, and an interface for it would add nothing
Further evidence	*To find:* an independent source on the cost of premature abstraction; the description of the over-abstraction cost in Chapter 6, Section 6.7, is a starting point and is not independent evidence
Who said what	Claim: the author. Application to the lending project: my inference. Search terms for further evidence: suggested by a tool, then checked by me

E.4 Evidence of individual contribution

The descriptor says that individual contribution on the project is assessed through commits, pull-request reviews and peer evaluation, and that the marks for the final project are moderated by individual contribution. Each of those is an input. None of them is the answer, and this book never ranks people by counts. Chapter 14, Section 14.6 explains why; this section shows how to gather evidence that does the job.

What each kind of evidence can and cannot show

Evidence	What it can show	What it cannot show
Commits	That a change was made, by which author identity, when, and what it touched	Effort, difficulty or importance. Size is a habit. Pair work is recorded under one author unless the other is named in a `Co-authored-by` line [2]. Formatting and regenerated files inflate counts. A squash merge turns many commits into one
Pull-request reviews	That someone read a change and what they said. A review can be a comment, an approval or a request for changes [3]	Whether the review was careful. An approval says the changes look ready to merge, and nothing about how thoroughly they were checked
Peer evaluation	How teammates experienced each other's work, including what no tool records	Objectivity. It is affected by friendship, conflict and what people happened to see

A contribution count can also be wrong for a plain reason. A commit appears in a person's contribution graph only when its author address is linked to their account, and only when it is on the default branch or reaches it through a merged pull request. Commits made in a fork do not count until they are merged [1]. A person whose commits carry the wrong address can appear to have done nothing.

Counts describe habits

Chapter 14 gives a constructed team of three. Its last iteration finished with these figures:

	Commits	Of which formatting only	Lines changed	Reviews given	Reviews that asked for changes
P1	38	17	410	4	1
P2	9	0	1,150	11	5
P3	24	3	620	7	0

Ranked by commits the order is P1, P3, P2. By lines changed it is P2, P3, P1; by reviews given, P2, P3, P1; by reviews that asked for changes, P2, P1, P3. Four rankings give three different orders. The numbers describe habits and the artefacts describe contributions. The honest use of a count is as a pointer to an artefact to read: "Why did P2 have so few commits?" is a question to ask, and the answer is in the diff and the review.

Build an evidence index

Write your contribution as claims, each with evidence that a reader can open. A short index is more useful than a long list:

Claim	Where to see it	What it shows	What it does not show	Who can confirm
(what you did, in one sentence)	*(a pull request, a file, a test, a review comment, a decision record)*	*(the part of the claim it supports)*	*(its limit)*	*(a teammate or the reviewer)*

A constructed row, for the team above:

Claim	Where to see it	What it shows	What it does not show	Who can confirm
P2 wrote the approval-file adapter and found two defects in review	The adapter's pull request and its review thread	The code, the nine commits behind it and the two comments that led to fixes	How much of the design came from discussion before the first commit	P1 and P3, who reviewed it

Habits that make the index easy to write: name your co-author in the commit when you pair [2]; keep review comments and the replies to them; record decisions in decision records (Appendix C); note the work that leaves no commit, such as a design session or a debugging session, and who can confirm it. Peer evaluation is collected as your module sets it. When you give one, say what you saw and where, and keep to the work.

E.5 The final handover and submission

The descriptor says that teams submit and present the complete system built across the sprints, together with a short technical report and an individual reflection. Each team gives a demonstration and a presentation followed by questions. The submission is made through Moodle as a repository link, the technical report and the individual reflection. The demonstrations and the final submission fall in Weeks 15 and 16. Week 13 is the final sprint, the retrospective, polish and the handover, and Week 14 has no classes. Follow the dates your module leader publishes.

What the submission must show

The descriptor says the submission must show clean object-oriented design, justified use of design patterns and architecture, automated tests with continuous integration, evidence of safe refactoring, and adaptability to the injected change in requirements. Use those five phrases as the headings of an **evidence index** for your own team. For each, name the evidence a reader can open and say what it does not show. Chapter 14, Section 14.7 gives the index for the book's project as a model.

