

SOFTWARE DESIGN COURSE

Object-Oriented Design with UML

Analysis and design with UML: use cases, domain models, class, sequence and state diagrams, checked against running code.

12 Chapters

Philip Osztromok

Generated with Claude

OBJECT-ORIENTED DESIGN WITH UML

Contents

Chapter 1	Why Model	2
Chapter 2	Diagrams as Code	10
Chapter 3	Use Cases	27
Chapter 4	Domain Model	44
Chapter 5	Class Diagrams	57
Chapter 6	Responsibilities	77
Chapter 7	Sequence Diagrams	102
Chapter 8	State and Activity	122
Chapter 9	Structure Diagrams	151
Chapter 10	UML and Code	176
Chapter 11	Patterns and Review	196
Chapter 12	Capstone	226

Object-Oriented Design with UML

Course 1 · Chapter 1 · Why Model: Analysis, Design and UML

You can write code without drawing anything, and for a small script you should. But sooner or later you will join a project where nobody can say which part owns the loan rules, or you will have to explain your design to someone who will not read your code, or you will find, three weeks in, that two classes each assumed the other did the checking. A **model** is a cheap way to find those problems before they are expensive. This course teaches how to think a design through, and how to draw it using the Unified Modeling Language (UML) so that others can read it.

> **WHAT WAS CHECKED, AND WHAT WAS NOT**
>
> The two diagrams and the Python in this chapter were produced and run on this machine: Mermaid's command-line renderer (`@mermaid-js/mermaid-cli` 12.0.0, Mermaid 12.1.0, drawing with Chrome) made the SVGs shown, and the Python (3.14) ran with the output pasted. The UML version and date come from OMG's own page for the specification, which I read. Two things I did *not* re-read for this chapter and give from general knowledge: the list of the 14 diagram types, and the three "modes" of using UML, which I attribute below to Martin Fowler's book *UML Distilled*. PlantUML (needs Java, which is not installed here) arrives in chapter 2.

Analysis and Design: Two Questions

Object-oriented analysis asks *what is the problem?* It looks at the situation (a library lending books) and finds the things in it, the goals people have and the rules that apply, without deciding how software will do it. **Object-oriented design** asks *what shape should the solution take?* It decides which classes exist, what each is responsible for, and how they cooperate. In practice the two overlap and you go back and forth, but keeping the questions apart prevents a common failure: designing a clever solution to a problem nobody checked.

	ANALYSIS	DESIGN
Question	What is the problem, in the user's world?	How will the software be organised to solve it?
Typical output	Use cases, a domain model, a glossary	Class design, interactions, states, components

	ANALYSIS	DESIGN
Who must understand it	Users and domain experts, as well as developers	Mainly developers
In this course	Chapters 3 and 4	Chapters 5 to 11

Requirements and design documents as parts of a whole project are covered in *The Software Development Lifecycle* 5 (Requirements & User Stories) and 8 (Design Documents & ADRs); this course is about the modelling skills inside those activities and does not repeat them.

What UML Is

UML is a standard, graphical vocabulary for describing software: boxes for classes, arrows for relationships, lifelines for objects exchanging messages. It is maintained by the Object Management Group (OMG). OMG's page for the specification lists **UML 2.5.1, published December 2017**, as the current formal version (2.0 was July 2005; 1.1, the first, December 1997). That page mentions no ISO number, so I make no claim about one. Its value is shared meaning: when you draw a hollow diamond, a reader who knows UML knows what you mean without asking.

Three Ways to Use UML

Martin Fowler's *UML Distilled* (from my memory of the book, not re-read) describes three modes, and knowing which one you are in settles most arguments about "how detailed should the diagram be":

MODE	WHAT IT IS	PRECISION	LIFETIME	EXAMPLE
Sketch	Drawings to communicate an idea or explore one	Low: leave out whatever does not matter	Minutes or days: thrown away, or redrawn when needed	Two developers at a whiteboard deciding how loans and reservations relate
Blueprint	Detailed designs a team builds from, often by a designer for others	High: every operation and type	Months, with a real risk of falling out of date	A full class design handed to another team

MODE	WHAT IT IS	PRECISION	LIFETIME	EXAMPLE
Programming language	Models complete enough to generate the program	Complete	The model *is* the source	Model-driven tools (chapter 10 explains why this faded)

This course works almost entirely in sketch mode, with the sketches kept as text beside the code. That is where UML pays for itself in most teams: a quick shared picture, cheap to change. Detail beyond what the reader needs is not extra rigour, it is extra maintenance.

The Diagram Types

UML 2 defines 14 kinds of diagram, in two groups: **structure** (what exists) and **behaviour** (what happens). Interaction diagrams are a sub-group of the behaviour ones. This course teaches the ones you will actually draw:

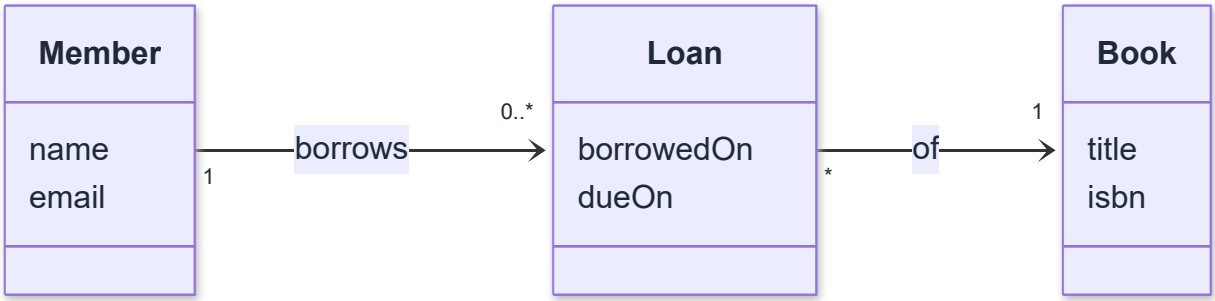
GROUP	DIAGRAM	ANSWERS	CHAPTER
Structure	Class	What kinds of things exist, and how are they related?	4, 5
Structure	Component, package, deployment	How is the system divided up, and where does it run?	9
Structure	Object	What does one example snapshot look like?	9
Behaviour	Use case	Who uses the system, and for what goals?	3
Behaviour (interaction)	Sequence	In what order do objects exchange messages in one scenario?	7
Behaviour	State machine	What states can one object be in, and what moves it between them?	8
Behaviour	Activity	What are the steps of a workflow, including choices and parallel work?	8

That covers nine of the fourteen: class, object, component, package, deployment (structure) and use case, sequence, state machine, activity (behaviour). The other five are composite structure, profile (structure) and communication, interaction overview, timing (behaviour). Communication

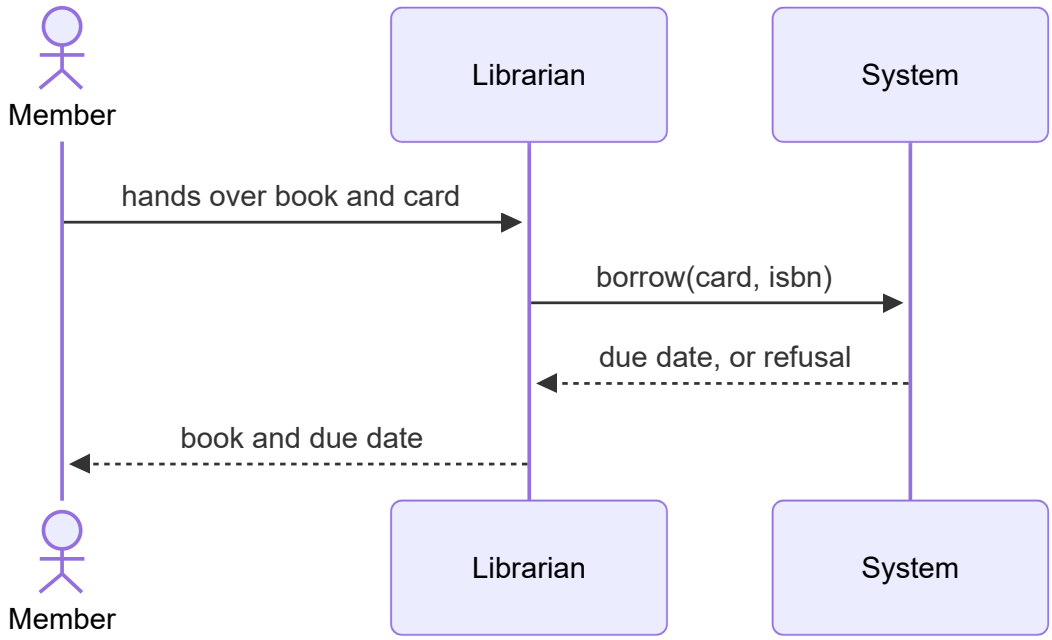
diagrams get a paragraph in chapter 7; the rest are specialist tools you can learn from the specification the day you need one. Not learning them is a sensible choice, not a gap.

A First Taste: One Idea, Two Pictures

The library lending system runs through most of this course, so meet it now. A community library has members, books, and loans. Here is the idea as a structure diagram (a small class diagram) and as a behaviour diagram (a sequence diagram). Both are Mermaid text, rendered to SVG by the real tool. How to write and render them is chapter 2; for now, just read them.



Structure: a member borrows zero or more loans (the arrow means a member can reach their loans); each loan is of exactly one book. A book can appear in many loans over time (the `*`).



Behaviour: one scenario, "borrow a book", with a person, the librarian and the system. Time runs downwards; dashed arrows are replies.

Neither picture says anything about databases, screens or Python. That is the point of a sketch: it is about the problem and the cooperation, and you could argue about it with a librarian.

A Diagram Says Less Than Code, On Purpose

Here is the structure sketch turned into Python (run on this machine). Read the comments: they show where each line of the diagram went.

```
from dataclasses import dataclass, field
from datetime import date, timedelta


@dataclass
class Book:
    title: str
    isbn: str


@dataclass
class Loan:
    book: Book                      # Loan "*" --> "1" Book
    borrowed_on: date
    due_on: date


@dataclass
class Member:
    name: str
    email: str
    loans: list[Loan] = field(default_factory=list)      # Member "1" --> "0..*" Loan

    def borrow(self, book: Book, today: date) -> Loan:
        loan = Loan(book, today, today + timedelta(days=21))
        self.loans.append(loan)
        return loan


ada = Member("Ada", "ada@example.org")
dune = Book("Dune", "978-0-00-000000-0")
loan = ada.borrow(dune, date(2026, 10, 10))
print(f"{ada.name} has {len(ada.loans)} loan(s); {loan.book.title} is due {loan.due_on}"
```

```
Ada has 1 loan(s); Dune is due 2026-10-31
```

Look at what the code had to decide that the diagram never mentioned: that a loan lasts 21 days, that it is the member who creates it, that a loan keeps its book as an object rather than an id, and that `borrow` exists at all. The diagram was not wrong; it was silent. Some of those silences are decisions for later chapters (chapter 6 asks who should create a loan); some are rules you must go and ask a librarian about (is 21 days right?). Finding the silences is half the value of drawing. Chapter 10 returns to how diagrams and code map onto each other, and why they do not stay in step by themselves.

What UML Is Not

- **Not a process.** UML says how to draw, not when, in what order, or by whom. It fits an agile team sketching on a whiteboard as well as a long, documented project.
- **Not code, and not automatically in sync with code.** A diagram drawn last month describes last month. Keeping it honest is a choice you make (small diagrams, kept as text next to the code, regenerated where possible); there is no magic.
- **Not a measure of quality.** A beautiful diagram can describe a poor design, and a messy sketch can describe a good one. Judging the design is its own skill (chapters 6 and 11).
- **Not the only notation.** Boxes and arrows on a whiteboard, a table of responsibilities, or a paragraph can be better. Use UML where its shared meaning helps.

How Much Modelling Is Worth It?

There is no formula, but these questions give a useful answer:

- **Is the thing hard to change later?** Relationships between your main concepts, the boundaries between components and the order of a tricky interaction are expensive to get wrong. The name of a private helper is not.
- **Is more than one person involved?** A diagram is a conversation tool; for a one-person throwaway script it is overhead.
- **Is there something you do not yet understand?** Drawing is a cheap way to discover it. If you can already see the whole design, writing the code is the faster check.

- **Who will maintain the diagram?** If the answer is "nobody", keep it a sketch and date it, or do not keep it.

A good stopping rule: **stop when the diagram answers the question you drew it for.** Then go and test the idea in code. Design and code are two ways of finding out whether an idea works, and the second is more truthful; use the first to avoid the slow, expensive part of the second.

MISTAKES WHILE PREPARING THIS CHAPTER

- **The first render was 190 KB per diagram.** Mermaid's renderer embeds a web font into every SVG, which would have bloated the page and could have clashed between diagrams that share an element id. What went wrong: I accepted the tool's default output. Asking for a plain system font and a distinct `--svgId` for each diagram made each file about 25 KB and gave every diagram its own id. Takeaway: check the size and the ids of generated SVG before inlining it in a page.
- **Pale, unreadable diagram text on the dark page.** The class diagram rendered fine in a PNG but, inlined, its labels were nearly invisible: Mermaid draws class-diagram text as HTML inside the SVG, which inherits the page's light text colour, on the diagram's white panel. The sequence diagram (SVG text) was fine, which is why only a screenshot of the real page showed it. Takeaway: look at the diagram in the page itself, on the real theme, not just in the tool's own output; the course stylesheet now sets a dark text colour on diagram panels.
- **My own count did not add up.** A first draft of the diagram-types section said "the other seven" were not taught, but the table taught nine of fourteen, leaving five. What went wrong: I wrote the number from memory instead of counting the list. Takeaway: when a list must total a number, write all the items out and count them (here 5 structure and behaviour types left over, listed above).

Hands-On Exercises

EXERCISE 1

Classify five situations as sketch, blueprint or programming-language mode, and say what you would draw (and what you would leave out) in each: (a) deciding with a colleague how to split a billing feature; (b) handing a payment-gateway design to an outsourced team; (c) a new teammate's first-day explanation of the system; (d) a tool that generates code from the diagram; (e) a ten-line script you will run once.

View solution

EXERCISE 2

Draw a small structure sketch (Mermaid class diagram) for the library's *reservation* idea: a member places reservations, each holding a place for one book. Render it, and say what the multiplicities mean in words.

View solution

EXERCISE 3

Turn your reservation sketch into Python, run it, and find a rule the diagram does not state that the code allows to be broken. List the questions you would take to a librarian.

View solution

CHAPTER 1 QUICK REFERENCE

- Analysis = what is the problem; design = what shape the solution takes; they overlap, but keep the questions apart
- UML 2.5.1 (OMG, December 2017) is the current formal version per OMG's page; it is a notation, not a process or a quality measure
- Three modes: sketch (this course), blueprint, programming language; the mode decides the detail
- 14 diagram types; taught here: class, use case, sequence, state machine, activity, component/package/deployment, object
- Diagrams say less than code, on purpose: the gaps are the questions to ask
- Model what is hard to change, shared by several people, or not yet understood; stop when the diagram answers its question; then test in code

COMING NEXT

Object-Oriented Design with UML 2 (Diagrams as Code): why text beats drag-and-drop, Mermaid and PlantUML compared, installing and rendering both for real, and keeping the sources next to your code.

Object-Oriented Design with UML

Course 1 · Chapter 2 · Diagrams as Code: Mermaid and PlantUML

Chapter 1 argued for sketches that are cheap to change. A diagram you draw with a mouse in a drawing tool is cheap to *make* and expensive to *change*: nobody can see what changed between two versions, two people cannot edit it at once, and it quietly rots in a folder nobody opens. A diagram written as **text** behaves like code: it goes in the repository, shows up in a pull request, and is regenerated by a command. This chapter sets up the two tools the course uses, for real, and shows where each is strong.

WHAT WAS RUN, AND WHAT WAS NOT

Everything on this page was run on Windows with these versions: **Mermaid 12.1.0** drawn by **@mermaid-js/mermaid-cli 12.0.0** (using the installed Chrome), and **PlantUML 1.2026.8** (released 2026-09-05), the *native Windows build* from the project's GitHub releases. Both install into a scratch folder, nothing system-wide. The diagrams you see are the tools' real output. **Not checked:** that the PlantUML download is authentic beyond coming from the project's release page (its release lists signature files for the jar files, and I did not verify any); how the tools behave on macOS or Linux; and which editors preview which format (I did not test VS Code or JetBrains extensions). Where the page says "GitHub renders Mermaid" that comes from GitHub's own documentation, which I read.

Why Text Beats Drag-and-Drop (for Developers)

	DIAGRAM AS TEXT	DRAWN IN A GUI TOOL
Reviewing a change	A normal diff, a few lines	"Binary file changed", or a screenshot someone must remember to update
Version history	Every change is a commit with a reason	Whatever the file format and your discipline allow
Merging two people's edits	Normal merge, normal conflict markers	Usually "pick one"
Layout	Automatic (you cannot place boxes exactly)	Fully yours (and you must move every box when something is added)
Learning curve	A small syntax to learn	A tool to learn

	DIAGRAM AS TEXT	DRAWN IN A GUI TOOL
Pretty, bespoke diagrams	Limited: you accept the tool's look	Better: this is where GUI tools win

The "automatic layout" row is both the cost and the benefit. You give up control of exactly where a box sits and gain the end of tidying. For a design sketch that is the right trade; for a poster, use a drawing tool. Here is the benefit measured on a real change: I committed the library diagram's source and its rendered SVG, then added one class and one relationship, and ran `git diff`.

```
diff --git a/docs/diagrams/loans.class.mmd b/docs/diagrams/loans.class.mmd
index 6334293..796fe9a 100644
--- a/docs/diagrams/loans.class.mmd
+++ b/docs/diagrams/loans.class.mmd
@@ -8,9 +8,13 @@ classDiagram
         borrowedOn
         dueOn
     }
+    class Reservation {
+        reservedOn
+    }
     class Book {
         title
         isbn
     }
     Member "1" --> "0..*" Loan : borrows
     Loan "*" --> "1" Book : of
+    Reservation "*" --> "1" Book : holds a place for
```

```
Source diff (the .mmd file):   483 bytes, 4 lines added
Rendered SVG diff (same change): 54,559 bytes, one 24 KB line replaced

$ git diff --stat
 docs/diagrams/loans.class.mmd     | 4 ++++
 docs/diagrams/loans.class.mmd.svg | 2 +-
 2 files changed, 5 insertions(+), 1 deletion(-)
```

The source change is readable at a glance (a new `Reservation` class and one arrow); the same change in the rendered SVG is a single 24 KB line, and the diff of it is 54,559 bytes of noise. So

review the source, not the picture. Whether to commit the rendered files at all is a choice: committing them lets a plain Markdown viewer show them without a build step; not committing them keeps diffs clean and avoids stale pictures. A common compromise is to commit the source and have a build or CI step render the pictures; I have not set up CI here, so I only describe it.

Two Tools, Compared

Mermaid is JavaScript. A diagram is a text block starting with its type (`classDiagram`, `sequenceDiagram`), and it is drawn in a browser or by a command-line wrapper that drives a headless browser. **PlantUML** is Java (here, a native build of it): a diagram sits between `@startuml` and `@enduml` and is rendered by a program to SVG, PNG and other formats. Layout for several PlantUML diagram types is done by a separate program, **Graphviz**.

	MERMAID	PLANTUML
Needs	Node and a browser (for the CLI)	Java, or the native build used here; Graphviz for some layouts (see below)
Renders natively on GitHub	Yes: a `mermaid` fenced block in Markdown (GitHub's documentation, read for this chapter)	Not mentioned in that documentation page, so plan on rendering it yourself
Class, sequence, state	Yes	Yes
Use case	Yes in 12.x, but the keyword is `usecase-beta`: treat it as unstable	Yes
Activity, component, deployment, object	Not as UML diagram types in the syntax I tried (below)	Yes, all four (below)
Look	Modern, themeable, light or dark by a setting	Plainer by default; many themes
Error behaviour	Fails with a parse error, non-zero exit	Draws the error into the image (below)

The coverage row was tested, not assumed. I wrote the smallest diagram I could for each type and rendered it with each tool:

```
--- Mermaid 12.1.0 (mmdc 12.0.0)
uc.mmd    (first line: usecase-beta): rendered
act.mmd   (first line: activity): FAILED: UnknownDiagramError: No diagram type detected
comp.mmd  (first line: componentDiagram): FAILED: UnknownDiagramError: No diagram type d
state.mmd (first line: stateDiagram-v2): rendered
--- PlantUML 1.2026.8 (native Windows build)
uc.puml   (use case):   rendered
act.puml  (activity):   rendered
comp.puml (component):  rendered
dep.puml  (deployment): rendered
obj.puml  (object):     rendered
state.puml (state):     rendered
```

Read that carefully. A failure for `activity` and `componentDiagram` proves only that *those words are not Mermaid diagram types*; Mermaid has flowcharts (which can draw an activity-like flow) and an `architecture-beta` type that I did not test. Mermaid's own syntax reference page (read for this chapter) lists use case but not activity, component, deployment or object diagrams. So the course draws the structure of the data and the interactions in Mermaid, and reaches for PlantUML for the diagram types Mermaid does not offer.

Installing Both (Without Touching Your System)

```
# Mermaid: Node.js 18+ is needed (24.18.0 here). Install into a project folder, not globa
mkdir ooad && cd ooad
npm install @mermaid-js/mermaid-cli@12.0.0
# mermaid-cli drives a browser; point it at one you already have instead of downloading a
#   set PUPPETEER_SKIP_DOWNLOAD=1 before npm install, and put the path in a puppeteer co

# PlantUML: download the native build for your OS from
#   https://github.com/plantuml/plantuml/releases   (native-plantuml-windows-amd64-1.202
# unzip it anywhere; it contains plantuml.exe and the runtime libraries it needs
plantuml.exe -version
```

```
$ plantuml -version
PlantUML version 1.2026.8 / 149874a [2026-09-05 15:59:30 UTC]
(GPL source distribution)
```

```
...
Java Runtime: GraalVM Runtime Environment
```

The usual way to run PlantUML is `java -jar plantuml.jar`; I did not use it because Java is not installed on this machine, and the native build needs none (it ships its own runtime, which the `Java Runtime: GraalVM` line shows). PlantUML also ran here with Graphviz present:

```
$ plantuml -testdot
GraphViz: dot - graphviz version 2.44.1 (20200629.0846)
Installation seems OK. File generation OK
```

Graphviz's `dot` was already on this machine's path; I do not know where it came from, and **I did not test PlantUML without it**. If a class or component diagram fails on your machine with a message about `dot`, install Graphviz (I understand sequence diagrams do not need it, but did not test that). The browser config for mermaid-cli (`-p`) and a config that sets a plain font (`-c`) are two small JSON files:

```
# cfg.json   (puppeteer: which browser to use)
{"executablePath": "C:/Program Files/Google/Chrome/Application/chrome.exe", "args": ["--

# mm.json    (mermaid: a system font, so the SVG does not embed a 160 KB web font)
{"theme": "default", "fontFamily": "Arial, Helvetica, sans-serif"}

mmdc -i diagram.mmd -o diagram.svg -p cfg.json -c mm.json --svgId unique-id
```

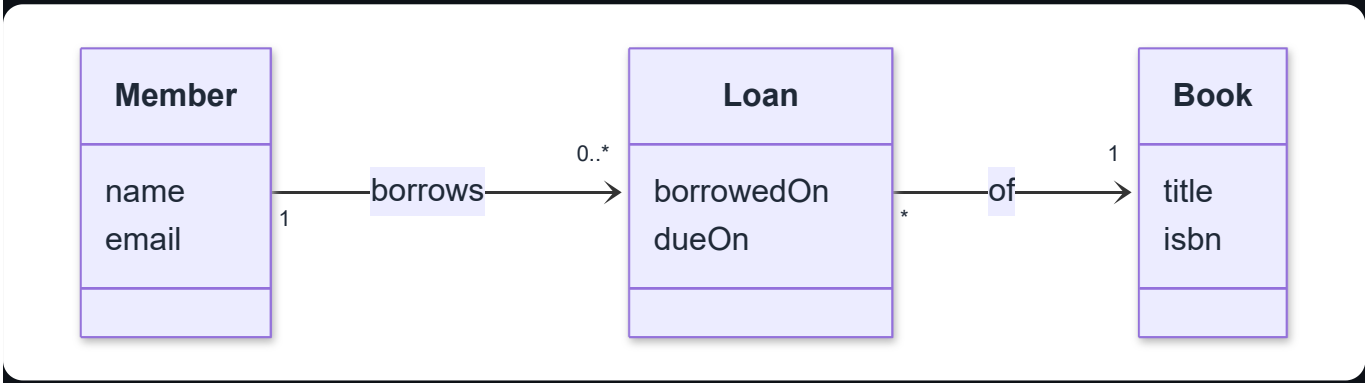
The Same Two Diagrams, in Both Tools

Chapter 1's class and sequence sketches, written first in Mermaid and then in PlantUML. Same meaning, different syntax. Compare how each says "a member has zero or more loans" and "the librarian asks the system".

Class diagram

```
classDiagram
    direction LR
    class Member {
```

```
        name
        email
    }
    class Loan {
        borrowedOn
        dueOn
    }
    class Book {
        title
        isbn
    }
    Member "1" --> "0..*" Loan : borrows
    Loan "*" --> "1" Book : of
```



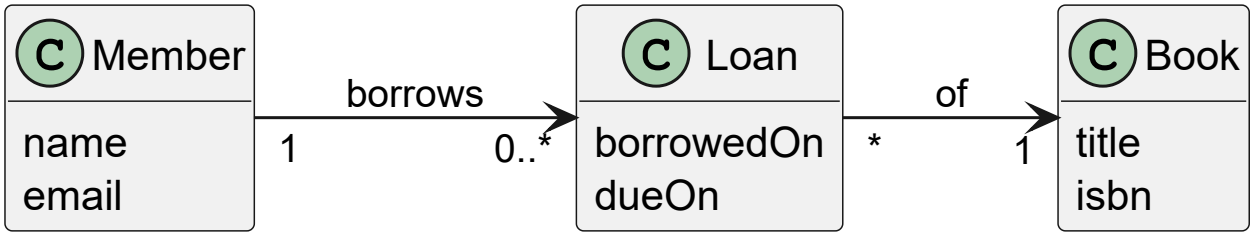
Mermaid

```
@startuml
scale 1.5
hide empty methods
left to right direction

class Member {
  name
  email
}
class Loan {
  borrowedOn
  dueOn
}
class Book {
  title
  isbn
}

Member "1" --> "0..*" Loan : borrows
```

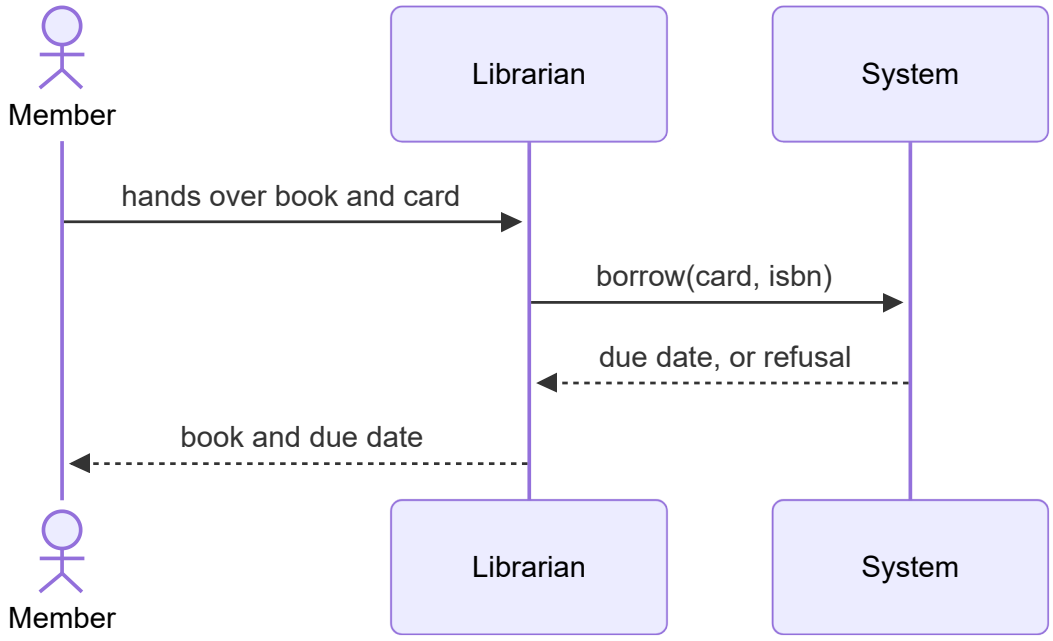
```
  Loan "*" --> "1" Book : of
  @enduml
```



PlantUML. `hide empty methods` hides the empty operations compartment and `scale 1.5` enlarges the picture; without it PlantUML's drawing is small on a web page.

Sequence diagram

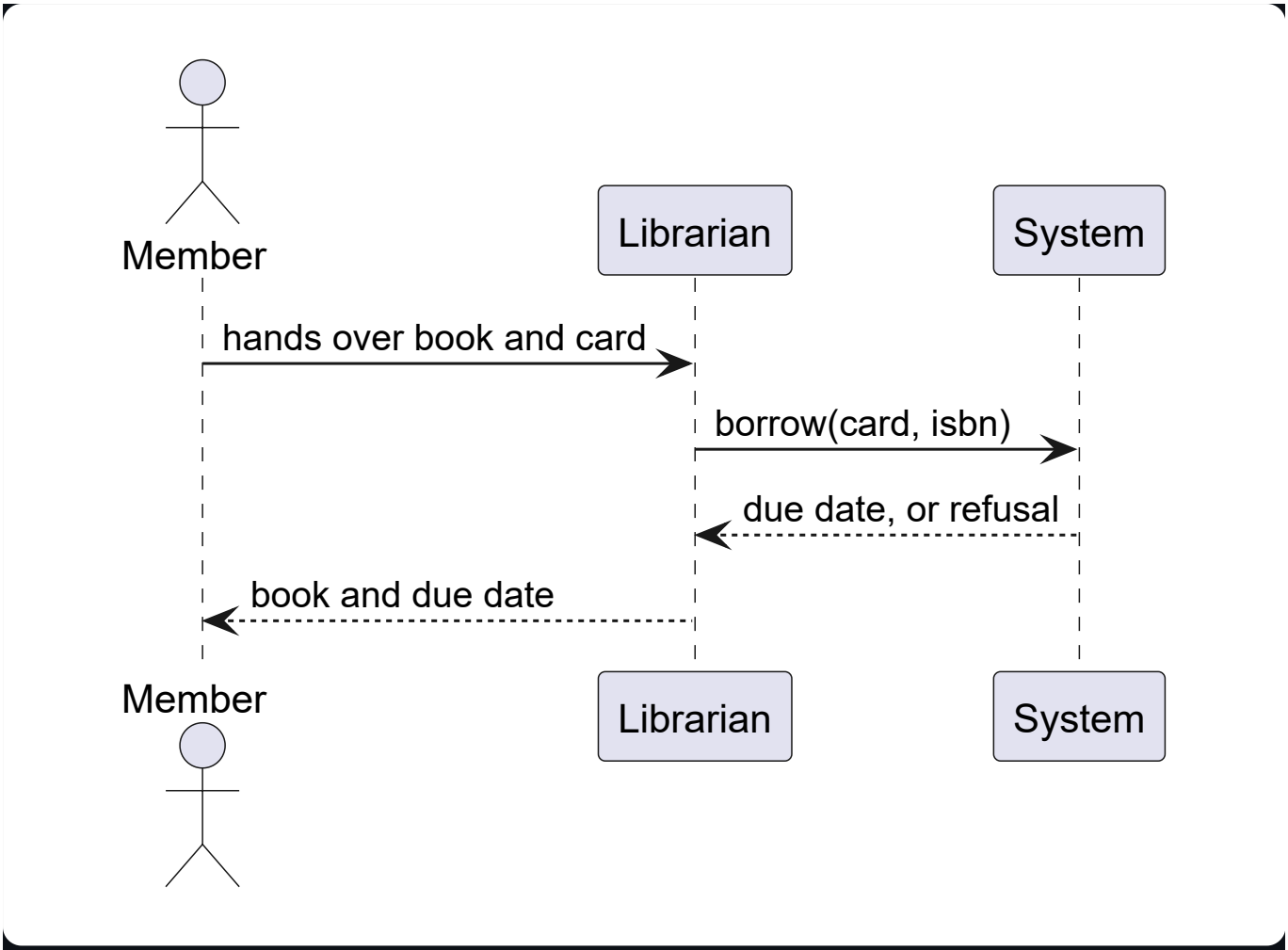
```
sequenceDiagram
    actor M as Member
    participant L as Librarian
    participant S as System
    M->>L: hands over book and card
    L->>S: borrow(card, isbn)
    S-->>L: due date, or refusal
    L-->>M: book and due date
```



Mermaid

```
@startuml
scale 1.5
actor Member as M
participant Librarian as L
participant System as S

M -> L : hands over book and card
L -> S : borrow(card, isbn)
S --> L : due date, or refusal
L --> M : book and due date
@enduml
```



PlantUML

IDEA	MERMAID	PLANTUML
Start	`classDiagram`	`@startuml` … `@enduml`
Attributes in a class	Lines inside `class X { }`	Lines inside `class X { }`
Association with multiplicity	`A "1" --> "0..*" B : label`	The same text
Call and reply in a sequence	`A->>B: text` and `B-->>A: text`	`A -> B : text` and `B --> A : text`
A person	`actor M as Member`	`actor Member as M` (the alias comes last)

The two languages are close enough that you can move between them in an afternoon, with one trap in the table: the order of name and alias in `actor` is reversed. In both tools the arrow `-->` in a class diagram is an association you can navigate in one direction (chapter 5 covers all the arrow types).

A Render Script

Typing `mmdc` and `plantuml` commands for every file is how diagrams go stale. This small script renders every source under a folder, skips the ones whose picture is already newer, and fails loudly (exit status 1) if a diagram has an error. It names each output after its source (`x.mmd.svg`, `x.puml.svg`), so a Mermaid and a PlantUML version of one diagram can sit side by side.

```
"""Render every diagram source under a folder to SVG, next to its source.

    py render_diagrams.py docs/diagrams

*.mmd  -> <name>.mmd.svg   Mermaid (mmdc, from @mermaid-js/mermaid-cli)
*.puml -> <name>.puml.svg  PlantUML (plantuml executable or `java -jar plantuml.jar`)
The output name keeps the source's extension, so a Mermaid and a PlantUML version of one
Only sources newer than their SVG are rendered, so it is quick to run before every commi
"""
import pathlib
import subprocess
import sys

MMDC = r"C:\lwdj\ooad\node_modules\.bin\mmdc.cmd"            # adjust: where you install
PUML = [r"C:\lwdj\ooad\puml\bin\plantuml.exe"]               # or ["java", "-jar", "plan
MMDC_ARGS = ["-p", r"C:\lwdj\ooad\ch1\cfg.json", "-c", r"C:\lwdj\ooad\ch1\mm.json", "-b"


def stale(src: pathlib.Path, out: pathlib.Path) -> bool:
    return not out.exists() or out.stat().st_mtime < src.stat().st_mtime


def main(folder: str) -> int:
    failures = 0
    for src in sorted(pathlib.Path(folder).rglob("*")):
        out = src.with_name(src.name + ".svg")
        if src.suffix == ".mmd":
            cmd = [MMDC, "-i", str(src), "-o", str(out), "--svgId", "d-" + src.stem.repl
        elif src.suffix == ".puml":
            cmd = [*PUML, "-tsvg", "-pipe"]                  # read the source on stdi
        else:
            continue
        if not stale(src, out):
            print(f"up to date  {src}")
            continue
        if src.suffix == ".puml":
```

```
            result = subprocess.run(cmd, input=src.read_bytes(), capture_output=True)
            # PlantUML reports a syntax error by drawing it into the image (and, with -p
            ok = result.returncode == 0 and result.stdout.lstrip().startswith((b"<?xml",
            if ok:
                out.write_bytes(result.stdout)
            result.stderr = "" if ok else (result.stderr.decode(errors="replace") or "Pl
        else:
            result = subprocess.run(cmd, capture_output=True, text=True)
        if result.returncode or not out.exists():
            failures += 1
            print(f"FAILED      {src}\n{(result.stderr or result.stdout).strip()[:300]}"
        else:
            print(f"rendered    {src}  ->  {out.name}  ({out.stat().st_size // 1024} KB)
    return 1 if failures else 0


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

The real output of running it twice on the folder holding the four diagram sources above, and then on two deliberately broken files:

```
rendered    docs\diagrams\borrow.sequence.mmd  ->  borrow.sequence.mmd.svg  (25 KB)
rendered    docs\diagrams\borrow.sequence.puml  ->  borrow.sequence.puml.svg  (4 KB)
rendered    docs\diagrams\loans.class.mmd  ->  loans.class.mmd.svg  (24 KB)
rendered    docs\diagrams\loans.class.puml  ->  loans.class.puml.svg  (7 KB)
```

```
up to date  docs\diagrams\borrow.sequence.mmd
up to date  docs\diagrams\borrow.sequence.puml
up to date  docs\diagrams\loans.class.mmd
up to date  docs\diagrams\loans.class.puml
```

```
FAILED      broken\bad.mmd
Error: Parse error on line 2:
...iagram  class A {
--------------------^
Expecting 'STRUCT_STOP', 'MEMBER', got 'EOF_IN_STRUCT'
Parser.parseError (https://mermaid-cli-intercept.invalid/C:/lwdj/ooad/node_modules/merma
Parser.parse (https:
FAILED      broken\bad.puml
```

```
ERROR

1

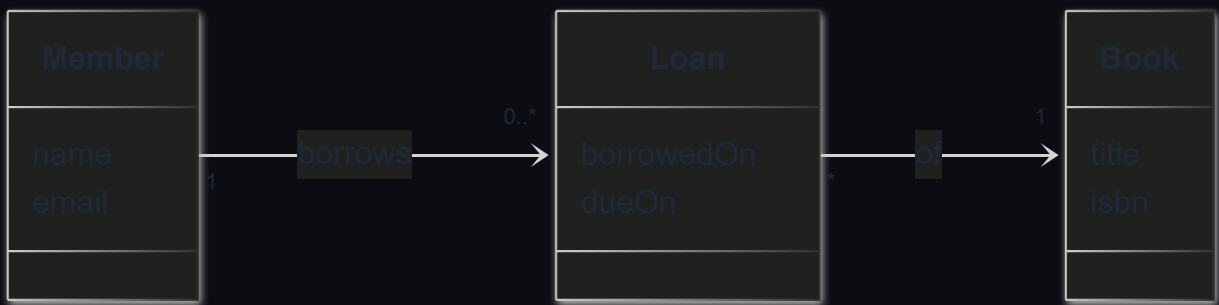
Syntax Error? (Assumed diagram type: sequence)
```

Note the last lines: both broken diagrams were reported as failures. The PlantUML one is the interesting one. PlantUML does not refuse to produce a picture for bad input; it **draws the error into the picture** (here "Syntax Error? (Assumed diagram type: sequence)") and, with `-pipe`, exits with status 200. The script treats both the exit status and the words "Syntax Error" in the output as a failure. A script that only checked for "a file appeared" would have published the error as if it were your design.

Themes and the Dark Page

UML is conventionally drawn dark on light. This course's pages are dark, so light diagrams sit on a white panel (the `.diagram` class in the course stylesheet). The alternative is to draw the diagram dark. Mermaid takes a theme in an `init` comment on the first line; PlantUML has built-in themes, selected with `!theme`. Both worked offline here:

```
%%{init: {'theme': 'dark', 'fontFamily': 'Arial, sans-serif'}}%%
classDiagram
    direction LR
    class Member {
        name
        email
    }
    class Loan {
        borrowedOn
        dueOn
    }
    class Book {
        title
        isbn
    }
    Member "1" --> "0..*" Loan : borrows
    Loan "*" --> "1" Book : of
```

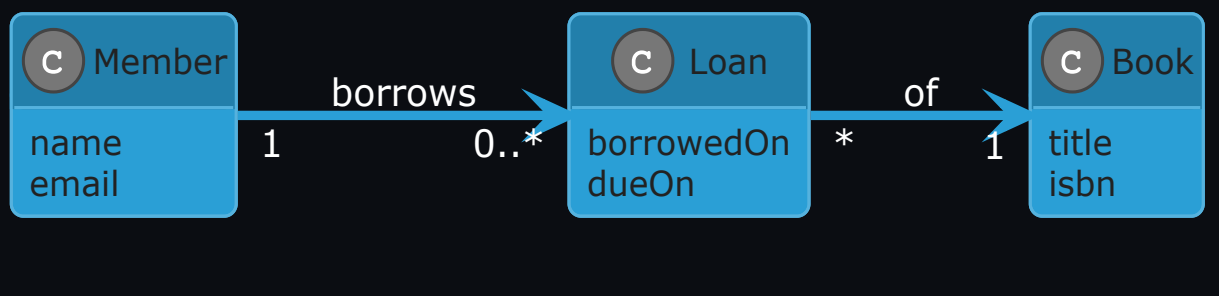


Mermaid, `theme: dark`

```
@startuml
scale 1.5
!theme cyborg
hide empty methods
left to right direction

class Member {
  name
  email
}
class Loan {
  borrowedOn
  dueOn
}
class Book {
  title
  isbn
}

Member "1" --> "0..*" Loan : borrows
Loan "*" --> "1" Book : of
@enduml
```



PlantUML, `!theme cyborg`. The theme names come from PlantUML's bundled list; I tried one, not all of them.

A caution about this convenience: the panel colour must match the diagram. A dark diagram placed on a white panel, or a light one on a dark page without its panel, is the commonest readability bug (chapter 1's class diagram had pale text for exactly that reason). Whichever you choose, look at the diagram on the real page.

Where the Sources Live

Keep the sources next to the code they describe, in the same repository, so a change to the code and a change to the diagram can be one pull request:

```
library/
├── src/
│   └── library/
├── docs/
│   └── diagrams/
│       ├── loans.class.mmd
│       ├── borrow.sequence.mmd
│       ├── components.puml
│       └── README.md          # how to render: one command
└── render_diagrams.py
```

- **Name by subject, then type, then tool** (`loans.class.mmd`): you find the diagram by what it is about.
- **One idea per diagram.** A diagram that needs scrolling has stopped being a sketch.
- **Mermaid inside Markdown renders on GitHub** with no build step, which is the cheapest way to keep a diagram where people already read. Putting a `mermaid` fence in your README works for the structure and interaction diagrams the course draws in Mermaid. PlantUML needs the render step above.
- **Mind what you send out.** PlantUML has a public web server that renders text you send it. A design for a real system may be confidential; render locally, as here.

The Course's Choice

DIAGRAM	TOOL IN THIS COURSE	WHY
Class (chapters 4, 5, 6, 11)	Mermaid	Renders on GitHub; the output is clean; enough notation for what we draw
Sequence (7)	Mermaid	Readable output, simple syntax
State (8)	Mermaid	Supported and clean; chapter 8 will say if PlantUML is needed for composite states or history
Use case (3)	PlantUML	Mermaid's is beta; PlantUML's is mature
Activity (8)	PlantUML	No Mermaid equivalent (a flowchart is the nearest)
Component, deployment, object (9)	PlantUML	Not Mermaid UML types

That is a decision with a cost: you learn two syntaxes. I judge it worth it, because the alternative is to leave out half the diagram types or draw them badly. If you prefer to use one tool, PlantUML alone covers everything in this course; Mermaid alone does not.

MISTAKES WHILE PREPARING THIS CHAPTER

- **My course outline was out of date.** It said the latest PlantUML was V1.2026.6 and that PlantUML needs Java, "not installed on this machine". Checking the releases page showed 1.2026.8 (2026-09-05), with a native Windows build that needs no Java. What went wrong: I carried a version and a prerequisite from notes written earlier. How to avoid it: re-check versions and requirements on the day you use them.
- **My render script overwrote its own output.** The first version named every picture after its source's stem, so `loans.class.mmd` and `loans.class.puml` both wrote `loans.class.svg`; the second silently replaced the first, and the "up to date" check then believed the wrong file. Fixed by keeping the source extension in the output name. Takeaway: when two inputs can map to one output name, the tool will lose one without telling you; test with the awkward case on purpose.
- **I tested error handling with input that was not an error.** My first "broken" PlantUML file was `A ->` with no target, which PlantUML accepts as a sequence message, so the script correctly reported success. I then assumed PlantUML exits 0 on syntax errors; with a really broken file it exits 200 and draws the error. Takeaway: before you write a check, produce a failure you have verified is a failure, and read the exit status instead of guessing it.
- **A failing probe proves less than it seems.** Mermaid rejected my guessed keywords `activity` and `componentDiagram`. That shows those words do not exist, not that Mermaid cannot draw such

things. I limited the claim in the comparison to what I had tested and what Mermaid's syntax reference lists.

Hands-On Exercises

EXERCISE 1

In Mermaid, draw the lifecycle of a library loan as a state diagram: it starts when the book is borrowed, can become overdue, and ends when it is returned (an overdue loan returns with a fine due). Render it with the script.

View solution

EXERCISE 2

In PlantUML, draw a use case diagram for the library: a Member can borrow, return and pay a fine; a Librarian can borrow and return on a member's behalf. Draw the system boundary. Render it with the script.

View solution

EXERCISE 3

Set up a diagrams folder with one Mermaid and one PlantUML source, and use the render script to show four things: the first render, a no-change run that skips both, a run after editing one file that re-renders only that one, and a run with a broken file that fails with exit status 1.

View solution

CHAPTER 2 QUICK REFERENCE

- Diagrams as text: reviewable diffs, history, merges, regenerated by command; review the source, not the picture (the SVG diff was 54,559 bytes against a 483-byte source diff)
- Mermaid 12.1.0 + mermaid-cli 12.0.0: `mmdc -i x.mmd -o x.svg -p cfg.json -c mm.json --svgId unique`; renders natively in GitHub Markdown; class, sequence, state; use case only as `usecase-beta`

- PlantUML 1.2026.8: `@startuml ... @enduml`; `plantuml -tsvg -pipe`; native build needs no Java; Graphviz for some layouts (untested without); covers use case, activity, component, deployment, object
- PlantUML draws syntax errors into the picture (exit status 200 with `-pipe`): check for it in scripts
- Keep sources beside the code, named subject.type.tool; one idea per diagram; render locally for confidential designs
- Dark page: light diagrams on a white panel, or draw dark (`theme: dark`, `!theme cyborg`); look at the real page
- Course choice: Mermaid for class, sequence, state; PlantUML for use case, activity, component, deployment, object

COMING NEXT

Object-Oriented Design with UML 3 (From Requirements to Use Cases): actors, goals and scope, the use case diagram and its limits, writing a use case description with a main flow and alternative flows, and finding the use cases of the library system.

Object-Oriented Design with UML

Course 1 · Chapter 3 · From Requirements to Use Cases

Analysis starts with a question that has nothing to do with classes: **who wants what from this system, and what has to happen for them to get it?** A **use case** answers it for one goal, as a short story with a happy path and the ways it can go wrong. Use cases come before the domain model (chapter 4) because the concepts you need are the ones the stories mention, and they come before any class design because you cannot decide who is responsible for something nobody has described.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> The two use case diagrams were written in PlantUML 1.2026.8 and rendered here. The "Borrow a Book" use case description is turned into Python (3.14) with one test per flow and the output pasted; a second run shows a deliberate bug being caught. **The library is invented**: the 21-day loan, the limit of five and the rule that fines block borrowing are my assumptions, listed as open questions for a real librarian. Three things come from general knowledge and were not re-read for this chapter: that use cases were popularised by Ivar Jacobson; the goal levels and the step-numbering convention for alternative flows, which I attribute to Alistair Cockburn's *Writing Effective Use Cases*; and the exact UML meaning of `include` and `extend`.

The Words

TERM	MEANING	LIBRARY EXAMPLE
Actor	Someone or something *outside* the system that interacts with it, in a role (a role, not a job title or a person)	Member, Librarian, the card payment service
Primary actor	The actor whose goal the use case serves	Member, for "Reserve a book"
Supporting actor	An actor the system calls on to reach the goal	The card payment service, for "Pay a fine"
Goal	What the primary actor wants to achieve; the name of the use case	"Borrow a book"
Scope	The system being designed, treated as a black box; everything outside it is an actor	The library *lending system*, not the whole library

TERM	MEANING	LIBRARY EXAMPLE
Use case	A goal-directed set of interactions between actors and the system, including failures along the way	"Borrow a Book", with its flows
Scenario	One path through a use case	Borrow succeeds; borrow refused because of fines

Two points that prevent most confusion. **Roles, not people:** a person can play both Member and Librarian (a librarian borrowing a book); the use case cares about the role. **Scope decides who is an actor:** inside the lending system the librarian is an actor, because the system is the software. If your scope were "the library as an organisation", the librarian would be inside the boundary. Always state the scope.

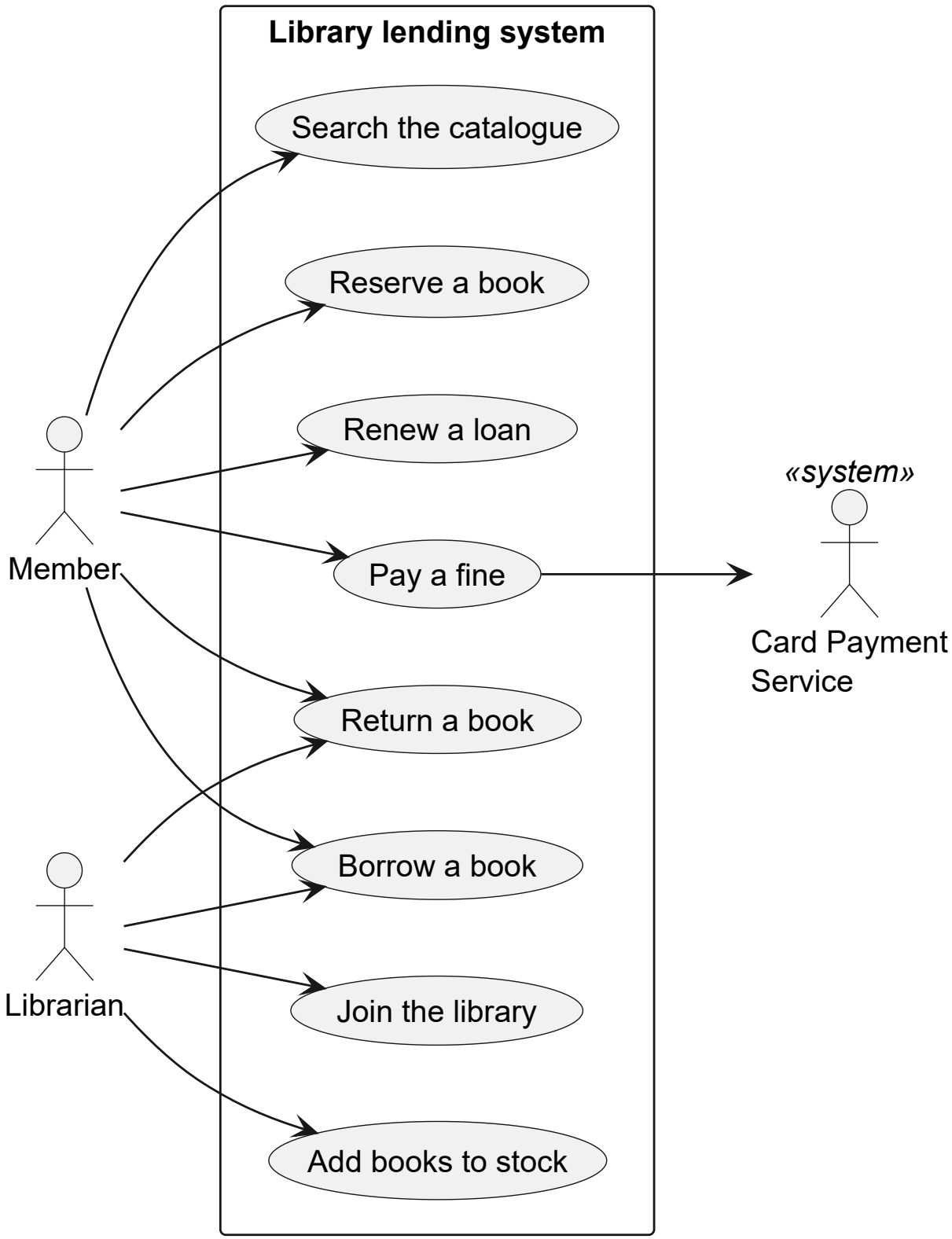
Finding Actors and Goals

A reliable method is to list actors first and then ask each one the same question: *what do you want to get done with this system?* Ask "what else?" until the answers dry up, then ask what happens on a schedule, with no person involved (a nightly job is an actor too). For the library:

ACTOR	GOALS
Member	Search the catalogue; borrow a book; return a book; reserve a book; renew a loan; pay a fine
Librarian	Borrow and return books at the desk for a member; join a new member; add books to stock
Card payment service (supporting)	Takes the payment for a fine
A nightly timer (exercise 2)	Send overdue notices

Goals have a **level**. Cockburn's names (from memory) are cloud, kite, sea level, fish and clam. The useful one is **sea level**, or "user goal": something a person does in one sitting that is worth doing for its own sake, such as "Borrow a book". Above it ("Run the library") is too big to describe; below it ("Enter the card number") is a step, not a goal. A quick test: after finishing it, would the actor consider the job done, and could they walk away?

The Use Case Diagram



PlantUML, rendered here. The rectangle is the scope; actors stand outside it; each line says "this actor takes part in this use case".

```
@startuml
scale 1.4
left to right direction

actor Member
actor Librarian
actor "Card Payment\nService" as Pay <<system>>

rectangle "Library lending system" {
  usecase "Search the catalogue" as Search
  usecase "Borrow a book" as Borrow
  usecase "Return a book" as Return
  usecase "Reserve a book" as Reserve
  usecase "Renew a loan" as Renew
  usecase "Pay a fine" as PayFine
  usecase "Join the library" as Join
  usecase "Add books to stock" as Stock
}

Member --> Search
Member --> Borrow
Member --> Return
Member --> Reserve
Member --> Renew
Member --> PayFine
Librarian --> Borrow
Librarian --> Return
Librarian --> Join
Librarian --> Stock
PayFine --> Pay
@enduml
```

That is the whole use case diagram, and it is useful for exactly one thing: **showing the scope and the cast on one page**, so a customer can say "you forgot that members can renew". Its limits are as important:

- **It says nothing about what happens.** "Borrow a book" is an ellipse; whether a member with fines can borrow, and what the librarian sees if not, is in the description below, not the picture.
- **It has no order and no flow.** Neither arrows nor position mean "first".

- **It does not scale.** Past a dozen ellipses it becomes wallpaper. Group by actor or by area into several diagrams.
- **The arrowheads are a drawing convention** (from memory of the UML notation, where the line between an actor and a use case is an association). Here the arrow runs from the primary actor to the use case, and from a use case to a supporting actor it needs.

Many teams keep only a *list* of use case names and put the effort into the descriptions. That is a fair choice.

The Use Case Description

The description is the real content. A common layout (Cockburn's, as I remember it) has these parts:

PART	WHAT GOES IN IT
Name, scope, level, primary actor	Name it as a verb phrase for the goal; say what the system is and who wants this
Stakeholders and interests	Everyone who cares about the outcome, even if they never touch the system, and what they want (this finds rules that the actors would not mention)
Precondition	What must already be true for the use case to start (not a check the use case performs)
Trigger	The event that starts it
Main success scenario	Numbered steps of the simplest path in which everything goes right
Extensions	What can go differently, attached to the step where it happens (3a, 3b...), each ending with either "use case ends" or "resume at step N"
Guarantees	*Success guarantee*: what is true afterwards if it worked. *Minimal guarantee*: what is true at least, however it ends

The library's "Borrow a Book", written out in full:

```
USE CASE: Borrow a Book
Scope:           Library lending system (black box)
Goal level:      User goal ("sea level")
Primary actor:   Librarian, acting for a Member at the desk
```

```
Stakeholders:     Member (wants the book and a clear due date)
                  Library (wants every loan recorded, and books returned)
Precondition:     The Librarian is signed in.
Trigger:          A Member hands over a book and their library card.
Success guarantee: A loan is recorded, the copy is marked on loan, and the due date is k
Minimal guarantee: If the loan is refused, nothing has changed, and the Librarian knows

MAIN SUCCESS SCENARIO
 1. The Member gives the Librarian the book and the card.
 2. The Librarian enters the card number and the book's ISBN.
 3. The system checks the membership is valid.
 4. The system checks the Member has no unpaid fines.
 5. The system checks the Member is under the loan limit (5).
 6. The system checks this copy can be lent to this Member.
 7. The system records the loan, due in 21 days.
 8. The system tells the Librarian the due date.
 9. The Librarian gives the book to the Member and tells them the date.

EXTENSIONS
 2a. The catalogue has no book with that ISBN:
     2a1. The system refuses, saying so. Use case ends.
 3a. The card is unknown, or the membership has expired:
     3a1. The system refuses, saying the membership is not valid. Use case ends.
 4a. The Member has unpaid fines:
     4a1. The system refuses and offers "Pay a fine". Use case ends (or continues from s
 5a. The Member already has 5 loans:
     5a1. The system refuses, saying the limit is reached. Use case ends.
 6a. The copy is already on loan:
     6a1. The system refuses and offers "Reserve a book". Use case ends.
 6b. The copy is being held for a different Member:
     6b1. The system refuses, saying it is held. Use case ends.
 6c. The copy is being held for THIS Member:
     6c1. The system continues at step 7, and the hold is fulfilled (cleared).

OPEN QUESTIONS FOR THE LIBRARIAN
 - Is 21 days right? Is 5 loans right? Do all books lend for the same time?
 - Should a Member with a small fine really be refused (4a)?
```

How It Reads

- **The steps are about the actor and the system, not the screen.** "The Librarian enters the card number and the ISBN" does not say whether that is a form, a scanner or a command line; deciding that is design.

- **Each check is its own step** (3 to 6), because each can fail differently, and each failure is an extension with the *same number*. This is where the value is: the main scenario is short and obvious, and the extensions list contains the decisions a developer would otherwise make silently.
- **Extensions say how the use case ends**, so no flow is left hanging. 4a even links to another use case (`Pay a fine`) and says whether to resume.
- **The minimal guarantee is a promise about failure**: a refused loan changes nothing. That sentence is cheap to write and expensive to forget, because a half-recorded loan is a bug nobody notices until the data is wrong.
- **The open questions are part of the document.** A use case written by someone who does not run the library will contain guesses; marking them is honest and tells you what to ask.

From Description to Check

A use case that cannot be turned into a test is probably vague. Here the description above became a Python function, with each numbered step and each extension as a comment in the code at the place it happens, and one test per flow. This is not the design you will end up with (who owns these checks is chapter 6's question); it is a way to make the description *execute*.

```
"""The 'Borrow a Book' use case as code. Each numbered step and each extension of the de
from dataclasses import dataclass, field
from datetime import date, timedelta

LOAN_DAYS = 21
MAX_LOANS = 5


class Refusal(Exception):
    """A use case extension that ends in failure. The message is what the Librarian read


@dataclass
class Member:
    card: str
    expires_on: date
    fines_pence: int = 0
    loans: list = field(default_factory=list)
```

```
@dataclass
class Copy:
    isbn: str
    on_loan: bool = False
    held_for: str | None = None          # card number of the member a reservation is he


@dataclass
class Loan:
    card: str
    isbn: str
    due_on: date


class Library:
    def __init__(self, members, copies):
        self.members = {m.card: m for m in members}
        self.copies = {c.isbn: c for c in copies}

    def borrow(self, card: str, isbn: str, today: date) -> Loan:
        member = self.members.get(card)
        # 2. The Librarian enters the card number and the book's ISBN.
        if isbn not in self.copies:
            raise Refusal("no such book in the catalogue")
        # 3. The system checks the membership is valid.
        if member is None or member.expires_on < today:
            raise Refusal("membership is not valid")
        # 4. The system checks the member has no unpaid fines.
        if member.fines_pence > 0:
            raise Refusal("unpaid fines: offer 'Pay a fine'")
        # 5. The system checks the member is under the loan limit.
        if len(member.loans) >= MAX_LOANS:
            raise Refusal("loan limit reached")
        # 6. The system checks the copy can be lent to this member.
        copy = self.copies[isbn]
        if copy.on_loan:
            raise Refusal("on loan: offer 'Reserve a book'")
        if copy.held_for not in (None, card):
            raise Refusal("held for another member")
        # 7. The system records the loan and the due date.        (6c: held for THIS mem
        loan = Loan(card, isbn, today + timedelta(days=LOAN_DAYS))
        copy.on_loan, copy.held_for = True, None
        member.loans.append(loan)
```

```
        # 8. The system tells the Librarian the due date.
        return loan
```

```
import copy as cp
import unittest
from datetime import date

from borrow_usecase import MAX_LOANS, Copy, Library, Loan, Member, Refusal

TODAY = date(2026, 10, 10)


def library(**overrides):
    member = Member(**{"card": "M1", "expires_on": date(2027, 1, 1), **overrides.get("me
    copies = [Copy("111", **overrides.get("c111", {})), Copy("222")]
    return Library([member], copies)


class MainFlow(unittest.TestCase):
    def test_success_postconditions(self):
        lib = library()
        loan = lib.borrow("M1", "111", TODAY)
        self.assertEqual(loan.due_on, date(2026, 10, 31))                # due date set
        self.assertTrue(lib.copies["111"].on_loan)                       # copy no long
        self.assertEqual(lib.members["M1"].loans, [loan])                # loan recorde


class Extensions(unittest.TestCase):
    def refused(self, lib, card="M1", isbn="111", expect=""):
        before = cp.deepcopy((lib.members, lib.copies))
        with self.assertRaises(Refusal) as ctx:
            lib.borrow(card, isbn, TODAY)
        self.assertIn(expect, str(ctx.exception))
        self.assertEqual((lib.members, lib.copies), before)              # minimal guar

    def test_2a_unknown_book(self):
        self.refused(library(), isbn="999", expect="catalogue")

    def test_3a_unknown_card(self):
        self.refused(library(), card="NOPE", expect="membership")

    def test_3a_expired_membership(self):
        self.refused(library(member={"expires_on": date(2026, 10, 9)}), expect="membersh
```

```
    def test_4a_unpaid_fines(self):
        self.refused(library(member={"fines_pence": 50}), expect="fines")

    def test_5a_loan_limit(self):
        loans = [Loan("M1", str(i), TODAY) for i in range(MAX_LOANS)]
        self.refused(library(member={"loans": loans}), expect="limit")

    def test_6a_on_loan(self):
        self.refused(library(c111={"on_loan": True}), expect="Reserve")

    def test_6b_held_for_someone_else(self):
        self.refused(library(c111={"held_for": "M2"}), expect="another member")

    def test_6c_held_for_this_member_succeeds_and_clears_the_hold(self):
        lib = library(c111={"held_for": "M1"})
        lib.borrow("M1", "111", TODAY)
        self.assertIsNone(lib.copies["111"].held_for)

    def test_membership_expiring_today_is_still_valid(self):
        lib = library(member={"expires_on": TODAY})
        self.assertEqual(lib.borrow("M1", "111", TODAY).isbn, "111")


if __name__ == "__main__":
    unittest.main(verbosity=2)
```

```
test_2a_unknown_book (test_borrow_usecase.Extensions.test_2a_unknown_book) ... ok
test_3a_expired_membership (test_borrow_usecase.Extensions.test_3a_expired_membership) .
test_3a_unknown_card (test_borrow_usecase.Extensions.test_3a_unknown_card) ... ok
test_4a_unpaid_fines (test_borrow_usecase.Extensions.test_4a_unpaid_fines) ... ok
test_5a_loan_limit (test_borrow_usecase.Extensions.test_5a_loan_limit) ... ok
test_6a_on_loan (test_borrow_usecase.Extensions.test_6a_on_loan) ... ok
test_6b_held_for_someone_else (test_borrow_usecase.Extensions.test_6b_held_for_someone_e
test_6c_held_for_this_member_succeeds_and_clears_the_hold (test_borrow_usecase.Extension
test_membership_expiring_today_is_still_valid (test_borrow_usecase.Extensions.test_membe
test_success_postconditions (test_borrow_usecase.MainFlow.test_success_postconditions) .
Ran 10 tests in 0.001s
OK
```

Two observations from doing this. **First, writing the code found a missing extension.** My first version of the description had no extension 2a: the code looked the ISBN up in the catalogue, and a book that was not there raised a Python `KeyError`, not a refusal the Librarian could read:

```
--- before extension 2a was written
KeyError: '999-not-in-catalogue'   <- not a Refusal: the description never said what hap
--- after
Refusal: no such book in the catalogue   <- the Librarian can read this out
```

The description was incomplete; the code was merely the first thing to ask the awkward question. I added extension 2a, the test for it, and ran again. **Second, I tested the tests.** The minimal guarantee ("a refusal changes nothing") is checked by comparing a deep copy of the data before and after every refusal. To see whether that test could fail, I ran the suite against a version of `borrow` with a deliberate bug that changes the copy's state before the checks that may refuse:

```
FAIL: test_3a_expired_membership (test_borrow_usecase.Extensions.test_3a_expired_membersh
FAIL: test_3a_unknown_card (test_borrow_usecase.Extensions.test_3a_unknown_card)
FAIL: test_4a_unpaid_fines (test_borrow_usecase.Extensions.test_4a_unpaid_fines)
FAIL: test_5a_loan_limit (test_borrow_usecase.Extensions.test_5a_loan_limit)
FAIL: test_6a_on_loan (test_borrow_usecase.Extensions.test_6a_on_loan)
FAIL: test_6b_held_for_someone_else (test_borrow_usecase.Extensions.test_6b_held_for_som
mutant run: tests failed: 6 of 10
```

Six tests failed, so the guarantee is really being checked. (Test code that never fails on broken code proves nothing; *Build a Chatbot: From API Call to Deployed Bot* 11 made the same point with simulations.) Note what the tests do *not* establish: that 21 days and a limit of five are right. Only the librarian can say.

Use Cases and User Stories

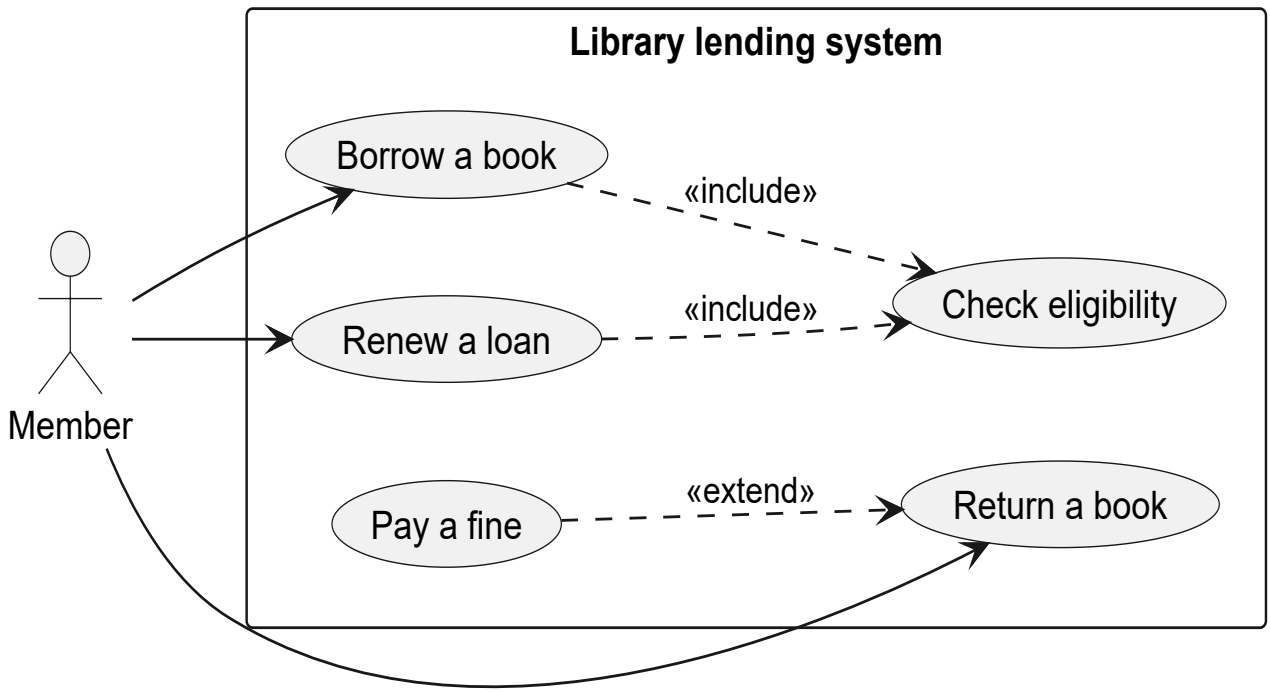
The Software Development Lifecycle 5 covers requirements and user stories; this section only relates the two. A story such as *"As a member, I want to borrow a book so that I can read it at home"* is a short reminder of a need, meant to start a conversation. A use case is what that conversation produces when the goal has real flows: the steps, the checks and the failures. They are not rivals.

	USER STORY	USE CASE
Size	One or two sentences	One to two pages
Strength	Planning, prioritising, small slices of value	Complete flows, rules and failures for one goal
Weakness	Detail lives in conversations and tests that can be lost	Heavier to write and to keep current
Use when	The goal is simple or the team talks daily	A goal has many rules and failure modes, or the people deciding are not in the room

A practical way to use both: let the use case supply the stories. "Borrow a book" gives one story for the main scenario and one for each extension that is worth building separately ("As a member with fines, I want to be told why I was refused and how to fix it"); the extensions double as acceptance criteria.

Include and Extend, and Why to Be Sparing

Use cases can relate to each other in two ways (from memory of the UML standard and its usual explanations). **Include**: the base use case *always* performs the included one, as a step; use it to share a piece of behaviour that several use cases need. **Extend**: the extending use case *optionally* adds behaviour to the base at a named point, under a condition; the arrow runs from the extension to the base. A small example, rendered:



Borrowing and renewing both include "Check eligibility". "Pay a fine" can extend "Return a book" when the return is late.

```
@startuml
scale 1.4
left to right direction

actor Member

rectangle "Library lending system" {
  usecase "Borrow a book" as Borrow
  usecase "Renew a loan" as Renew
  usecase "Check eligibility" as Check
  usecase "Return a book" as Return
  usecase "Pay a fine" as PayFine
}

Member --> Borrow
Member --> Renew
Member --> Return
Borrow ..> Check : <<include>>
Renew ..> Check : <<include>>
```

```
  PayFine ..> Return : <<extend>>
  @enduml
```

Read the arrows: an `include` arrow points *to the included* use case; an `extend` arrow points *to the base*. People draw them backwards constantly. And a warning: **the temptation is to overuse both**. Splitting use cases into include-pieces turns the description into a functional decomposition (a flowchart in disguise), and the readers lose the story. Reach for them only when the sharing is real (the same several steps in several use cases) or the optional behaviour genuinely belongs elsewhere. Writing the extension inline ("4a. The Member has unpaid fines...") is usually clearer. In the Borrow description above I did exactly that, and referred to *Pay a fine* by name instead of drawing an extend relationship.

Common Mistakes

MISTAKE	EXAMPLE	BETTER
A use case that is a screen or a data operation	"Manage books" (create, read, update, delete)	Name the goals: "Add books to stock", "Withdraw a damaged book"
A step dressed as a use case	"Enter card number"	Make it a step inside "Borrow a book"
The user interface in the flow	"The Librarian clicks the green Borrow button"	"The Librarian asks to borrow the book" (the interface is a later decision)
The system as an actor, or an actor inside the system	"The Database" as an actor	Inside the scope it is design. An actor is outside it: a payment service, a timer
Only the happy path	A one-page description with no extensions	Walk each step and ask "what if this fails?"; write it down
Too many levels at once	A diagram mixing "Run the library" and "Scan a barcode"	Keep to sea level; mention the others in prose
Include and extend everywhere	Twenty ellipses joined by dotted arrows	Inline the extensions; share only what is truly shared
Never taking it to the people who know	A design finished without asking the librarian	Read the description aloud to them; it is written in their language, which is its point

MISTAKES WHILE PREPARING THIS CHAPTER

- **I identified a physical book by its ISBN.** The description and the code say "the book's ISBN" and look the copy up by it. That conflates two things: an ISBN identifies a *title*, and a library may own several physical copies of one title. Chapter 4's domain model exposes it (its demonstration shows two copies of one ISBN collapsing to one in this chapter's code, so a second member is refused a copy that is on the shelf). The fix, not made here, is to say "copy barcode" in the steps and add a Title/Copy distinction. Takeaway: a use case written before the domain model is a draft of its vocabulary; expect the model to correct it.
- **A broken test helper.** My first run of the use case tests failed two tests with `TypeError: Member.__init__() got multiple values for argument 'expires_on'`. What went wrong: the test helper passed `expires_on` by position and then again through the overrides. How to avoid it: build test fixtures from a dictionary of defaults that overrides replace, never from a positional argument plus overrides.
- **An incomplete use case, found late.** The description had no extension for an unknown ISBN until the code raised `KeyError` (shown above). The takeaway is the practice, not the apology: turn each description into a runnable check as early as is cheap, because code is the strictest reader a requirement will ever have.
- **A demonstration that printed the wrong thing.** My first "gap" demo printed "not a Refusal" on every line, even after the fix, because the message was written as a constant instead of depending on the exception type. It said the opposite of what happened. Takeaway: make a demonstration's commentary depend on what it observes.
- **A command that did not do what I thought.** A `sed` command to remove a label from a diagram silently matched nothing (the file held a literal backslash-n, and my pattern assumed a newline), and the render script still said "rendered". I only noticed because I looked at the picture. Takeaway: after an automated edit, check the file or the picture, not just the exit status.

Hands-On Exercises

EXERCISE 1

Write the use case description for "Return a Book" in the full layout: scope, level, primary actor, stakeholders, precondition, trigger, a main success scenario, extensions (no open loan for the copy, an overdue book with a fine, another member waiting in the reservation queue, a damaged book), and the two guarantees. List any rule you had to assume.

View solution

EXERCISE 2

In PlantUML, draw a use case diagram for the library with Member, Librarian and a nightly timer as actors (the timer triggers "Send overdue notice"). Draw the scope boundary. Say whether the Member is a primary or supporting actor for the notice, and why.

View solution

EXERCISE 3

Turn your "Return a Book" description into Python with a test for the main scenario and each extension, and run it. Look for a case your description did not decide, and write down the question for the librarian.

View solution

CHAPTER 3 QUICK REFERENCE

- Actor = a role outside the system; scope = the system as a black box; goal = what the primary actor wants; state the scope
- Find use cases by asking each actor "what do you want to get done?" and "what else?"; include scheduled jobs; aim for sea-level goals
- The use case diagram shows scope and cast on one page and nothing more; keep it small; many teams use a list instead
- The description is the content: precondition, trigger, numbered main scenario, extensions numbered by the step they attach to, each ending or resuming; success and minimal guarantees
- One check per step, so each failure gets its own extension; keep the user interface out of the steps
- Make the description executable early: it finds missing extensions, and test the tests by breaking the code on purpose
- Story = reminder for a conversation; use case = the detail it produces; extensions double as acceptance criteria
- `include` arrow points to the included use case (always performed); `extend` arrow points to the base (optional, conditional); use sparingly, inline extensions first

COMING NEXT

Object-Oriented Design with UML 4 (The Domain Model): finding the concepts that the use cases mention (noun and verb analysis and its weaknesses), attributes, associations and multiplicity, a glossary, and the library's domain model as a class diagram.

Object-Oriented Design with UML

Course 1 · Chapter 4 · The Domain Model

Chapter 3 described what people want to do. This chapter describes **the world those wishes live in**: the things the library cares about, and how they relate. That picture is the **domain model**: a class diagram, but of *real-world concepts* (a member, a loan, a fine), not of software classes. It is the vocabulary the team will share, and it is the cheapest place to find a misunderstanding, because you can show it to a librarian.

WHAT WAS RUN, AND WHAT WAS NOT

Run on this machine: a word-frequency script over the chapter 3 use cases; a Python model of the library's concepts that produces *both* the Mermaid diagram (Mermaid 12.1.0, rendered) and the plain-English sentences used to review it; a small Python design with a rule the diagram cannot express; and a demonstration that chapter 3's code has a flaw this model exposes. From general knowledge, not re-read: that this kind of conceptual class diagram is called a "domain model" in Craig Larman's *Applying UML and Patterns* and a "conceptual" perspective in Martin Fowler's *UML Distilled*, and Larman's idea of a category list for finding concepts. The library is invented, as before.

Conceptual Model, Design Model

The same notation, drawn for two readers, answers two different questions:

	DOMAIN (CONCEPTUAL) MODEL	DESIGN MODEL
Describes	Things in the problem world	Classes in the software
Contains	Concepts, attributes, associations, multiplicity	Also operations, visibility, types, interfaces, helper classes
A "Member" class has...	name, card number, expiry date	Those, plus methods like `borrow()` and perhaps a database id
Has a `LoanService`, a `Database`, a `Controller`?	No: a librarian has never heard of them	Yes, probably
Can be reviewed by	A librarian	A developer

	DOMAIN (CONCEPTUAL) MODEL	DESIGN MODEL
Chapters	This one	5 to 7

The domain model is also not a database schema. It will often map onto one, but its job is understanding, so it ignores keys, indexes, join tables and anything the librarian would not recognise.

Finding the Concepts

Three sources, in order of reliability: **the people who know the domain** (ask what they call things, and what they confuse), **the documents you already have** (the use cases), and **the existing paper and forms** (a library's membership card and fine slip name real concepts). The classic technique is **noun and verb analysis**: underline the nouns in the requirements as candidate concepts and attributes, and the verbs as candidate relationships.

I ran a crude version over the two use case descriptions from chapter 3 (Borrow a Book and Return a Book): a script that counts the words, ignoring a hand-made list of common words and obvious verbs. There is no language-analysis library on this machine, so this is just counting; the point is the shape of the result:

```
"""A crude 'noun analysis': which words does the library's requirements text use most? C
import collections
import pathlib
import re

STOP = set("""a an the of to in on at by for with and or if is are be it its this that t
has have had will must which who whom what when where how all any each every one two thr
use case ends end step steps system checks check enters enter gives give tells tell reco
offers offer says said saying works out resume resumes main success scenario extensions
actor stakeholders hands hand over mark marks notes note mention adds add finds find clo
confirms confirm reports report puts put checks check read down up we what too still yet
instead because about first another other same such own whether""".split())

text = "\n".join(pathlib.Path(f).read_text(encoding="utf8") for f in ("borrow_a_book.use
words = re.findall(r"[A-Za-z][A-Za-z']+", text)
counts = collections.Counter(w.lower() for w in words if w.lower() not in STOP and len(w
```

```
for word, n in counts.most_common(45):
    print(f"{n:>3}  {word}")
```

```
 19  member
 16  librarian
 14  book
 12  loan
 11  copy
  6  fine
  5  library
  5  due
  5  date
  5  held
  4  wants
  4  recorded
  4  card
  4  fines
  4  available
  3  refused
  3  isbn
  3  membership
  3  open
  3  right
  3  back
  2  lending
  2  black
  2  box
  2  acting
  2  desk
  2  books
  2  signed
  2  minimal
  2  nothing
  2  changed
  2  book's
  2  valid
  2  unpaid
  2  limit
  2  days
  2  loans
  2  being
  2  hold
  2  shelf
```

```
  1  sea
  1  clear
  1  their
  1  marked
  1  known
```

The top of the list is encouraging: member, librarian, book, loan, copy, fine, card, membership. But it is a list of **candidates**, and the work is deciding what each word is:

WORD	DECISION	REASON
member	Concept	Has its own facts and relationships
librarian	Not in the model	An *actor*: a role of someone using the system (chapter 3). If the library also needs to know which librarian did what, a Staff concept may come back later
book and copy	Two concepts: **Title** and **Copy**	The text uses "book" for both an ISBN-identified work and a physical item (see below)
loan	Concept	A *transaction* with its own dates; it comes from the verb "borrows"
fine	Concept	It has its own amount, date and "paid" status, and arises from a particular loan
due, date, days	Attributes	"Due date" is a fact about a loan
card, isbn	Attributes	Identifiers of a Member and of a Title (`cardNumber`, `isbn`)
membership, valid	Attribute (`expiresOn`)	Whether it is valid is worked out from a date; it is not a thing
held, hold, shelf	A status of a Copy	The hold is a state, not a separate thing. The words are the librarian's, which is why they are in the glossary
available, open, unpaid	Statuses	Values of an attribute
limit, lending	Rules, not concepts	"At most 5 loans" is a business rule; it goes in the use case or a constraint
library, desk, box, sea	Not in the model	The library is the scope; the rest is noise (the word "box" came from "black box", "sea" from "sea level")
reservation	Concept	It appears only once in the text, so counting would have missed it; I knew it from the stakeholders and from chapter 3's other use cases

Where Noun Analysis Fails

- **It finds words, not meanings.** "Book" and "copy" are different concepts hiding behind overlapping words (a later section shows what it cost).
- **It misses what the text rarely says.** "Reservation" appears once. Counting would drop it; a stakeholder would not.
- **It cannot tell a concept from an attribute from a status.** Every row of the table above needed a human decision. A useful test: *if it only has a value (a number, a date, a piece of text), it is an attribute; if it has its own facts or relationships, it is a concept.*
- **Plurals and synonyms split the count.** "fine" (6) and "fines" (4), "loan" and "loans", "book" and "book's" are counted apart. A real analysis also has to merge "reservation" and "hold" or decide that they differ.
- **Actors and interface words look like concepts.** "Librarian" scored second; it does not belong.
- **Verbs hide concepts.** "A member *borrows* a copy" is where *Loan* came from: the relationship has facts of its own (when, until when, returned or not), so it became a thing. Look at verbs for this.

Larman's alternative (from memory) is a *category list*: prompt yourself with kinds of thing (physical objects, transactions, roles, places, records, events, catalogues) and ask whether the domain has one of each. Used beside noun analysis it finds what counting misses. Neither replaces asking.

Attributes, Associations and Multiplicity

- **Attributes** are the facts a concept holds: a Member has a `name`, `cardNumber`, `joinedOn`, `expiresOn`. In a domain model keep them to what is meaningful in the domain and leave out types and ids.
- **Associations** are meaningful relationships between concepts: "a Member borrows a Loan"; "a Loan is of a Copy". Name each one with a short verb phrase, so the diagram reads as sentences.
- **Multiplicity** says how many of one end may relate to one of the other: `1` exactly one, `0..1` optional, `0..*` any number, `1..*` at least one. Read it as the number of things at *that* end for each one at the other end.

The Library's Domain Model

Rather than draw the diagram by hand and then write the sentences by hand (and let them drift), I wrote the model once, as data, and a small script produces both. Associations are written as "each LEFT has *this many* RIGHT".