The submission must show	Evidence to point at	What it may not show
Clean object-oriented design	Your responsibilities, dependencies and architecture checks	That it is the best design, or that anyone else reviewed it
Justified patterns and architecture	Your decision records, with alternatives considered	A pattern that you did not use
Automated tests with CI	Your test suite and pipeline; a run you can link to	A hosted run that has never happened; a workflow file is not a run
Evidence of safe refactoring	Characterisation tests, small steps, a record of what you ran	Behaviour on inputs your tests did not produce
Adaptability to the change	The footprint of the change, the tests that pin it and the requirements table	That the rule you implemented is the right policy

The handover bundle

A handover is the material that lets someone else set the project up, trust what it says and understand who did what. Chapter 14, Section 14.3 describes six documents, each answering a different question:

Document	The question it answers	A template
`README.md`	What is this, and where do I start?	
`docs/HANDOVER.md`	How do I set it up, run it and understand it in an hour?	
`docs/REQUIREMENTS-TO-EVIDENCE.md`	Which test checks each requirement?	C.10
`docs/KNOWN-ISSUES.md`	What is wrong or missing, and what was decided?	C.10
`docs/ADR-0NN-*.md`	Why is it built this way?	C.5
`docs/RELEASE-NOTES-*.md`	What changed in this version?	C.11

Keep each document true by a check wherever you can: the book's project has a test that fails when a name in the requirements table disappears, and a script that follows the handover in a fresh copy and compares what it finds with what the document promises (Chapter 14, Sections 14.4 and 14.5). A script can check that a name exists; only a reviewer can check that it is the right evidence. Then ask someone outside your team to follow the handover while you watch and say nothing, and write down every place they hesitate.

A sequence

1. **Week 13.** Finish. Run the retrospective (C.12) and decide what the handover must say.
2. **Freeze.** Agree a time after which only fixes to the handover and blockers are allowed. Run your handover check on the frozen version.
3. **Week 14.** There are no classes. Review the whole bundle as a stranger would, fix what the dry run finds, and ask someone outside the team to follow it.
4. **Before Weeks 15 and 16.** Prepare the demonstration and the report from the evidence index, and write your reflection from your own work.

The descriptor does not prescribe the format or length of the technical report. Follow the assignment brief. The evidence index above is a sound spine for it.

E.6 The individual reflection

The descriptor lists an individual reflection among the items submitted with the final project and does not give its length or format. Your assignment brief does. What follows are prompts for thinking, taken from Chapter 14, Section 14.6, and are not a rubric.

Part	Prompts
What I did	Three to five specific pieces of work, each with a link or path: a pull request, a file, a test, a review comment
What I decided	A design or process decision, the alternatives and why
What went wrong and what I did about it	A real fault or misjudgement, how it was found, the fix and what changed afterwards
How I used AI	Where, with what input, what you accepted or rejected and how you verified it
What I did not do	Honest limits: work you started and did not finish, work you relied on others for
What I would change	One thing about your own practice

A reflection makes claims and gives the evidence for each. It does not say "I did a lot." A constructed excerpt, for the team of Section E.4, shows the habit. It is invented and it is not a model to copy:

> *(Constructed.)* I wrote the approval-file adapter. The pull request and its review thread are the evidence: nine commits, and two review comments from a teammate that changed my code. The first was that a row with an unknown status must be an operational failure and not a rejection; I had treated it as a rejection, which would have let a damaged register look like a sponsor's "no". I changed the adapter and added a test for the damaged row. I did not write the pipeline script; P1 did, and I relied on it without reading it closely. Next time I will read the scripts that gate my own work.

Notice what is in it: a named piece of work, a specific fault and how it was found, what changed, and an honest limit. Notice also what is not in it: any count.

E.7 Sources

1. GitHub Docs, **Why are my contributions not showing up on my profile?**. The author email, the default branch and the fork conditions for a commit to count.
2. GitHub Docs, **Creating a commit with multiple authors**. The `Co-authored-by` trailer.
3. GitHub Docs, **About pull request reviews**. A review can be a comment, an approval or a request for changes.