```
"""The library domain model as DATA. One source produces the diagram and the plain-Engli
import pathlib

CLASSES = {
    "Member": ["name", "cardNumber", "joinedOn", "expiresOn"],
    "Title": ["isbn", "title", "author"],
    "Copy": ["barcode", "status"],
    "Loan": ["borrowedOn", "dueOn", "returnedOn"],
    "Reservation": ["placedOn"],
    "Fine": ["amountPence", "issuedOn", "paid"],
}

# (left, left multiplicity, name, right multiplicity, right)   read as: each LEFT has <r
ASSOCIATIONS = [
    ("Member", "1", "borrows", "0..*", "Loan"),
    ("Loan", "0..*", "is of", "1", "Copy"),
    ("Copy", "0..*", "is a copy of", "1", "Title"),
    ("Member", "1", "places", "0..*", "Reservation"),
    ("Reservation", "0..*", "is for", "1", "Title"),
    ("Member", "1", "owes", "0..*", "Fine"),
    ("Fine", "0..1", "arose from", "1", "Loan"),
]

WORDS = {"0..*": "zero or more", "1": "exactly one", "0..1": "at most one", "1..*": "one

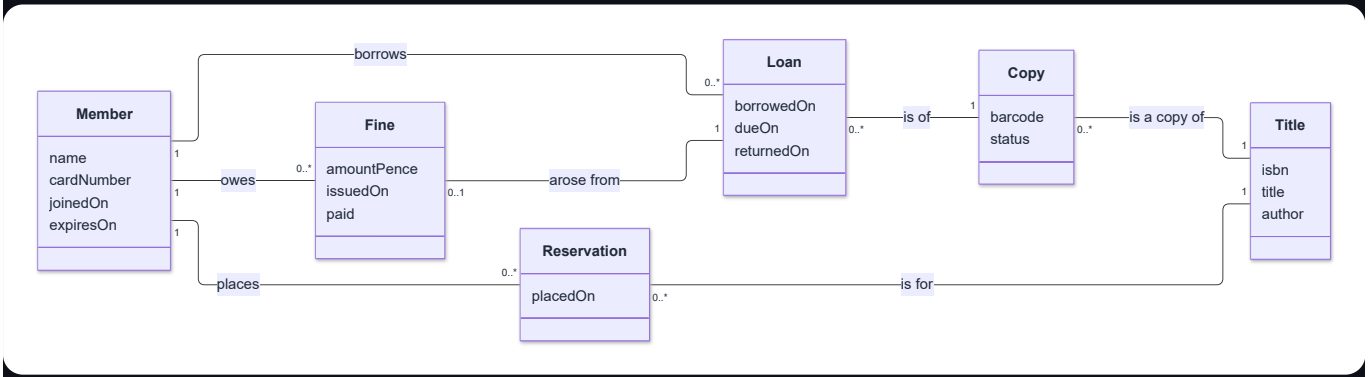

def mermaid(classes=None, associations=None) -> str:
    classes, associations = classes or CLASSES, associations or ASSOCIATIONS
    lines = ["classDiagram", "    direction LR"]
    for name, attrs in classes.items():
        lines.append(f"    class {name} {{")
        lines += [f"        {a}" for a in attrs]
        lines.append("    }")
    for left, lm, label, rm, right in associations:
        lines.append(f'    {left} "{lm}" -- "{rm}" {right} : {label}')
    return "\n".join(lines) + "\n"
```

```
def _noun(name: str, mult: str) -> str:
    """Pluralise for 'zero or more'. Done by hand-written rule, so it needs a test (see
    if mult not in ("0..*", "1..*"):
        return name
    return name[:-1] + "ies" if name.endswith("y") else name + "s"


def sentences(associations=None, reverse=None) -> list[str]:
    associations, reverse = associations or ASSOCIATIONS, reverse or _reverse
    out = []
    for left, lm, label, rm, right in associations:
        out.append(f"Each {left} {label} {WORDS[rm]} {_noun(right, rm)}.")
        out.append(f"   ...and each {right} {reverse(label)} {WORDS[lm]} {_noun(left, lm
    return out


def _reverse(label: str) -> str:
    return {"borrows": "is borrowed by", "is of": "is the subject of", "is a copy of": "
            "is for": "is wanted by", "owes": "is owed by", "arose from": "gave rise to"


if __name__ == "__main__":
    pathlib.Path("library.domain.mmd").write_text(mermaid(), encoding="utf8")
    print("\n".join(sentences()))
```



Mermaid, generated by `model.py`. Lines are plain associations: at the domain level, who can navigate to whom is a design decision, not yet made. The diagram is wide; the text is easier to read in the sentences below.

Reviewing It With a Non-Programmer

A librarian will not read UML, and does not need to. Read the model aloud as sentences; the script's output, verbatim:

```
Each Member borrows zero or more Loans.
   ...and each Loan is borrowed by exactly one Member.
Each Loan is of exactly one Copy.
   ...and each Copy is the subject of zero or more Loans.
Each Copy is a copy of exactly one Title.
   ...and each Title has as a copy zero or more Copies.
Each Member places zero or more Reservations.
   ...and each Reservation is placed by exactly one Member.
Each Reservation is for exactly one Title.
   ...and each Title is wanted by zero or more Reservations.
Each Member owes zero or more Fines.
   ...and each Fine is owed by exactly one Member.
Each Fine arose from exactly one Loan.
   ...and each Loan gave rise to at most one Fine.
```

Every sentence is a claim the librarian can accept or reject, and a few of these are not obviously right, which is the point. "*Each Fine arose from exactly one Loan*": what about a fine for a lost book, or for a late payment of something else? "*Each Reservation is for exactly one Title*": could a member ask for a specific edition? "*Each Member owes zero or more Fines*": is a fine ever paid in part? I recorded these as open questions rather than answering them myself. Showing sentences instead of boxes also hides nothing: a librarian reading "each Title has as a copy zero or more Copies" notices that a title with no copies is allowed, and can say whether a title nobody owns any more should stay on record.

A Glossary

A domain model is only useful if the whole team uses its words the same way, so write them down. One line per term, in the domain's own language, plus the words you decided *not* to use:

```
GLOSSARY: Library lending system                                  (draft 2026-10-10: to

Title        A published work, identified by its ISBN. "We have Dune." Not a physical ob
Copy         One physical item that can be lent, identified by its barcode. "We have thr
             (Never just "book": the word means Title in one sentence and Copy in the ne
Member       A person registered to borrow. Identified by card number. Has a membership
Loan         A Member having a Copy, from the day it is borrowed until it is returned. O
Reservation  A Member's request to be first in the queue for the next available Copy of
Fine         An amount a Member owes because a Loan came back late.
Hold         The state of a returned Copy kept aside for the Member who is first in a Ti
```

```
              (A status of a Copy, not a separate thing: see "Copy.status".)
  Librarian   A role played by library staff using the system. An actor in the use cases;
```

The glossary entry for **Copy** says "never just 'book'", and that rule came from a real mistake, below.

What the Model Found in Chapter 3

In chapter 3 the Borrow use case, and the Python written from it, identified the item being lent by **ISBN**. The domain model says an ISBN identifies a *Title*, and the library may own several copies of one title. Here is chapter 3's code with the physical copies of each title on the shelf and two members:

```
physical copies owned: 2   copies the chapter-3 code knows about: 1
second member, second copy on the shelf, but: on loan: offer 'Reserve a book'
```

The code's dictionary of copies is keyed by ISBN, so the two copies collapse into one, and the second member is told the book is on loan when a copy is sitting on the shelf. This is not a coding slip; it is a *vocabulary* error in the requirement, which the model's two separate boxes make obvious. Chapter 3's page now records it. The fix belongs in the use case ("the copy's barcode") and in the model, not in the dictionary.

What Multiplicity Cannot Say

The diagram says a Copy can have `0..*` Loans. That is true over time. What it cannot say is the rule that matters: **at most one of them may be open at once**. That is a *constraint*, and it belongs beside the diagram as a sentence (UML has a formal constraint language, OCL, which I do not teach here). Here is the model as a first plain Python design, with the rule enforced, and a run that shows both halves:

```
"""The domain model as a first, plain Python design. Names and associations come from th
from dataclasses import dataclass, field
from datetime import date
```

```
@dataclass
class Title:                                    # a published work: what the ISBN identif
    isbn: str
    title: str
    author: str


@dataclass
class Copy:                                     # one physical item on one shelf: what th
    barcode: str
    title: Title                                # Copy "0..*" -- "1" Title
    status: str = "available"


@dataclass
class Loan:
    copy: Copy                                  # Loan "0..*" -- "1" Copy
    borrowed_on: date
    due_on: date
    returned_on: date | None = None


@dataclass
class Member:
    name: str
    card_number: str
    loans: list[Loan] = field(default_factory=list)   # Member "1" -- "0..*" Loan

    def open_loans(self) -> list[Loan]:
        return [l for l in self.loans if l.returned_on is None]


# A rule the multiplicities cannot say: Copy "0..*" -- Loan allows many loans per copy (
def lend(member: Member, copy: Copy, today: date, due_on: date, all_members: list[Member
    for m in all_members:
        for l in m.open_loans():
            if l.copy is copy:
                raise ValueError(f"copy {copy.barcode} is already on loan to {m.name}")
    loan = Loan(copy, today, due_on)
    member.loans.append(loan)
    return loan
```

```
from datetime import date

from domain import Copy, Member, Title, lend

dune = Title("978-0-00-000000-0", "Dune", "F. Herbert")
copy1, copy2 = Copy("B-001", dune), Copy("B-002", dune)
ada, bo = Member("Ada", "M1"), Member("Bo", "M2")
members = [ada, bo]
today, due = date(2026, 10, 10), date(2026, 10, 31)

lend(ada, copy1, today, due, members)
lend(bo, copy2, today, due, members)
print("two members borrowed Dune at once, using two copies of one title:",
      [(m.name, l.copy.barcode, l.copy.title.title) for m in members for l in m.loans])
try:
    lend(bo, copy1, today, due, members)
except ValueError as e:
    print("the same copy twice:", e)
ada.loans[0].returned_on = today
lend(bo, copy1, today, due, members)
print("after Ada returns it, Bo can borrow that copy; copy1 has", sum(1 for m in members
```

```
two members borrowed Dune at once, using two copies of one title: [('Ada', 'B-001', 'Dun
the same copy twice: copy B-001 is already on loan to Ada
after Ada returns it, Bo can borrow that copy; copy1 has 2 loans in its history, 1 open
```

Two members borrow Dune at once with two copies of one title, which the old ISBN-keyed code could not do. The same copy cannot go out twice, and can go out again after it returns, leaving two loans in its history with one open. A domain model plus its constraints is more than the picture; keep a short list of such rules next to it.

MISTAKES WHILE PREPARING THIS CHAPTER

- **My own earlier chapter had the wrong vocabulary.** Chapter 3 treated an ISBN as the identity of a physical book. What went wrong: I used the word "book" for two concepts and never asked which. How to avoid it: before writing flows, write the glossary; and when a single noun is used in two senses ("we have the book"/"the book is on loan"), split it.
- **"Copys".** My generated sentences read "each Title has as a copy zero or more Copys", because my pluraliser just added an "s". It is a small thing in a document the librarian is meant to read aloud,

and an embarrassing one. I replaced it with a rule for words ending in "y". Takeaway: read generated text, and test hand-written pluralisation on the words your model really uses.

- **A word count mistaken for analysis.** My first instinct was to treat the top of the frequency list as the concept list. It contains an actor (librarian), two overlapping words (book, copy), statuses (held, available, open) and rules (limit), and misses "reservation". The table above is the real work. Takeaway: a tool that produces candidates is a prompt for judgment, not a replacement.
- **Mermaid text does not say who reads it.** The diagram has no arrowheads on purpose; at the domain level, navigation is undecided. Drawing arrows too early would have claimed a design decision (that a Copy cannot find its Loans) I had not made.

Hands-On Exercises

EXERCISE 1

Here is a short description of a bike repair shop (in the solution file). Run a word count over it, then classify every candidate as concept, attribute, status, rule, actor or noise, and say what the counting missed or got wrong.

View solution

EXERCISE 2

Write the bike shop's domain model as data (concepts with attributes, associations with multiplicities), generate the Mermaid diagram and the review sentences, render it, and name two sentences you would ask the shop owner about.

View solution

EXERCISE 3

The bike shop has a rule its multiplicities cannot express: a bike can have only one job in progress at a time. Write it as a constraint in words, enforce it in Python, and write tests showing that many jobs over time are fine, two at once are refused, and finishing a job frees the bike. Run them.

View solution

CHAPTER 4 QUICK REFERENCE

- Domain model = class diagram of real-world concepts: attributes, associations, multiplicity; no operations, helper classes or database detail
- Sources: domain experts, use cases, existing forms; noun analysis for concepts and attributes, verbs for relationships (and for hidden concepts like Loan); a category list as a second prompt
- Concept vs attribute: if it only has a value, it is an attribute; if it has its own facts or relationships, it is a concept
- Noun analysis is a source of candidates: it misses rarely mentioned things, confuses actors, statuses and rules with concepts, and cannot see that one word means two things
- Multiplicity: `1`, `0..1`, `0..*`, `1..*`; read as "for each one at the other end, how many at this end"
- Multiplicity cannot express "at most one open at a time": write such rules as constraints beside the diagram and enforce them in code
- Review with non-programmers by reading associations as sentences; keep a glossary, including the words not to use
- A domain model usually corrects the use cases written before it; expect to go back

COMING NEXT

Object-Oriented Design with UML 5 (Class Diagrams in Depth): the full notation (visibility, static, abstract, interfaces), association versus aggregation versus composition, dependency, generalisation and realisation, each with a runnable code example, navigability, association classes and the usual mistakes.

Object-Oriented Design with UML

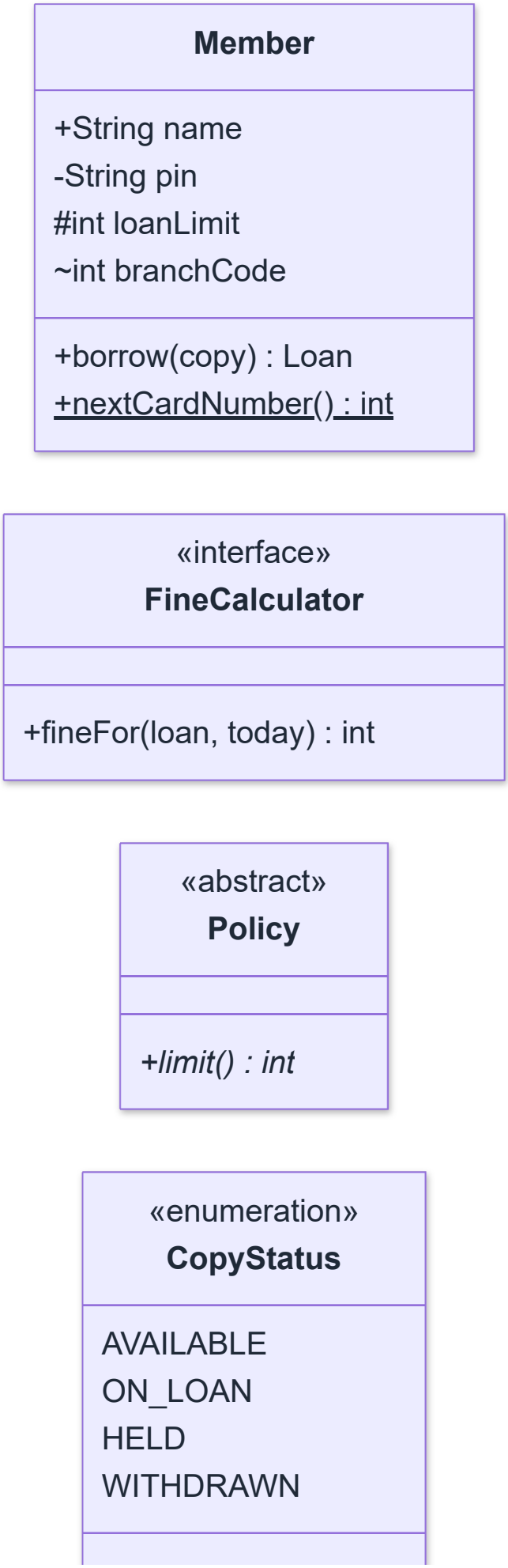
Course 1 · Chapter 5 · Class Diagrams in Depth

Chapter 4's domain model used the class diagram's simplest features: boxes, attributes, plain lines. A **design** class diagram uses the rest: visibility, operations, interfaces, and six kinds of arrow that each make a different promise about code. This chapter teaches the notation and, for each relationship, shows the Python it becomes and runs it, so that you can tell what a diagram is claiming and check that code keeps the claim.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> All diagrams were rendered here with Mermaid 12.1.0 (class boxes and relationships) or PlantUML 1.2026.8 (association classes, which Mermaid cannot draw: I ran the attempt and it failed with a lexical error). All Python (3.14) was run with output pasted. I checked the arrow shapes against the rendered pictures, so the descriptions below say what the tools actually draw. **From general knowledge, not re-read:** the exact wording of the UML specification on aggregation (that "shared aggregation" has little precise meaning), and Martin Fowler's advice to avoid it. The library is invented.

Anatomy of a Class Box

A class is a box with up to three compartments: **name**, **attributes**, **operations**. In a domain model you show only the first two; in a design you show what the reader needs, and no more. This diagram uses every piece of box notation this chapter covers:


Member
+String name
-String pin
#int loanLimit
~int branchCode
+borrow(copy) : Loan
+nextCardNumber() : int
«interface»
FineCalculator
+fineFor(loan, today) : int
«abstract»
Policy
+limit() : int
«enumeration»
CopyStatus
AVAILABLE
ON_LOAN
HELD
WITHDRAWN

Mermaid. Top to bottom: a class with all four visibility marks and a static operation; an interface; an abstract class with an abstract operation; an enumeration.

```
%%{init: {'themeVariables': {'fontSize': '20px'}}}%%
classDiagram
    direction LR
    class Member {
        +String name
        -String pin
        #int loanLimit
        ~int branchCode
        +borrow(copy) Loan
        +nextCardNumber()$ int
    }
    class FineCalculator {
        <<interface>>
        +fineFor(loan, today) int
    }
    class Policy {
        <<abstract>>
        +limit()* int
    }
    class CopyStatus {
        <<enumeration>>
        AVAILABLE
        ON_LOAN
        HELD
        WITHDRAWN
    }
```

NOTATION	MEANING	IN PYTHON
`+` public	Anyone may use it	An ordinary name
`-` private	Only this class	`__name` (name-mangled), or `_name` by convention
`#` protected	This class and its subclasses	`_name`, by convention only
`~` package	Same package or module	No direct equivalent: a leading underscore on a module-level name, or a convention

NOTATION	MEANING	IN PYTHON
Underlined (`$` in Mermaid)	Static: belongs to the class, not an instance	`@staticmethod`, `@classmethod`, a class attribute
Italic name or operation (`*`)	Abstract: cannot be instantiated; must be implemented by a subclass	`ABC` with `@abstractmethod`
`«interface»`	A contract: operations without an implementation	A `typing.Protocol` (structural) or an ABC (nominal)
`«enumeration»`	A fixed set of named values	`enum.Enum`

Python enforces much less than the notation promises. The run below (the first sections of the output) shows what the language actually does with each mark:

```
"""Class-diagram notation, one concept at a time, as Python. Each comment names the diagr
from abc import ABC, abstractmethod
from dataclasses import dataclass, field
from datetime import date
from enum import Enum
from itertools import count
from typing import Protocol, runtime_checkable


# --- enumeration: <<enumeration>> CopyStatus ----------------------------------------
class CopyStatus(Enum):
    AVAILABLE = "available"
    ON_LOAN = "on loan"
    HELD = "held"
    WITHDRAWN = "withdrawn"


# --- interface and realisation: <<interface>> FineCalculator <|.. DailyRateFine -------
@runtime_checkable
class FineCalculator(Protocol):
    def fine_for(self, due_on: date, returned_on: date) -> int: ...


class DailyRateFine:                                    # realises FineCalculator witho
    def __init__(self, pence_per_day: int = 10):
        self.pence_per_day = pence_per_day

    def fine_for(self, due_on: date, returned_on: date) -> int:
```

```
        return max(0, (returned_on - due_on).days) * self.pence_per_day


class NoFine:
    def fine_for(self, due_on: date, returned_on: date) -> int:
        return 0


# --- abstract class and generalisation: <<abstract>> Policy <|-- StandardPolicy, Junior
class Policy(ABC):
    @abstractmethod
    def limit(self) -> int: ...                          # +limit()* : the abstract oper


class StandardPolicy(Policy):
    def limit(self) -> int:
        return 5


class JuniorPolicy(Policy):
    def limit(self) -> int:
        return 3


# --- visibility and static: +name, -pin, #loanLimit, +nextCardNumber()$ ---------------
class Member:
    _cards = count(1)                                    # class-level state: the "stati

    def __init__(self, name: str, pin: str):
        self.name = name                                 # +  public
        self.__pin = pin                                 # -  private (name-mangled)
        self._loan_limit = 5                             # #  protected (a convention on
        self.loans: list["Loan"] = []                    # association Member 1 --> 0..*

    @staticmethod
    def next_card_number() -> int:                       # +nextCardNumber()$ : underlin
        return next(Member._cards)


@dataclass
class Loan:                                              # note: no reference back to Me
    copy: "Copy"
    borrowed_on: date
```

```python
# --- composition: Title *-- Copy (a Copy cannot exist without its Title, and goes when t
@dataclass
class Copy:
    barcode: str
    title: "Title"                                  # required: there is no Copy wi
    status: CopyStatus = CopyStatus.AVAILABLE


@dataclass
class Title:
    isbn: str
    name: str
    copies: list[Copy] = field(default_factory=list)

    def add_copy(self, barcode: str) -> Copy:       # the whole creates and owns its
        copy = Copy(barcode, self)
        self.copies.append(copy)
        return copy

    def withdraw(self) -> list[Copy]:               # the whole's end of life ends t
        gone, self.copies = self.copies, []
        for c in gone:
            c.status = CopyStatus.WITHDRAWN
        return gone


# --- aggregation: Shelf o-- Copy (copies are grouped by a shelf but live on without it)
@dataclass
class Shelf:
    label: str
    copies: list[Copy] = field(default_factory=list)


# --- dependency: LoanDesk ..> FineCalculator (used in a call, never stored) -----------
class LoanDesk:
    def close(self, due_on: date, returned_on: date, calc: FineCalculator) -> int:
        return calc.fine_for(due_on, returned_on)


# --- association class: (Member, Title) .. Reservation --------------------------------
@dataclass
class Reservation:
    member: Member
    title: Title
    placed_on: date
```

```
class ReservationBook:
    """One Reservation per (Member, Title) pair: that is what an association class means

    def __init__(self):
        self._by_pair: dict[tuple[int, str], Reservation] = {}

    def place(self, member: Member, title: Title, today: date) -> Reservation:
        key = (id(member), title.isbn)
        if key in self._by_pair:
            raise ValueError(f"{member.name} has already reserved {title.name}")
        self._by_pair[key] = Reservation(member, title, today)
        return self._by_pair[key]
```

```
from datetime import date

from library_classes import (Copy, CopyStatus, DailyRateFine, FineCalculator, JuniorPoli
                             ReservationBook, Shelf, StandardPolicy, Title)

print("== visibility: Python only has conventions")
ada = Member("Ada", "1234")
print("  +name:", ada.name)
print("  #_loan_limit readable from outside:", ada._loan_limit)
try:
    ada.__pin
except AttributeError as e:
    print("  -__pin from outside:", type(e).__name__, "(name-mangled)")
print("  ...but reachable as _Member__pin:", ada._Member__pin, " <- private by politenes

print("== static: +nextCardNumber()$")
print("  ", Member.next_card_number(), Member.next_card_number(), Member.next_card_numbe

print("== abstract: <<abstract>> Policy")
try:
    Policy()
except TypeError as e:
    print("  Policy():", e)
print("  generalisation: StandardPolicy.limit() =", StandardPolicy().limit(), " JuniorPo

print("== enumeration: CopyStatus")
try:
    CopyStatus("lost")
```

```
except ValueError as e:
    print("  CopyStatus('lost'):", e)

print("== realisation: DailyRateFine realises FineCalculator without inheriting")
print("  isinstance(DailyRateFine(), FineCalculator):", isinstance(DailyRateFine(), Fine
print("  isinstance(object(), FineCalculator):", isinstance(object(), FineCalculator))

print("== dependency: LoanDesk uses a calculator only for the length of one call")
desk = LoanDesk()
print("  fields stored on LoanDesk:", vars(desk), "| fine (3 days late, 10p/day):", desk
      "| with NoFine:", desk.close(date(2026, 10, 10), date(2026, 10, 13), NoFine()))

print("== association with navigability: Member --> Loan")
dune = Title("111", "Dune"); c1 = dune.add_copy("B-1")
loan = Loan(c1, date(2026, 10, 10)); ada.loans.append(loan)
print("  member -> loans: ", len(ada.loans), "loan(s) | loan -> member: has attribute 'm

print("== composition: Title *-- Copy")
try:
    Copy("B-9")
except TypeError as e:
    print("  Copy('B-9') alone:", e)
dune.add_copy("B-2")
gone = dune.withdraw()
print("  withdrawing the title:", [(c.barcode, c.status.value) for c in gone], "| title

print("== aggregation: Shelf o-- Copy")
t2 = Title("222", "Emma"); a, b = t2.add_copy("E-1"), t2.add_copy("E-2")
shelf = Shelf("F1", [a, b]); del shelf
print("  shelf deleted; copies still exist and are", [c.status.value for c in (a, b)])

print("== association class: (Member, Title) .. Reservation")
book = ReservationBook()
book.place(ada, t2, date(2026, 10, 10))
try:
    book.place(ada, t2, date(2026, 10, 11))
except ValueError as e:
    print("  second reservation of the same title by the same member:", e)
```

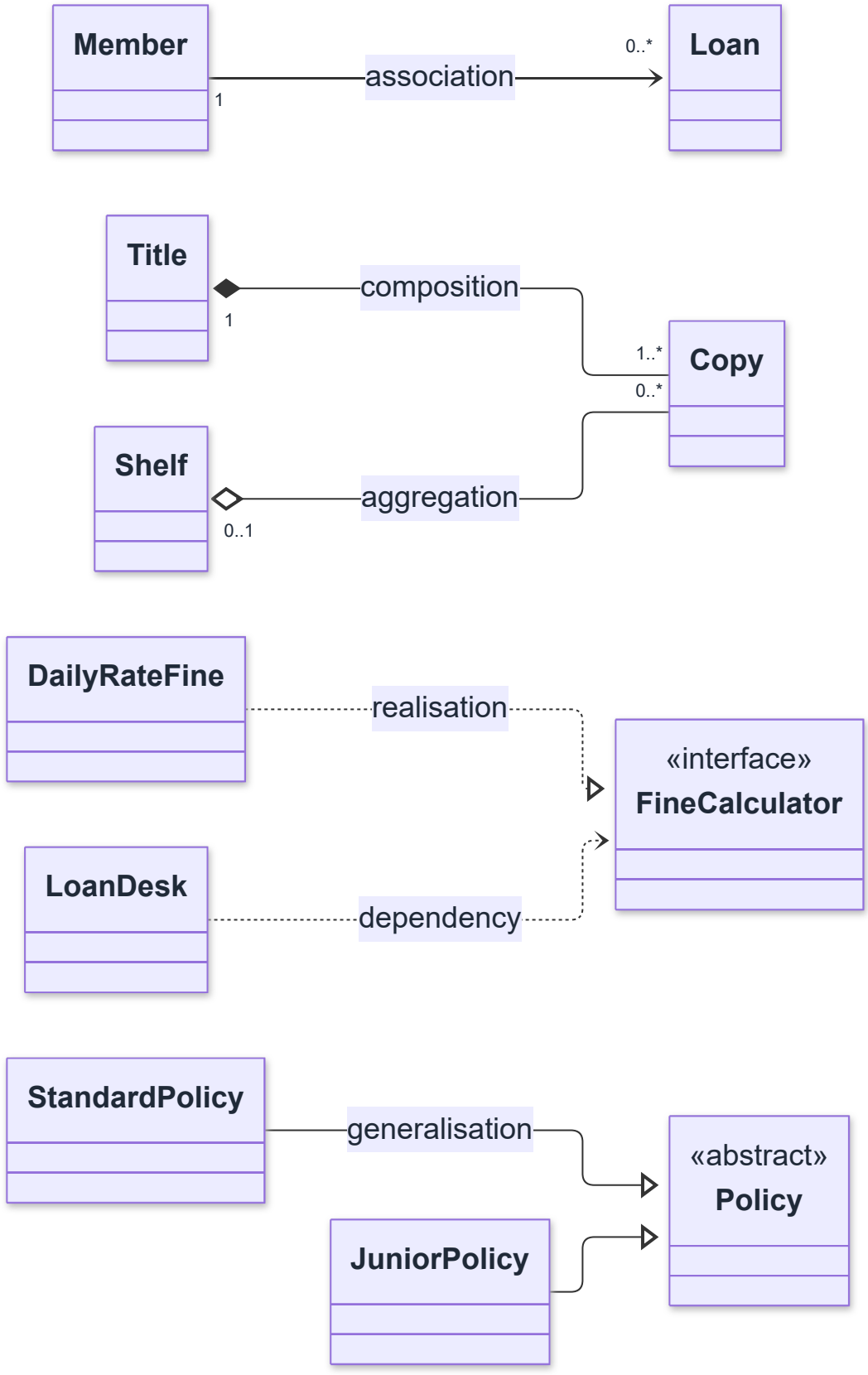
```
== visibility: Python only has conventions
  +name: Ada
  #_loan_limit readable from outside: 5
  -__pin from outside: AttributeError (name-mangled)
```

```
    ...but reachable as _Member__pin: 1234  <- private by politeness, not enforcement
  == static: +nextCardNumber()$
     1 2 3 (shared counter, no instance needed)
  == abstract: <<abstract>> Policy
    Policy(): Can't instantiate abstract class Policy without an implementation for abstra
    generalisation: StandardPolicy.limit() = 5  JuniorPolicy.limit() = 3
  == enumeration: CopyStatus
    CopyStatus('lost'): 'lost' is not a valid CopyStatus
  == realisation: DailyRateFine realises FineCalculator without inheriting
    isinstance(DailyRateFine(), FineCalculator): True | DailyRateFine.__mro__: ['DailyRate
    isinstance(object(), FineCalculator): False
  == dependency: LoanDesk uses a calculator only for the length of one call
    fields stored on LoanDesk: {} | fine (3 days late, 10p/day): 30 | with NoFine: 0
  == association with navigability: Member --> Loan
    member -> loans:  1 loan(s) | loan -> member: has attribute 'member'? False
  == composition: Title *-- Copy
    Copy('B-9') alone: Copy.__init__() missing 1 required positional argument: 'title'
    withdrawing the title: [('B-1', 'withdrawn'), ('B-2', 'withdrawn')] | title now has 0
  == aggregation: Shelf o-- Copy
    shelf deleted; copies still exist and are ['available', 'available']
  == association class: (Member, Title) .. Reservation
    second reservation of the same title by the same member: Ada has already reserved Emma
```

The visibility section is the one to read twice. `-pin` is "private" in the diagram and in the code (`__pin`), yet it is reachable as `_Member__pin`: Python's privacy is politeness. When you draw `-` you are stating the *design intent*; in Java or C# the compiler would enforce it, in Python a test or a reviewer must. The abstract class and the enumeration, by contrast, are enforced: `Policy()` and `CopyStatus("lost")` both fail.

The Six Relationships

Every line between classes is one of these. They run, roughly, from the weakest dependency to the strongest coupling. Here they are in the library, rendered:



Mermaid. Each label names the relationship drawn. Check the shapes: an open arrow, a filled diamond at the whole, a hollow diamond at the whole, a hollow triangle on a solid line, a hollow triangle on a dotted line, and a dotted line with an open arrow.

```
%%{init: {'themeVariables': {'fontSize': '18px'}}}%%
classDiagram
    direction LR
    class Member
    class Loan
    class Title
    class Copy
    class Shelf
    class Policy {
        <<abstract>>
    }
    class StandardPolicy
    class JuniorPolicy
    class FineCalculator {
        <<interface>>
    }
    class DailyRateFine
    class LoanDesk

    Member "1" --> "0..*" Loan : association
    Title "1" *-- "1..*" Copy : composition
    Shelf "0..1" o-- "0..*" Copy : aggregation
    StandardPolicy --|> Policy : generalisation
    JuniorPolicy --|> Policy
    DailyRateFine ..|> FineCalculator : realisation
    LoanDesk ..> FineCalculator : dependency
```

RELATIONSHIP	DRAWN AS	TEST QUESTION	IN CODE
Dependency	Dotted line, open arrow, to the thing used	Does it use the other only briefly: a parameter, a local variable, a call?	No field. `LoanDesk.close(..., calc)` takes the calculator as an argument and keeps nothing
Association	Solid line, usually an open arrow showing who can navigate to whom	Does it know about the other over time?	A field: `Member.loans`
Aggregation	Solid line, hollow diamond at the whole	"Has a", and the parts live on without the whole?	A field holding a collection; the parts have a life elsewhere (a `Shelf` of `Copy` s)
Composition	Solid line, filled diamond at the	If the whole dies, do the parts die with it, and can	The whole creates and owns the parts (`Title.add_copy`); the part cannot be

RELATIONSHIP	DRAWN AS	TEST QUESTION	IN CODE
	whole	a part belong to only one whole?	built without it
Generalisation	Solid line, hollow triangle at the parent	Can the child be used anywhere the parent is expected, and behave as promised?	Subclassing: `StandardPolicy(Policy)`
Realisation	Dotted line, hollow triangle at the interface	Does the class fulfil the contract of an interface?	Implementing a Protocol/ABC: `DailyRateFine` for `FineCalculator`

The same run's output for each relationship (read the lines under each heading in the output above):

- **Dependency.** `fields stored on LoanDesk: {}`. It computed a fine with `DailyRateFine` (30p for 3 days) and with `NoFine` (0) and remembered neither. Dependency is the cheapest coupling: change the calculator and nothing that merely passes it in is affected.
- **Association with navigability.** A member reaches its loans; `hasattr(loan, "member")` is `False`. See the next section.
- **Composition.** `Copy("B-9")` alone fails for want of its title, and withdrawing the title withdraws its copies. *Honesty about the demonstration:* Python garbage-collects, so "lifetime" here is bookkeeping I wrote (marking copies withdrawn), not a language guarantee.
- **Aggregation.** `del shelf` and the copies are still there. That is true of any plain reference, which is why the next point matters.
- **Generalisation and realisation.** Abstract `Policy` cannot be created; `StandardPolicy` (limit 5) and `JuniorPolicy` (limit 3) can. `DailyRateFine` satisfies `FineCalculator` with no inheritance at all: `isinstance` is true, and its inheritance chain is just `DailyRateFine` and `object`. That is realisation in Python: a Protocol matches by shape.

Aggregation: Usually Not Worth Drawing

Aggregation is the relationship people argue about, and the argument is the lesson. As I remember it, the UML specification says aggregation (the hollow diamond) carries no precise meaning beyond "association", and Fowler's *UML Distilled* advises not using it. The code agrees: a `Shelf` holding `Copy` references is indistinguishable from any association with a list. A

practical rule: **draw an association unless the filled diamond (composition) is genuinely true**. If you feel you must use the hollow diamond, say in a note what you mean by it.

Composition is meaningful, because it makes two promises: *exclusive ownership* (a part belongs to one whole) and *shared lifetime*. Use it when both are true. The library's "a Copy exists only as a copy of a Title" qualifies, if the library's policy is that withdrawing a title withdraws its copies; if copies could be re-catalogued under a different title, it would not.

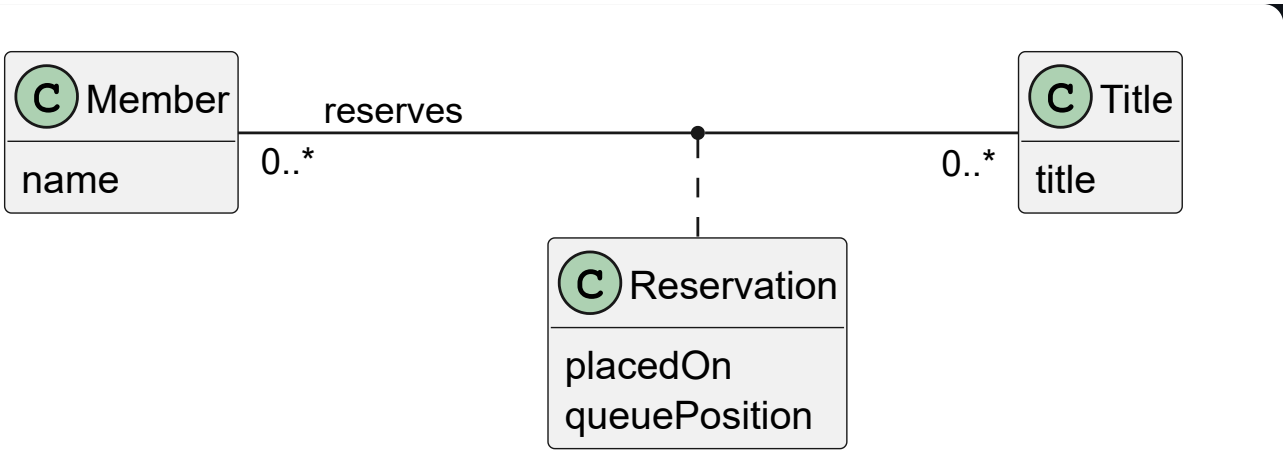
Multiplicity and Navigability

Multiplicity works as in chapter 4. **Navigability** is the arrowhead: it says which class holds the reference. In the code above `Member` has `loans`, but `Loan` has no `member`: the diagram's arrow from Member to Loan is exactly that, one way. A line with no arrowheads says "undecided" (chapter 4's model) or "both ways"; be sure which you mean. Navigability has real costs:

- **One way is simpler.** From a loan you cannot reach its member without searching every member. If you need that, add a reference, not a hope.
- **Two ways must be kept in step.** If `Member.loans` and `Loan.member` both exist, every place that creates or removes a loan must update both, and a mismatch is a bug that tests rarely catch. Decide who owns the link, and have only that side change it.

Association Classes

Sometimes a fact belongs to the *relationship*, not to either end: a reservation's date belongs to the pairing of a member and a title. The notation is a class attached by a dashed line to the middle of the association. Mermaid cannot draw it (I tried: `Lexical error on line 6. Unrecognized text.`), so this one is PlantUML:



PlantUML. `Reservation` holds the facts about one (Member, Title) link.

```
@startuml
scale 1.4
hide empty methods
left to right direction

class Member {
  name
}
class Title {
  title
}
class Reservation {
  placedOn
  queuePosition
}

Member "0..*" -- "0..*" Title : reserves
(Member, Title) .. Reservation
@enduml
```

The last two lines of the notation demo show the catch. An association class means **one instance per pair**: a member cannot have two reservations for the same title, and the second attempt fails. If your domain allows it (reserve again after a lapsed hold), the thing is an ordinary class with its own identity and two associations, not an association class. Choose the notation by the rule you mean.

Inheritance for Reuse: The Classic Mistake

Generalisation is the most overused arrow. The question is not "does a librarian have a name and a list of loans like a member?" (reuse) but "**can a librarian be used anywhere a member is expected and still behave as the member's code assumes?**" (substitutability). *Clean Code, SOLID & Refactoring* 6 and 7 cover the Liskov Substitution Principle properly; here is the failure it prevents, run:

```
"""Inheritance for reuse, and what it does to code that trusts the parent's promises. Th
from dataclasses import dataclass, field


@dataclass
class Member:
    name: str
    loans: list = field(default_factory=list)

    def loan_limit(self) -> int:
        return 5


def can_borrow(member: Member) -> bool:                  # written for Members, trusting
    return len(member.loans) < member.loan_limit()


# --- the shortcut: a Librarian "is a" Member, because it saves re-typing name and loans
class LibrarianIsAMember(Member):
    def loan_limit(self):
        return None                                      # "staff have no limit": the pa


print("1. Member: can_borrow ->", can_borrow(Member("Ada")))
try:
    print("2. Librarian-as-Member: can_borrow ->", can_borrow(LibrarianIsAMember("Bo")))
except TypeError as e:
    print("2. Librarian-as-Member:", type(e).__name__ + ":", e)


# --- the alternative: a Librarian HAS a membership, and does not pretend to be one ---
@dataclass
class Librarian:
    name: str
    membership: Member | None = None                     # staff may also borrow, as ord
```

```
bo = Librarian("Bo", membership=Member("Bo"))
print("3. Librarian with an ordinary membership: can_borrow ->", can_borrow(bo.membershi
print("   isinstance(librarian, Member):", isinstance(bo, Member), "(no lie told, so no
```

```
1. Member: can_borrow -> True
2. Librarian-as-Member: TypeError: '<' not supported between instances of 'int' and 'Non
3. Librarian with an ordinary membership: can_borrow -> True
   isinstance(librarian, Member): False (no lie told, so no code is surprised)
```

`can_borrow` was written for members and trusts that `loan_limit()` returns a number. A librarian-as-member returns `None` ("no limit") and the check crashes: the subclass broke the parent's promise. The alternative in the last lines, a librarian who *has* an ordinary membership, tells no lie, so no code is surprised. On a diagram this is the difference between a hollow triangle and an association. *Design Patterns* chapters on Strategy and State show the usual way to vary behaviour without subclassing the owner.

Reading a Class Diagram Quickly

1. **Find the centre.** The class with the most lines is usually the heart (here, `Loan` or `RepairJob`).
2. **Read the nouns first,** then the diamonds and triangles: ownership and "is a" are the strong claims.
3. **Follow one arrow in words:** "A Member holds zero or more Loans; a Loan cannot find its Member."
4. **Check each multiplicity against reality:** exactly-one and zero-or-more are where wrong assumptions hide.
5. **Look for what is missing:** the thing in the use cases that has no box; the arrow you would need to answer a question the system must answer.

Usual Mistakes

MISTAKE	WHY IT HURTS	FIX
Inheritance to share code	Subclass breaks the parent's promises; fragile hierarchies	Composition and delegation; inheritance only for true substitutability
Hollow diamond everywhere	It adds no meaning; readers guess	Association; composition only when ownership and lifetime are real
Diamond on the wrong end	Says the part owns the whole	The diamond sits at the *whole*
Arrows both ways by default	Two references to keep in step	One direction unless both are needed
Every attribute and operation shown	The diagram stops being readable and goes out of date at once	Show what the reader needs; omit getters and setters
Interface and implementation arrows confused	Solid triangle line means inheritance; dotted means implementation	Check solid versus dotted
Drawing the database	Join tables and keys are not classes	Keep the domain model; map later

MISTAKES WHILE PREPARING THIS CHAPTER

- **A shell command that wasted a step.** I wrote the demonstration files with a shell here-document whose text contained `<<enumeration>>` and similar, and the shell tried to parse them as redirections, failing with a syntax error before anything ran. How to avoid it: write source files with a file-writing tool, not a here-document, whenever the text contains angle brackets or quotes of its own.
- **My first notation diagram was tiny.** Mermaid draws class boxes in a small font, so four stacked boxes occupied a corner of the picture. Raising the font size with an `init` directive made it legible. Takeaway: look at a diagram at the size the reader will see it.
- **A demonstration that proves less than it appears to.** Python's garbage collection means `del shelf` leaves copies alive for any reason at all, and "withdrawing a title" is bookkeeping I wrote. The page now says so beside the output. Takeaway: when a diagram promise (lifetime) is not something the language enforces, say what the demo does and does not show.
- **A claim tested only by trying it.** I believed Mermaid could not draw association classes and verified it by running a diagram (it fails). I still cannot say it never will: it is a limit of Mermaid 12.1.0 as run here.

Hands-On Exercises

EXERCISE 1

Read the bike shop diagram in the solution file (do not render it first) and answer: (a) when a RepairJob is deleted, what happens to its JobParts and to the Parts? (b) can you get from a Mechanic to their jobs? (c) can a Bike exist without a Customer? (d) which line is a dependency, and what does that say about RepairJob's fields? (e) can you create a Pricing object?

View solution

EXERCISE 2

Here is some Python (in the solution file). Draw its class diagram in Mermaid, choosing for each link whether it is an association, composition, realisation or dependency, and say how you decided. Check by running the code.

View solution

EXERCISE 3

A mechanic can be certified for several bike types and a bike type has several certified mechanics; each certification has a date. Draw it with an association class in PlantUML, implement it in Python with tests (many-to-many, a fact on the link, one certification per pair), and say when an association class is the wrong choice.

View solution

CHAPTER 5 QUICK REFERENCE

- Class box: name / attributes / operations; `+ - # ~` visibility (intent only in Python); underlined = static; italic = abstract; `«interface»`, `«enumeration»`
- Dependency (dotted, open arrow): used briefly, no field. Association (solid, open arrow): a field. Composition (filled diamond at the whole): owns, shared lifetime. Aggregation (hollow diamond): rarely worth it. Generalisation (solid, hollow triangle): substitutable. Realisation (dotted, hollow triangle): fulfils an interface
- Navigability = the arrowhead = who holds the reference; two-way links must be kept in step

- Association class: a fact about a pair; implies one instance per pair; Mermaid cannot draw it, PlantUML can
- Inheritance is for substitutability, not reuse; prefer composition when the child would break the parent's promises
- Read a diagram: find the centre, read nouns then diamonds and triangles, follow one arrow in words, check multiplicities, look for what is missing

COMING NEXT

Object-Oriented Design with UML 6 (Assigning Responsibilities): CRC cards, the GRASP patterns that matter (Information Expert, Creator, Controller, Low Coupling, High Cohesion, Polymorphism, Pure Fabrication) with measured before and after Python examples, the anemic domain model, and deciding who does what in the library.

Object-Oriented Design with UML

Course 1 · Chapter 6 · Assigning Responsibilities

A class diagram shows *what exists*. The harder design question is **who does what**: when the librarian returns a book late, which class works out the fine? Get the answer wrong and you get a diagram that looks fine and code where one class knows everything and the rest are bags of data. This chapter gives you a way to decide (CRC cards), a set of reasoning rules (GRASP), a name for the failure (the anemic domain model), and a measured comparison of the same behaviour written both ways.

> **WHAT WAS RUN, AND WHAT WAS NOT; AND HOW FAR THE "MEASUREMENTS" GO**
>
> I wrote the library's borrow and return behaviour twice in Python 3.14: version A (data classes plus one service holding all the rules) and version B (responsibilities assigned with GRASP). **One suite of 26 behaviour tests runs against both** and passes on both. Then I measured both with small scripts I wrote for this chapter, applied the same two change requests to each, and tried a deliberately incomplete change. **The caveats matter.** I wrote both versions, so version A reflects my idea of a typical procedural design, not a proven average; the program is tiny (about 70 against 95 lines); my metrics are crude counts, not validated measures of quality; and one result below goes *against* the design B, which I report. From general knowledge, not re-read: that GRASP is from Craig Larman's *Applying UML and Patterns* (nine patterns; this chapter covers seven), that CRC cards come from Kent Beck and Ward Cunningham, and that "anemic domain model" is Martin Fowler's term.

CRC Cards: Deciding Before Drawing

A **CRC card** is an index card for one class with three things on it: the **C**lass name, its **R**esponsibilities (things it knows or does, in plain words), and its **C**ollaborators (the other classes it needs to do them). You lay the cards on a table and *role-play* a use case: "A member hands over a book. Who receives it?" Each time you say "and then someone must...", you either find a card that should do it or you write a new card. The physical limit (a card has room for a few lines) is the feature: a class with too many responsibilities will not fit, and that is a design signal.

Here are the cards for the library's borrow and return use cases, written *before* I compared them to the code:

CLASS	RESPONSIBILITIES	COLLABORATORS
LoanDesk	Receive "borrow" and "return" requests; find the member, copy and loan involved; hand the work to them	Catalogue, Member, Copy, Loan
Catalogue	Find a member by card; a copy by barcode; the open loan for a copy	Member, Copy, Loan
Member	Know card, expiry, fines, loans, kind; decide whether they may borrow; create their Loans; take a book back and add any fine	Loan, Copy, MemberKind
Loan	Know barcode and dates; say whether it is open; say how many days overdue; work out the fine for a kind of member	MemberKind
Copy	Know barcode and whether on loan; say whether it can be lent	(none)
MemberKind	Give the loan limit; give the fine per day	(none)

Cards are a thinking tool, but I wanted to check mine against something. The script below lists the class names each class's code actually mentions, and compares:

```
"""CRC cards for version B, written by hand first; then compared with the class names ea
import ast
import pathlib

CARDS = {
    "LoanDesk": {
        "responsibilities": ["receive 'borrow' and 'return' requests", "find the member,
        "collaborators": {"Catalogue", "Member", "Copy", "Loan"},
    },
    "Catalogue": {
        "responsibilities": ["find a member by card", "find a copy by barcode", "find th
        "collaborators": {"Member", "Copy", "Loan"},
    },
    "Member": {
        "responsibilities": ["know card, expiry, fines, loans, kind", "decide whether th
        "collaborators": {"Loan", "Copy", "MemberKind"},
    },
    "Loan": {
        "responsibilities": ["know barcode and dates", "say whether it is open", "say ho
        "collaborators": {"MemberKind"},
    },
    "Copy": {"responsibilities": ["know barcode and whether on loan", "say whether it ca
    "MemberKind": {"responsibilities": ["give the loan limit", "give the fine per day"],
```

```
}


def named_classes(path: str) -> dict[str, set[str]]:
    tree = ast.parse(pathlib.Path(path).read_text(encoding="utf8"))
    classes = {n.name for n in tree.body if isinstance(n, ast.ClassDef)}
    out = {}
    for node in tree.body:
        if isinstance(node, ast.ClassDef):
            names = {n.id for n in ast.walk(node) if isinstance(n, ast.Name) and n.id in
            for base in node.bases:
                names.discard(getattr(base, "id", ""))
            out[node.name] = names - {"Refusal"}
    return out


if __name__ == "__main__":
    code = named_classes(str(pathlib.Path(__file__).with_name("library_b.py")))
    for name, card in CARDS.items():
        on_card, in_code = card["collaborators"], code.get(name, set())
        print(f"{name}")
        print(f"   card says collaborates with : {sorted(on_card) or '-'}")
        print(f"   code names                  : {sorted(in_code) or '-'}")
        only_card, only_code = sorted(on_card - in_code), sorted(in_code - on_card)
        if only_card:
            print(f"   on the card, never named in the code: {only_card}")
        if only_code:
            print(f"   named in the code, missing from the card: {only_code}")
```

```
LoanDesk
   card says collaborates with : ['Catalogue', 'Copy', 'Loan', 'Member']
   code names                  : ['Catalogue', 'Loan']
   on the card, never named in the code: ['Copy', 'Member']
Catalogue
   card says collaborates with : ['Copy', 'Loan', 'Member']
   code names                  : ['Copy', 'Loan', 'Member']
Member
   card says collaborates with : ['Copy', 'Loan', 'MemberKind']
   code names                  : ['Copy', 'Loan', 'MemberKind']
Loan
   card says collaborates with : ['MemberKind']
   code names                  : ['MemberKind']
Copy
```

```
    card says collaborates with : -
    code names                  : -
  MemberKind
    card says collaborates with : -
    code names                  : -
```

One mismatch, and it is instructive. My card says `LoanDesk` collaborates with `Member` and `Copy`, but its code never names either: it gets them from the catalogue and calls them in a chain (`self.catalogue.member(card).borrow(...)`). Both statements are true. The card describes *who it works with at run time*; the code shows *which names it depends on*. The second kind, the names a class depends on, is what makes a class costly to change. Also note that the script found only a difference of names: it cannot tell whether a responsibility is on the right card. That is still a judgement.

The Problem Both Versions Solve

The behaviour (from chapter 3's use cases, simplified): **borrow** checks the card is valid, no unpaid fines, under the loan limit (5 for standard members, 3 for juniors), and the copy is free, then records a 21-day loan; **return** finds the open loan, works out any fine (10p a day for standard members, 5p for juniors), records it, and frees the copy. Chapter 4's lesson is applied: items are identified by barcode (a Copy), not ISBN.

Version A: One Service, Passive Data

```
"""Version A: data classes plus one service that holds all the rules (the 'anemic domain
from dataclasses import dataclass, field
from datetime import date, timedelta


class Refusal(Exception):
    pass


@dataclass
class Member:
    card: str
    expires_on: date
    kind: str = "standard"                # "standard" or "junior"
    fines_pence: int = 0
```

```
    loans: list = field(default_factory=list)


@dataclass
class Copy:
    barcode: str
    on_loan: bool = False

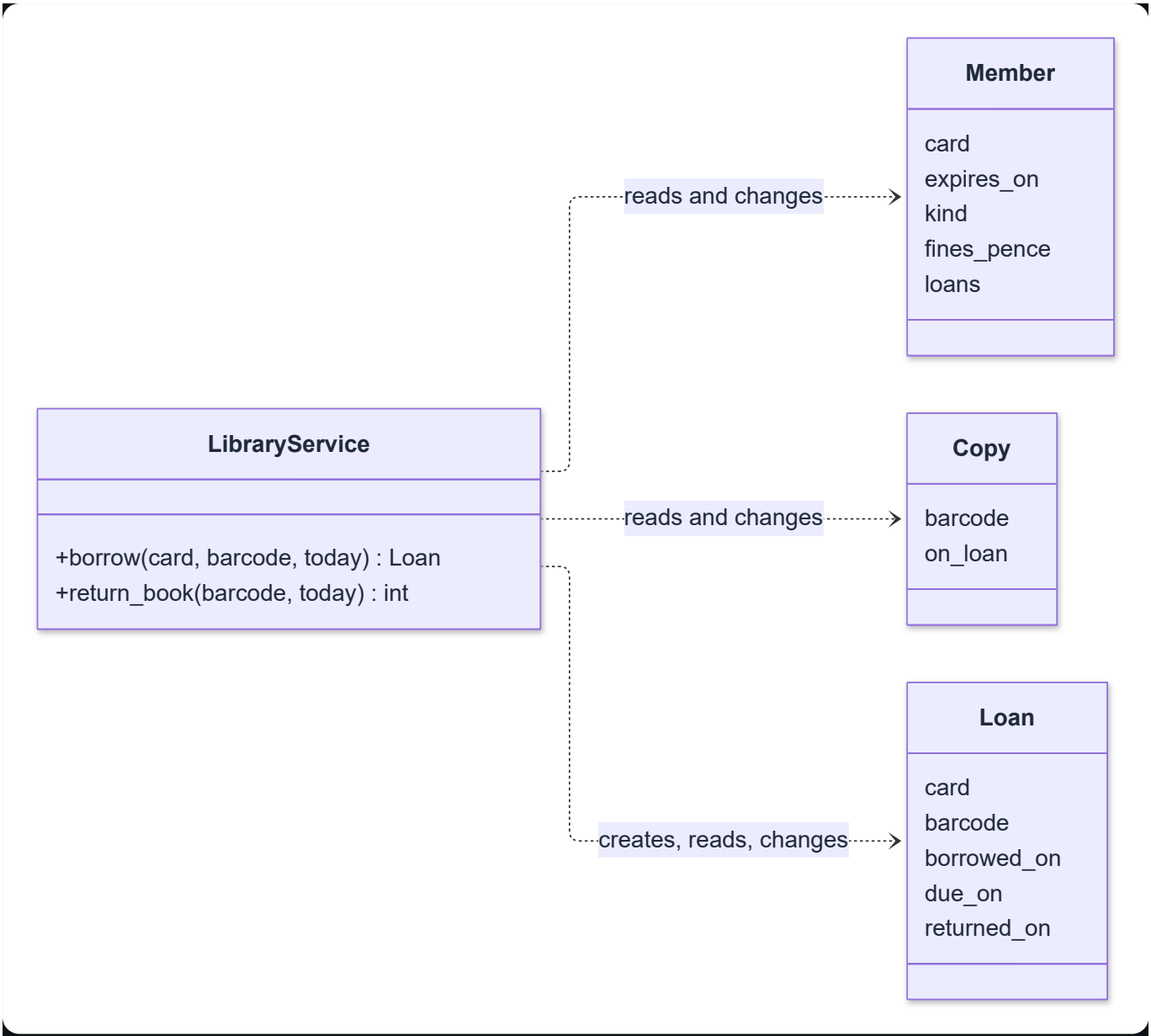

@dataclass
class Loan:
    card: str
    barcode: str
    borrowed_on: date
    due_on: date
    returned_on: date | None = None


class LibraryService:
    def __init__(self, members, copies):
        self.members = {m.card: m for m in members}
        self.copies = {c.barcode: c for c in copies}
        self.loans: list[Loan] = []

    def borrow(self, card: str, barcode: str, today: date) -> Loan:
        member = self.members.get(card)
        if member is None or member.expires_on < today:
            raise Refusal("membership is not valid")
        if member.fines_pence > 0:
            raise Refusal("unpaid fines")
        if member.kind == "standard":
            limit = 5
        elif member.kind == "junior":
            limit = 3
        else:
            raise ValueError(member.kind)
        open_loans = [l for l in self.loans if l.card == card and l.returned_on is None]
        if len(open_loans) >= limit:
            raise Refusal("loan limit reached")
        copy = self.copies.get(barcode)
        if copy is None:
            raise Refusal("no such copy")
        if copy.on_loan:
            raise Refusal("already on loan")
        loan = Loan(card, barcode, today, today + timedelta(days=21))
```

```
        copy.on_loan = True
        self.loans.append(loan)
        member.loans.append(loan)
        return loan

    def return_book(self, barcode: str, today: date) -> int:
        loan = next((l for l in self.loans if l.barcode == barcode and l.returned_on is N
        if loan is None:
            raise Refusal("that copy is not on loan")
        member = self.members[loan.card]
        days_late = max(0, (today - loan.due_on).days)
        if member.kind == "standard":
            rate = 10
        elif member.kind == "junior":
            rate = 5
        else:
            raise ValueError(member.kind)
        fine = days_late * rate
        member.fines_pence += fine
        loan.returned_on = today
        self.copies[barcode].on_loan = False
        return fine
```



Version A. Three classes of data, and one service that reads and changes all of them.

This is not an absurd design; it is how many first drafts look, and it has virtues: everything is in one place, and you can read the whole behaviour top to bottom. Its problem has a name.

The Anemic Domain Model

An **anemic domain model** (Fowler's term, from memory) has classes for the domain concepts, but they are only data with getters and setters, while all the behaviour lives in "service" classes that manipulate them. It looks object-oriented (there is a `Member` class) but works like procedural code over records. Version A is anemic: `Member`, `Copy` and `Loan` have no methods at all. The costs, to be checked against the measurements below:

- **The rules about a member are not with the member.** To learn when a member may borrow, you read the service, not `Member`.
- **The service knows everything about everyone's fields**, so any change to a data class can break it, and it grows with every rule.
- **Nothing protects the data's own invariants.** Any code can set `fines_pence` or `on_loan` to anything.

A caution in the other direction: services are not evil. Something has to coordinate a use case across several objects (that is the Controller below), and some logic belongs to no single object. The smell is when *all* the behaviour is there and the domain classes are empty.

The GRASP Patterns That Matter

GRASP stands for General Responsibility Assignment Software Patterns (from memory of Larman's book). They are not design patterns in the Gang of Four sense; they are *questions to ask when giving a responsibility to a class*. Seven of the nine, each with where it shows up in version B:

PATTERN	THE QUESTION AND THE ANSWER	IN VERSION B
Information Expert	Who has the information needed? Give the responsibility to that class	`Loan.days_overdue` and `Loan.fine`: the loan holds the due date. `Member.check_can_borrow`: the member holds expiry, fines, loans
Creator	Who should create an object of class X? The class that contains or records X, closely uses it, or has the data to initialise it	`Member.borrow` creates the `Loan`: the member records its loans (the composition in chapter 5's notation)
Controller	Who receives a request from outside the domain (a screen, a message)? A class that represents the use case or the system, and *delegates* to domain objects	`LoanDesk`: receives borrow and return, holds no rule
Low Coupling	Assign responsibilities so that classes depend on few others	The domain classes `Loan` and `Copy` name no other domain class (measured below)
High Cohesion	Keep a class's responsibilities closely related and focused	Each class's card fits on a card; `Copy` only answers questions about a copy

PATTERN	THE QUESTION AND THE ANSWER	IN VERSION B
Polymorphism	When behaviour varies by type, give each type an operation, rather than testing the type	`MemberKind` with `Standard` and `Junior`: the limit and the fine rate. No `if kind == ...` anywhere
Pure Fabrication	When no domain concept is the right home, invent a class that is not in the domain, to keep the others cohesive	`Catalogue` (lookup) and arguably `LoanDesk`: neither is a library concept; they exist so that `Member` and `Loan` need not know how to find each other

The other two (Indirection and Protected Variations) are about inserting an intermediate object or interface to isolate a part that is likely to change. Polymorphism and Pure Fabrication here are doing a version of that job.

These overlap with SOLID: high cohesion is close to the single-responsibility principle, and polymorphism instead of type tests is close to open-closed; see *Clean Code, SOLID & Refactoring* 6 and 7, which this course does not repeat. That mapping is mine, not an official one.

Version B: Responsibilities Assigned

```
"""Version B: the same behaviour, with responsibilities given to the classes that hold t
from abc import ABC, abstractmethod
from dataclasses import dataclass, field
from datetime import date, timedelta


class Refusal(Exception):
    pass


# --- Polymorphism: what varies by kind of member lives in one class per kind ----------
class MemberKind(ABC):
    @abstractmethod
    def loan_limit(self) -> int: ...

    @abstractmethod
    def fine_per_day(self) -> int: ...


class Standard(MemberKind):
    def loan_limit(self) -> int:
        return 5
```

```
    def fine_per_day(self) -> int:
        return 10


class Junior(MemberKind):
    def loan_limit(self) -> int:
        return 3

    def fine_per_day(self) -> int:
        return 5


KINDS = {"standard": Standard(), "junior": Junior()}


# --- Information Expert: each class does the work for which it holds the data ---------
@dataclass
class Copy:
    barcode: str
    on_loan: bool = False

    def check_can_be_lent(self) -> None:
        if self.on_loan:
            raise Refusal("already on loan")


@dataclass
class Loan:
    barcode: str
    borrowed_on: date
    due_on: date
    returned_on: date | None = None

    def is_open(self) -> bool:
        return self.returned_on is None

    def days_overdue(self, today: date) -> int:
        return max(0, (today - self.due_on).days)

    def fine(self, today: date, kind: MemberKind) -> int:
        return self.days_overdue(today) * kind.fine_per_day()


@dataclass
```

```
class Member:
    card: str
    expires_on: date
    kind: MemberKind
    fines_pence: int = 0
    loans: list[Loan] = field(default_factory=list)

    def open_loans(self) -> list[Loan]:
        return [l for l in self.loans if l.is_open()]

    def check_can_borrow(self, today: date) -> None:
        if self.expires_on < today:
            raise Refusal("membership is not valid")
        if self.fines_pence > 0:
            raise Refusal("unpaid fines")
        if len(self.open_loans()) >= self.kind.loan_limit():
            raise Refusal("loan limit reached")

    def borrow(self, copy: Copy, today: date) -> Loan:        # Creator: the Member reco
        self.check_can_borrow(today)
        copy.check_can_be_lent()
        loan = Loan(copy.barcode, today, today + timedelta(days=21))
        copy.on_loan = True
        self.loans.append(loan)
        return loan

    def give_back(self, loan: Loan, copy: Copy, today: date) -> int:
        fine = loan.fine(today, self.kind)
        self.fines_pence += fine
        loan.returned_on = today
        copy.on_loan = False
        return fine


# --- Pure Fabrication: not a library concept, but it keeps lookup out of the domain cla
class Catalogue:
    def __init__(self, members, copies):
        self.members = {m.card: m for m in members}
        self.copies = {c.barcode: c for c in copies}

    def member(self, card: str) -> Member:
        if card not in self.members:
            raise Refusal("membership is not valid")
        return self.members[card]
```

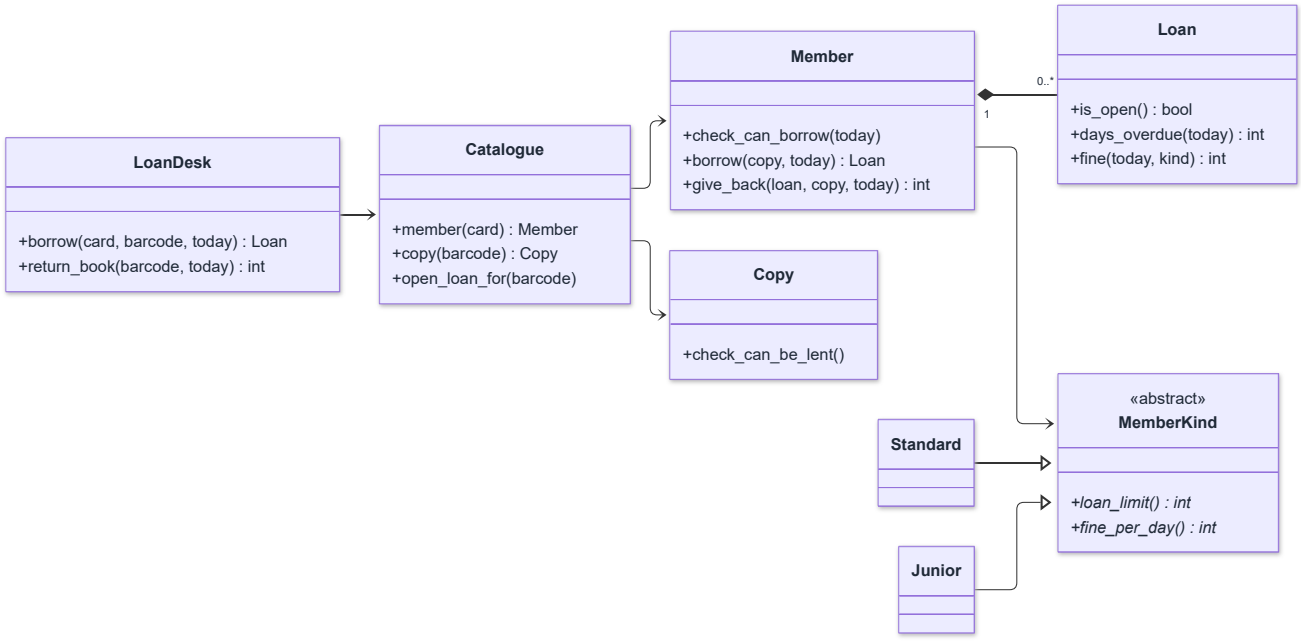
```
    def copy(self, barcode: str) -> Copy:
        if barcode not in self.copies:
            raise Refusal("no such copy")
        return self.copies[barcode]

    def open_loan_for(self, barcode: str) -> tuple[Member, Loan]:
        for m in self.members.values():
            for l in m.open_loans():
                if l.barcode == barcode:
                    return m, l
        raise Refusal("that copy is not on loan")


# --- Controller: receives the use case's request and delegates; it holds no business ru
class LoanDesk:
    def __init__(self, members, copies):
        self.catalogue = Catalogue(members, copies)

    def borrow(self, card: str, barcode: str, today: date) -> Loan:
        return self.catalogue.member(card).borrow(self.catalogue.copy(barcode), today)

    def return_book(self, barcode: str, today: date) -> int:
        member, loan = self.catalogue.open_loan_for(barcode)
        return member.give_back(loan, self.catalogue.copy(barcode), today)
```



Version B. More classes, each smaller. The request enters at LoanDesk, which hands off; the rules sit with the objects that hold the data.

Do They Behave the Same?

Before comparing structure, check the behaviour is identical. One suite of 26 tests (13 behaviours, each run through a small adapter against A and against B):

```
"""One behaviour suite, run against BOTH versions through a small adapter, to show they
import unittest
from datetime import date

import library_a as a
import library_b as b

TODAY = date(2026, 10, 10)
LATER = date(2026, 11, 3)          # 3 days after a loan taken on TODAY falls due (21 da


def make_a(members, barcodes):
    ms = [a.Member(card, exp, kind=kind, fines_pence=fines) for card, exp, kind, fines i
    return a.LibraryService(ms, [a.Copy(c) for c in barcodes]), a.Refusal


def make_b(members, barcodes):
    ms = [b.Member(card, exp, b.KINDS[kind], fines_pence=fines) for card, exp, kind, fin
    return b.LoanDesk(ms, [b.Copy(c) for c in barcodes]), b.Refusal


class Behaviour:
    make = None

    def world(self, kind="standard", fines=0, expires=date(2027, 1, 1), barcodes=("B1",
        return self.make([("M1", expires, kind, fines)], barcodes)

    def test_borrow_sets_due_date(self):
        desk, _ = self.world()
        self.assertEqual(desk.borrow("M1", "B1", TODAY).due_on, date(2026, 10, 31))

    def test_expired_membership_refused(self):
        desk, Refusal = self.world(expires=date(2026, 10, 9))
        with self.assertRaises(Refusal):
            desk.borrow("M1", "B1", TODAY)

    def test_unknown_card_refused(self):
        desk, Refusal = self.world()
```

```
        with self.assertRaises(Refusal):
            desk.borrow("NOPE", "B1", TODAY)

    def test_fines_block_borrowing(self):
        desk, Refusal = self.world(fines=10)
        with self.assertRaises(Refusal):
            desk.borrow("M1", "B1", TODAY)

    def test_standard_limit_is_five(self):
        desk, Refusal = self.world()
        for code in ("B1", "B2", "B3", "B4", "B5"):
            desk.borrow("M1", code, TODAY)
        with self.assertRaises(Refusal):
            desk.borrow("M1", "B6", TODAY)

    def test_junior_limit_is_three(self):
        desk, Refusal = self.world(kind="junior")
        for code in ("B1", "B2", "B3"):
            desk.borrow("M1", code, TODAY)
        with self.assertRaises(Refusal):
            desk.borrow("M1", "B4", TODAY)

    def test_copy_on_loan_refused(self):
        desk, Refusal = self.world()
        desk.borrow("M1", "B1", TODAY)
        with self.assertRaises(Refusal):
            desk.borrow("M1", "B1", TODAY)

    def test_unknown_copy_refused(self):
        desk, Refusal = self.world()
        with self.assertRaises(Refusal):
            desk.borrow("M1", "ZZZ", TODAY)

    def test_return_on_time_no_fine(self):
        desk, _ = self.world()
        desk.borrow("M1", "B1", TODAY)
        self.assertEqual(desk.return_book("B1", date(2026, 10, 31)), 0)

    def test_standard_fine_is_10p_a_day(self):
        desk, _ = self.world()
        desk.borrow("M1", "B1", TODAY)
        self.assertEqual(desk.return_book("B1", LATER), 30)

    def test_junior_fine_is_5p_a_day(self):
        desk, _ = self.world(kind="junior")
```

```
        desk.borrow("M1", "B1", TODAY)
        self.assertEqual(desk.return_book("B1", LATER), 15)

    def test_return_frees_the_copy_and_unknown_return_refused(self):
        desk, Refusal = self.world()
        desk.borrow("M1", "B1", TODAY)
        desk.return_book("B1", date(2026, 10, 31))
        desk.borrow("M1", "B1", date(2026, 11, 1))                  # available again
        with self.assertRaises(Refusal):
            desk.return_book("B2", TODAY)

    def test_a_fine_then_blocks_the_next_loan(self):
        desk, Refusal = self.world()
        desk.borrow("M1", "B1", TODAY)
        desk.return_book("B1", LATER)
        with self.assertRaises(Refusal):
            desk.borrow("M1", "B2", LATER)


class VersionA(Behaviour, unittest.TestCase):
    make = staticmethod(make_a)


class VersionB(Behaviour, unittest.TestCase):
    make = staticmethod(make_b)


if __name__ == "__main__":
    unittest.main(verbosity=1)
```

```
Ran 26 tests in 0.001s

OK
```

A passing suite shows the versions agree on *these* 13 behaviours; it could be too weak. So I tested the tests: I broke B on purpose (junior fine 6p instead of 5p) and A on purpose (fines no longer block borrowing), and ran the same suite:

```
unmodified: 0 of 26 tests fail []
B with junior fine 6p: 1 of 26 tests fail ['VersionB.test_junior_fine_is_5p_a_day']
```

```
A with fines ignored: 2 of 26 tests fail ['VersionA.test_a_fine_then_blocks_the_next_loa
restored: 0 of 26 tests fail []
```

Each bug was caught, by one and two tests respectively. That is reasonable evidence that the suite is looking at the right things, not that it is complete.

What I Measured

```
"""Three crude, honest measurements of a module of classes.

  named     distinct OTHER domain classes whose names the class mentions (a rough couplin
  reach-in  reads or writes of another class's data fields (x.field where field belongs t
  kind-tests comparisons of a '.kind' attribute: branching on a type tag, which polymorph

Exceptions (classes ending in Refusal or deriving from Exception) are not counted as dom
These are my own simple measures, written for this chapter. They count; they do not judg
"""
import ast
import pathlib
import sys


def analyse(path: str) -> list[dict]:
    tree = ast.parse(pathlib.Path(path).read_text(encoding="utf8"))
    classes = {n.name: n for n in tree.body if isinstance(n, ast.ClassDef)}
    domain = {name for name, node in classes.items() if not name.endswith("Refusal") and
        isinstance(b, ast.Name) and b.id in ("Exception", "ABC") for b in node.bases) an

    def fields(node: ast.ClassDef) -> set[str]:
        out = {s.target.id for s in node.body if isinstance(s, ast.AnnAssign) and isinst
        for fn in node.body:
            if isinstance(fn, ast.FunctionDef) and fn.name == "__init__":
                for n in ast.walk(fn):
                    if isinstance(n, ast.Attribute) and isinstance(n.value, ast.Name) an
                        out.add(n.attr)
        return out

    field_owner: dict[str, set[str]] = {}
    for name in domain:
        for f in fields(classes[name]):
            field_owner.setdefault(f, set()).add(name)
```

```
    rows = []
    for name, node in classes.items():
        if name not in domain and not name.startswith("Standard") and not name.startswit
            continue
        own = fields(node)
        named, reach, chains = set(), 0, 0
        sites = []
        for n in ast.walk(node):
            if isinstance(n, ast.Name) and n.id in domain and n.id != name:
                named.add(n.id)
            if isinstance(n, ast.Attribute) and not (isinstance(n.value, ast.Name) and n
                owners = field_owner.get(n.attr, set())
                if owners and n.attr not in own and name not in owners:
                    reach += 1
                    sites.append((n.lineno, ast.unparse(n)))
            if isinstance(n, ast.Compare) and any(isinstance(x, ast.Attribute) and x.att
                chains += 1
        methods = [f.name for f in node.body if isinstance(f, ast.FunctionDef)]
        rows.append({"class": name, "methods": len(methods), "named": len(named), "reach_
    return rows


def show(title: str, path: str, detail: bool = False) -> None:
    rows = analyse(path)
    print(title)
    print(f"  {'class':<14}{'methods':>8}{'named':>7}{'reach-in':>10}{'kind-tests':>11}"
    for r in rows:
        print(f"  {r['class']:<14}{r['methods']:>8}{r['named']:>7}{r['reach_in']:>10}{r[
    if detail:
        for r in rows:
            for line, text in r["sites"]:
                print(f"    reach-in in {r['class']}, line {line}: {text}")
    print(f"  {'TOTAL':<14}{sum(r['methods'] for r in rows):>8}{'':>7}{sum(r['reach_in']


if __name__ == "__main__":
    for arg in [a for a in sys.argv[1:] if not a.startswith('--')]:
        show(arg, arg, detail="--detail" in sys.argv)
        print()
```

```
library_a.py
  class          methods  named  reach-in kind-tests
```

```
  Member                0      0          0          0
  Copy                  0      0          0          0
  Loan                  0      0          0          0
  LibraryService        3      1         21          4
  TOTAL                 3                21          4

library_b.py
  class           methods  named  reach-in kind-tests
  Standard              2      0          0          0
  Junior                2      0          0          0
  Copy                  1      0          0          0
  Loan                  3      0          0          0
  Member                4      2          4          0
  Catalogue             4      3          3          0
  LoanDesk              3      2          0          0
  TOTAL                19                 7          0
```

What the table says, and does not say:

- **Reach-in** counts places where a class reads or writes another class's data. Version A's service does it 21 times; version B 7. This is the Information Expert measure: in A the service reaches through everyone; in B most of the questions are answered by the object that holds the data.
- **Kind-tests** counts comparisons of a type tag (`.kind`). A has 4 (two if-chains, one in each method); B has none, because `MemberKind` answers instead.
- **Methods and size** go the other way. A has 3 methods and about 69 non-blank, non-comment lines; B has 19 methods and about 95. **Version B is more code and more classes.** That is the price, and for a program this small it may not be worth paying; the comparison favours B as the rules grow, not at 70 lines.
- **The seven reach-ins that remain in B** are worth reading rather than ignoring. Three (`m.card`, `c.barcode`, `l.barcode` in `Catalogue`) read identifying values to build or search an index, which is hard to avoid. Four are in `Member`: it writes `copy.on_loan` twice, reads `copy.barcode`, and sets `loan.returned_on`. Those are real Information Expert lapses: a `Member` should not set another object's fields. Exercise 3 fixes them and measures again.

Two Change Requests, Applied to Both

CR1: a new "senior" member kind with limit 8 and fine 5p a day. **CR2:** fines are capped at 500p per loan for everyone. I applied each by script to both versions (so the edits are exact and

countable) and diffed the source:

```
"""Apply the same two change requests to both versions, and measure what each change tou

CR1: a new 'senior' member kind: loan limit 8, fine 5p a day.
CR2: fines are capped at 500p per loan, for every kind of member.
"""
import ast
import difflib
import pathlib

HERE = pathlib.Path(__file__).parent


def replace_once(text: str, old: str, new: str) -> str:
    assert text.count(old) == 1, f"expected exactly one match for: {old!r} (found {text.
    return text.replace(old, new)


def make_a(cr: set[str]) -> str:
    text = (HERE / "library_a.py").read_text(encoding="utf8")
    if "cr1" in cr:
        text = replace_once(text, '        elif member.kind == "junior":\n            li
                            '        elif member.kind == "junior":\n            limit = 
        text = replace_once(text, '        elif member.kind == "junior":\n            ra
                            '        elif member.kind == "junior":\n            rate = 5
    if "cr2" in cr:
        text = replace_once(text, "        fine = days_late * rate\n", "        fine = m
    return text


def make_b(cr: set[str]) -> str:
    text = (HERE / "library_b.py").read_text(encoding="utf8")
    if "cr1" in cr:
        text = replace_once(text, 'KINDS = {"standard": Standard(), "junior": Junior()}'
                            'class Senior(MemberKind):\n    def loan_limit(self) -> int:
                            'KINDS = {"standard": Standard(), "junior": Junior(), "senio
    if "cr2" in cr:
        text = replace_once(text, "        return self.days_overdue(today) * kind.fine_p
                            "        return min(self.days_overdue(today) * kind.fine_per_
    return text


def functions(source: str) -> dict[str, str]:
```

```
    """Qualified name -> source text of every function and class-level statement group, 
    tree = ast.parse(source)
    out = {}
    for node in tree.body:
        if isinstance(node, ast.ClassDef):
            out[node.name] = ast.dump(ast.Module(body=[b for b in node.body if not isins
            for fn in node.body:
                if isinstance(fn, ast.FunctionDef):
                    out[f"{node.name}.{fn.name}"] = ast.dump(fn)
        elif isinstance(node, ast.Assign):
            out[ast.unparse(node.targets[0])] = ast.dump(node)
    return out


def measure(before: str, after: str) -> dict:
    diff = list(difflib.unified_diff(before.splitlines(), after.splitlines(), lineterm="
    removed = sum(1 for l in diff if l.startswith("-") and not l.startswith("---"))
    added = sum(1 for l in diff if l.startswith("+") and not l.startswith("+++"))
    f0, f1 = functions(before), functions(after)
    edited = sorted(k for k in f0 if k in f1 and f0[k] != f1[k])
    new = sorted(k for k in f1 if k not in f0)
    return {"removed": removed, "added": added, "edited": edited, "new": new}


if __name__ == "__main__":
    for name, maker, orig in (("A", make_a, "library_a.py"), ("B", make_b, "library_b.py
        before = (HERE / orig).read_text(encoding="utf8")
        for label, cr in (("CR1 senior kind", {"cr1"}), ("CR2 fine cap", {"cr2"})):
            m = measure(before, maker(cr))
            print(f"Version {name}, {label}: existing lines changed/removed {m['removed'
            print(f"    existing functions/classes edited: {m['edited'] or 'none'}")
            print(f"    new functions/classes: {m['new'] or 'none'}")
        (HERE / f"library_{name.lower()}_cr.py").write_text(maker({"cr1", "cr2"}), encod
```

```
Version A, CR1 senior kind: existing lines changed/removed 0, lines added 4
    existing functions/classes edited: ['LibraryService.borrow', 'LibraryService.return_
    new functions/classes: none
Version A, CR2 fine cap: existing lines changed/removed 1, lines added 1
    existing functions/classes edited: ['LibraryService.return_book']
    new functions/classes: none
Version B, CR1 senior kind: existing lines changed/removed 1, lines added 9
    existing functions/classes edited: ['KINDS']
    new functions/classes: ['Senior', 'Senior.fine_per_day', 'Senior.loan_limit']
```

```
Version B, CR2 fine cap: existing lines changed/removed 1, lines added 1
    existing functions/classes edited: ['Loan.fine']
    new functions/classes: none
```

Read this honestly: the line counts do not favour B. CR1 in A is four added lines in two existing functions; in B it is nine added lines (a new class) and one changed line (the registry). CR2 is one changed line in each. What differs is *where* the change goes and *what can go wrong*:

- In A, CR1 means editing the *logic* of two existing functions; each must be remembered.
- In B, CR1 adds a class and leaves existing logic untouched (the one changed line is a registry entry).
- CR2 is a single place in both, because the fine was a single calculation in both. Polymorphism did nothing for it, and it should not be expected to.

Both modified versions pass the behaviour suite for the new rules (8 tests, run on each):

```
----------------------------------------------------------------------
Ran 8 tests in 0.000s

OK
```

The Change Made "Almost Completely"

The cost of the A style shows up when one of those places is forgotten. I applied CR1 to A but left out the fine chain, and to B but left out the fine method of the new class:

```
"""What happens when a change is made 'almost completely': the senior kind is added, but
import importlib
import pathlib
import sys
import types
from datetime import date

HERE = pathlib.Path(__file__).parent
sys.path.insert(0, str(HERE))
import apply_changes  # noqa: E402


def load(name: str, source: str) -> types.ModuleType:
    module = types.ModuleType(name)
    exec(compile(source, name, "exec"), module.__dict__)
```

```
    return module


# --- Version A: add 'senior' to the limit chain in borrow() but forget the fine chain i
text = apply_changes.make_a({"cr1"})
text = text.replace('        elif member.kind == "senior":\n            rate = 5\n', "")
a = load("a_partial", text)
svc = a.LibraryService([a.Member("M1", date(2027, 1, 1), kind="senior")], [a.Copy("B0")]
svc.borrow("M1", "B0", date(2026, 10, 10))
print("A: the senior member borrowed a book. Now they bring it back...")
try:
    svc.return_book("B0", date(2026, 10, 31))
except Exception as e:
    print(f"A: {type(e).__name__}: {e!s}   (the loan is still open: {svc.loans[0].return

# --- Version B: add Senior but forget one of its two methods ---
text = apply_changes.make_b({"cr1"})
text = text.replace("    def fine_per_day(self) -> int:\n        return 5\n\n\nKINDS = {
try:
    load("b_partial", text)
    print("B: loaded without complaint")
except TypeError as e:
    print(f"B: TypeError at import time: {e}")
```

```
A: the senior member borrowed a book. Now they bring it back...
A: ValueError: senior   (the loan is still open: True, so the book can never be returned
B: TypeError at import time: Can't instantiate abstract class Senior without an implemen
```

In A the mistake is invisible until a senior member returns a book, which then can never be returned (the loan stays open). In B it fails *immediately*, when the class is defined, because `MemberKind` declares `fine_per_day` abstract and Python refuses to create an incomplete `Senior`. That is the real argument for polymorphism: not shorter code, but a change that cannot be left half done. (An enumeration with a lookup table in A would also help; this is a comparison of two styles, not of every way to write A.)

Controller, Pure Fabrication and Where Rules Live

- **Keep the controller thin.** `LoanDesk.borrow` is one line: find the member, find the copy, ask the member to borrow. If a rule appears in the controller, version A is growing back.
- **A pure fabrication is allowed to be ugly.** `Catalogue` is not a thing a librarian would mention. It is a legitimate answer to "where does lookup go?" when the alternative is giving `Member` a reference to every other member.
- **Resist a class per noun.** More classes is the cost; a class should earn its place by a cohesive responsibility.

MISTAKES WHILE PREPARING THIS CHAPTER

- **The line counts do not favour B, and I reported that.** The tempting move is to pick the measure that makes the preferred design look best. The honest one is to decide what you will count, report all of it (including that B is more code), and make the argument the data supports: here, the half-finished change, not the line counts.
- **My "named classes" metric hid the thing I cared about.** It counts only the names a class mentions, so A's service, which never writes `Member` or `Copy` but manipulates them constantly, scored one. I kept the column to show the limit and relied on the reach-in count instead.
- **A metric with the wrong name.** The first version of the script labelled its type-tag count "chains", but it counts comparisons: version A has four comparisons in two if-chains. A column heading is a claim; I renamed it "kind-tests".
- **A card and the code disagreed, and both were right.** See the CRC section: run-time collaboration is not compile-time naming. Takeaway: say which one a diagram or card claims.
- **Leftover scaffolding.** My first draft of the exercise 2 test script contained a helper function that did nothing (a half-finished idea for counting edits). It ran fine, which is why I nearly left it. Takeaway: re-read a script you are going to publish for code that cannot matter.

Deciding Who Does What: A Checklist

1. Write the use case step in plain words ("work out the fine").
2. Ask **who has the information?** (Expert). Start there.
3. Ask **who contains or records the thing being created?** (Creator).
4. Does the responsibility vary by type? Put it on the type (Polymorphism), not in an if-chain.
5. Does it make a class do two unrelated jobs, or depend on many classes? Move it, or invent a helper (Cohesion, Coupling, Pure Fabrication).
6. Is it a request from outside the domain? Controller: delegate, hold no rule.

7. Fit the class on a card. If it will not fit, split it.

Hands-On Exercises

EXERCISE 1

In the bike shop, an invoice is the labour (hours at 30 pounds an hour) plus the parts used (price times quantity). Write the "one function reaches into everything" version and the Information Expert version (JobPart works out its own cost, the job its labour and total), prove they agree on three jobs, and show that a trade discount on parts only touches one class.

View solution

EXERCISE 2

The solution file contains shipping-cost code with two if-chains over a method name. Replace the chains with polymorphism, prove the behaviour is unchanged, count the branches before and after, and say what adding a "drone" method costs in each version.

View solution

EXERCISE 3

Version B's `Member` still writes into `Copy` and `Loan`. Give those two classes the operations they should own, refactor, show the behaviour suite still passes, and measure the reach-ins again. Which ones remain, and are they acceptable?

View solution

CHAPTER 6 QUICK REFERENCE

- CRC card: Class, Responsibilities, Collaborators; role-play a use case; a class that will not fit on a card does too much; a card shows run-time collaboration, which may differ from the names in the code
- Anemic domain model: domain classes with data only, all rules in a service; costs: rules far from their data, a service coupled to every field, no invariants protected

- Information Expert (who has the data?), Creator (who contains or records it?), Controller (delegate, no rules), Low Coupling, High Cohesion, Polymorphism (no type tests), Pure Fabrication (an invented class to keep others cohesive)
- Measured here: reach-in 21 (A) to 7 (B) to 4 (B after exercise 3); kind-tests 4 to 0; but B is more code (95 against 69 lines) and more classes
- Polymorphism's real benefit: a change that cannot be left half done (the forgotten place failed at start-up in B, and as a book that could never be returned in A); it does nothing for a rule that is already in one place
- Test the tests (mutants), and decide what to measure before looking at the result

COMING NEXT

Object-Oriented Design with UML 7 (Sequence Diagrams): lifelines, messages, return values, activation, the `alt`, `opt`, `loop`, `par` and `ref` fragments, going from a use case scenario to a sequence diagram to method signatures, and drawing "borrow a book" for this design and running it.

Object-Oriented Design with UML

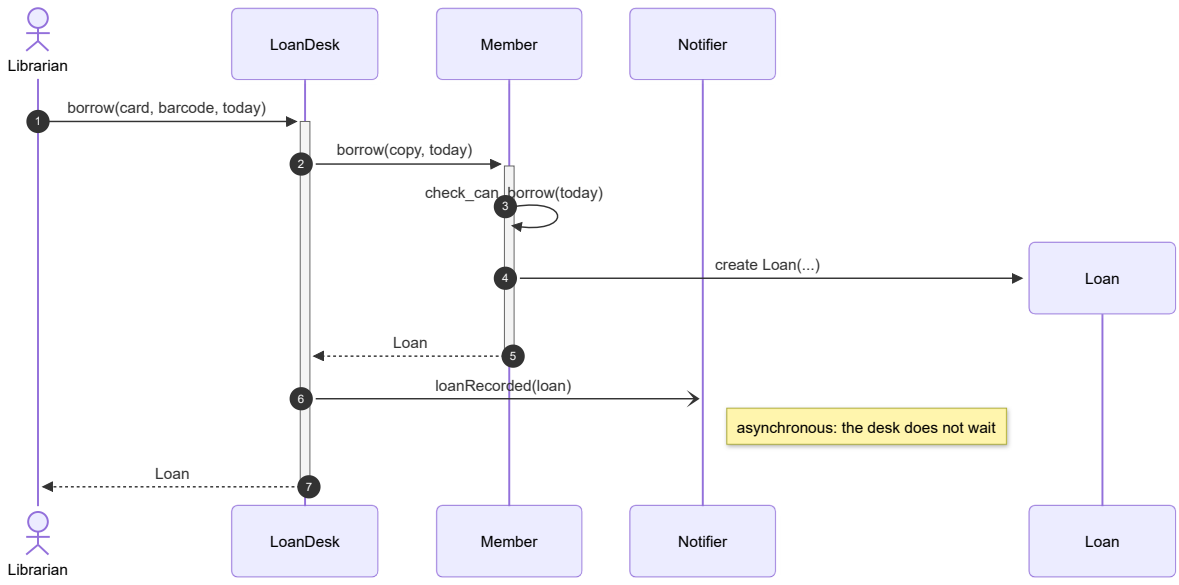
Course 1 · Chapter 7 · Sequence Diagrams

A class diagram shows who exists; it does not show how they cooperate. Chapter 6 gave the library's classes their responsibilities, but a reader still cannot see what happens when a librarian borrows a book: who is called first, who answers, and what happens when a rule says no. A **sequence diagram** shows one scenario as a conversation between objects, with time running downwards. It is the diagram that turns a use case into operations on the classes.

WHAT WAS RUN, AND WHAT WAS NOT

Every diagram was rendered here with Mermaid 12.1.0, except the one with a `ref` fragment, which Mermaid cannot draw (I tried: parse error) and which PlantUML 1.2026.8 drew. The Python (3.14) is chapter 6's version B, run for real. New in this chapter: a **call tracer I wrote** that records what the code actually does and draws it as a sequence diagram, a **conformance check** that compares a hand-drawn diagram with the code, and a check of the operations a sequence diagram needs against the class diagram. They are small scripts, shown in full, with their limits stated. **Not done:** communication, timing and interaction-overview diagrams (the paragraph on communication diagrams is from general knowledge, with nothing drawn or run); any tool that generates sequence diagrams from other languages.

Reading the Notation



Mermaid. The numbers come from `autonumber`; the Loan box appears partway down because it is created during the scenario.