All three were opened on 13 October 2026. The module descriptor is the source for the assessments, deadlines and submission described in E.1, E.4 and E.5. Chapters 1, 9 and 14 are the source for the reading record, the reading point, the contribution table, the handover documents and the reflection prompts.

Appendix F: Software Engineering for AI and Data Components

> *Applying the book's principles to software that relies on a model or a data service: a contract the domain owns, tests that tolerate change, an enforced time limit and a cautious fallback, and a staged rollout with a record of each decision.*

OO Development and Agile Practices
Designing, Testing and Evolving Software with AI Assistance

How to use this appendix

Some of the software you will build or maintain depends on something that is not ordinary code: a trained model that scores a request, a service that returns data you do not control, a pipeline whose output changes when someone retrains it. The risk is not that such a component is clever. The risk is that its answers change without a code change, that it can be slow or silent, and that its types and units leak into the code around it.

This appendix applies the book's principles to that situation: a contract that the domain owns, tests that do not depend on the model, a bounded wait, a cautious fallback, and a staged rollout that leaves a record. It does so through a small stand-alone project, the **risk lab**, whose code compiles for Java 17, has 67 tests and is listed in full in the appendix's companion. The lab does not touch the lending project. It asks one question, "how risky is it to lend this item to this borrower?", and puts a model behind the answer.

- **F.1** separates the two ways AI appears in this book.
- **F.2** replaces a leaky integration with a contract the domain owns.
- **F.3** shows how to test when an answer can change.
- **F.4** names the technical debt that machine learning systems collect.
- **F.5** deals with failure: a bounded wait and a cautious fallback.
- **F.6** rolls a model out in stages and records what each decision rested on.
- **F.7** is an exercise on the lab.

The appendix assumes Chapters 6 (interfaces and dependency inversion), 7 (test doubles), 8 (adapters) and 13 (boundaries, timeouts and flags). It does not teach machine learning. It says nothing about whether a model is accurate: no test here can.

F.1 Two ways AI appears in this book

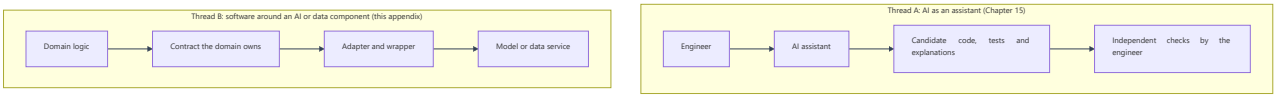


Figure F.1. The two threads are independent.

In **Thread A** an assistant helps you write software, and the software may have nothing to do with machine learning. Chapter 15 is about that. In **Thread B** the software you write contains or calls an AI or data component. The component is an architectural dependency: you must encapsulate it, test around it, bound it and evolve it safely. The lending project has already met this thread. The borrower directory of Chapters 8 and 13 is a data component outside your control, and Chapter 13 sets its policy: one attempt, a bounded wait, no retry, and a failure that is never reported as "not found".

Neither thread replaces the other. A strong assistant cannot rescue an application that is tangled with its model, and a clean domain model cannot make up for a model nobody is watching.

F.2 A contract the domain owns

The starting point

The exercise begins with a class that works on a good day. This is its one method:

```
public boolean approve(String borrowerId, String itemId) {
    try {
        HttpURLConnection connection = (HttpURLConnection)
                URI.create(serviceUrl + "/risk?borrower=" + borrowerId + "&item=" + itemId).toURL(
        BufferedReader in = new BufferedReader(new InputStreamReader(connection.getInputStream()))
        String line = in.readLine();
        String[] parts = line.split(" ");
        float score = Float.parseFloat(parts[1]);
        if (parts[0].equals("LOW")) {
            return score > 0.8f;
        }
        return false;
    } catch (Exception anything) {
        return false;
    }
}
```

Run against a small local service, it behaves like this:

```
good day            : true
confidence of 1.7   : true
service that is slow: true after 1502 ms
service that is down: false
```

The time on the third line varies from run to run. Read the class and the output together, and notice what is wrong:

1. **No timeout.** The slow service held the caller for about a second and a half, and nothing limits that. A stalled service would hold it for as long as it liked.
2. **An outage looks like a refusal.** The service that is down gives `false`, the same answer as a high-risk borrower. The caller cannot tell the two apart.