```
sequenceDiagram
    autonumber
    actor Librarian
    participant LoanDesk
    participant Member
    participant Notifier
    Librarian->>+LoanDesk: borrow(card, barcode, today)
    LoanDesk->>+Member: borrow(copy, today)
    Member->>Member: check_can_borrow(today)
    create participant Loan
    Member->>Loan: create Loan(...)
    Member-->>-LoanDesk: Loan
    LoanDesk-)Notifier: loanRecorded(loan)
    Note right of Notifier: asynchronous: the desk does not wait
    LoanDesk-->>-Librarian: Loan
```

ELEMENT	LOOKS LIKE	MERMAID	MEANS
Participant, lifeline	A box with a dashed or solid line down	`participant Member`, `actor Librarian`	One object (or class, or person) taking part; time passes down the line. Name an *object's role or class*, not a method
Synchronous message	Solid line, filled head	`A->>B: op(args)`	A calls an operation of B and waits
Return	Dashed line, open head	`B-->>A: value`	The result comes back. Optional if obvious; useful when the value matters
Asynchronous message	Solid line, open head	`A-)B: event`	A sends and carries on without waiting (a notification, a queue)
Activation	A thin bar on a lifeline	`A->>+B` and `B-->>-A`	The period when B is executing because of the call
Self message	An arrow looping back	`M->>M: check(today)`	An object calling its own operation
Creation	A box appearing mid-diagram	`create participant Loan`, then a message to it	The scenario makes a new object

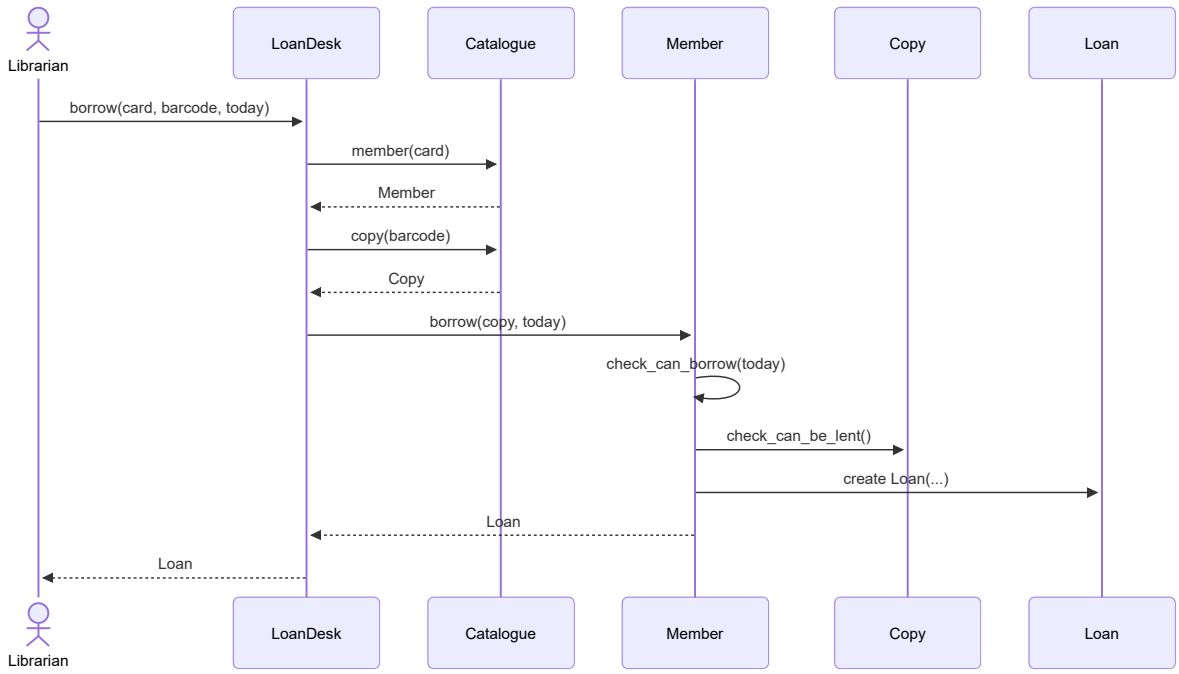
ELEMENT	LOOKS LIKE	MERMAID	MEANS
Failure	Line ending in a cross	`A--xB: Refusal`	An exception or refusal passed back (a common convention; not the only one)
Note	A yellow label	`Note right of N: text`	A comment

From Use Case to Sequence Diagram

In *Object-Oriented Design with UML* 3 the Borrow use case treated the system as a black box: "the system checks the membership...". Chapter 6 opened the box and assigned those checks to objects. A sequence diagram is where you show which object does each step. The method:

1. **Pick one scenario**: the main success scenario, or a single extension. One diagram, one path (fragments, below, fold a few alternatives in).
2. **Name the participants:** the actor, then the controller that receives the request (`LoanDesk`), then the objects the responsibilities landed on.
3. **Turn each step into a message to the object that should do it**, using chapter 6's reasoning (Expert, Creator, Controller). Show a return when the value is used later.
4. **Read the diagram out loud as the use case**, and check no step was lost.

USE CASE STEP (CHAPTER 3)	MESSAGE IN THE DIAGRAM BELOW
2. The Librarian enters the card number and ISBN (copy barcode, after chapter 4's correction)	`Librarian ->> LoanDesk: borrow(card, barcode, today)`
(finding the member and the copy: part of how the system answers)	`LoanDesk ->> Catalogue: member(card)`, `copy(barcode)`
3, 4, 5. Check membership, fines, loan limit	`Member ->> Member: check_can_borrow(today)`
6. Check this copy can be lent	`Member ->> Copy: check_can_be_lent()`
7. Record the loan, due in 21 days	`Member ->> Loan: create Loan(...)`
8. Tell the Librarian the due date	The two `Loan` returns, back to the Librarian



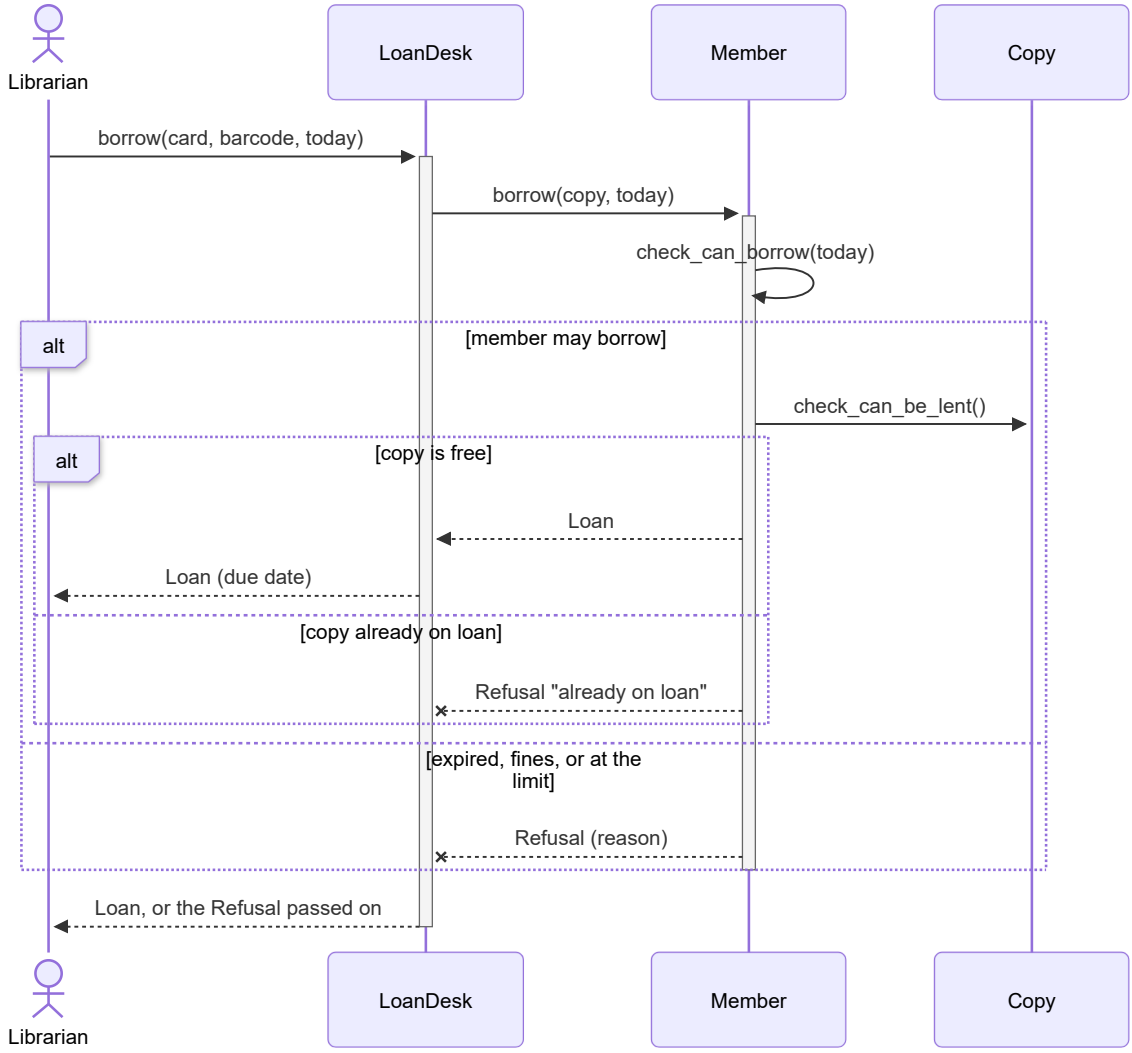
"Borrow a book", main success scenario, for chapter 6's design. Drawn by hand from chapter 6's design. (Full disclosure: I had already seen the tracer's output for this scenario when I wrote it, so it is not a blind test; exercise 1 is the blind one.)

```
sequenceDiagram
    actor Librarian
    participant LoanDesk
    participant Catalogue
    participant Member
    participant Copy
    participant Loan
    Librarian->>LoanDesk: borrow(card, barcode, today)
    LoanDesk->>Catalogue: member(card)
    Catalogue-->>LoanDesk: Member
    LoanDesk->>Catalogue: copy(barcode)
    Catalogue-->>LoanDesk: Copy
    LoanDesk->>Member: borrow(copy, today)
    Member->>Member: check_can_borrow(today)
    Member->>Copy: check_can_be_lent()
    Member->>Loan: create Loan(...)
    Member-->>LoanDesk: Loan
    LoanDesk-->>Librarian: Loan
```

Alternatives and Failure: Fragments

A **fragment** is a labelled box around some messages with a condition, so one diagram can hold several paths. The ones you will use:

FRAGMENT	MEANING	MERMAID
alt	One of several branches, chosen by a condition (if / else)	`alt ... else ... end`
opt	Happens only if the condition holds (if with no else)	`opt ... end`
loop	Repeated	`loop ... end`
par	Branches happen in parallel	`par ... and ... end`
ref	"See another diagram here": a reference, to keep one diagram small	**Not supported** (parse error when tried); PlantUML has it

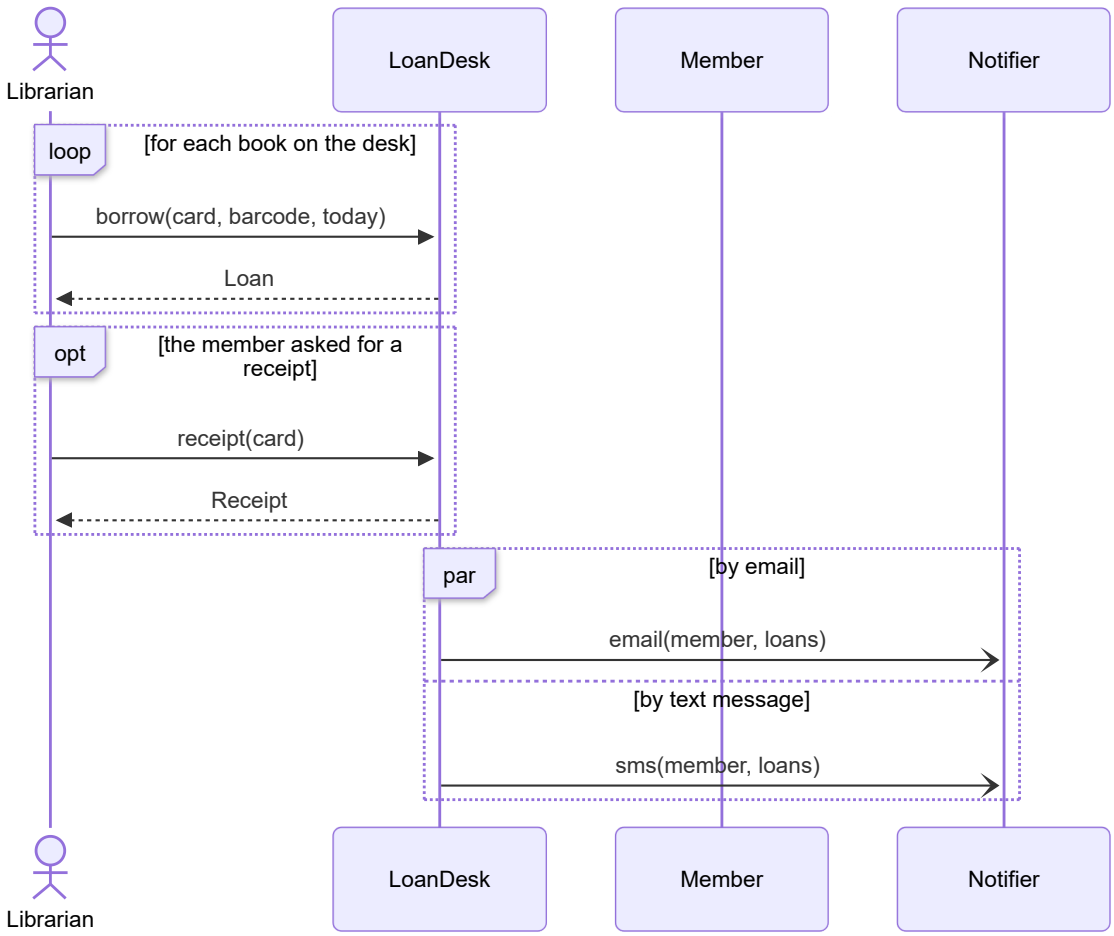


The borrow scenario with its refusals folded in: the first `alt` separates "member may borrow" from the failures the member's own checks raise, the nested one separates a free copy from one already on loan.

```
sequenceDiagram
    actor Librarian
    participant LoanDesk
    participant Member
    participant Copy
    Librarian->>+LoanDesk: borrow(card, barcode, today)
    LoanDesk->>+Member: borrow(copy, today)
    Member->>Member: check_can_borrow(today)
    alt member may borrow
        Member->>Copy: check_can_be_lent()
        alt copy is free
            Member-->>LoanDesk: Loan
            LoanDesk-->>Librarian: Loan (due date)
        else copy already on loan
            Member--xLoanDesk: Refusal "already on loan"
        end
    else expired, fines, or at the limit
        Member--xLoanDesk: Refusal (reason)
    end
    deactivate Member
    LoanDesk-->>-Librarian: Loan, or the Refusal passed on
```

Look at who throws. The refusal for an unavailable copy comes from `Copy`, not `Member`; the diagram draws it from the member's call because the member passes the exception up. I simplified it, and a stricter reader would add a second arrow. That is the usual trade: a diagram is a model, and every line you add makes it harder to read.

Loop, Opt and Par

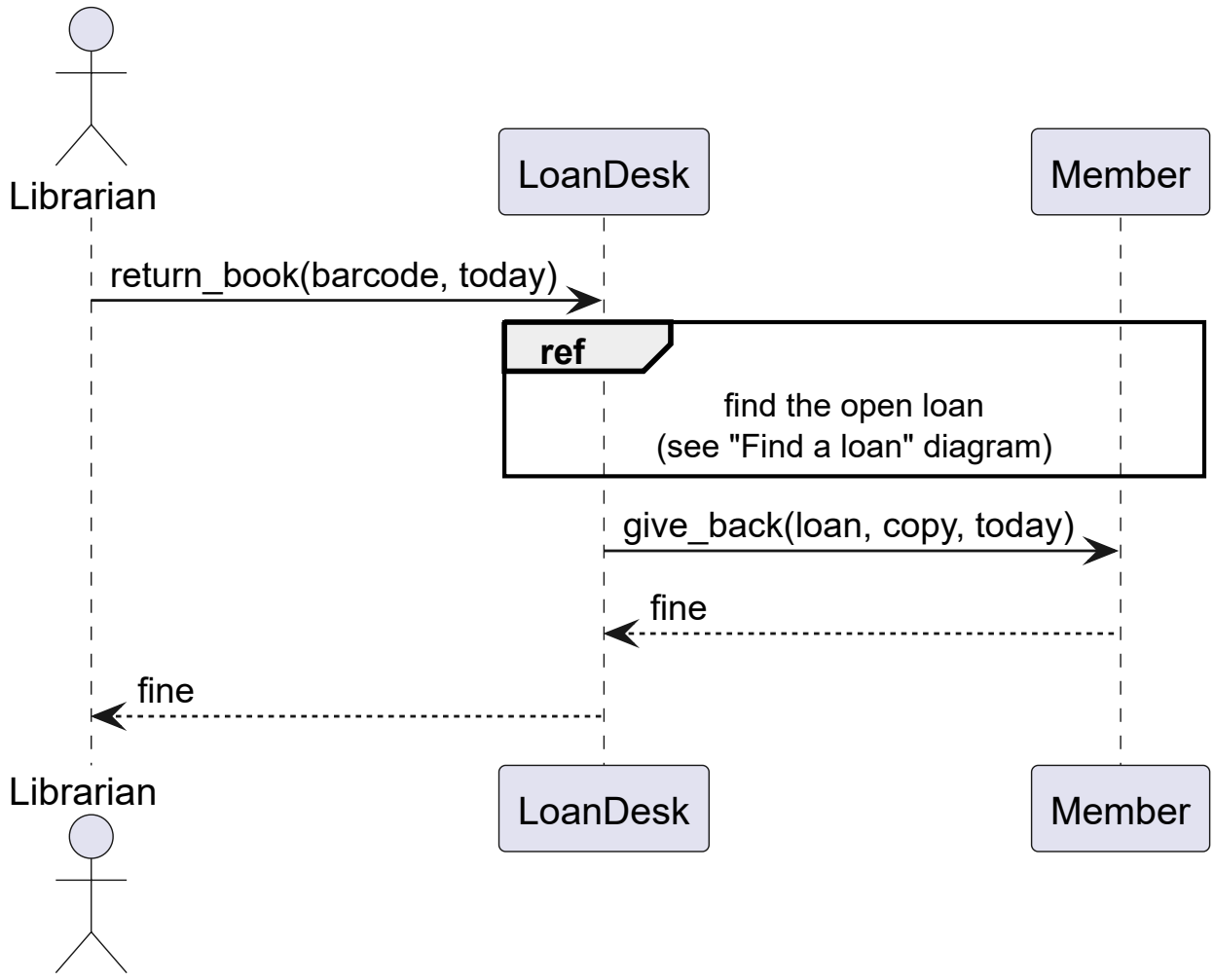


A desk session: loop over books, optional receipt, then two notifications sent in parallel (asynchronously).

```
sequenceDiagram
    actor Librarian
    participant LoanDesk
    participant Member
    participant Notifier
    loop for each book on the desk
        Librarian->>LoanDesk: borrow(card, barcode, today)
        LoanDesk-->>Librarian: Loan
    end
    opt the member asked for a receipt
        Librarian->>LoanDesk: receipt(card)
        LoanDesk-->>Librarian: Receipt
    end
    par by email
        LoanDesk-)Notifier: email(member, loans)
    and by text message
```

```
        LoanDesk-)Notifier: sms(member, loans)
    end
```

And `ref`, which only PlantUML draws. The box stands for another diagram, so this one stays short:



PlantUML. The `ref` box names a sub-scenario drawn elsewhere.

```
@startuml
scale 1.4
actor Librarian
participant LoanDesk
participant Member

Librarian -> LoanDesk : return_book(barcode, today)
ref over LoanDesk, Member : find the open loan\n(see "Find a loan" diagram)
LoanDesk -> Member : give_back(loan, copy, today)
Member --> LoanDesk : fine
```

```
  LoanDesk --> Librarian : fine
  @enduml
```

From Sequence Diagram to Operations

A message is a promise that the receiver has that operation. So reading a sequence diagram gives you the operations for the class diagram: `LoanDesk ->> Member: borrow(copy, today)` means `Member` needs `borrow(copy, today)`. This is the link between the two diagrams, and it can be checked mechanically. The script reads the Mermaid sources, lists each (receiver, operation), and compares with the operations on chapter 6's class diagram for the same design (the second half of its output):

```
"""Check the hand-drawn sequence diagram against (1) what the code really does and (2) t
import pathlib
import re

D = pathlib.Path(__file__).parent / "diagrams"
def calls(path: pathlib.Path) -> set[tuple[str, str, str]]:
    """Only the calls (solid arrows ->> and async -)), as (sender, receiver, operation n
    out = set()
    for line in path.read_text(encoding="utf8").splitlines():
        m = re.match(r"^\s*(\w+)(?:->>|-\))(\w+):\s*(.+)$", line)
        if m:
            s, r, label = m.groups()
            op = "create" if label.startswith("create ") else label.split("(")[0]
            out.add((s, r, op))
    return out


def class_operations(path: pathlib.Path) -> dict[str, set[str]]:
    ops, current = {}, None
    for line in path.read_text(encoding="utf8").splitlines():
        c = re.match(r"\s*class (\w+)", line)
        if c:
            current = c.group(1)
            ops.setdefault(current, set())
            continue
        o = re.match(r"\s*\+(\w+)\(.*\)", line)
        if o and current:
            ops[current].add(o.group(1))
    return ops
```

```
if __name__ == "__main__":
    drawn, traced = calls(D / "borrow.drawn.mmd"), calls(D / "trace_borrow_ok.mmd")
    print("1. hand-drawn 'borrow' versus what the code did (calls only)")
    print(f"   drawn: {len(drawn)} distinct calls | traced: {len(traced)} distinct calls
    for c in sorted(drawn - traced):
        print(f"   drawn but NOT in the trace : {c[0]} -> {c[1]}: {c[2]}")
    for c in sorted(traced - drawn):
        print(f"   in the trace, NOT drawn    : {c[0]} -> {c[1]}: {c[2]}")

    print()
    print("2. operations the sequence diagrams ask for, against the class diagram of cha
    ops = class_operations(D / "version_b.class.mmd")
    for name in ("borrow.drawn.mmd", "session.fragments.mmd"):
        missing = sorted((r, op) for (_, r, op) in calls(D / name) if op != "create" and
        unknown = sorted({r for (_, r, op) in calls(D / name)} - set(ops) - {"Librarian"
        print(f"   {name}")
        print(f"      operations not on the receiving class in the class diagram: {missi
        print(f"      receivers that are not classes in the class diagram        : {unkn
```

```
1. hand-drawn 'borrow' versus what the code did (calls only)
   drawn: 7 distinct calls | traced: 10 distinct calls
   in the trace, NOT drawn    : Member -> Loan: is_open
   in the trace, NOT drawn    : Member -> Member: open_loans
   in the trace, NOT drawn    : Member -> Standard: loan_limit

2. operations the sequence diagrams ask for, against the class diagram of chapter 6 (ver
   borrow.drawn.mmd
      operations not on the receiving class in the class diagram: none
      receivers that are not classes in the class diagram        : none
   session.fragments.mmd
      operations not on the receiving class in the class diagram: [('LoanDesk', 'receipt
      receivers that are not classes in the class diagram        : ['Notifier']
```

Read part 2. The borrow diagram asks for nothing the class diagram lacks. The *desk session* diagram asks for `LoanDesk.receipt` and for a `Notifier` that is not in the class diagram at all. Neither is a mistake yet; each is a decision waiting to be made, found by comparing two drawings. Either the class diagram gains a `Notifier` and a `receipt` operation, or the sequence diagram is showing something out of scope. Exercise 2 repeats this for a new diagram.

Does the Drawing Match the Code?

Part 1 of that output compares the drawn borrow diagram with a **trace of the real code**. The tracer (full source below) uses Python's `sys.settrace` to record each call into one of the traced classes, who made it (the `self` of the calling frame), and what came back, then writes a Mermaid diagram. It is a small tool and it has limits: it sees only the classes in one file, draws the outermost caller as an actor, and attributes a call to the caller's *class*, not to the particular object.

```
"""Record what really happens when a function runs, and draw it as a Mermaid sequence di

Uses sys.settrace. Only frames in the traced module's classes are recorded. A call is dr
(the 'self' of the calling frame) to the object whose method was entered. The outermost
"""
import sys
from dataclasses import dataclass


@dataclass
class Event:
    kind: str                 # call | return | raise
    caller: str
    callee: str
    label: str
    depth: int = 0            # how deep in the call stack the callee is (1 = called by


def trace(module_file: str, func, *args, actor: str = "Librarian"):
    events: list[Event] = []
    stack: list[str] = [actor]                    # class names of the active frames w

    def owner(frame) -> str | None:
        obj = frame.f_locals.get("self")
        if obj is not None:
            return type(obj).__name__
        # a dataclass-generated __init__ or a plain function: the code object's qualifie
        qual = frame.f_code.co_qualname
        return qual.split(".")[0] if "." in qual and not qual.startswith("<") else None

    def global_tracer(frame, event, arg):
        if event != "call":
            return None
        self_obj = frame.f_locals.get("self")
        from_module = frame.f_code.co_filename.endswith(module_file)
```

```
        # a dataclass writes __init__ with exec(), so its code object has no file name: r
        generated_init = frame.f_code.co_name == "__init__" and self_obj is not None and
        if not (from_module or generated_init):
            return None
        who = owner(frame)
        if who is None:
            return None
        name = frame.f_code.co_name
        argnames = frame.f_code.co_varnames[1: frame.f_code.co_argcount] if frame.f_local
        # for a dataclass __init__, 'self' is also first
        if name == "__init__":
            argnames = frame.f_code.co_varnames[1: frame.f_code.co_argcount]
            label = f"create {who}({', '.join(argnames)})"
        else:
            label = f"{name}({', '.join(argnames)})"
        events.append(Event("call", stack[-1], who, label, len(stack)))
        stack.append(who)

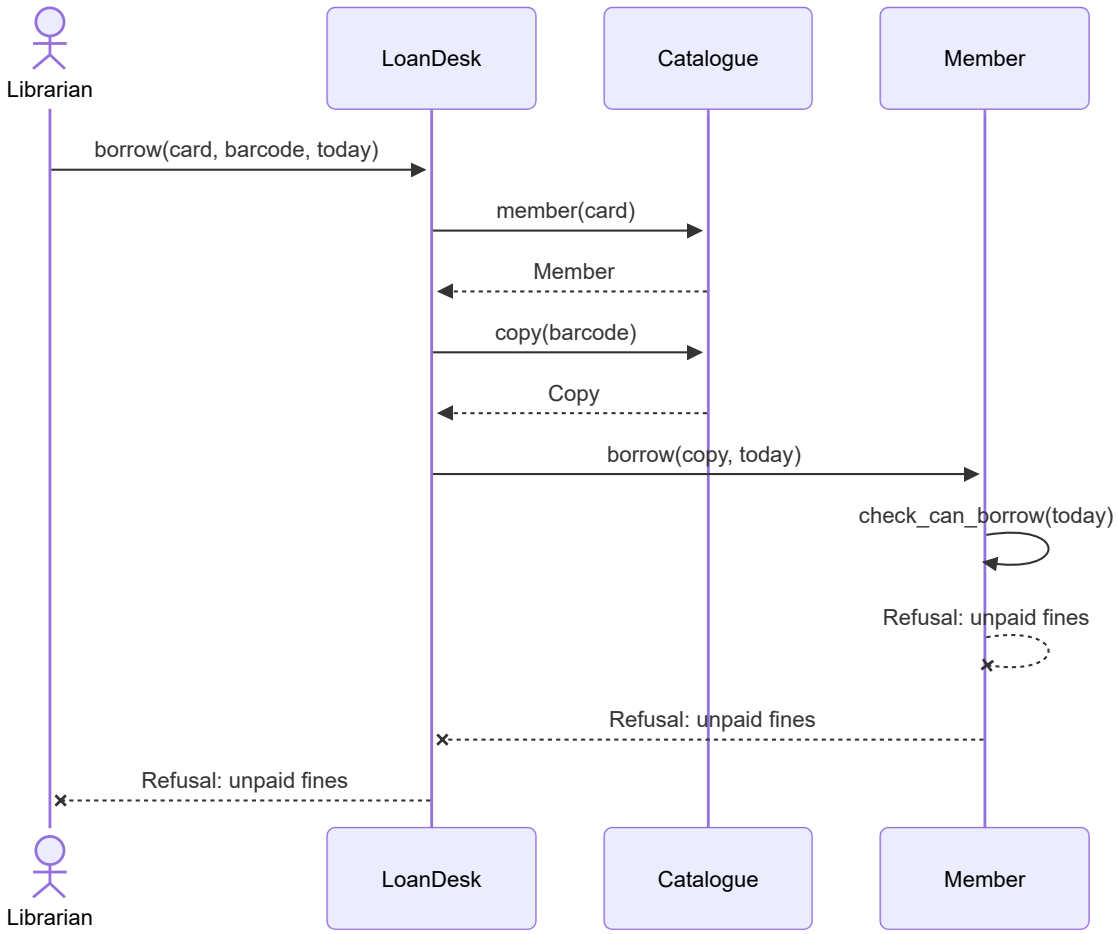
        def local_tracer(frame, event, arg):
            if event == "exception":
                local_tracer.exc = arg[1]
            elif event == "return":
                caller = stack[-2] if len(stack) > 1 else actor
                exc = getattr(local_tracer, "exc", None)
                if arg is None and exc is not None:
                    events.append(Event("raise", who, caller, f"{type(exc).__name__}: {ex
                elif arg is not None and name != "__init__":
                    events.append(Event("return", who, caller, type(arg).__name__, len(st
                stack.pop()
            return local_tracer
        local_tracer.exc = None
        return local_tracer

    sys.settrace(global_tracer)
    try:
        result = func(*args)
        error = None
    except Exception as e:      # the traced code's own refusals
        result, error = None, e
    finally:
        sys.settrace(None)
    return events, result, error


def to_mermaid(events: list[Event], actor: str = "Librarian", skip: set[str] | None = Non
```

```
    skip = skip or set()
    order = [actor]
    for e in events:
        for who in (e.caller, e.callee):
            if who not in order:
                order.append(who)
    lines = ["sequenceDiagram"]
    for who in order:
        lines.append(f"    actor {who}" if who == actor else f"    participant {who}")
    for e in events:
        if any(s in e.label for s in skip):
            continue
        if e.kind == "call":
            lines.append(f"    {e.caller}->>{e.callee}: {e.label}")
        elif e.kind == "return":
            lines.append(f"    {e.caller}-->>{e.callee}: {e.label}")
        else:
            lines.append(f"    {e.caller}--x{e.callee}: {e.label}")
    return "\n".join(lines) + "\n"
```

The refused-borrow scenario (the member has unpaid fines), as the code actually ran it:



Generated from a real run of chapter 6's `library_b.py`; I wrote none of these arrows. The refusal travels back up the chain: from the member, to the desk, to the librarian.

The comparison in part 1 of the output found three calls in the trace that the hand-drawn diagram omits: `Member -> Member.open_loans`, `Member -> Loan.is_open` and `Member -> Standard.loan_limit`. The first two are helpers, and leaving them out is a legitimate choice. **The third deserves a second look**: it is the call to the member's kind, the polymorphic collaboration chapter 6 built (the loan limit differs by kind of member). My drawing left out the one call that carries the design's main idea. Detail is a choice you make per audience; just notice what you are choosing to hide.

The Diagram as a Test

Once the hidden calls are agreed, the comparison can be a test: do the code's calls, filtered to the diagram's level of detail, equal the diagram's calls in the same order? The check below runs it on the real code, and on a version of the code where I made a plausible refactoring (looking the copy up before the member):

```
"""A sequence diagram as a test: does the code make the calls the diagram shows, in the o

The drawn diagram is deliberately less detailed than the code, so the trace is filtered t
helper calls the diagram chooses not to show (open_loans, is_open, and the call to the m
"""
import difflib
import pathlib
import re
import types
from datetime import date

from tracetool import to_mermaid, trace

HERE = pathlib.Path(__file__).parent
HIDDEN = {"open_loans", "is_open", "loan_limit"}          # abstraction chosen by the di


def ordered_calls(text: str) -> list[str]:
    out = []
    for line in text.splitlines():
        m = re.match(r"^\s*(\w+)->>(\w+):\s*(.+)$", line)
        if m:
            sender, receiver, label = m.groups()
            op = "create" if label.startswith("create ") else label.split("(")[0]
            if op not in HIDDEN:
                out.append(f"{sender} -> {receiver}.{op}")
    return out


def load(source: str):
    """Run library_b's source as a module with the same file name, so the tracer treats i
    module = types.ModuleType("library_b")
    exec(compile(source, str(HERE / "library_b.py"), "exec"), module.__dict__)
    return module


def check(name: str, module) -> bool:
    desk = module.LoanDesk([module.Member("M1", date(2027, 1, 1), module.KINDS["standard
    events, result, error = trace("library_b.py", desk.borrow, "M1", "B0", date(2026, 10
    actual = ordered_calls(to_mermaid(events))
    drawn = ordered_calls((HERE / "diagrams" / "borrow.drawn.mmd").read_text(encoding="ut
    if actual == drawn:
        print(f"{name}: the code makes the {len(drawn)} calls the diagram shows, in the s
        return True
```

```
        print(f"{name}: DOES NOT MATCH the diagram ('-' drawn, '+' actual)")
        for line in difflib.unified_diff(drawn, actual, lineterm="", n=0):
            if not line.startswith(("---", "+++", "@@")):
                print("   ", line)
        return False


    if __name__ == "__main__":
        source = (HERE / "library_b.py").read_text(encoding="utf8")
        check("code as written     ", load(source))
        # a plausible refactoring: look the copy up before the member
        old = "        return self.catalogue.member(card).borrow(self.catalogue.copy(barcode
        new = "        copy = self.catalogue.copy(barcode)\n        return self.catalogue.me
        assert source.count(old) == 1
        check("copy looked up first", load(source.replace(old, new)))
```

```
code as written     : the code makes the 7 calls the diagram shows, in the same order
copy looked up first: DOES NOT MATCH the diagram ('-' drawn, '+' actual)
    +LoanDesk -> Catalogue.copy
    -LoanDesk -> Catalogue.copy
```

The refactoring is flagged, correctly in a literal sense and unhelpfully in a practical one: the two lookups are independent, and the order does not matter. That is the cost of a strict conformance test, so use it for sequences where order *is* the point (a protocol, a payment flow), not for every diagram. Be clear about what it cannot do: it checks the calls this one scenario makes, so it says nothing about the other paths, and a passing check does not make the design good.

When a Sequence Diagram Is Too Big

A faithful trace of one short desk session (three borrows and one return) is this big:

```
one desk session (3 borrows + 1 return): 41 calls, 73 arrows in all, fine 30p
for comparison, the hand-drawn 'borrow' diagram has 11 arrows (7 of them calls)
rendered size (width x height, px): 1568.5 x 3759
```

Seventy-three arrows, over 3,700 pixels tall: nobody will read it. The hand-drawn diagram has 11 arrows because its author decided what mattered. The same trace drawn at four levels of detail

(the call depth kept, from the exercise 3 output):