3. **A nonsense answer is accepted.** A confidence of 1.7 cannot exist, and the class approved it.
4. **A magic number and a string comparison.** `0.8f` and `"LOW"` are in the middle of the method with no name and no owner.
5. **Identifiers are pasted into the address** without encoding.
6. **It cannot be tested** without a running service.

All six are boundary failures, and they have a common cause: the domain's decision and the service's wire format are in the same place.

The contract

The remedy is Chapter 6's dependency inversion applied to a model. The domain defines the question in its own words, and everything else adapts to it.

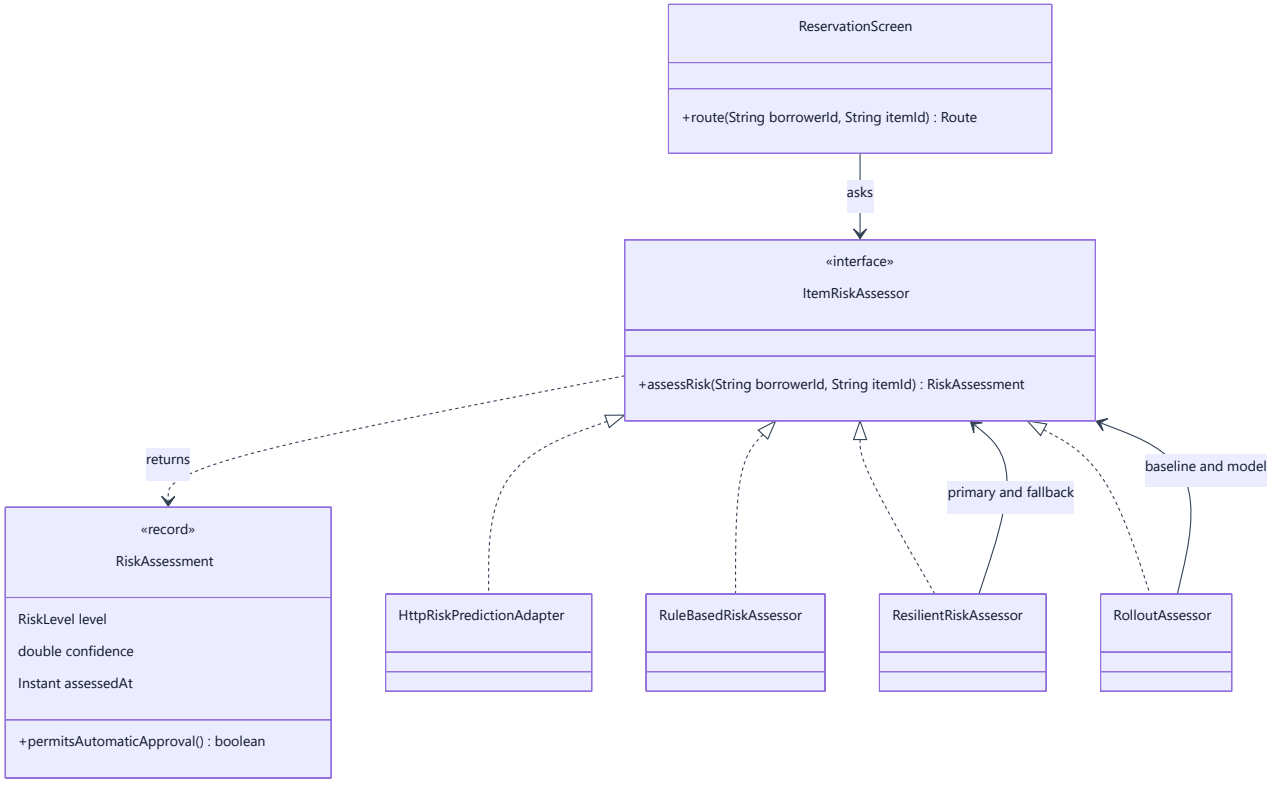


Figure F.2. The domain owns `ItemRiskAssessor` and `RiskAssessment`. Everything else implements or uses the contract.