```
full trace: 73 arrows (41 calls)
detail kept   arrows   what a reader sees
  depth <= 1       8   the desk's four requests only
  depth <= 2      32   plus what each request asks of the catalogue and member
  depth <= 3      45   plus the member's own checks and loan lookups
  depth <= 9      73   everything
```

The ways to stay readable:

- **One scenario per diagram.** Borrow, and return, and refusals, not a whole session.
- **Choose the level.** Showing only the first two levels of calls (32 arrows here) tells the story; the third level is for the person implementing the member's checks.
- **Hide helpers, show collaborations.** A call that carries a design decision stays; a loop's housekeeping goes.
- **Use** `ref` (in PlantUML) to name a sub-scenario and draw it once.
- **Fewer participants.** If five objects are not needed to make the point, leave them out and say so.
- **Prefer a different diagram** when the point is the set of states (chapter 8) or the shape of a workflow (activity diagram) rather than who calls whom.

Communication Diagrams, in a Paragraph

A **communication diagram** (called a collaboration diagram in UML 1) carries the same information as a sequence diagram but draws the objects as boxes joined by lines, with numbered messages along the lines instead of a time axis. It shows *which objects are linked* more clearly and the order less clearly. It is a sensible choice for a small group of objects where the links matter. I did not draw or render one for this chapter, and this paragraph is from general knowledge; in practice most teams use sequence diagrams.

Usual Mistakes

MISTAKE	WHY IT HURTS	FIX
Drawing every call	Unreadable, and out of date within a week	One scenario; hide helpers; choose a level
Messages that are not operations	"Check everything" is not a method the receiver can have	Name each message as the operation the receiver will implement
Everything sent to one "manager" object	The anemic service of chapter 6 again, drawn as a star	If one lifeline receives every message, ask who has the data (Expert)
Arrows pointing the wrong way	The arrow goes from the caller to the object that has the operation, not from data to its user	Read each arrow as "A asks B to..."
No failure path	The diagram shows the sunny day the use case already described	Add the extensions that matter as `alt`
A diagram nobody checks against code	It describes the design as intended, not as built	Trace, or review with the code open; date the diagram

MISTAKES WHILE PREPARING THIS CHAPTER

- **My own probe of Mermaid was wrong.** I tested several features in one diagram, and the first run failed with "Trying to inactivate an inactive participant". I had written `M-->>-D` (which deactivates M) after activating D. The error was mine, not Mermaid's, and I nearly concluded a feature was missing. I fixed the diagram and re-ran: create, async, activation, loop, opt and par all work; only `ref` fails. Takeaway: test one new thing at a time, and make sure a failure is caused by the thing you are testing.
- **The tracer silently missed two things on its first run.** It drew no creation of a `Loan` (a dataclass builds its `__init__` with `exec`, so the function has no file name and my filter rejected it) and no refusal arrows (I tested whether an exception was in flight with `sys.exc_info`, which apparently did not report one inside a trace callback; removing that test fixed it). I only noticed because I read the generated diagram instead of trusting the count. Takeaway: a recorder that is wrong looks like a quiet program; read its output against a case where you know the answer.
- **A number I typed instead of computing.** My first draft said the hand-drawn diagram has 8 arrows. It has 11. I had written the figure into a print statement from memory. The script now counts the lines.
- **A scripted edit that corrupted its own file, for the third time.** I patched `conformance.py` with a Python script containing `\n` inside a string that was itself inside a heredoc; the newlines became real line breaks and the file would not parse (`SyntaxError: unterminated string literal`). I had

made the same mistake twice before, while writing the chatbot course. How to avoid it: for edits that contain escape sequences, use the editor's replace tool, and run the file immediately after any scripted edit.

Hands-On Exercises

EXERCISE 1

Draw the sequence diagram for "Return a Book" (the main scenario, with a late return) in Mermaid, using chapter 6's design, *from reading the code, without running it*. Then check it against the real code with the tracer and say what differed, if anything.

View solution

EXERCISE 2

Draw "send overdue notices each night": a timer triggers the desk; for each overdue loan an email is sent to the member asynchronously and, only if the member has a mobile number, a text as well. Then check which operations the diagram needs that the class diagram lacks.

View solution

EXERCISE 3

Trace a desk session (three borrows, one return) and draw it at four levels of detail by limiting the call depth. Report how many arrows each has, and say which level you would put in a design document and which you would never publish.

View solution

CHAPTER 7 QUICK REFERENCE

- Participants across the top, time down; `->>` call, `-->>` return, `-)` asynchronous, activation bars, self messages, creation, a cross for a failure

- Method: one scenario; name actor, controller and the objects that hold the responsibilities; each step becomes a message to the object that should do it; read it aloud as the use case
- Fragments: `alt`, `opt`, `loop`, `par` in Mermaid; `ref` only in PlantUML (Mermaid 12.1.0 fails on it)
- Every message is an operation on the receiver: compare the diagram with the class diagram to find missing operations and missing classes
- Check a diagram against the code by tracing (`sys.settrace`); a diagram can be a test, but order-strict tests flag harmless refactorings
- Too big? One scenario, a chosen level of detail (here 73 arrows down to 32 at two levels), hide helpers but not design-defining calls, use `ref`, fewer participants
- Communication diagrams show the same facts around links rather than time (not drawn here)

COMING NEXT

Object-Oriented Design with UML 8 (State Machines and Activity Diagrams): states, events, guards, actions, entry and exit, composite states and history; the lifecycle of a loan as a state machine, implemented and tested; the link to the State pattern in *Design Patterns* 8; activity diagrams with decisions, fork and join and swimlanes, and modelling reservation handling.

Object-Oriented Design with UML

Course 1 · Chapter 8 · State Machines and Activity Diagrams

Class diagrams show structure and sequence diagrams show one conversation. Two questions remain. **What can happen to a single object over its life, and in what order?** (A loan is borrowed, may be renewed, becomes overdue, is returned or lost.) That is a **state machine**. And: **what are the steps of a process that crosses several people and systems?** (A reserved book comes back; who does what, and in what order, and what runs in parallel?) That is an **activity diagram**.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> PlantUML 1.2026.8 drew the state and activity diagrams; Mermaid 12.1.0 drew a flat version of the state diagram and was probed for the rest. Python 3.14 runs everything else: the loan lifecycle written twice (a transition table, and the State pattern), 28 tests that run against both, a transition matrix, and a reservation workflow with 10 tests. **The tools' fidelity is part of the story:** I tested what Mermaid does with history states and entry/exit actions, and the answer is that it *accepts them without error and draws them wrongly*. That is why the full diagram is PlantUML. **From general knowledge, not re-read:** the UML rule that an event which triggers no transition is simply discarded, how entry actions behave when a state is re-entered through history, and the token semantics of activity diagrams (not covered here).

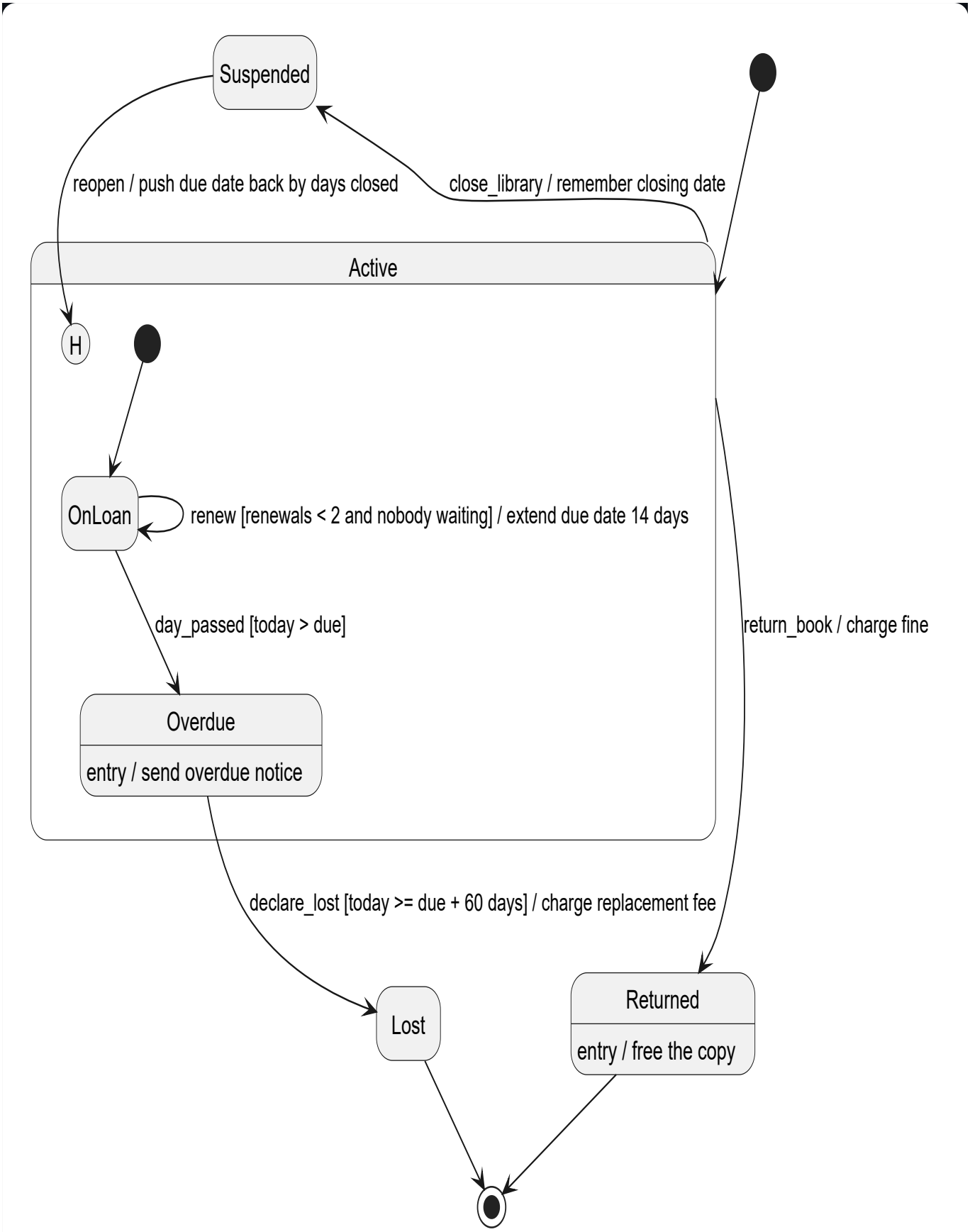
State Machines: The Vocabulary

TERM	NOTATION	MEANING	IN THE LOAN LIFECYCLE
State	Rounded box	A condition an object stays in, in which it reacts to some events and not others	OnLoan, Overdue, Suspended, Returned, Lost
Event	Label on an arrow	Something that happens and may cause a transition	`return_book`, `renew`, `day_passed`
Transition	Arrow	A move from one state to another when an event occurs	OnLoan to Overdue on `day_passed`
Guard	`[condition]`	A test that must be true for the transition to fire	`[today > due]`

TERM	NOTATION	MEANING	IN THE LOAN LIFECYCLE
Action	`/ action` on an arrow	Work done during the transition	`/ charge fine`
Entry and exit action	`entry / ...` inside the box	Work done every time the state is entered or left, however you got there	Overdue: `entry / send overdue notice`
Initial and final	Filled dot; circled dot	Where the object starts; where its life ends	Starts OnLoan; ends Returned or Lost
Composite state	A box containing states	A group of states that share transitions	`Active` holds OnLoan and Overdue
History state	Circled **H**	"Go back to whichever substate you were in last"	Reopening the library resumes OnLoan or Overdue

A Loan's Life

The rules, from earlier chapters and a few new ones: a loan starts **OnLoan**. It can be **renewed** for 14 days up to twice, unless someone else is waiting. When the due date passes it becomes **Overdue**, and an overdue notice goes out. It can be **returned** from either of those, with a fine of 10p a day if late. An overdue loan 60 days late can be **declared lost**, for a 25-pound replacement fee. If the library **closes** (a snow day, say), the loan is **Suspended**, and when it reopens the loan goes back to what it was, with the due date pushed back by the days closed.



PlantUML. **Generated from the transition table** in `loan_sm.py` (below), not drawn by hand. `Active` is a composite state: `return_book` and `close_library` are drawn once, from the box, and apply to both OnLoan and Overdue. The **H** is the history state.

```
@startuml
scale 1.3
hide empty description
[*] --> Active
state Active {
  [*] --> OnLoan
  state OnLoan
  state Overdue
  Overdue : entry / send overdue notice
}
Returned : entry / free the copy
OnLoan --> Overdue : day_passed [today > due]
OnLoan --> OnLoan : renew [renewals < 2 and nobody waiting] / extend due date 14 days
Active --> Returned : return_book / charge fine
Overdue --> Lost : declare_lost [today >= due + 60 days] / charge replacement fee
Active --> Suspended : close_library / remember closing date
Suspended --> Active[H] : reopen / push due date back by days closed
Returned --> [*]
Lost --> [*]
@enduml
```

Three things the diagram makes visible that a paragraph hides. **The composite state saves arrows:** without `Active`, `return_book` and `close_library` would each need two arrows. **History is the right tool for "go back":** reopening does not need to know whether the loan was overdue. **Nothing leaves Returned or Lost** except to the final dot: a returned loan cannot be renewed, which is a rule you can see. What the diagram does *not* show: events that are deliberately ignored (the daily clock tick in Overdue, Suspended, Returned and Lost) are conventionally left off, which is a reason to have the matrix below.

The Same Loan in Code, From a Table

A state machine is data: a list of (source state, event, guard, action, target). Writing the code as that list makes the diagram and the code the same thing, because the diagram above was generated from the list. Guards and actions are named, so the diagram text and the code cannot drift.

```
"""The lifecycle of a library loan as a table-driven state machine.

States: OnLoan and Overdue (both inside the composite state Active), Suspended, Returned
```

```
Events: renew, day_passed, return_book, declare_lost, close_library, reopen.
The transition table is the single source: the diagram is generated from it, and so is t
"""
from dataclasses import dataclass
from datetime import date, timedelta

FINE_PER_DAY = 10
LOST_AFTER_DAYS = 60
LOST_FEE = 2500
MAX_RENEWALS = 2
RENEWAL_DAYS = 14

COMPOSITES = {"Active": ["OnLoan", "Overdue"]}
INITIAL = "OnLoan"


class IllegalTransition(Exception):
    pass


@dataclass
class Loan:
    due_on: date
    reserved_by_other: bool = False        # someone else is waiting for this title
    renewals: int = 0
    notices_sent: int = 0
    fine_pence: int = 0
    copy_freed: bool = False
    closed_on: date | None = None


@dataclass(frozen=True)
class Transition:
    source: str                            # a leaf state, or a composite (applies to al
    event: str
    target: str                            # a state, or 'Active.history'
    guard: str = ""                        # shown on the diagram
    action: str = ""                       # shown on the diagram


def leaves(state: str) -> list[str]:
    return COMPOSITES.get(state, [state])


# --- guards, actions, entry and exit, by name, so the diagram text and the code cannot
```

```
def guard(name: str, loan: Loan, today: date | None) -> bool:
    if name == "":
        return True
    if name == "today > due":
        return today > loan.due_on
    if name == "renewals < 2 and nobody waiting":
        return loan.renewals < MAX_RENEWALS and not loan.reserved_by_other
    if name == "today >= due + 60 days":
        return today >= loan.due_on + timedelta(days=LOST_AFTER_DAYS)
    raise KeyError(name)


def action(name: str, loan: Loan, today: date | None) -> None:
    if name == "":
        return
    if name == "extend due date 14 days":
        loan.due_on += timedelta(days=RENEWAL_DAYS)
        loan.renewals += 1
    elif name == "charge fine":
        loan.fine_pence += max(0, (today - loan.due_on).days) * FINE_PER_DAY
    elif name == "charge replacement fee":
        loan.fine_pence += LOST_FEE
    elif name == "remember closing date":
        loan.closed_on = today
    elif name == "push due date back by days closed":
        loan.due_on += today - loan.closed_on
        loan.closed_on = None
    else:
        raise KeyError(name)


ENTRY = {"Overdue": "send overdue notice", "Returned": "free the copy"}
EXIT = {"Suspended": "(none)"}


def do_entry(state: str, loan: Loan) -> None:
    if state == "Overdue":
        loan.notices_sent += 1
    elif state == "Returned":
        loan.copy_freed = True


TRANSITIONS = [
    Transition("OnLoan", "day_passed", "Overdue", guard="today > due"),
    Transition("OnLoan", "renew", "OnLoan", guard="renewals < 2 and nobody waiting", act
```

```
    Transition("Active", "return_book", "Returned", action="charge fine"),
    Transition("Overdue", "declare_lost", "Lost", guard="today >= due + 60 days", action=
    Transition("Active", "close_library", "Suspended", action="remember closing date"),
    Transition("Suspended", "reopen", "Active.history", action="push due date back by da
]
IGNORED = {("Overdue", "day_passed"), ("Suspended", "day_passed"), ("Returned", "day_pas
STATES = ["OnLoan", "Overdue", "Suspended", "Returned", "Lost"]
EVENTS = ["renew", "day_passed", "return_book", "declare_lost", "close_library", "reopen


class LoanMachine:
    def __init__(self, loan: Loan):
        self.loan = loan
        self.state = INITIAL
        self._last_active = INITIAL          # the history of the composite state Active

    def send(self, event: str, today: date | None = None) -> str:
        if (self.state, event) in IGNORED:
            return self.state
        candidates = [t for t in TRANSITIONS if t.event == event and self.state in leave
        if not candidates:
            raise IllegalTransition(f"{event} is not allowed in state {self.state}")
        for t in candidates:
            if guard(t.guard, self.loan, today):
                self._take(t, today)
                return self.state
        return self.state                    # guards all false: the event is discarded

    def _take(self, t: Transition, today: date | None) -> None:
        if self.state in COMPOSITES["Active"]:
            self._last_active = self.state   # remember where we were inside Active
        action(t.action, self.loan, today)
        self.state = self._last_active if t.target == "Active.history" else t.target
        do_entry(self.state, self.loan)


def matrix() -> dict[tuple[str, str], str]:
    """For each state and event: what happens, with the probe arguments chosen so that e
    due = date(2026, 11, 1)
    args = {"renew": None, "day_passed": due + timedelta(days=1), "return_book": due + t
            "close_library": due + timedelta(days=1), "reopen": due + timedelta(days=2)}
    out = {}
    for s in STATES:
        for e in EVENTS:
            m = machine_in(s, due)
```

```
            try:
                out[(s, e)] = m.send(e, args[e]) if (s, e) not in IGNORED else "ignored"
            except IllegalTransition:
                out[(s, e)] = "illegal"
    return out


def machine_in(state: str, due: date) -> LoanMachine:
    """Drive a fresh machine into a state by legal events."""
    m = LoanMachine(Loan(due))
    if state == "Overdue":
        m.send("day_passed", due + timedelta(days=1))
    elif state == "Suspended":
        m.send("close_library", due - timedelta(days=5))
    elif state == "Returned":
        m.send("return_book", due)
    elif state == "Lost":
        m.send("day_passed", due + timedelta(days=1))
        m.send("declare_lost", due + timedelta(days=61))
    return m


def to_mermaid() -> str:
    """A flat Mermaid state diagram (no composite, no history: see the chapter for why).
    lines = ["stateDiagram-v2", f"    [*] --> {INITIAL}"]
    for t in TRANSITIONS:
        label = t.event + (f" [{t.guard}]" if t.guard else "") + (f" / {t.action}" if t.
        target = "OnLoan" if t.target == "Active.history" else t.target
        for src in leaves(t.source):
            note = " (to the state it left)" if t.target == "Active.history" else ""
            lines.append(f"    {src} --> {target} : {label}{note}")
    lines += ["    Returned --> [*]", "    Lost --> [*]"]
    return "\n".join(lines) + "\n"


def to_plantuml() -> str:
    lines = ["@startuml", "scale 1.3", "hide empty description", "[*] --> Active", "state
    for s in COMPOSITES["Active"]:
        lines.append(f"  state {s}")
        if s in ENTRY:
            lines.append(f"  {s} : entry / {ENTRY[s]}")
    lines.append("}")
    for s, text in ENTRY.items():
        if s not in COMPOSITES["Active"]:
            lines.append(f"{s} : entry / {text}")
```

```
    for t in TRANSITIONS:
        label = t.event + (f" [{t.guard}]" if t.guard else "") + (f" / {t.action}" if t.
        target = "Active[H]" if t.target == "Active.history" else t.target
        src = t.source
        lines.append(f"{src} --> {target} : {label}")
    lines += ["Returned --> [*]", "Lost --> [*]", "@enduml"]
    return "\n".join(lines) + "\n"
```

Four decisions in that code are decisions you must make for any state machine in code, because the notation does not make them for you:

- **An event that fits no transition in the current state.** UML's rule, as I remember it, is that such an event is simply discarded. In code that hides bugs, so I made it an **error** (`IllegalTransition`) unless I list it as deliberately **ignored** (the clock tick in four states).
- **A guard that is false.** The event is discarded: `renew` after two renewals changes nothing, quietly. (Whether that should instead tell the member is a business question; the machine reports the same state.)
- **Where history is remembered.** `_last_active` records which substate of Active we left. It is the code for the H.
- **What entry means when you come back through history.** Re-entering Overdue runs its entry action again, so a member whose library closed for two days gets *a second overdue notice* when it reopens. A test pins that behaviour down, so it is a visible decision rather than an accident (see below). Whether it is desirable is for the librarian.

Every State, Every Event: The Matrix

A diagram shows the transitions that exist. A **transition matrix** shows what happens for *every* state and event, including the ones that do not exist. It is computed by driving a fresh machine into each state and sending each event (with arguments chosen so each guard is satisfied):

```
from the transition table
             renew          day_passed     return_book    declare_lost   close_library
  OnLoan     OnLoan         Overdue        Returned       illegal        Suspended
  Overdue    illegal        ignored        Returned       Lost           Suspended
  Suspended  illegal        ignored        illegal        illegal        illegal
  Returned   illegal        ignored        illegal        illegal        illegal
  Lost       illegal        ignored        illegal        illegal        illegal
```

```
the State-pattern implementation, driven into every state: identical to the table
```

Read the empty-looking cells as design: `illegal` everywhere in Returned and Lost except the ignored clock tick (a finished loan accepts nothing else), `declare_lost` only from Overdue, `reopen` only from Suspended, `renew` only from OnLoan. Two caveats: the guards are all assumed true in this table (a guard that is false in a given case discards the event, as the tests below show), and the `reopen` cell says OnLoan because the matrix reached Suspended from OnLoan; reached from Overdue it returns to Overdue.

The Same Loan With the State Pattern

Design Patterns 8 teaches the State pattern; here is the same lifecycle written that way, as one class per state with each handling the events it allows, so you can compare. The behaviour is checked by the same tests, below.

```
"""The same loan lifecycle written with the State pattern: one class per state, each han
Same public behaviour as loan_sm.LoanMachine (the shared tests and the matrix check prov
from datetime import date, timedelta

from loan_sm import FINE_PER_DAY, LOST_AFTER_DAYS, LOST_FEE, MAX_RENEWALS, RENEWAL_DAYS,


class State:
    name = "?"

    def renew(self, m, today=None):
        raise IllegalTransition(f"renew is not allowed in state {self.name}")

    def day_passed(self, m, today=None):
        raise IllegalTransition(f"day_passed is not allowed in state {self.name}")

    def return_book(self, m, today=None):
        raise IllegalTransition(f"return_book is not allowed in state {self.name}")

    def declare_lost(self, m, today=None):
        raise IllegalTransition(f"declare_lost is not allowed in state {self.name}")

    def close_library(self, m, today=None):
```

```
        raise IllegalTransition(f"close_library is not allowed in state {self.name}")

    def reopen(self, m, today=None):
        raise IllegalTransition(f"reopen is not allowed in state {self.name}")

    def enter(self, m):
        pass


class Active(State):
    """Behaviour shared by OnLoan and Overdue: the composite state."""

    def return_book(self, m, today=None):
        m.loan.fine_pence += max(0, (today - m.loan.due_on).days) * FINE_PER_DAY
        m.go(Returned())

    def close_library(self, m, today=None):
        m.last_active = self
        m.loan.closed_on = today
        m.go(Suspended())


class OnLoan(Active):
    name = "OnLoan"

    def renew(self, m, today=None):
        if m.loan.renewals < MAX_RENEWALS and not m.loan.reserved_by_other:
            m.loan.due_on += timedelta(days=RENEWAL_DAYS)
            m.loan.renewals += 1

    def day_passed(self, m, today=None):
        if today > m.loan.due_on:
            m.go(Overdue())


class Overdue(Active):
    name = "Overdue"

    def enter(self, m):
        m.loan.notices_sent += 1

    def day_passed(self, m, today=None):
        pass                                                # still overdue: nothing

    def declare_lost(self, m, today=None):
```

```
        if today >= m.loan.due_on + timedelta(days=LOST_AFTER_DAYS):
            m.loan.fine_pence += LOST_FEE
            m.go(Lost())


class Suspended(State):
    name = "Suspended"

    def day_passed(self, m, today=None):
        pass

    def reopen(self, m, today=None):
        m.loan.due_on += today - m.loan.closed_on
        m.loan.closed_on = None
        m.go(m.last_active)                                  # history: back to the st


class Returned(State):
    name = "Returned"

    def day_passed(self, m, today=None):
        pass

    def enter(self, m):
        m.loan.copy_freed = True


class Lost(State):
    name = "Lost"

    def day_passed(self, m, today=None):
        pass


class LoanStateMachine:
    def __init__(self, loan: Loan):
        self.loan = loan
        self.current = OnLoan()
        self.last_active = self.current

    @property
    def state(self) -> str:
        return self.current.name

    def go(self, new: State) -> None:
```

```
        self.current = new
        new.enter(self)

    def send(self, event: str, today: date | None = None) -> str:
        getattr(self.current, event)(self, today)
        return self.state
```

	TRANSITION TABLE	STATE PATTERN (ONE CLASS PER STATE)
Diagram to code	Direct: the rows are the arrows, and the diagram and matrix are generated from them	Indirect: the arrows are spread across methods, and there is no generated diagram
Behaviour that differs a lot by state	Awkward: actions are named strings dispatched by `if`	Natural: each state class holds its own code, and can have its own data
Adding a state	New rows, plus any new action code	A new class, and a new method wherever the other states must handle its events
Forgetting a case	Visible in the matrix, and unhandled events raise an error	The base class raises an error for anything not overridden; the *ignored* events must be remembered in each class
Fits when	Many states and events, little behaviour per state; the diagram matters	A few states with a lot of behaviour each

Neither is "the" answer. The table suits this loan, where the transitions carry the interest. I have not benchmarked either; the comparison is of what the code is like to change.

One Test Suite, Two Implementations, and What It Caught

28 tests (14 behaviours run against each implementation) cover every transition, the guards, the ignored events, the history cases, and the illegal ones. The first run did not pass:

```
ERROR: test_becoming_overdue_sends_exactly_one_notice (test_loan_states.TableDriven.test_
    raise IllegalTransition(f"{event} is not allowed in state {self.state}")
loan_sm.IllegalTransition: day_passed is not allowed in state Overdue
Ran 28 tests in 0.002s
FAILED (errors=1)
```

```
the State-pattern implementation, driven into every state: DIFFERENCES
   in Overdue, event day_passed: table says 'illegal', pattern implementation says 'Over
```

What went wrong: in the table I listed the daily clock tick as ignored in Suspended, Returned and Lost, and **forgot Overdue**. A nightly job that sends `day_passed` to every loan would have crashed on the first overdue loan it met. The State-pattern version, where I had written the no-op in the Overdue class, passed the same test. Two independent checks found the same hole: the test, and the matrix comparison (the pattern implementation was driven into every state and compared to the table, in `matrix_check.py`). I added the row, and the whole suite and the matrix comparison now agree:

```
Ran 28 tests in 0.001s

OK
```

```
"""One suite, run against both implementations of the loan lifecycle."""
import unittest
from datetime import date, timedelta

import loan_sm
import loan_state_pattern
from loan_sm import IllegalTransition, Loan

DUE = date(2026, 11, 1)


class Behaviour:
    make = None

    def machine(self, **kw):
        return self.make(Loan(DUE, **kw))

    def test_on_time_return(self):
        m = self.machine()
        self.assertEqual(m.send("return_book", DUE), "Returned")
        self.assertEqual((m.loan.fine_pence, m.loan.copy_freed), (0, True))

    def test_late_return_charges_ten_pence_a_day(self):
        m = self.machine()
```

```
        m.send("day_passed", DUE + timedelta(days=1))
        self.assertEqual(m.send("return_book", DUE + timedelta(days=3)), "Returned")
        self.assertEqual(m.loan.fine_pence, 30)

    def test_day_passed_before_or_on_the_due_date_changes_nothing(self):
        m = self.machine()
        self.assertEqual(m.send("day_passed", DUE - timedelta(days=1)), "OnLoan")
        self.assertEqual(m.send("day_passed", DUE), "OnLoan")

    def test_becoming_overdue_sends_exactly_one_notice(self):
        m = self.machine()
        self.assertEqual(m.send("day_passed", DUE + timedelta(days=1)), "Overdue")
        m.send("day_passed", DUE + timedelta(days=2))
        m.send("day_passed", DUE + timedelta(days=3))
        self.assertEqual(m.loan.notices_sent, 1)

    def test_renew_extends_by_fourteen_days_twice_and_then_is_refused_quietly(self):
        m = self.machine()
        m.send("renew")
        m.send("renew")
        self.assertEqual((m.loan.due_on, m.loan.renewals), (DUE + timedelta(days=28), 2)
        self.assertEqual(m.send("renew"), "OnLoan")
        self.assertEqual(m.loan.renewals, 2)                    # guard false: the even

    def test_renew_is_refused_when_someone_is_waiting(self):
        m = self.machine(reserved_by_other=True)
        m.send("renew")
        self.assertEqual((m.loan.due_on, m.loan.renewals), (DUE, 0))

    def test_renew_is_illegal_once_overdue(self):
        m = self.machine()
        m.send("day_passed", DUE + timedelta(days=1))
        with self.assertRaises(IllegalTransition):
            m.send("renew")

    def test_declare_lost_needs_sixty_days_overdue(self):
        m = self.machine()
        m.send("day_passed", DUE + timedelta(days=1))
        self.assertEqual(m.send("declare_lost", DUE + timedelta(days=59)), "Overdue")
        self.assertEqual(m.send("declare_lost", DUE + timedelta(days=60)), "Lost")
        self.assertEqual(m.loan.fine_pence, 2500)

    def test_declare_lost_is_illegal_while_on_loan(self):
        with self.assertRaises(IllegalTransition):
            self.machine().send("declare_lost", DUE + timedelta(days=100))
```

```
    def test_closure_while_on_loan_resumes_on_loan_with_the_due_date_pushed_back(self):
        m = self.machine()
        m.send("close_library", DUE - timedelta(days=3))
        self.assertEqual(m.state, "Suspended")
        self.assertEqual(m.send("day_passed", DUE + timedelta(days=5)), "Suspended")
        self.assertEqual(m.send("reopen", DUE - timedelta(days=3) + timedelta(days=8)),
        self.assertEqual(m.loan.due_on, DUE + timedelta(days=8))

    def test_closure_while_overdue_resumes_overdue(self):
        m = self.machine()
        m.send("day_passed", DUE + timedelta(days=1))
        m.send("close_library", DUE + timedelta(days=2))
        self.assertEqual(m.send("reopen", DUE + timedelta(days=4)), "Overdue")

    def test_history_reruns_the_entry_action(self):
        """A consequence to decide about: coming back to Overdue through history enters
        m = self.machine()
        m.send("day_passed", DUE + timedelta(days=1))
        m.send("close_library", DUE + timedelta(days=2))
        m.send("reopen", DUE + timedelta(days=4))
        self.assertEqual(m.loan.notices_sent, 2)

    def test_cannot_return_while_the_library_is_closed(self):
        m = self.machine()
        m.send("close_library", DUE - timedelta(days=1))
        with self.assertRaises(IllegalTransition):
            m.send("return_book", DUE)

    def test_terminal_states_accept_nothing_but_clock_ticks(self):
        m = self.machine()
        m.send("return_book", DUE)
        for event in ("renew", "return_book", "declare_lost", "close_library", "reopen")
            with self.assertRaises(IllegalTransition):
                m.send(event, DUE)
        self.assertEqual(m.send("day_passed", DUE + timedelta(days=9)), "Returned")


class TableDriven(Behaviour, unittest.TestCase):
    make = staticmethod(loan_sm.LoanMachine)


class StatePattern(Behaviour, unittest.TestCase):
    make = staticmethod(loan_state_pattern.LoanStateMachine)
```

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

```
"""The transition matrix: for every state and event, what happens. Computed from the tab
driving the State-pattern implementation into each state and sending it each event. If t
from datetime import date, timedelta

import loan_sm
import loan_state_pattern as sp
from loan_sm import EVENTS, STATES, IllegalTransition, Loan

DUE = date(2026, 11, 1)
ARGS = {"renew": None, "day_passed": DUE + timedelta(days=1), "return_book": DUE + timed
        "close_library": DUE + timedelta(days=1), "reopen": DUE + timedelta(days=2)}


def pattern_machine_in(state: str) -> sp.LoanStateMachine:
    m = sp.LoanStateMachine(Loan(DUE))
    if state == "Overdue":
        m.send("day_passed", DUE + timedelta(days=1))
    elif state == "Suspended":
        m.send("close_library", DUE - timedelta(days=5))
    elif state == "Returned":
        m.send("return_book", DUE)
    elif state == "Lost":
        m.send("day_passed", DUE + timedelta(days=1))
        m.send("declare_lost", DUE + timedelta(days=61))
    return m


def pattern_matrix() -> dict:
    out = {}
    for s in STATES:
        for e in EVENTS:
            m = pattern_machine_in(s)
            try:
                before = m.state
                after = m.send(e, ARGS[e])
                out[(s, e)] = after
            except IllegalTransition:
                out[(s, e)] = "illegal"
    return out
```

```
def show(title: str, matrix: dict) -> None:
    print(title)
    print(f"  {'':<11}" + "".join(f"{e:<15}" for e in EVENTS))
    for s in STATES:
        print(f"  {s:<11}" + "".join(f"{matrix[(s, e)]:<15}" for e in EVENTS))


if __name__ == "__main__":
    table = loan_sm.matrix()
    show("from the transition table", table)
    pat = pattern_matrix()
    print()
    # the table version reports an event it deliberately ignores as 'ignored'; the patte
    normal = {k: (k[0] if v == "ignored" else v) for k, v in table.items()}
    differences = {k: (normal[k], pat[k]) for k in normal if normal[k] != pat[k]}
    print("the State-pattern implementation, driven into every state:", "identical to th
    for (s, e), (a, b) in sorted(differences.items()):
        print(f"   in {s}, event {e}: table says {a!r}, pattern implementation says {b!r
```

A matrix shows *what* each implementation does and the tests say *whether that is intended*; you need both. Note the history test (`test_history_reruns_the_entry_action`): it documents the second overdue notice rather than judging it.

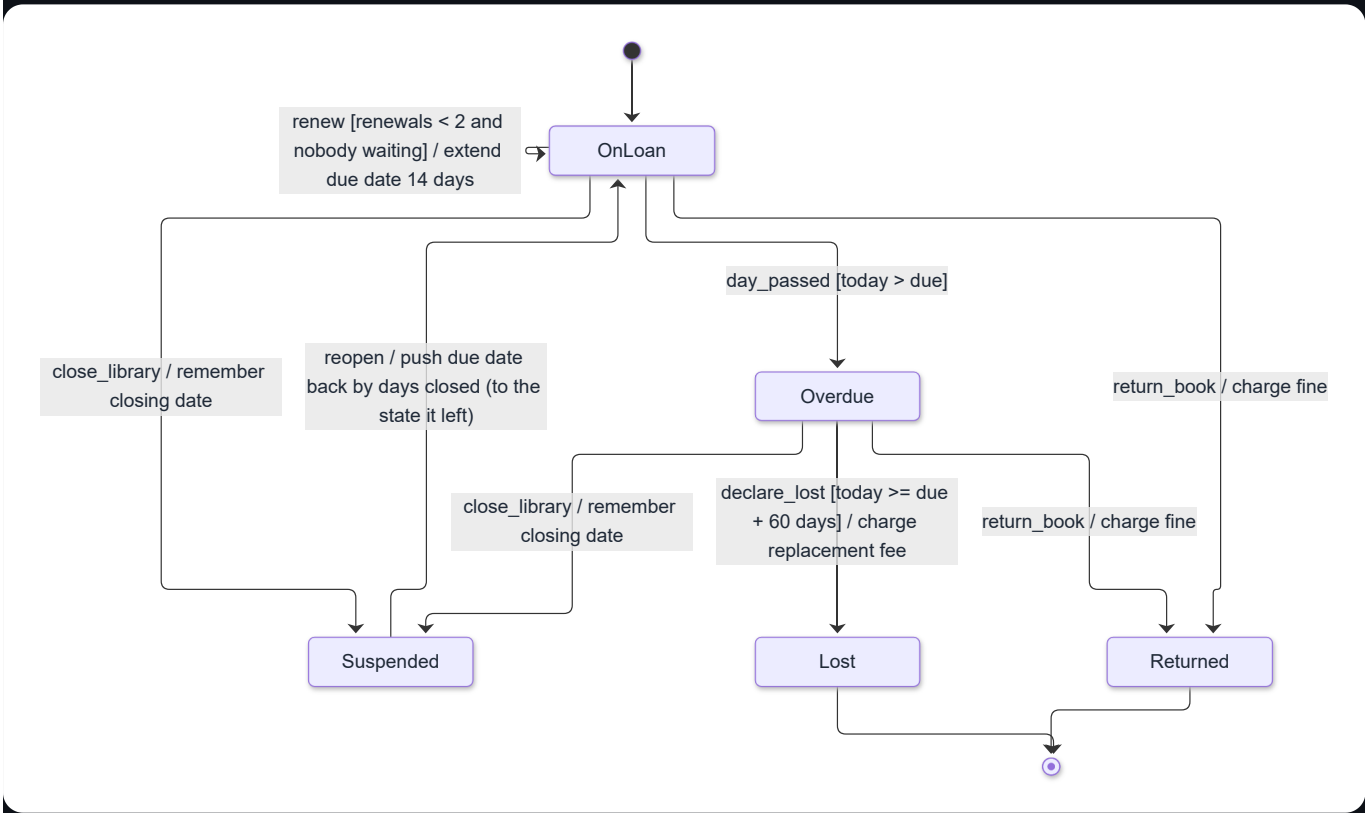
What Mermaid Does With the Same Diagram

Chapter 2 said the course would use Mermaid for state diagrams unless composite states or history needed more. I tested each feature in Mermaid 12.1.0, looking at the rendered picture rather than only the exit status, because the exit status was misleading. (I inspected the pictures for the history and entry/exit cases; for the others I confirmed they rendered without error and did not check every detail.)

FEATURE	RESULT IN MERMAID 12.1.0
Composite state (`state Active { ... }`)	Works
Choice, fork and join pseudo-states	Work (`<<choice>>`, `<<fork>>`, `<<join>>`)

FEATURE	RESULT IN MERMAID 12.1.0
Notes	Work
Guards and actions as part of the arrow label	Work (they are only text)
History state `[H]`	**No error, wrong result**: drawn as an ordinary state literally named "[H]"
Entry and exit actions written as `Overdue : entry / send notice`	**No error, wrong result**: the description line replaced the state's name, so the box showed "entry / send notice" and "exit / stop notices" and no longer said "Overdue"
Workaround for entry and exit	`state "Overdue entry / send notice exit / stop notices" as Overdue` draws a box with the name and the lines

So Mermaid can draw a *flat* state diagram well, and the same table generates one (below, for comparison), but not history, and entry/exit actions only by a text workaround. For this loan, **PlantUML is the right tool**. The rule to take away is wider than the tool: *a diagram tool that accepts your syntax without complaint may still be drawing something else*; look at the picture.



Mermaid, flat, generated from the same table. The composite state is gone (`return_book` and `close_library` each appear twice) and the history is only a note on the arrow back.

Activity Diagrams

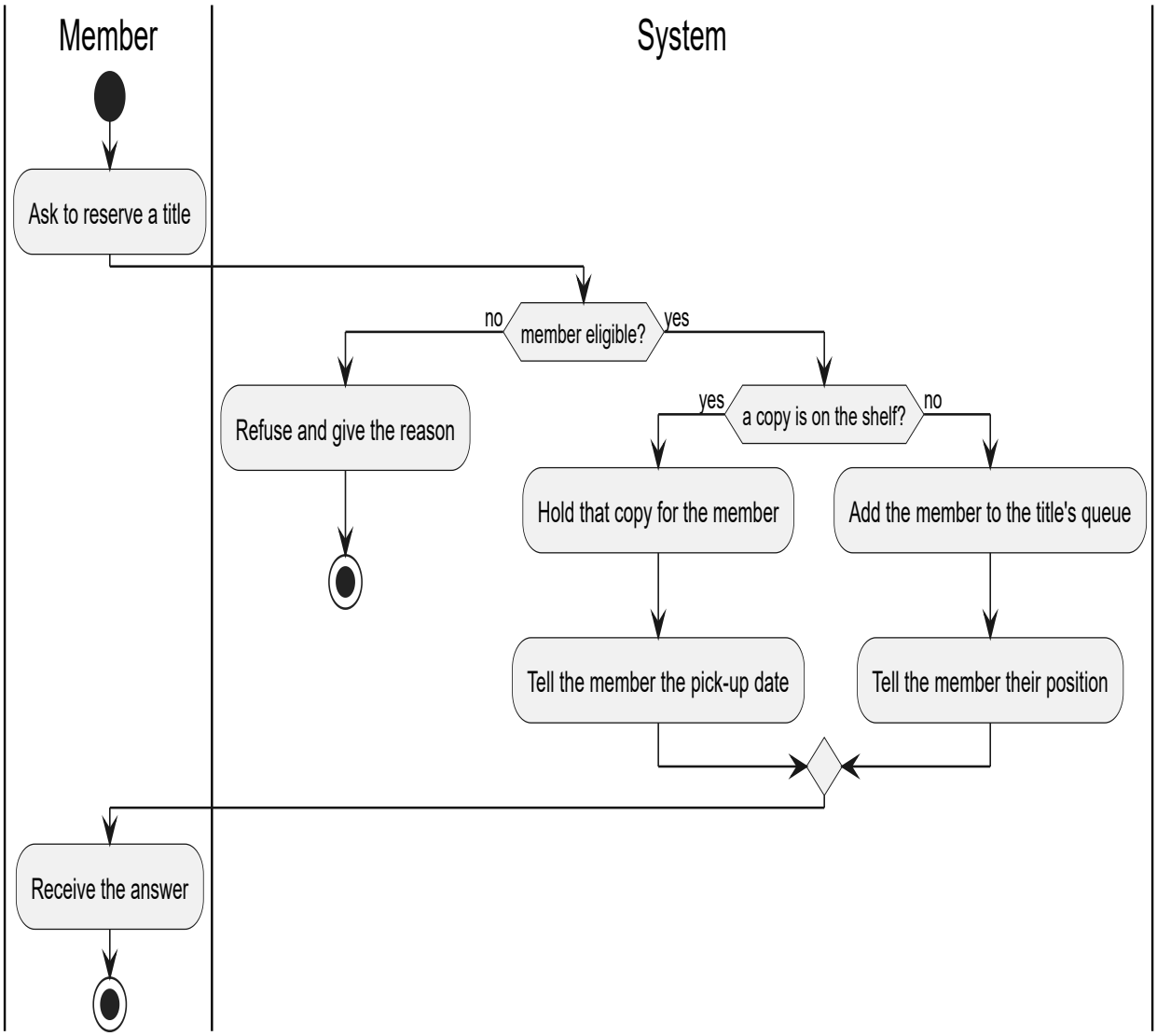
A state diagram follows one object. An **activity diagram** follows a *process*: a sequence of actions with decisions, parallel branches and the people or systems responsible. It is the diagram for a workflow.

ELEMENT	NOTATION	MEANING
Action	Rounded box	One step
Start, end	Filled dot; circled dot	Where the process begins and where a path ends
Decision, merge	Diamond	A choice with guards on the outgoing arrows; a merge brings paths together again
Fork, join	Thick bar	Split into parallel paths; wait for all of them before continuing
Swimlane (partition)	Vertical column per actor	Who is responsible for each action

When an Activity Diagram Beats a Flowchart

A flowchart describes an *algorithm*: the steps one program follows. An activity diagram earns its place when the process involves **several actors** (swimlanes) or **things happening at once** (fork and join). If there is one actor and no concurrency, a flowchart (or just a use case description) is simpler and no worse.

The reservation workflow, in two diagrams. First, a member asks to reserve a title:

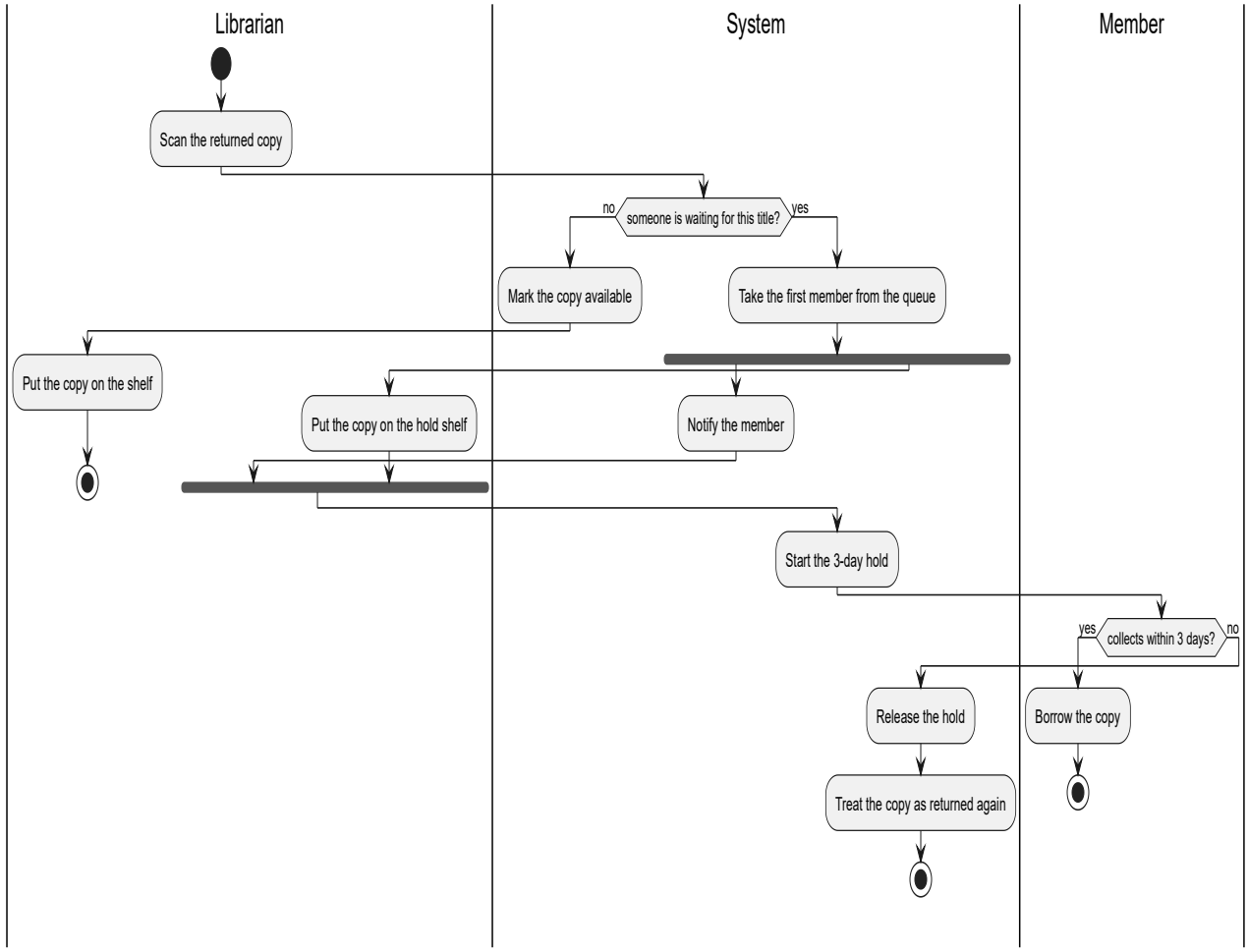


PlantUML. Two lanes. Two nested decisions, and the paths merge before the member receives the answer.

```
@startuml
scale 1.3
|Member|
start
:Ask to reserve a title;
|System|
if (member eligible?) then (no)
  :Refuse and give the reason;
  stop
else (yes)
  if (a copy is on the shelf?) then (yes)
    :Hold that copy for the member;
    :Tell the member the pick-up date;
```

```
    else (no)
      :Add the member to the title's queue;
      :Tell the member their position;
    endif
  endif
  |Member|
  :Receive the answer;
  stop
  @enduml
```

Second, a reserved title's copy comes back (this is where parallel work and three actors matter):



PlantUML. When someone is waiting, the system notifies them *and* the librarian puts the copy on the hold shelf: neither has to come first, and the process waits for both (the join) before starting the three-day hold. If the member does not collect, the hold is released and the copy is treated as returned again, so the next member in the queue gets a turn.

```
  @startuml
  scale 1050 width
```

```
  |Librarian|
  start
  :Scan the returned copy;
  |System|
  if (someone is waiting for this title?) then (no)
    |System|
    :Mark the copy available;
    |Librarian|
    :Put the copy on the shelf;
    stop
  else (yes)
    |System|
    :Take the first member from the queue;
    fork
      |System|
      :Notify the member;
    fork again
      |Librarian|
      :Put the copy on the hold shelf;
    end fork
    |System|
    :Start the 3-day hold;
    |Member|
    if (collects within 3 days?) then (yes)
      :Borrow the copy;
      stop
    else (no)
      |System|
      :Release the hold;
      :Treat the copy as returned again;
      stop
    endif
  endif
  @enduml
```

The Workflow as Running Code

I implemented both activities (`reservations.py`, with 10 tests) so that the diagram's decisions are checked, not just drawn. The story below follows the diagram: two members queue for the only copy of *Dune*, it comes back, the first does not collect in three days, the hold expires, the second collects. The log is the real output:

```
"""Reservation handling: the workflow the activity diagrams describe, as running code wi
from dataclasses import dataclass, field
from datetime import date, timedelta

HOLD_DAYS = 3


class Refusal(Exception):
    pass


@dataclass
class Hold:
    member: str
    expires_on: date


@dataclass
class Reservations:
    eligible: set                                          # cards allowed to
    queue: dict = field(default_factory=dict)              # title -> [cards w
    holds: dict = field(default_factory=dict)              # copy barcode -> H
    shelf: set = field(default_factory=set)                # barcodes of copie
    log: list = field(default_factory=list)

    def say(self, text: str) -> None:
        self.log.append(text)

    # --- activity 1: a member asks to reserve a title ---
    def place(self, card: str, title: str, copy_on_shelf: str | None, today: date) -> st
        if card not in self.eligible:                      # decision: member
            self.say(f"{card} refused: not eligible")
            raise Refusal("not eligible")
        if card in self.queue.get(title, []):
            raise Refusal("already queued for this title")
        if copy_on_shelf is not None:                      # decision: a copy
            self.shelf.discard(copy_on_shelf)
            self.holds[copy_on_shelf] = Hold(card, today + timedelta(days=HOLD_DAYS))
            self.say(f"{card}: copy {copy_on_shelf} held until {self.holds[copy_on_shelf
            return "held"
        self.queue.setdefault(title, []).append(card)
        self.say(f"{card}: queued for {title} at position {len(self.queue[title])}")
        return "queued"
```

```
    # --- activity 2: a copy comes back ---
    def copy_returned(self, barcode: str, title: str, today: date) -> str:
        waiting = self.queue.get(title, [])
        if not waiting:
            self.shelf.add(barcode)
            self.say(f"copy {barcode} returned: nobody waiting, back on the shelf")
            return "shelved"
        card = waiting.pop(0)
        self.holds[barcode] = Hold(card, today + timedelta(days=HOLD_DAYS))
        # the two actions of the fork: they have no order between them
        self.say(f"  (fork) notify {card}")
        self.say(f"  (fork) put copy {barcode} on the hold shelf")
        self.say(f"  (join) {card} has until {self.holds[barcode].expires_on}")
        return "held"

    def collect(self, card: str, barcode: str, today: date) -> None:
        hold = self.holds.get(barcode)
        if hold is None or hold.member != card:
            raise Refusal("no hold for this member on this copy")
        if today > hold.expires_on:
            raise Refusal("the hold has expired")
        del self.holds[barcode]
        self.say(f"{card} collected copy {barcode}: it becomes a loan")

    def day_passed(self, today: date, titles: dict) -> None:
        """titles maps a barcode to its title, so an expired hold can pass the copy to t
        for barcode, hold in list(self.holds.items()):
            if today > hold.expires_on:
                del self.holds[barcode]
                self.say(f"hold on {barcode} for {hold.member} expired")
                self.copy_returned(barcode, titles[barcode], today)
```

```
ada: queued for Dune at position 1
bo: queued for Dune at position 2
  (fork) notify ada
  (fork) put copy B1 on the hold shelf
  (join) ada has until 2026-11-08
hold on B1 for ada expired
  (fork) notify bo
  (fork) put copy B1 on the hold shelf
  (join) bo has until 2026-11-12
bo collected copy B1: it becomes a loan
```

```
Ran 10 tests in 0.000s

OK
```

The two `(fork)` lines are the parallel branches: the code does them in one order, but the diagram says the order does not matter, and that is a statement about the *process*, not about the program. Look also at what the code needed that the diagram did not say: whether the last day of the hold is still valid (a test pins "yes"), and what an expired hold does when nobody is waiting (the copy goes back to the shelf).

State Machine or Activity Diagram?

IF YOUR QUESTION IS...	DRAW
"What can happen to this one thing, and what is it allowed to do in each condition?"	State machine
"What are the steps of this process, who does each, and what happens in parallel?"	Activity diagram
"What is the order of calls for this one scenario?"	Sequence diagram (chapter 7)
"What can this system do for whom?"	Use cases (chapter 3)

They connect: each box in the reservation activity ("Hold that copy", "Release the hold") is a change of state in some object's state machine (a copy becomes Held; a hold ends). An activity says *when* in the process, the state machine says *what that object is permitted to do next*.

Usual Mistakes

MISTAKE	FIX
A state that is really an action ("Calculating fine")	States are conditions an object stays in; actions happen on transitions
A boolean flag instead of a state, and later a second flag, and a third	If flags combine into impossible combinations (`overdue` and `returned`), you have a state machine hiding

MISTAKE	FIX
Unhandled events silently discarded in code	Choose per state: illegal (raise) or deliberately ignored (list it)
Drawing only the happy path	Draw the loss, the cancellation, the timeout; they are where bugs live
A state with no way out that is not a final state	Run a reachability check (Exercise 2)
Activity diagram as a program flowchart	Use lanes and concurrency, or use a flowchart and say so
Everything in one huge diagram	One object's life, or one process, per diagram

MISTAKES WHILE PREPARING THIS CHAPTER

- **"It rendered" proved nothing.** Every Mermaid probe returned success, including the two that drew the wrong thing. If I had checked exit codes, I would have written that Mermaid supports history and entry actions. Takeaway: for diagram syntax, the check is the picture.
- **I forgot an event in my own table.** The clock tick was missing for Overdue (above). I had it right in the State-pattern version and wrong in the table. Takeaway: write the matrix for every state machine; a hole you cannot see in the diagram is visible in the matrix.
- **My first exercise claimed one check would find two mistakes.** I wrote the "find the holes" exercise (Exercise 2) for a copy lifecycle with two missing rows, and said a reachability check finds both. It finds only one (a copy in repair can never come back). The other (a held copy has no way out if nobody collects it) is not a reachability problem, because the copy can still leave Held by being collected; it needs a different kind of check, a rule about states that wait for a person. I rewrote the exercise around the two checks. Takeaway: run the analysis before describing what it finds.
- **PlantUML swimlanes carried over.** In my first returned-copy diagram I changed lanes in one branch of a decision, and the other branch started in the same lane, putting the system's "take the first member" step in the librarian's column. I only saw it in the picture. The fix is to name the lane at the start of every branch. The diagram was also too wide, so it now sets a width.
- **Leftover junk in a demonstration script.** The reservation story script briefly contained a line ending in `if False else None`, a leftover from an idea I dropped. It did nothing, which is exactly why it nearly stayed. I deleted it.

Hands-On Exercises

EXERCISE 1

A reservation passes through Queued, Held, Collected, Expired and Cancelled. A copy becoming available moves Queued to Held (with a three-day deadline); collecting in time ends in Collected; a day passing after the deadline moves Held to Expired; cancelling is allowed from Queued and Held. Write it as a transition table, generate the transition matrix, write tests (including collecting one day late and cancelling twice), and generate the Mermaid diagram.

View solution

EXERCISE 2

The solution file has the state table for a physical copy of a book with two missing rows. Write checks that find them (hint: they are different kinds of mistake), show the output before and after adding the rows, and say why one check cannot find both.

View solution

EXERCISE 3

Draw "join the library" as an activity diagram with three swimlanes (Applicant, Librarian, System): the applicant fills in the form and shows proof of address; the librarian checks it; if it is not acceptable the applicant is told what is missing; if it is, the system creates the record, and in parallel the system records the card number and the librarian prints the card; then the card is handed over and signed for.

View solution

CHAPTER 8 QUICK REFERENCE

- State machine = one object's life: states, events, transitions with `[guard]` and `/ action`, entry and exit actions, initial and final, composite states (shared transitions), history (return to the substate you left)
- In code, decide: an unhandled event is illegal or deliberately ignored; a false guard discards the event; where history lives; whether entry runs again on return through history (it did here: a second notice)
- A transition table makes the diagram and the matrix generate from the code; the State pattern fits when each state has a lot of behaviour; neither is the answer for all cases

- Write the matrix (every state by every event) and the tests, and run reachability checks, plus rules for states that wait for a person: a diagram hides what is missing
- Mermaid 12.1.0: composite, choice, fork, notes work; history `[H]` and `State : entry / ...` are accepted and drawn wrongly; use PlantUML for those
- Activity diagram = a process: actions, decisions and merges, fork and join, swimlanes; beats a flowchart when there are several actors or parallel work
- State machine for what one thing may do; activity for the steps of a process; sequence for the order of calls; use cases for goals

COMING NEXT

Object-Oriented Design with UML 9 (Components, Packages and Deployment): component diagrams with provided and required interfaces, package diagrams and dependency direction, deployment diagrams and object diagrams; how they relate to the C4 model in *Software Architecture Fundamentals* 9; keeping dependencies acyclic, and the library's components and deployment.

Object-Oriented Design with UML

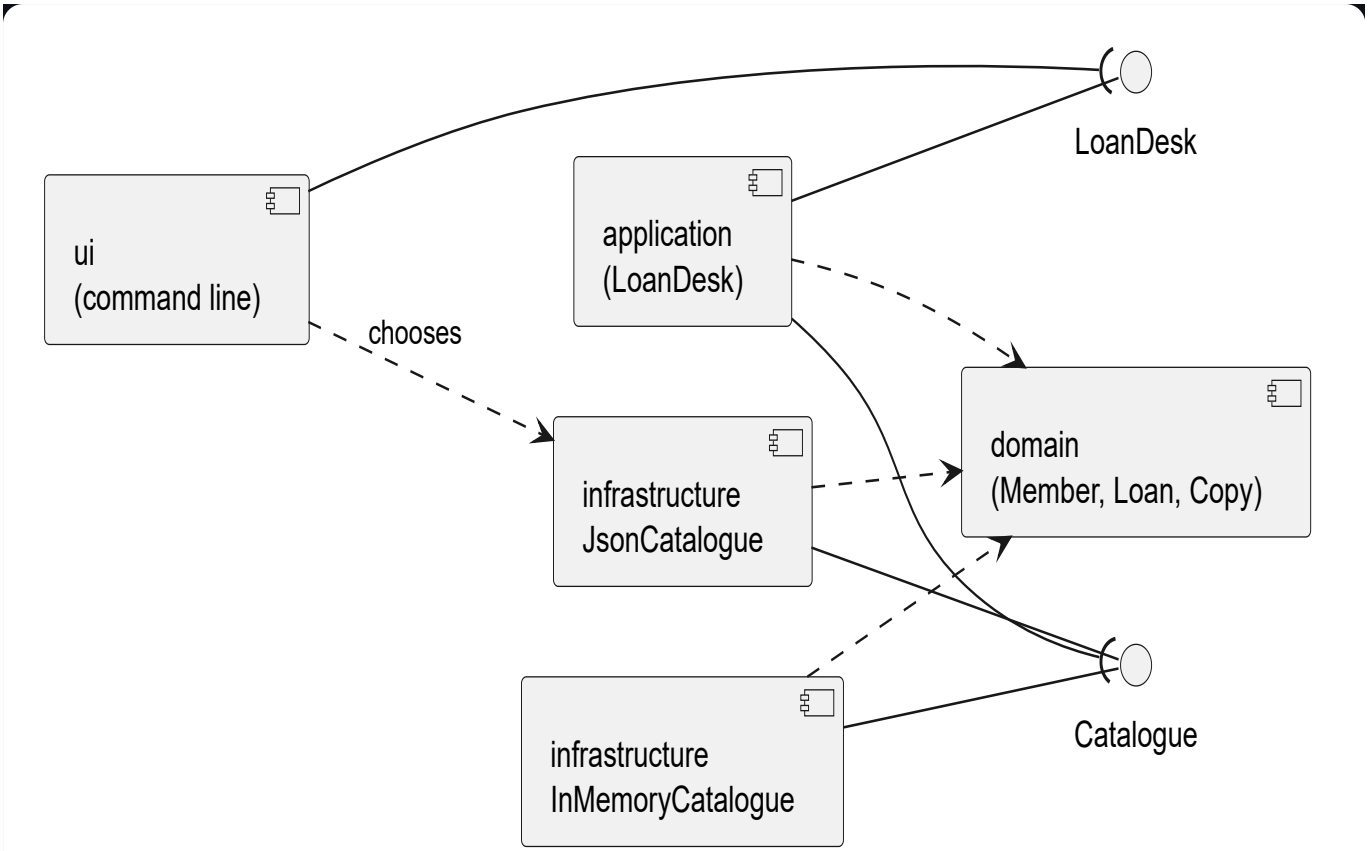
Course 1 · Chapter 9 · Components, Packages and Deployment

Class diagrams are about classes. A real system is also about **chunks bigger than a class**: the pieces you can build, replace and deploy separately, and which of them may depend on which. Four diagrams cover that level: **component** diagrams (the pieces and the interfaces between them), **package** diagrams (how the code is grouped, and the direction of dependencies), **deployment** diagrams (what runs where) and **object** diagrams (a snapshot of real objects at one moment). This chapter restructures the library into a real Python package, so the diagrams describe something that exists and can be checked.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Run here (Python 3.14, PlantUML 1.2026.8): the library as a four-package project (`domain`, `application`, `infrastructure`, `ui`) with 10 behaviour tests over two interchangeable storage components; a command-line program run against a real JSON file; a script that reads the code's imports and checks three dependency rules (on the real tree, on tampered copies, and as a test); an object diagram generated from live objects; and a demonstration of what two desks sharing one file do. **Not done:** nothing was deployed anywhere: the deployment diagram is a *proposal*, labelled as such. The C4 comparison is from my memory of Simon Brown's model, not a re-read of *Software Architecture Fundamentals* 9 (Documenting Architecture: ADRs and the C4 Model), which is the chapter to read for it.

Component Diagrams

A **component** is a replaceable part of a system with a clear boundary. Its boundary is made of **interfaces**: a **provided interface** (a lollipop: what the component offers) and a **required interface** (a socket: what it needs from others). A component fits another when one's lollipop goes into the other's socket. The notation is small, and the idea is large: the component says *what it needs*, not *who supplies it*.



PlantUML. The library's components. `application` provides the `LoanDesk` interface (to the UI) and *requires* `Catalogue`; two `infrastructure` components each provide `Catalogue`. The dashed arrows are plain dependencies.

```
@startuml
scale 1.3
left to right direction

component "ui\n(command line)" as UI
component "application\n(LoanDesk)" as App
component "domain\n(Member, Loan, Copy)" as Domain
component "infrastructure\nJsonCatalogue" as Json
component "infrastructure\nInMemoryCatalogue" as Mem

interface "Catalogue" as Cat
interface "LoanDesk" as Desk

UI -( Desk
App -- Desk
App -( Cat
Json -- Cat
Mem -- Cat
App ..> Domain
```

```
Json ..> Domain
Mem ..> Domain
UI ..> Json : chooses
@enduml
```

Here is the point of the diagram in code. The application component declares the interface it requires, in its own package:

```
"""The interface the application REQUIRES from whatever stores the library's data. Defin
from typing import Protocol, runtime_checkable

from library.domain import Copy, Loan, Member


@runtime_checkable
class Catalogue(Protocol):
    def member(self, card: str) -> Member: ...

    def copy(self, barcode: str) -> Copy: ...

    def open_loan_for(self, barcode: str) -> tuple[Member, Loan]: ...

    def save(self) -> None: ...
```

```
"""The use cases of the lending desk. Depends on the domain and on the Catalogue interfa
from datetime import date

from library.application.ports import Catalogue
from library.domain import Loan


class LoanDesk:
    def __init__(self, catalogue: Catalogue):
        self.catalogue = catalogue

    def borrow(self, card: str, barcode: str, today: date) -> Loan:
        loan = self.catalogue.member(card).borrow(self.catalogue.copy(barcode), today)
        self.catalogue.save()
        return loan

    def return_book(self, barcode: str, today: date) -> int:
```

```
        member, loan = self.catalogue.open_loan_for(barcode)
        fine = member.give_back(loan, self.catalogue.copy(barcode), today)
        self.catalogue.save()
        return fine
```

`LoanDesk` receives anything that fits and never names a storage class. Two components provide it, written without reference to `LoanDesk`:

```
"""A Catalogue that keeps everything in memory. Provides the interface the application r
from library.domain import Copy, Loan, Member, Refusal


class InMemoryCatalogue:
    def __init__(self, members=(), copies=()):
        self.members = {m.card: m for m in members}
        self.copies = {c.barcode: c for c in copies}

    def member(self, card: str) -> Member:
        if card not in self.members:
            raise Refusal("membership is not valid")
        return self.members[card]

    def copy(self, barcode: str) -> Copy:
        if barcode not in self.copies:
            raise Refusal("no such copy")
        return self.copies[barcode]

    def open_loan_for(self, barcode: str) -> tuple[Member, Loan]:
        for m in self.members.values():
            for l in m.open_loans():
                if l.barcode == barcode:
                    return m, l
        raise Refusal("that copy is not on loan")

    def save(self) -> None:
        pass
```

```
"""A Catalogue that keeps its data in a JSON file and saves after every change. A second
import json
import os
```

```
import pathlib
import tempfile
from datetime import date

from library.domain import Copy, Loan, Member
from library.infrastructure.memory import InMemoryCatalogue


def _d(value: str | None) -> date | None:
    return date.fromisoformat(value) if value else None


class JsonCatalogue:
    def __init__(self, path: str):
        self.path = pathlib.Path(path)
        self._data = self._load() if self.path.exists() else InMemoryCatalogue()

    # the provided interface: same operations as InMemoryCatalogue, delegated
    def member(self, card):
        return self._data.member(card)

    def copy(self, barcode):
        return self._data.copy(barcode)

    def open_loan_for(self, barcode):
        return self._data.open_loan_for(barcode)

    def add(self, member: Member | None = None, copy: Copy | None = None) -> None:
        if member:
            self._data.members[member.card] = member
        if copy:
            self._data.copies[copy.barcode] = copy

    def save(self) -> None:
        raw = {
            "members": [{"card": m.card, "expires_on": m.expires_on.isoformat(), "kind":
                         "loans": [{"barcode": l.barcode, "borrowed_on": l.borrowed_on.i
                                    "returned_on": l.returned_on.isoformat() if l.return
                        for m in self._data.members.values()],
            "copies": [{"barcode": c.barcode, "on_loan": c.on_loan} for c in self._data.
        }
        fd, tmp = tempfile.mkstemp(dir=self.path.parent, suffix=".tmp")      # write to
        with os.fdopen(fd, "w", encoding="utf8") as f:
            json.dump(raw, f, indent=2)
        os.replace(tmp, self.path)
```

```
    def _load(self) -> InMemoryCatalogue:
        raw = json.loads(self.path.read_text(encoding="utf8"))
        members = [Member(m["card"], date.fromisoformat(m["expires_on"]), m["kind"], m["
                          [Loan(l["barcode"], date.fromisoformat(l["borrowed_on"]), date
                   for m in raw["members"]]
        return InMemoryCatalogue(members, [Copy(c["barcode"], c["on_loan"]) for c in raw
```

The interface is owned by the consumer. `Catalogue` lives in `application`, because it describes what the desk needs, not what a database offers. That inversion is what lets you swap the storage component without touching the application. (*Clean Code, SOLID & Refactoring* 6 and 7 cover the dependency inversion principle properly; here it is as an architectural picture.) One consequence worth noticing: because Python's protocols match by shape, **neither storage class imports** `Catalogue`. In the code, the infrastructure package does not depend on the application package at all; the "provides" arrow in the diagram is a statement about the design, not an `import`.

The tests prove that both providers really do fit, and that the whole thing works with either (10 tests, the same behaviour suite over both):

```
"""Tests for the layered library: one behaviour suite over both providers of the Catalogu
import os
import tempfile
import unittest
from datetime import date

from library.application import Catalogue, LoanDesk
from library.domain import Copy, Member, Refusal
from library.infrastructure import InMemoryCatalogue, JsonCatalogue

TODAY = date(2026, 10, 10)


def seed(catalogue):
    if isinstance(catalogue, JsonCatalogue):
        catalogue.add(member=Member("M1", date(2027, 1, 1), "standard"))
        catalogue.add(member=Member("J1", date(2027, 1, 1), "junior"))
        for code in ("B1", "B2", "B3", "B4"):
            catalogue.add(copy=Copy(code))
        catalogue.save()
    return catalogue
```

```
class Behaviour:
    def catalogue(self):
        raise NotImplementedError

    def test_provider_satisfies_the_required_interface(self):
        self.assertIsInstance(self.catalogue(), Catalogue)

    def test_borrow_and_return_with_a_fine(self):
        desk = LoanDesk(self.catalogue())
        self.assertEqual(desk.borrow("M1", "B1", TODAY).due_on, date(2026, 10, 31))
        self.assertEqual(desk.return_book("B1", date(2026, 11, 3)), 30)

    def test_junior_limit_and_junior_fine(self):
        desk = LoanDesk(self.catalogue())
        for code in ("B1", "B2", "B3"):
            desk.borrow("J1", code, TODAY)
        with self.assertRaises(Refusal):
            desk.borrow("J1", "B4", TODAY)
        self.assertEqual(desk.return_book("B1", date(2026, 11, 3)), 15)

    def test_refusals(self):
        desk = LoanDesk(self.catalogue())
        for card, code in (("NOPE", "B1"), ("M1", "ZZZ")):
            with self.assertRaises(Refusal):
                desk.borrow(card, code, TODAY)
        with self.assertRaises(Refusal):
            desk.return_book("B1", TODAY)


class InMemory(Behaviour, unittest.TestCase):
    def catalogue(self):
        return InMemoryCatalogue([Member("M1", date(2027, 1, 1), "standard"), Member("J1
                                  [Copy(c) for c in ("B1", "B2", "B3", "B4")])


class JsonFile(Behaviour, unittest.TestCase):
    def setUp(self):
        self.dir = tempfile.TemporaryDirectory()
        self.path = os.path.join(self.dir.name, "library.json")

    def tearDown(self):
        self.dir.cleanup()
```

```
    def catalogue(self):
        return seed(JsonCatalogue(self.path))

    def test_state_survives_a_restart(self):
        desk = LoanDesk(self.catalogue())
        desk.borrow("M1", "B1", TODAY)
        fresh = LoanDesk(JsonCatalogue(self.path))                    # a brand-new obje
        self.assertEqual(fresh.return_book("B1", date(2026, 10, 31)), 0)
        with self.assertRaises(Refusal):                             # and that return wa
            LoanDesk(JsonCatalogue(self.path)).return_book("B1", date(2026, 10, 31))

    def test_no_temporary_files_are_left_behind(self):
        LoanDesk(self.catalogue()).borrow("M1", "B1", TODAY)
        self.assertEqual(os.listdir(self.dir.name), ["library.json"])


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

```
..........
----------------------------------------------------------------------
Ran 10 tests in 0.032s

OK
```

And the command-line front end, run for real against a JSON file. It borrows a copy, is refused a second loan of it, returns it late, and the file keeps the result:

```
"""A command-line front end for the lending desk.

    py -m library.ui.cli --data library.json init
    py -m library.ui.cli --data library.json borrow M1 B1 --today 2026-10-10
    py -m library.ui.cli --data library.json return B1 --today 2026-11-03
"""
import argparse
import sys
from datetime import date

from library.application import LoanDesk
from library.domain import Copy, Member, Refusal
from library.infrastructure import JsonCatalogue
```

```
def main(argv: list[str]) -> int:
    parser = argparse.ArgumentParser(prog="library")
    parser.add_argument("--data", required=True)
    sub = parser.add_subparsers(dest="command", required=True)
    sub.add_parser("init")
    b = sub.add_parser("borrow")
    b.add_argument("card")
    b.add_argument("barcode")
    b.add_argument("--today", default=date.today().isoformat())
    r = sub.add_parser("return")
    r.add_argument("barcode")
    r.add_argument("--today", default=date.today().isoformat())
    args = parser.parse_args(argv)

    catalogue = JsonCatalogue(args.data)
    if args.command == "init":
        catalogue.add(member=Member("M1", date(2027, 12, 31), "standard"))
        catalogue.add(member=Member("J1", date(2027, 12, 31), "junior"))
        for code in ("B1", "B2", "B3"):
            catalogue.add(copy=Copy(code))
        catalogue.save()
        print("initialised:", args.data)
        return 0
    desk = LoanDesk(catalogue)
    try:
        if args.command == "borrow":
            loan = desk.borrow(args.card, args.barcode, date.fromisoformat(args.today))
            print(f"loan recorded: {loan.barcode} due {loan.due_on}")
        else:
            fine = desk.return_book(args.barcode, date.fromisoformat(args.today))
            print(f"returned; fine {fine}p")
    except Refusal as e:
        print(f"refused: {e}")
        return 1
    return 0


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

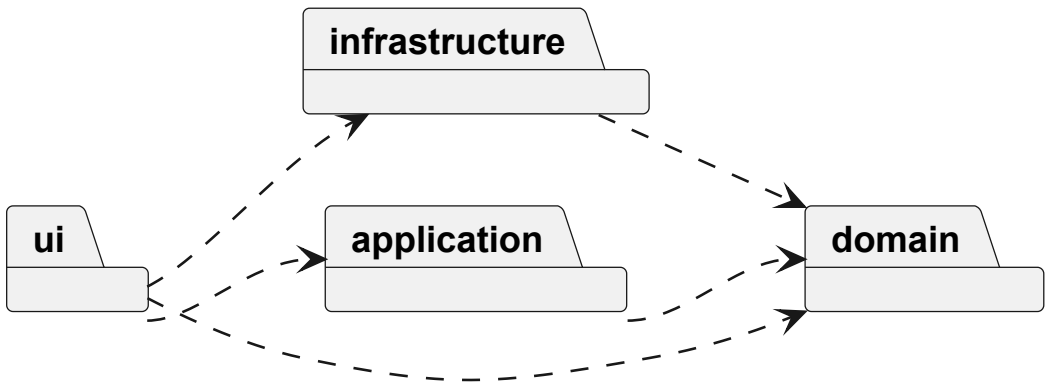
```
$ py -m library.ui.cli --data library.json init
initialised: library.json
```

```
$ py -m library.ui.cli --data library.json borrow M1 B1 --today 2026-10-10
loan recorded: B1 due 2026-10-31
$ py -m library.ui.cli --data library.json borrow J1 B1 --today 2026-10-10    (the same
refused: already on loan
$ py -m library.ui.cli --data library.json return B1 --today 2026-11-03
returned; fine 30p

the file the desk wrote (first member only):
{
 "card": "M1",
 "expires_on": "2027-12-31",
 "kind": "standard",
 "fines_pence": 30,
 "loans": [
  {
   "barcode": "B1",
   "borrowed_on": "2026-10-10",
   "due_on": "2026-10-31",
   "returned_on": "2026-11-03"
  }
 ]
}
```

Package Diagrams and Dependency Direction

A **package** is a grouping of code (in Python, a directory with an `__init__.py`). A package diagram shows packages and the dependencies between them: an arrow from A to B means "A's code uses B's code" (in Python, almost always an `import`). The diagram is easy to draw and easy to let rot, so here it is *generated from the imports*:



PlantUML, generated by `deps.py` from the real imports. The infrastructure package depends on the domain, not on the application (see above). Everything points toward `domain`.

The architecture has rules about those arrows. This one is the common "layered" shape: the domain depends on nothing; the application depends on the domain; the infrastructure serves the application; the UI sits on top and wires things together (it is the only place that chooses `JsonCatalogue`). Written as data, with two further rules, in a script that reads the code:

```
"""Read the imports of the `library` package, build its package dependency graph, and ch

    py deps.py [path-to-folder-containing-library]  [--puml out.puml]

R1 direction : which packages may depend on which (the architecture's rule, written down
R2 acyclic   : no cycle between packages
R3 public API: other packages import a package, never its inner modules (library.domain,
"""
import ast
import pathlib
import sys

ALLOWED = {                                  # package -> packages it may import
    "domain": set(),
    "application": {"domain"},
    "infrastructure": {"domain", "application"},
    "ui": {"domain", "application", "infrastructure"},
}


def package_of(path: pathlib.Path, root: pathlib.Path) -> str:
    rel = path.relative_to(root / "library").parts
    return rel[0] if len(rel) > 1 else "(root)"


def collect(root: pathlib.Path):
    edges = []
    for path in sorted((root / "library").rglob("*.py")):
        src = package_of(path, root)
        for node in ast.walk(ast.parse(path.read_text(encoding="utf8"))):
            names = []
            if isinstance(node, ast.ImportFrom) and node.module and node.module.startswi
                names.append(node.module)
            elif isinstance(node, ast.Import):
                names += [a.name for a in node.names if a.name.startswith("library.")]
            for module in names:
```

```
                parts = module.split(".")
                dst = parts[1]
                if dst != src:
                    edges.append({"from": src, "to": dst, "file": str(path.relative_to(r
                                  "deep": len(parts) > 2})
    return edges


def find_cycles(edges) -> list[list[str]]:
    graph: dict[str, set[str]] = {}
    for e in edges:
        graph.setdefault(e["from"], set()).add(e["to"])
    cycles, state = [], {}

    def visit(node, path):
        state[node] = 1
        for nxt in sorted(graph.get(node, ())):
            if state.get(nxt) == 1:
                cycles.append(path[path.index(nxt):] + [nxt])
            elif nxt not in state:
                visit(nxt, path + [nxt])
        state[node] = 2

    for n in sorted(graph):
        if n not in state:
            visit(n, [n])
    return cycles


def check(root: pathlib.Path, show_edges: bool = True) -> int:
    files = list((root / "library").rglob("*.py"))
    edges = collect(root)
    print(f"analysed {len(files)} files, found {len(edges)} cross-package imports")
    if not files:
        print("result: NOTHING WAS ANALYSED, so 'clean' would mean nothing: is the path r
        return 1
    if not edges:
        print("note: no imports between packages at all, so there is nothing for the rul
    if show_edges:
        print("package dependencies found in the code (from -> to : first place):")
        seen = set()
        for e in edges:
            key = (e["from"], e["to"])
            if key not in seen:
                seen.add(key)
```

```
                print(f"   {e['from']:<15} -> {e['to']:<15} {e['file']}:{e['line']}  ({e
    problems = 0
    print("R1 direction:")
    for e in edges:
        if e["from"] in ALLOWED and e["to"] not in ALLOWED[e["from"]]:
            problems += 1
            print(f"   VIOLATION {e['from']} may not depend on {e['to']}: {e['file']}:{e
    print("R2 no cycles:")
    cycles = find_cycles(edges)
    for c in cycles:
        problems += 1
        print("   VIOLATION cycle: " + " -> ".join(c))
    print("R3 public API only:")
    for e in edges:
        if e["deep"]:
            problems += 1
            print(f"   VIOLATION {e['file']}:{e['line']} reaches inside {e['to']} ({e['m
    print("result:", "clean" if problems == 0 else f"{problems} violation(s)")
    return problems


def to_plantuml(edges) -> str:
    lines = ["@startuml", "scale 1.3", "left to right direction"]
    for pkg in ("ui", "infrastructure", "application", "domain"):
        lines.append(f'package "{pkg}" as {pkg}')
    for key in sorted({(e["from"], e["to"]) for e in edges}):
        lines.append(f"{key[0]} ..> {key[1]}")
    lines.append("@enduml")
    return "\n".join(lines) + "\n"


if __name__ == "__main__":
    argv = sys.argv[1:]
    puml = None
    if "--puml" in argv:
        i = argv.index("--puml")
        puml = pathlib.Path(argv[i + 1])
        del argv[i:i + 2]
    root = pathlib.Path(argv[0]) if argv else pathlib.Path(__file__).parent
    code = check(root)
    if puml:
        puml.write_text(to_plantuml(collect(root)), encoding="utf8")
    sys.exit(1 if code else 0)
```

The three rules, with their output on the real tree:

- **R1 direction:** which package may import which. The domain importing the UI would be a violation.
- **R2 no cycles:** if the domain imports infrastructure and infrastructure imports the domain, neither can be understood, built or tested without the other.
- **R3 public API only:** each package's `__init__.py` lists its public names (`__all__`), and other packages import from the package, not from its inner modules. That is a *module boundary* you can see and check: inside `domain`, files can be reorganised freely.

```
analysed 11 files, found 7 cross-package imports
package dependencies found in the code (from -> to : first place):
   application     -> domain          library\application\desk.py:5  (library.domain)
   infrastructure  -> domain          library\infrastructure\jsonfile.py:8  (library.dom
   ui              -> application     library\ui\cli.py:11  (library.application)
   ui              -> domain          library\ui\cli.py:12  (library.domain)
   ui              -> infrastructure  library\ui\cli.py:13  (library.infrastructure)
R1 direction:
R2 no cycles:
R3 public API only:
result: clean
```

The Check Can Fail

A rule checker that has never reported a violation has not been shown to work. I copied the library, added two shortcuts a hurried developer might add (the domain importing the infrastructure to build a default storage; the UI importing the domain's inner module), and ran it:

```
analysed 11 files, found 8 cross-package imports
package dependencies found in the code (from -> to : first place):
R1 direction:
   VIOLATION domain may not depend on infrastructure: library\domain\model.py:3 imports
R2 no cycles:
   VIOLATION cycle: domain -> infrastructure -> domain
R3 public API only:
   VIOLATION library\ui\cli.py:12 reaches inside domain (library.domain.model); import l
result: 3 violation(s)
```

All three problems are reported with file and line. And here is why R2 matters: the same tampered copy does not merely look untidy, it *does not run*:

```
$ py -c "import library.ui.cli"      (the tree with the shortcut in domain/model.py)
    from library.infrastructure.jsonfile import JsonCatalogue
  File "C:\lwdj\ooad\ch9\bad\library\infrastructure\jsonfile.py", line 8, in <module>
    from library.domain import Copy, Loan, Member
ImportError: cannot import name 'Copy' from partially initialized module 'library.domain

$ py -c "import library.domain"
  File "C:\lwdj\ooad\ch9\bad\library\infrastructure\jsonfile.py", line 8, in <module>
    from library.domain import Copy, Loan, Member
ImportError: cannot import name 'Copy' from partially initialized module 'library.domain
```

Python sometimes tolerates import cycles, depending on which module is loaded first, so a cycle can lurk until an unrelated change reorders the imports. Do not rely on luck; check.

The Rules as a Test

A rule nobody runs is a hope. `test_architecture.py` runs the check on the real tree (it must find no violations) and on a deliberately tampered copy (it must find some):

```
"""The architecture rules as a test, so a dependency in the wrong direction fails the bu
import contextlib
import io
import pathlib
import shutil
import tempfile
import unittest

import deps

HERE = pathlib.Path(__file__).parent


def violations(root: pathlib.Path) -> int:
    with contextlib.redirect_stdout(io.StringIO()):
        return deps.check(root, show_edges=False)


class Architecture(unittest.TestCase):
    def test_the_library_follows_its_dependency_rules(self):
```

```
        self.assertEqual(violations(HERE), 0)

    def test_the_check_can_fail(self):
        """Tested on a copy with a deliberate bad import, so the real tree is never touc
        with tempfile.TemporaryDirectory() as d:
            root = pathlib.Path(d)
            shutil.copytree(HERE / "library", root / "library")
            model = root / "library" / "domain" / "model.py"
            model.write_text("from library.ui import cli\n" + model.read_text(encoding="
            self.assertGreaterEqual(violations(root), 1)


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

```
----------------------------------------------------------------------
Ran 2 tests in 0.123s

OK
```

Known gap, left for Exercise 1: a package that is not listed in the rules is not checked at all. A new package that imports the UI passes as "clean" until someone writes its rules down.