The contract itself is small:

```
package risk.domain;

/**
 * The domain's question about risk, in the domain's words. An implementation may be a model, a ta
 * of rules or a test fake. The contract: the result is never null; if there is no trustworthy ans
 * the call throws {@link RiskAssessmentException}. It never returns a default that looks like an
 */
public interface ItemRiskAssessor {
    RiskAssessment assessRisk(String borrowerId, String itemId);
}
```

```
package risk.domain;

import java.time.Instant;
import java.util.Objects;

/** One answer to the question “how risky is it to lend this item to this borrower?”. */
public record RiskAssessment(RiskLevel level, double confidence, Instant assessedAt) {

    /** The least confidence at which a LOW assessment may approve a request without a person. */
    public static final double AUTOMATIC_APPROVAL_CONFIDENCE = 0.80;

    public RiskAssessment {
        Objects.requireNonNull(level, "level");
        Objects.requireNonNull(assessedAt, "assessedAt");
        if (Double.isNaN(confidence) || confidence < 0.0 || confidence > 1.0) {
            throw new IllegalArgumentException("Confidence must be between 0.0 and 1.0: " + confi
        }
    }

    /** Only a LOW level held with enough confidence may skip a person. */
    public boolean permitsAutomaticApproval() {
        return level == RiskLevel.LOW && confidence ≥ AUTOMATIC_APPROVAL_CONFIDENCE;
    }
}
```

Notice what the domain decided, and where. The confidence is validated in the constructor, so a value of 1.7 cannot become an assessment (an invariant, Chapter 2). The threshold has a name and an owner, the domain. `UNKNOWN` is a level that can never approve. Failure is a different thing from an answer: the contract says a failed call throws and never returns a default that looks like an answer. And the class that makes the decision knows none of this about models:

```
public Route route(String borrowerId, String itemId) {
    requireText(borrowerId, "borrowerId");
    requireText(itemId, "itemId");
    RiskAssessment assessment = assessor.assessRisk(borrowerId, itemId);
    if (assessment.level() == RiskLevel.HIGH) {
        return Route.REJECT;
    }
    return assessment.permitsAutomaticApproval() ? Route.AUTO_APPROVE : Route.SUPERVISOR_REVIEW;
}
```

`ReservationScreen` does not catch `RiskAssessmentException`. That is deliberate. An outage must not be turned into an approval by a convenient `catch`; where to fall back is a policy decision, made in one place (F.5).

F.3 Testing when an answer can change

A model's answer to the same input can change after retraining. It can vary between calls, and it can arrive late. If your unit tests call the model, your build becomes **flaky**: it fails without a code change, and the team stops believing it. The book's rule from Chapter 7 holds here: tests that decide whether the code is right must be deterministic. So each layer of the lab replaces the model with something controlled.

Layer	Tests	Count	What stands in for the model
The domain's own rules	`RiskAssessmentTest`, `ReservationScreenTest`	12 and 10	A hand-written `FakeItemRiskAssessor`
The cautious fallback, with a metamorphic property	`RuleBasedRiskAssessorTest`	16	A one-line history
The wrapper's policy	`ResilientRiskAssessorTest`	8	Fakes and a latch that never opens
The rollout	`RolloutAssessorTest`	10	Fakes that count their calls
The adapter against a real client	`HttpRiskPredictionAdapterContractTest` (tagged `integration`)	11	A local stub service built on the JDK's HTTP server

The lab has **67 invocations**: 56 run without the `integration` tag and 11 with it. Offline evaluation of the model itself (precision, recall, drift) is a different activity. It needs real data, runs out of process and is not in the lab. A green lab says that the code around a model keeps its promises. It says nothing about whether the model is any good.

A property instead of an exact value

You cannot write an exact assertion for every answer of a model, but you can assert a relation that must hold whatever the model learns. A **metamorphic** test states how two answers must relate. Here, a borrower with more late returns must never be assessed as less risky:

```
@ParameterizedTest
@CsvSource({"0, 1", "0, 3", "1, 2", "2, 3", "0, 8", "3, 4", "2, 9"})
void moreViolationsNeverLowerTheRisk(int fewer, int more) {
    // a metamorphic property: it does not say what the level is, only how two answers relate
    assertTrue(assess(more).level().compareTo(assess(fewer).level()) ≥ 0);
}
```

The test does not say what the level is. It checks that raising an input never lowers the output. The lab runs it on the rule-based assessor, which is deterministic. For a real model the same relation can be run over a sample of borrowers; a failure then tells you something about the model, and may not be a bug in your code.

A stub that misbehaves on purpose

The adapter is tested against a service you control, so that you can make it misbehave: `StubRiskService` answers well, answers nonsense, answers 500, sends nothing, and, most usefully, **sends its headers and the first byte of the body and then stalls**. Chapter 13 explained why that case matters: a client's request timeout need not cover a body that stops. The lab's adapter bounds the whole answer, and its test has an outer deadline (`@Timeout(5)`) so that a missing bound shows as a failed test and not as a hang. Fault E11 in the companion removes the bound and shows exactly that.

F.4 Technical debt in machine learning systems

Sculley and colleagues argued that machine learning systems collect a particular kind of technical debt [1]. Their first figure makes the point visually: only a small fraction of a real-world system is the model code, and the surrounding infrastructure is vast and complex. In the paper's section on glue code the authors say that a mature system might end up being at most 5% machine learning code and at least 95% glue code. The paper names several risks. The ones that this book's techniques address are:

The paper's risk	What it means	A move from this book	Where in the lab
Entanglement ("changing anything changes everything")	No input is really independent, so a change to one signal can change all the answers	Give clients a small, focused view of the answer (interface segregation, Chapter 6)	`RiskAssessment` carries a level and a confidence, not a feature vector
Boundary erosion	Complex models blur the abstraction boundaries that encapsulation provides	Let the domain own the abstraction (dependency inversion, Chapter 6)	`ItemRiskAssessor`
Glue code and pipeline jungles	Code that surrounds a general-purpose package, and pipelines that grow without design	Wrap a black-box package in a common interface; keep each step small and testable	`HttpRiskPredictionAdapter`
Data dependencies	Inputs that change meaning or quality without a compiler error	Treat incoming data as untrusted input: validate and reject malformed rows with a clear failure (Chapter 8)	The adapter's `parse` and the `RiskAssessment` constructor
Changes in the external world	The data the model sees drifts	Record what each decision rested on and be able to switch back	`ProvenanceRecord` and `RolloutMode.OFF` (F.6)

The paper names other risks, among them hidden feedback loops and undeclared consumers, that the lab does not address. A second paper, Breck and colleagues' *ML Test Score*, is a checklist of tests and monitoring for production readiness [2]; read it after this appendix.

F.5 Failure, a bounded wait and a cautious fallback

When your software depends on a component that can be slow, wrong or unavailable, decide in advance what it does. The book has three rules, from Chapters 8 and 13 and from the change specification of Chapter 12:

1. **An outage is never "not found".** If the service does not answer, the system must not assume the record does not exist, or that the risk is zero.
2. **Never default to a category.** If a borrower's category cannot be established, the request fails. Chapter 8 notes that defaulting to staff would be worse than defaulting to student, because it grants more, and Chapter 13 tests that an unknown code is defaulted to neither.
3. **Bound the whole wait.** A call to an outside component has a limit that covers the entire answer, and the limit is tested.

Fail, or fall back?

For the borrower directory, Chapter 13 chose to **fail**: one attempt, a bounded wait, no retry and no fallback, because a category cannot be guessed safely. For a risk assessment there is a safe fallback, which the directory lacks: a cautious answer that cannot do harm. The lab's `RuleBasedRiskAssessor` is deterministic and never approves without a person, because its confidence is fixed below the automatic-approval threshold. If the model cannot answer, a request goes to a supervisor, or is rejected if the history is bad, and is never approved on a guess. The decision between failing and falling back is a design decision. Record it in a decision record (Appendix C).

A wrapper that enforces its limit

```
@Override
public RiskAssessment assessRisk(String borrowerId, String itemId) {
    Future<RiskAssessment> pending = executor.submit(() → primary.assessRisk(borrowerId, itemId)
    try {
        return Objects.requireNonNull(pending.get(limit.toNanos(), TimeUnit.NANOSECONDS),
                "the primary assessor returned null");
    } catch (TimeoutException late) {
        pending.cancel(true);
        return fallBackBecause("no answer within " + limit.toMillis() + " ms", borrowerId, itemId
    } catch (InterruptedException interrupted) {
        pending.cancel(true);
        Thread.currentThread().interrupt();
        throw new RiskAssessmentException("Interrupted while waiting for the risk service.", inte
    } catch (ExecutionException failed) {
        Throwable cause = failed.getCause();
        if (cause instanceof RiskAssessmentException) {
            return fallBackBecause(cause.getMessage(), borrowerId, itemId);
        }
        if (cause instanceof RuntimeException runtime) {
            throw runtime;
        }
        if (cause instanceof Error error) {
            throw error;
        }
        throw new IllegalStateException("The primary assessor failed unexpectedly.", cause);
    }
}
```

The wrapper runs the primary on its own thread and waits for it for at most the limit. When the limit passes it asks the thread to stop with `cancel(true)` and answers from the fallback. Asking is all it can do: a primary that ignores interruption keeps running, but the caller no longer depends on it, because the caller has stopped waiting [3]. Look at what else the method decides:

- **Only the expected failure leads to the fallback.** A `RiskAssessmentException` does. An `IllegalStateException` or any other programming error propagates, so a bug is found and not hidden by a quiet fallback.
- **A primary that returns null is a bug** and is reported as one.
- **An interrupted caller keeps its interrupt flag** and gets a failure, not a fallback.
- **If the fallback also fails, the caller sees the failure.** An outage never becomes an approval.

- **The fallback is counted** (`fallbackCount()`), so that a monitor can notice that the model has been silent.

Each of those has a test. The companion's fault experiments remove them one at a time. E1, a wait of an hour in place of the limit, makes two tests fail by their deadline; E2, no `cancel`, fails the test that checks the stalled call is asked to stop; E3, any runtime failure falling back, fails the test for a programming error; E4, no restored flag, fails the interrupt test.

F.6 Rolling a model out in stages

Replacing a model is a change in behaviour even though no line of your code changes. Chapter 13 introduced the feature flag for exactly this: ship the change turned off, test both settings, and plan its removal. For a model the same idea has four settings:

Mode	Who decides	What happens to the model
`OFF`	The baseline	Never called
`SHADOW`	The baseline	Called, and its answer is only recorded. A failure of the model never affects the decision
`CANARY`	The model for a stable share of borrowers, the baseline for the rest	Called for the share
`ON`	The model	Called for everyone

The class that chooses is short:

```
public boolean inCanary(String borrowerId) {
    return Math.floorMod(borrowerId.hashCode(), 100) < canaryPercent;
}
```

```
@Override
public RiskAssessment assessRisk(String borrowerId, String itemId) {
    return switch (mode) {
        case OFF → baseline.assessRisk(borrowerId, itemId);
        case SHADOW → shadow(borrowerId, itemId);
        case CANARY → inCanary(borrowerId) ? modelDecides(borrowerId, itemId) : baseline.assessR
        case ON → modelDecides(borrowerId, itemId);
    };
}
```

The canary's share is **stable**: the same borrower is always in or always out for a given percentage, so a person does not get one answer today and another tomorrow. And every decision in which the model took part is recorded with the model version, the model's answer (empty if it did not answer), the assessment actually used, and whether the model decided (`ProvenanceRecord`). That record is what lets you answer, after an incident, "what was the model doing, and who decided?". Record identifiers and decisions, and no more personal data than the review needs; Chapter 15, Section 15.7 discusses what not to log.

Be clear about what a flag does not do. It switches which component decides. It cannot undo decisions that have been made, and it does not repair drift. Setting the mode to `OFF` is a rollback of future decisions only. As in Chapter 13, a flag is inventory with a cost: plan how it will be removed, and test both settings until then.

F.7 Practical exercise: wrap a scoring service

The book's project. Work on the risk lab. **Your team's project.** Apply the same steps to a model or data service in your own system, or to the model-serving mapping suggested in Appendix D, Section D.5.

The companion supplies the starting class `UnsafeRiskScorer`, the probe that runs it, and the lab's tests. The tests are the acceptance tests for the exercise. The companion also lists a model solution; attempt each part before reading it.

Part	Do	Evidence
A	Run the probe. Read `UnsafeRiskScorer` and list every way it can fail. The probe shows three. Find at least five more by reading	Your list, with the line each comes from
B	In package `risk.domain`, write `RiskLevel`, `RiskAssessment`, `RiskAssessmentException` and `ItemRiskAssessor` so that `RiskAssessmentTest` compiles and passes	The test result
C	Write `Route`, `ReservationScreen` and a `FakeItemRiskAssessor`. Make `ReservationScreenTest` pass	The test result
D	Write `RuleBasedRiskAssessor`. Make `RuleBasedRiskAssessorTest`, including the metamorphic test, pass	The test result
E	Write `HttpRiskPredictionAdapter`. Make the adapter's contract test pass, including both stalls	The test result and the time the stalled cases took
F	Write `ResilientRiskAssessor`. Make its tests pass. Then remove the limit in a scratch copy and read what fails	The failure, and what it told you
G	Write `RolloutAssessor`, `RolloutMode` and the provenance types. Make the rollout tests pass	The test result

Closing note. A short note: your list of flaws from part A, the number of tests that pass, one fault you introduced and the test that caught it, and one thing the tests do not show. A transfer note for a team on another domain: replace "borrower" and "item" by the entities of your system, and keep the contract, the stub, the bound and the fallback policy.

F.8 Sources

Opened on 13 October 2026. The lab, its tests and the explanations are original.

1. Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F. and Dennison, D., **Hidden Technical Debt in Machine Learning Systems**, *Advances in Neural Information Processing Systems* 28, 2015. The first figure (a small fraction of the system is model code), the CACE principle, boundary erosion, glue code and pipeline jungles, data dependencies; the remark about 5% and 95% is in the glue-code section.

2. Breck, E., Cai, S., Nielsen, E., Salib, M. and Sculley, D., **The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction**, IEEE Big Data, 2017. A checklist of tests and monitoring for production readiness.

3. Oracle, **HttpClient (Java SE 17)** and **CompletableFuture (Java SE 17)**. Cancelling attempts to cancel the exchange and is not guaranteed to be prompt; `get` with a timeout throws `TimeoutException` without cancelling. Also cited in Chapter 13.

4. Oracle, **HttpRequest.Builder (Java SE 17)** and **com.sun.net.httpserver (Java SE 17)**. The request timeout, and the embedded server used by the stub. Also cited in Chapter 13.

Object Oriented Development and Agile Practices

Software changes. Requirements evolve, new features challenge earlier decisions, and working code must remain understandable to the people who maintain it.

OO Development and Agile Practices brings object-oriented design and Agile teamwork together through an evolving Java project. Following a campus equipment lending system from its first responsibilities to a tested change and final handover, readers learn how design decisions hold up as a system grows.

The book connects interfaces, composition, design patterns and architecture with practical work in requirements, testing, refactoring, integration and delivery. Worked examples, exercises and accompanying code explanations make each idea open to inspection: readers can trace a decision, run a test, examine a failure and justify a correction.

AI assistance appears throughout as a source of suggestions to question and verify. Readers learn to evaluate generated code, tests and explanations while retaining responsibility for the result.

Written for students who know programming fundamentals, this book develops the judgement needed to build software that can change and to explain, with evidence, why a change is ready.



by Polla Fattah
and Collection of LLMs