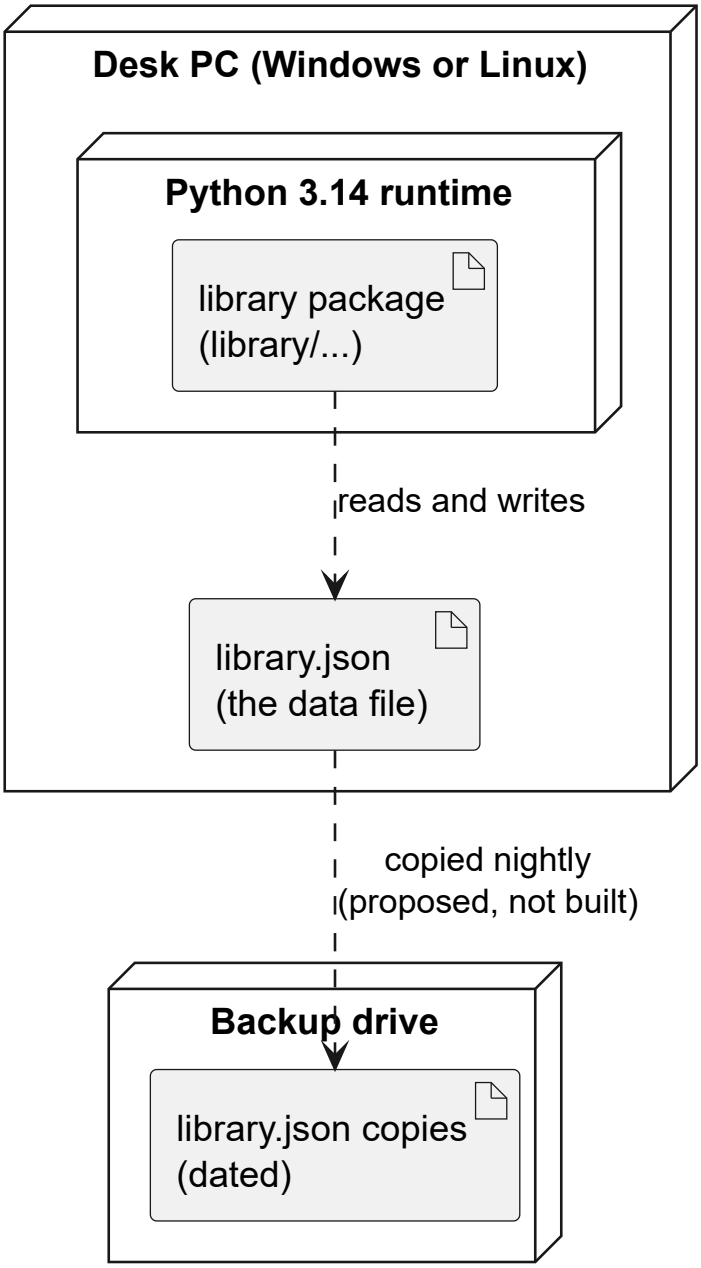
Keeping Dependencies Acyclic

- **Invert the dependency.** If A needs B and B needs A, declare an interface in A's package for what A needs and let B provide it (the `Catalogue` pattern).
- **Move the shared piece down.** If two packages need the same type, put it in a package both may depend on (the domain).
- **Merge** packages that cannot be understood apart: a cycle may be telling you they are one thing.
- **Check automatically**, as above, so a cycle fails the build the day it is introduced.

Deployment Diagrams

A **deployment diagram** shows where software runs: **nodes** (machines or execution environments, drawn as 3D boxes) and the **artifacts** placed on them (programs, files), with lines

for how nodes communicate. It answers "what has to be installed where?". The library's, as I would propose it for a small branch:



PlantUML. **A proposal; nothing was deployed.** The program and its data file are on the desk PC; the nightly copy to a backup drive is marked "proposed, not built".

```
@startuml
scale 1.3
node "Desk PC (Windows or Linux)" as PC {
  node "Python 3.14 runtime" as Py {
    artifact "library package\n(library/...)" as Pkg
  }
```

```
  artifact "library.json\n(the data file)" as Data
}
node "Backup drive" as Backup {
  artifact "library.json copies\n(dated)" as Copies
}
Pkg ..> Data : reads and writes
Data ..> Copies : copied nightly\n(proposed, not built)
@enduml
```

Drawing the deployment found a real problem, because the obvious next question is "what if there are two desks?". The JSON catalogue reads the whole file when it starts and rewrites it after every change. Two desks sharing one file, both started before either lent anything:

```
"""Two desks sharing one data file: what does the JSON catalogue do when both write?"""
import os
import tempfile
from datetime import date

from library.application import LoanDesk
from library.domain import Copy, Member
from library.infrastructure import JsonCatalogue

with tempfile.TemporaryDirectory() as d:
    path = os.path.join(d, "library.json")
    seed = JsonCatalogue(path)
    seed.add(member=Member("M1", date(2027, 1, 1), "standard"))
    seed.add(member=Member("J1", date(2027, 1, 1), "junior"))
    for code in ("B1", "B2"):
        seed.add(copy=Copy(code))
    seed.save()

    desk_a = LoanDesk(JsonCatalogue(path))          # desk A loads the file
    desk_b = LoanDesk(JsonCatalogue(path))          # desk B loads the same file, at the
    desk_a.borrow("M1", "B1", date(2026, 10, 10))   # A lends B1 and saves
    desk_b.borrow("J1", "B2", date(2026, 10, 10))   # B lends B2 and saves, from its old

    final = JsonCatalogue(path)
    print("after both desks have saved, the file says:")
    for code in ("B1", "B2"):
        print(f"   copy {code}: on loan = {final.copy(code).on_loan}")
    print("   M1's loans:", [l.barcode for l in final.member("M1").loans], "| J1's loans
```

```
after both desks have saved, the file says:
   copy B1: on loan = False
   copy B2: on loan = True
   M1's loans: [] | J1's loans: ['B2']
```

Desk A's loan **vanished**: copy B1 is not on loan and member M1 has no loans, though A saw its loan recorded. Desk B saved from an older picture of the world and overwrote A's change, with no error. The JSON component is fine for one desk and is a *correctness* bug for two. The deployment diagram has to say "one desk PC", or the storage component has to be replaced (a database that handles concurrent writers) behind the same `Catalogue` interface, which is exactly what the component diagram makes cheap. I did not build that replacement.

Object Diagrams

An **object diagram** is a snapshot: specific instances, with their actual field values, and the links between them at one moment. Where a class diagram says "a member has any number of loans", an object diagram shows *Ada has these two*. It is for examples: explaining a tricky situation, or checking that your model can represent one. The script below writes one from *live objects* after a short session (two loans for M1, one returned three days late so that a fine was charged, one loan for the junior member):

```
"""Walk live dataclass objects and write a PlantUML object diagram: instances as boxes w
import dataclasses

names: dict[int, str] = {}
labels: dict[int, str] = {}


def name_of(obj) -> str:
    if id(obj) not in names:
        label = {"Member": getattr(obj, "card", ""), "Copy": getattr(obj, "barcode", "")
        names[id(obj)] = f"{type(obj).__name__.lower()}_{len(names)}"
        labels[id(obj)] = f"{label.lower()} : {type(obj).__name__}"
    return names[id(obj)]


def walk(roots) -> str:
    lines, links, seen = ["@startuml", "scale 1.2", "hide empty members"], [], set()
    queue = list(roots)
```

```
    while queue:
        obj = queue.pop(0)
        if id(obj) in seen:
            continue
        seen.add(id(obj))
        ident = name_of(obj)
        lines.append(f'object "{labels[id(obj)]}" as {ident} {{')
        for f in dataclasses.fields(obj):
            value = getattr(obj, f.name)
            children = value if isinstance(value, list) else [value]
            if children and all(dataclasses.is_dataclass(c) for c in children):
                for c in children:
                    queue.append(c)
                    links.append((ident, c, f.name))
            elif not isinstance(value, list):
                lines.append(f"  {f.name} = {value}")
        lines.append("}")
    for src, child, label in links:
        lines.append(f"{src} --> {name_of(child)} : {label}")
    lines.append("@enduml")
    return "\n".join(lines) + "\n"
```

```
"""Draw an OBJECT diagram of what is really in memory: run a few operations, then walk th

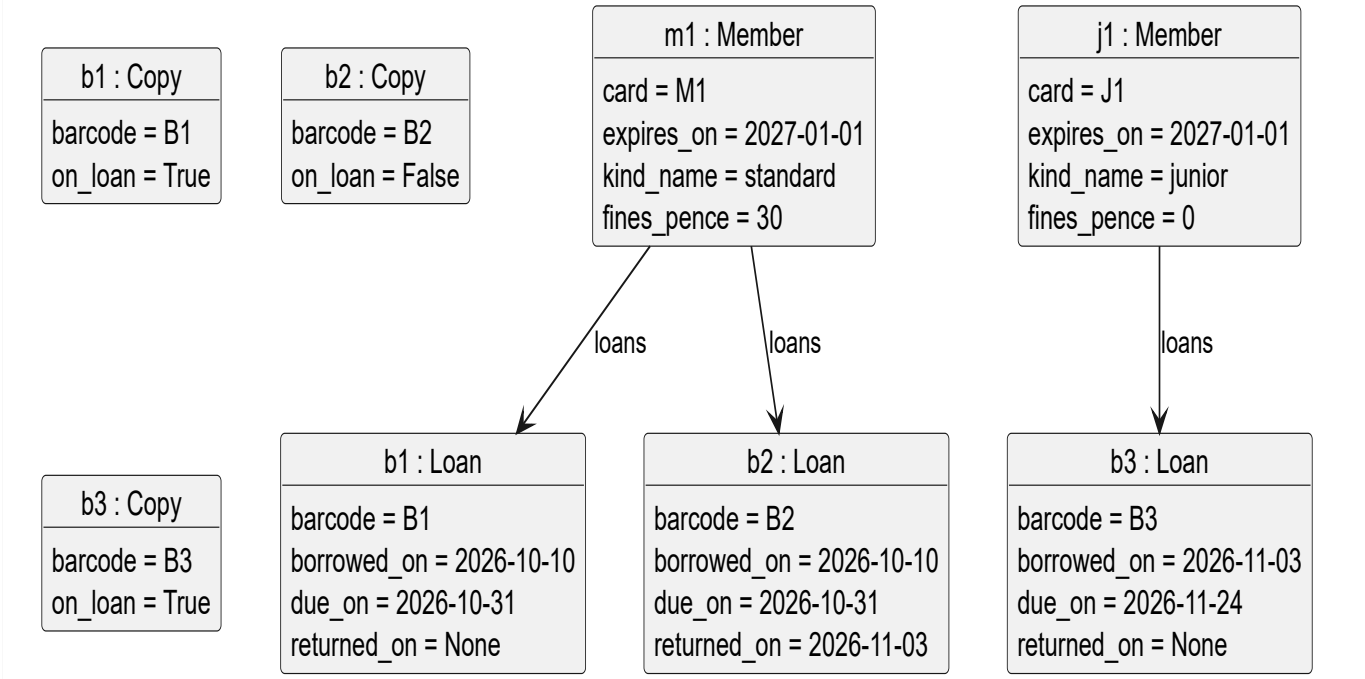
Instances become boxes with their field values; fields that hold other objects (or lists
"""
import pathlib
from datetime import date

from library.application import LoanDesk
from library.domain import Copy, Member
from library.infrastructure import InMemoryCatalogue
from objectdiagram import walk

catalogue = InMemoryCatalogue([Member("M1", date(2027, 1, 1), "standard"), Member("J1", d
                               [Copy("B1"), Copy("B2"), Copy("B3")])
desk = LoanDesk(catalogue)
desk.borrow("M1", "B1", date(2026, 10, 10))
desk.borrow("M1", "B2", date(2026, 10, 10))
desk.return_book("B2", date(2026, 11, 3))              # three days late: a fine
desk.borrow("J1", "B3", date(2026, 11, 3))

text = walk(list(catalogue.members.values()) + list(catalogue.copies.values()))
```

```
pathlib.Path(__file__).with_name("diagrams").joinpath("snapshot.object.puml").write_text
print(text)
```



PlantUML, generated from the running objects: nothing typed by hand. M1's fine of 30 pence, the returned loan, and a copy that is back on the shelf are all real values.

Look for what it shows that the class diagram did not make obvious: **the loans are linked to their members but not to their copies**. A `Loan` holds the copy's barcode as text, not a reference to the `Copy` object, so the three copies sit unconnected beside the loans. That is a design fact (chapter 6's choice), and a few reasons it can be good (a loan record that survives a copy being deleted) and bad (nothing stops a loan pointing at a barcode that does not exist). Exercise 3 draws another scenario.

How These Relate to the C4 Model

The C4 model (Simon Brown; from memory, see *Software Architecture Fundamentals* 9 for the real treatment) describes software at four levels of zoom: **context**, **containers**, **components** and **code**. The diagrams in this course line up roughly like this:

C4 LEVEL	QUESTION IT ANSWERS	NEAREST UML IN THIS COURSE
1 Context	What is the system and who and what does it talk to?	The use case diagram's scope and actors (chapter 3)
2 Containers	What are the separately running, deployable things (programs, databases)?	The deployment diagram's nodes and artifacts
3 Components	What are the major parts inside one container and how do they interact?	The component diagram, backed by the package diagram
4 Code	How is one component built?	The class diagram (chapters 5 and 6)

The two are complementary, not rivals: C4 is a simple, consistent way to organise the zoom levels and name the boxes; UML supplies detailed notations (interfaces, states, interactions) C4 deliberately leaves out. Use C4's levels to decide *which* diagram to draw and UML's notation when a level needs detail.

Usual Mistakes

MISTAKE	WHY IT HURTS	FIX
Components named after files or classes	The diagram repeats the class diagram	A component is a unit you could replace; name what it is for
The interface owned by the provider	The consumer depends on what the provider happens to offer	Declare it with the consumer; providers fit it
A package diagram drawn once, from memory	It stops matching the code at the next import	Generate it from the imports and check rules in a test
Cycles accepted "for now"	They get cemented; one can break imports	Invert, move down, or merge
A deployment diagram with no failure story	It hides single points of failure and concurrent writers (the two-desk loss above)	Ask "what if there are two?" and "what if it dies?" of every node and artifact
Hand-drawn object diagrams of the happy case	They prove the model handles only what you imagined	Draw the awkward situation (the limit, the late return) from real objects

MISTAKES WHILE PREPARING THIS CHAPTER

- **My dependency checker said "clean" while checking nothing.** Its first run printed an empty list of dependencies and "result: clean". My argument parser had taken the output file name after `--puml` as the folder to check, so it analysed zero files. A checker that finds nothing and reports success is worse than no checker. It now prints how many files and imports it analysed, and fails if it finds no files (run on a wrong path it prints "analysed 0 files, found 0 cross-package imports" and "NOTHING WAS ANALYSED"). Correction made in chapter 11: my first guard also failed when there were files but no imports between packages, which is legitimate for independent packages, and chapter 11's exercise 3 ran into it; the script now only fails on zero files. Takeaway: when a check passes, ask what it actually looked at.
- **I assumed an arrow that the code does not have.** My rules allow `infrastructure` to import `application` (to implement its interface). The generated package diagram shows no such edge: with structural protocols, the storage classes never import the interface. I had drawn the component diagram's "provides" and half-expected a matching import. Takeaway: generate the diagram from the code, and be willing to be surprised by it.
- **A rule list with a hole.** Packages not named in the rules are unchecked (above, and Exercise 1). I found it by trying to break my own checker with a new package, and I am leaving the hole in the chapter's script on purpose, with the exercise to close it; a reader should see that a checker's blind spots are a design decision.
- **A tidy-up that was not.** The first version of the object-diagram generator numbered its boxes 0, 2, 4, 6, because labels and names shared a dictionary and doubled the counter. Harmless, but the sort of thing a reader trips on. I split them.

Hands-On Exercises

EXERCISE 1

Add a `reporting` package to a copy of the library whose code imports the UI (it should not). Show that the checker as written calls the tree clean, then write the package's rules and show the violation, then write the rule that would have caught the omission by itself (a package with no rules is an error).

View solution

EXERCISE 2

The bike shop has components Booking, Invoicing, PartsStock and two pricing components, StandardPricing and TradePricing. Invoicing requires a Pricing interface and a PartLookup interface. Draw the component diagram with provided and required interfaces, then write the Python skeleton with each interface declared beside the component that requires it, and tests that both pricing components plug in.

View solution

EXERCISE 3

Generate an object diagram of this situation from live objects: a junior member who already has three loans (the limit) asks for a fourth copy and is refused. Say what the diagram shows and what it cannot show.

View solution

CHAPTER 9 QUICK REFERENCE

- Component: a replaceable unit with a boundary; provided interface (lollipop), required interface (socket); declare the interface with the consumer, so providers fit it
- Package: a grouping of code; arrows = imports; generate the diagram from the code; rules: allowed directions, no cycles, public API only (`__all__`)
- Check the rules as a test; a checker must be seen to fail, and must report how much it analysed; unlisted packages are a blind spot unless you forbid them
- A cycle is not only untidy: here it broke the imports outright
- Deployment: nodes and artifacts; ask "what if there are two?": two desks sharing the JSON file silently lost a loan
- Object diagram: a snapshot of real instances and links; generate it from live objects, and draw the awkward case
- C4 levels (context, containers, components, code) pick which diagram to draw; UML gives the detailed notation for a level

COMING NEXT

Object-Oriented Design with UML 10 (From UML to Code and Back): mapping diagrams to Python (`dataclass`, `ABC`, `Protocol`, composition, enums), what each relationship becomes in code, a note

on the same design in Java (labelled not run here), reverse engineering with `pyreverse` and what it shows and omits, and keeping diagrams honest.

Object-Oriented Design with UML

Course 1 · Chapter 10 · From UML to Code and Back

A diagram and the code it describes are two descriptions of one thing, and nothing keeps them in step. This chapter looks at the three movements between them: **forward** (from a diagram to code), **reverse** (from code to a diagram), and **keeping them honest** (noticing when they have drifted apart). Each is done for real with Python, and the results include some disappointments, which are the useful part.

WHAT WAS RUN, AND WHAT WAS NOT

Run here: a **generator I wrote** (about 150 lines) that reads a Mermaid class diagram and writes a Python skeleton; tests of what that skeleton does; `pyreverse` from **pylint 4.1.2** (with astroid 4.3.4, installed in a scratch virtual environment, not globally) run on the library from chapter 9 and on the generated code; a round-trip comparison; and a small drift check. Python 3.14, Mermaid 12.1.0 for rendering. **Not run:** anything in Java (**Java is not installed here**), so the Java column below is from general knowledge and was *not compiled*, and any commercial model-driven tool. The history of model-driven generation and why it faded is also general knowledge, not research.

What Each Diagram Element Becomes

Chapter 5 showed each relationship as running Python. Here they are in one table, with the Java equivalent for readers coming from there. **The Java column was not compiled or run**; treat it as a guide, not a tested reference.

IN THE DIAGRAM	PYTHON	JAVA (NOT RUN)
Class with attributes	`@dataclass` with typed fields (or a plain class with `__init__`)	A class with fields, or a `record` for an immutable value
Operation	A method; the type hints carry the signature	A method
Visibility `+ - # ~`	Name conventions (`_x`, `__x`); not enforced	`public private protected` and package-private; enforced
Static member	`@staticmethod`, `@classmethod`, a class attribute	`static`

IN THE DIAGRAM	PYTHON	JAVA (NOT RUN)
Abstract class / operation	`ABC` with `@abstractmethod`	`abstract class`, `abstract` method
Interface («interface»)	`typing.Protocol` (structural) or an ABC (nominal)	`interface` (nominal: a class must say `implements`)
Enumeration	`enum.Enum`	`enum`
Generalisation	Subclass: `class Standard(Policy)`	`extends`
Realisation	A class with the right methods (Protocol), or inheriting an ABC	`implements`
Association, multiplicity 1	A field holding the object; required in `__init__`	A field, set in the constructor
0..1	`Type | None = None`	A nullable field (or `Optional`)
0..* or 1..*	`list[Type]` with `field(default_factory=list)`	`List<Type>`
Composition	The whole creates the parts (`add_copy`); parts hold no independent reference	Same: the whole creates and owns; `final` list
Dependency	A parameter or local variable; no field	Same
Constraint (1..*, "at most one open")	Code you must write (a check in a method), plus a test	Same

For illustration only, and not compiled, the library's `Member`, `MemberKind` and `Loan` would read something like this in Java:

```
// NOT COMPILED OR RUN: Java is not installed on the machine this course was written on
public interface MemberKind { int loanLimit(); int finePerDay(); }

public final class Standard implements MemberKind {
    public int loanLimit() { return 5; }
    public int finePerDay() { return 10; }
}

public record Loan(String barcode, java.time.LocalDate borrowedOn, java.time.LocalDate d

public class Member {
    private final String card;
```

```
    private final MemberKind kind;
    private final java.util.List<Loan> loans = new java.util.ArrayList<>();   // composi
    public Member(String card, MemberKind kind) { this.card = card; this.kind = kind; }
}
```

The main differences to notice are the ones the table points at: Java enforces visibility and requires interfaces to be declared with `implements`, while Python's `Protocol` lets the "provides" arrow of chapter 9 exist only in the design.

Forward: From a Diagram to a Skeleton

The input is a Mermaid class diagram (the notation of chapter 5):

```
classDiagram
    class CopyStatus {
        <<enumeration>>
        AVAILABLE
        ON_LOAN
        WITHDRAWN
    }
    class FineCalculator {
        <<interface>>
        +fine_for(due_on: date, returned_on: date) int
    }
    class DailyRateFine {
        +pence_per_day: int
        +fine_for(due_on: date, returned_on: date) int
    }
    class Policy {
        <<abstract>>
        +limit()* int
    }
    class StandardPolicy
    class JuniorPolicy
    class Title {
        +isbn: str
        +name: str
    }
    class Copy {
        +barcode: str
        +status: CopyStatus
```

```
        +lend() None
    }
    class Member {
        +name: str
        +card: str
        +policy: Policy
        +borrow(copy: Copy, today: date) Loan
    }
    class Loan {
        +borrowed_on: date
        +due_on: date
    }
    DailyRateFine ..|> FineCalculator
    StandardPolicy --|> Policy
    JuniorPolicy --|> Policy
    Title "1" *-- "1..*" Copy
    Member "1" --> "0..*" Loan
    Loan "0..*" --> "1" Copy
```

A small generator reads it. Its source is the best documentation of what it does and what it cannot:

```
"""Generate a Python skeleton from a Mermaid class diagram (a small subset of the notati

    py uml2py.py design.mmd generated_library.py

Understood: <<enumeration>>, <<interface>>, <<abstract>>; attributes 'name: type'; opera
relationships: --|> (generalisation), ..|> (realisation), *-- (composition), --> (associ
What it cannot do is the whole point of the chapter: it generates structure, and no beha
"""
import re
import sys
from dataclasses import dataclass, field

REL = re.compile(r'^\s*(\w+)\s*(?:"([^"]+)"\s*)?(--\|>|\.\.\|>|\*--|-->)\s*(?:"([^"]+)"\


@dataclass
class Cls:
    name: str
    kind: str = "class"                      # class | enumeration | interface | abstra
    attrs: list = field(default_factory=list)         # (name, type)
    ops: list = field(default_factory=list)           # (name, params, ret, abstract)
```

```
    values: list = field(default_factory=list)
    base: str | None = None
    realises: list = field(default_factory=list)
    parts: list = field(default_factory=list)          # (type, multiplicity) composed
    refs: list = field(default_factory=list)           # (type, multiplicity) associated


def parse(text: str) -> dict[str, Cls]:
    classes: dict[str, Cls] = {}
    current = None
    for raw in text.splitlines():
        line = raw.strip()
        if not line or line == "classDiagram":
            continue
        m = re.match(r"class (\w+)\s*(\{)?$", line)
        if m:
            current = classes.setdefault(m.group(1), Cls(m.group(1)))
            if not m.group(2):
                current = None
            continue
        if line == "}":
            current = None
            continue
        if current is not None:
            stereo = re.match(r"<<(\w+)>>", line)
            if stereo:
                current.kind = stereo.group(1)
                continue
            op = re.match(r"\+(\w+)\(([^)]*)\)(\*)?\s*(\w[\w\[\], |]*)?$", line)
            if op:
                current.ops.append((op.group(1), op.group(2), op.group(4) or "None", boo
                continue
            attr = re.match(r"\+(\w+):\s*(\S+)$", line)
            if attr:
                current.attrs.append((attr.group(1), attr.group(2)))
                continue
            if current.kind == "enumeration" and re.match(r"^\w+$", line):
                current.values.append(line)
                continue
            raise ValueError(f"cannot read a member line in {current.name}: {line!r}")
        r = REL.match(line)
        if r:
            left, lm, arrow, rm, right = r.groups()
            classes.setdefault(left, Cls(left))
            classes.setdefault(right, Cls(right))
```

```
            if arrow == "--|>":
                classes[left].base = right
            elif arrow == "..|>":
                classes[left].realises.append(right)
            elif arrow == "*--":
                classes[left].parts.append((right, rm or "1"))
            else:
                classes[left].refs.append((right, rm or "1"))
            continue
        raise ValueError(f"cannot read line: {line!r}")
    return classes


def link_name(target: str, mult: str) -> str:
    """The field name for a link: CamelCase to snake_case, plural for 'many' links (Copy
    name = re.sub(r"(?<!^)(?=[A-Z])", "_", target).lower()
    if mult in ("0..*", "1..*", "*"):
        name = name[:-1] + "ies" if name.endswith("y") else name + "s"
    return name


def field_type(target: str, mult: str) -> tuple[str, str]:
    """(annotation, default) for a link to `target` with this multiplicity."""
    if mult in ("0..*", "1..*", "*"):
        return f"list[{target}]", " = field(default_factory=list)"
    if mult == "0..1":
        return f"{target} | None", " = None"
    return target, ""


def emit(classes: dict[str, Cls]) -> str:
    order = [c for c in classes.values() if c.kind == "enumeration"] + [c for c in class
    placed = {c.name for c in order}
    rest = [c for c in classes.values() if c.name not in placed]
    while rest:                                                  # bases before s
        for c in list(rest):
            if c.base is None or c.base in placed:
                order.append(c)
                placed.add(c.name)
                rest.remove(c)
    uses_date = any("date" in t for c in classes.values() for _, t in c.attrs) or any("d
    out = ['"""Generated from the class diagram by uml2py.py. Structure only: every oper
           "from __future__ import annotations", "", "from abc import ABC, abstractmetho
           "from dataclasses import dataclass, field"]
    if uses_date:
```

```
        out.append("from datetime import date")
    out += ["from enum import Enum", "from typing import Protocol, runtime_checkable", "
    for c in order:
        out += ["", ""]
        if c.kind == "enumeration":
            out.append(f"class {c.name}(Enum):")
            out += [f'    {v} = "{v}"' for v in c.values] or ["    pass"]
        elif c.kind == "interface":
            out += ["@runtime_checkable", f"class {c.name}(Protocol):"]
            out += [f"    def {n}(self{', ' + p if p else ''}) -> {r}: ..." for n, p, r,
        else:
            bases = [c.base] if c.base else (["ABC"] if c.kind == "abstract" else [])
            if c.kind != "abstract":
                out.append("@dataclass")
            out.append(f"class {c.name}({', '.join(bases)}):" if bases else f"class {c.n
            body = []
            required = [f"    {n}: {t}" for n, t in c.attrs]
            defaulted = []
            for target, mult in c.parts + c.refs:
                ann, default = field_type(target, mult)
                name = link_name(target, mult)
                (defaulted if default else required).append(f"    {name}: {ann}{default}
            if c.kind != "abstract":
                body += required + defaulted
            own = {n for n, _, _, _ in c.ops}
            stubs = list(c.ops)
            for iface in c.realises:                                    # a realised in
                stubs += [(n, p, r, a) for n, p, r, a in classes[iface].ops if n not in
            for n, p, r, abstract in stubs:
                if abstract or c.kind == "abstract" and abstract:
                    body += ["", "    @abstractmethod", f"    def {n}(self{', ' + p if p
                else:
                    body += ["", f"    def {n}(self{', ' + p if p else ''}) -> {r}:", f'
            out += body or ["    pass"]
    return "\n".join(out) + "\n"


if __name__ == "__main__":
    src, dst = sys.argv[1], sys.argv[2]
    code = emit(parse(open(src, encoding="utf8").read()))
    open(dst, "w", encoding="utf8", newline="\n").write(code)
    print(f"wrote {dst}: {len(code.splitlines())} lines")
```

The skeleton it wrote:

```
"""Generated from the class diagram by uml2py.py. Structure only: every operation is a s
from __future__ import annotations

from abc import ABC, abstractmethod
from dataclasses import dataclass, field
from datetime import date
from enum import Enum
from typing import Protocol, runtime_checkable



class CopyStatus(Enum):
    AVAILABLE = "AVAILABLE"
    ON_LOAN = "ON_LOAN"
    WITHDRAWN = "WITHDRAWN"


@runtime_checkable
class FineCalculator(Protocol):
    def fine_for(self, due_on: date, returned_on: date) -> int: ...


@dataclass
class DailyRateFine:
    pence_per_day: int

    def fine_for(self, due_on: date, returned_on: date) -> int:
        raise NotImplementedError("DailyRateFine.fine_for")


class Policy(ABC):

    @abstractmethod
    def limit(self) -> int: ...


@dataclass
class StandardPolicy(Policy):
    pass


@dataclass
```

```
class JuniorPolicy(Policy):
    pass


@dataclass
class Title:
    isbn: str
    name: str
    copies: list[Copy] = field(default_factory=list)


@dataclass
class Copy:
    barcode: str
    status: CopyStatus

    def lend(self) -> None:
        raise NotImplementedError("Copy.lend")


@dataclass
class Member:
    name: str
    card: str
    policy: Policy
    loans: list[Loan] = field(default_factory=list)

    def borrow(self, copy: Copy, today: date) -> Loan:
        raise NotImplementedError("Member.borrow")


@dataclass
class Loan:
    borrowed_on: date
    due_on: date
    copy: Copy
```

What the Skeleton Keeps, and What It Lost

Nine tests ask the skeleton what the diagram promised. Six check promises the code keeps; three record what the diagram never said (note the second group's names):

```
"""What does the generated skeleton actually do? Each test states a claim the diagram ma
import unittest
from datetime import date

import generated_library as g


class WhatTheDiagramPromises(unittest.TestCase):
    def test_enumeration(self):
        self.assertEqual([s.name for s in g.CopyStatus], ["AVAILABLE", "ON_LOAN", "WITHDR
        with self.assertRaises(ValueError):
            g.CopyStatus("LOST")

    def test_abstract_class_cannot_be_created(self):
        with self.assertRaises(TypeError):
            g.Policy()

    def test_composition_gives_the_whole_a_list_of_parts(self):
        t = g.Title("111", "Dune")
        t.copies.append(g.Copy("B1", g.CopyStatus.AVAILABLE))
        self.assertEqual(len(t.copies), 1)

    def test_association_with_exactly_one_is_a_required_field(self):
        with self.assertRaises(TypeError):
            g.Loan(date(2026, 10, 10), date(2026, 10, 31))             # the diagram says

    def test_the_realising_class_has_the_interface_operation(self):
        self.assertIsInstance(g.DailyRateFine(10), g.FineCalculator)

    def test_operations_are_stubs(self):
        with self.assertRaises(NotImplementedError):
            g.DailyRateFine(10).fine_for(date(2026, 10, 10), date(2026, 10, 12))


class WhatTheDiagramDidNotSay(unittest.TestCase):
    def test_the_concrete_policies_cannot_be_created_yet(self):
        """The diagram draws StandardPolicy under Policy but, as UML conventionally does
        The generated StandardPolicy therefore inherits an abstract operation it never i
        with self.assertRaises(TypeError):
            g.StandardPolicy()

    def test_the_generation_gap_way_to_finish_it(self):
        """Do not edit the generated file (the next generation overwrites it): add behav
        class Standard(g.StandardPolicy):
```

```
            def limit(self) -> int:
                return 5
        self.assertEqual(Standard().limit(), 5)

    def test_nothing_enforces_the_rules_the_diagram_could_not_say(self):
        """Multiplicity 1..* (a Title has at least one Copy) is not enforced: a Title ca
        self.assertEqual(g.Title("111", "Dune").copies, [])


if __name__ == "__main__":
    unittest.main(verbosity=2)
```

```
----------------------------------------------------------------------
Ran 9 tests in 0.000s

OK
```

- **Kept:** the enumeration rejects an unknown value; the abstract class cannot be created; composition gave `Title` a list of `Copy`; a link of multiplicity exactly one is a required field (omitting it raises `TypeError`); the realising class has the interface's operation; the operations are stubs.
- **Not said, and therefore broken:** `StandardPolicy` and `JuniorPolicy` cannot be created. The diagram draws them under `Policy` and, as UML conventionally does, does not repeat the inherited operation `limit()` in each subclass box, but the code needs each concrete class to implement it. *A diagram can leave out what a compiler cannot.*
- **Not expressible:** multiplicity `1..*` (a title has at least one copy) is not enforced; a `Title` is created with none. The generator cannot turn a constraint into behaviour.

Never Edit the Generated File

Generated code that you edit is overwritten the next time you generate, and generated code that you do not edit stays a stub. The standard way out is the **generation gap**: keep the generated classes untouched and add behaviour in hand-written *subclasses* in another file. One test above does exactly that (`class Standard(g.StandardPolicy)` with `limit()`). Exercise 1 does it for a whole design.

What the generator cannot do at all is the point: it produced structure, and none of the behaviour (the fines, the limits, the rules). Behaviour is chapters 7 and 8's diagrams plus your

head. Forward generation is a head start on boilerplate, not on design.

Reverse: From Code to a Diagram

`pyreverse`, part of `pylint`, reads Python source and writes class and package diagrams (I used its Mermaid output; its help lists other output formats, which I did not try). I ran it on the library from chapter 9 (`pyreverse -o mmd -p library library`) and rendered the Mermaid it wrote:



Mermaid rendered from pyreverse's output on the real code; I changed nothing. It is wide because it is one row of eleven boxes with only two connected groups.

Compared with the diagram I drew by hand for the same design, this is what it **shows** well:

- **Every class and every method with its types**, read from the code (`fine(today: date, kind: MemberKind) int`): more complete and more exact than my hand drawing.
- **Inheritance** (`Junior`, `Standard` to `MemberKind`), and a hollow-diamond link where one class receives another in its constructor (`LoanDesk` holds a `Catalogue`).

And what it **leaves out or gets wrong**. I checked each against the code:

WHAT THE CODE HAS	WHAT PYREVERSE DREW
`Member.loans: list[Loan]`, a member owns its loans	The text `loans : list[Loan]` in the box, **no arrow to** `Loan`: a collection of objects is not drawn as an association
`InMemoryCatalogue` and `JsonCatalogue` provide `Catalogue`	**No realisation arrows.** They match the Protocol by shape and never name it, so nothing in the source says "implements". `Catalogue` appears as a plain class, not an interface
`InMemoryCatalogue.save()` is an empty method that does nothing, on purpose	Drawn as `save()*`: **marked abstract**, which it is not. The tool treats an empty-bodied method as abstract. The same happened to stubs that `raise NotImplementedError` in the generated code

WHAT THE CODE HAS	WHAT PYREVERSE DREW
`Member.kind`, a property computed from `kind_name`	Both `kind : MemberKind` and `kind_name : str` as attributes, with no link to `MemberKind`
`Loan` refers to its copy by barcode text (chapter 9's object diagram)	Nothing: correctly, there is no link in the code
`Refusal`, an exception class used everywhere	A lone empty box, connected to nothing
Multiplicities, roles, constraints, and the reason for any design choice	None: the code does not contain them

Its package diagram is also a different thing from chapter 9's. It draws *modules* as boxes (`desk`, `ports`, `model`, `jsonfile` and so on) with 14 import arrows, where chapter 9's script reduced the same code to four packages and five arrows with rules attached. Neither is wrong; the second answers an architectural question and the first lists imports.

The Round Trip

The cleanest test of reverse engineering is to go **diagram, code, diagram**: reverse-engineer the generated code from above and compare the relationships with those in the original design:

```
"""Diagram -> code -> diagram. Reverse-engineer code with pyreverse and compare the resu

    py roundtrip.py            (design.mmd against generated_library.py)
"""
import pathlib
import re
import subprocess

import uml2py

HERE = pathlib.Path(__file__).parent
PYREVERSE = HERE.parent / "pylintenv" / "Scripts" / "pyreverse.exe"


def expected_relationships(design: dict) -> set:
    """What the design says, as (kind, a, b). An attribute whose type is another class i
    out = set()
    for c in design.values():
        if c.base:
```

```
            out.add(("inherits", c.name, c.base))
        for i in c.realises:
            out.add(("realises", c.name, i))
        for t, _ in c.parts + c.refs:
            out.add(("has", c.name, t))
    return out


def found_relationships(mermaid: str) -> set:
    out = set()
    for line in mermaid.splitlines():
        m = re.match(r"\s*(\w+) --\|> (\w+)", line)
        if m:
            out.add(("inherits", m.group(1), m.group(2)))
        m = re.match(r"\s*(\w+) --([*o]) (\w+)", line)             # 'Part --* Whole' an
        if m:
            out.add(("has", m.group(3), m.group(1)))
        m = re.match(r"\s*(\w+) --> (\w+)", line)
        if m:
            out.add(("has", m.group(1), m.group(2)))
    return out


def compare(design_path, code_path, workdir) -> str:
    design = uml2py.parse(pathlib.Path(design_path).read_text(encoding="utf8"))
    out = pathlib.Path(workdir)
    out.mkdir(exist_ok=True)
    subprocess.run([str(PYREVERSE), "-o", "mmd", "-p", "rt", str(code_path)], cwd=out, c
    recovered = (out / "classes_rt.mmd").read_text(encoding="utf8")
    expected, found = expected_relationships(design), found_relationships(recovered)
    lines = [f"design: {len(design)} classes, {len(expected)} relationships | recovered
    lines += [f"   in the design, NOT recovered : {e[1]} {e[0]} {e[2]}" for e in sorted(
    lines += [f"   recovered, NOT in the design : {f[1]} {f[0]} {f[2]}" for f in sorted(
    lines += [f"   recovered                   : {e[1]} {e[0]} {e[2]}" for e in sorted(
    return "\n".join(lines)


if __name__ == "__main__":
    print(compare(HERE / "design.mmd", HERE / "generated_library.py", HERE / "rt"))
```

```
design: 10 classes, 6 relationships | recovered from the code: 5
   in the design, NOT recovered : Member has Loan
   in the design, NOT recovered : Title has Copy
```

```
in the design, NOT recovered : DailyRateFine realises FineCalculator
recovered, NOT in the design : Copy has CopyStatus
recovered, NOT in the design : Member has Policy
recovered                    : Loan has Copy
recovered                    : JuniorPolicy inherits Policy
recovered                    : StandardPolicy inherits Policy
```

Three relationships made the whole trip (inheritance twice, and `Loan` to `Copy`). Three were lost: `Member` to `Loan` and `Title` to `Copy` (both collections), and the realisation of `FineCalculator`. Two appeared that were not in the relationship list: `Member` to `Policy` and `Copy` to `CopyStatus`. Those two are not errors: my design wrote `policy: Policy` as an attribute, and pyreverse correctly reads an attribute of another class's type as an association. The difference is notational, and it is a reminder that the same fact can be drawn as an attribute or as an arrow. The lesson: **a round trip is lossy.** Whatever you draw by hand should not depend on a tool being able to recover it.

Model-Driven Generation, and Why It Faded

From general knowledge, not researched for this course: in the 2000s there was a strong push for **model-driven** development, where the model, not the code, was the source: tools generated whole applications from UML and you never edited the output (Fowler's "UML as programming language" mode from chapter 1). It faded for mostly practical reasons. Behaviour needed a code-like language anyway, so the model became a clumsy programming language. Generated code was awkward to read and debug. Keeping model and hand-written code in step was painful (the generation-gap problem above, at scale), and the heavy tools got in the way of ordinary tools: version control, diffing, code review, refactoring. What survived are the *narrow* forms: generating code from interface and schema definitions, ORM models, and API descriptions, where the input is small, precise and has no behaviour. Mine, above, is the same narrow form.

Keeping Diagrams Honest

A diagram that disagrees with the code is worse than none: it is trusted and wrong. The choices, from cheapest up:

- **Draw less, and say when.** A small sketch dated at the top, kept as text next to the code, is honest about being a snapshot.
- **Generate it** (pyreverse; chapter 9's package diagram; the object diagram), so it cannot drift, at the price of being only what the tool can read.
- **Check it.** Chapter 7's conformance test checks a sequence diagram against calls; chapter 9's rules check package dependencies; here is the equivalent for class diagrams, comparing the operations on the hand-drawn chapter 6 diagram with the methods in the code:

```
"""Has the hand-drawn class diagram drifted from the code? Compare the operations on eacl
import ast
import pathlib
import re
import sys

HERE = pathlib.Path(__file__).parent


def diagram_ops(path: pathlib.Path) -> dict[str, set[str]]:
    ops, current = {}, None
    for line in path.read_text(encoding="utf8").splitlines():
        c = re.match(r"\s*class (\w+)", line)
        if c:
            current = c.group(1)
            ops.setdefault(current, set())
            continue
        o = re.match(r"\s*\+(\w+)\(.*\)", line)
        if o and current:
            ops[current].add(o.group(1))
    return ops


def code_ops(path: pathlib.Path) -> dict[str, set[str]]:
    out = {}
    for node in ast.parse(path.read_text(encoding="utf8")).body:
        if isinstance(node, ast.ClassDef):
            out[node.name] = {f.name for f in node.body if isinstance(f, ast.FunctionDef
    return out


if __name__ == "__main__":
    diagram, code = diagram_ops(HERE / "version_b.class.mmd"), code_ops(HERE / "library_l
    print(f"diagram: {len(diagram)} classes | code: {len(code)} classes")
```

```
    for name in sorted(set(diagram) | set(code)):
        if name not in code:
            print(f"   {name}: in the diagram, no such class in the code")
        elif name not in diagram:
            print(f"   {name}: in the code, not in the diagram (methods: {sorted(code[na
        else:
            gone, undrawn = sorted(diagram[name] - code[name]), sorted(code[name] - diag
            if gone or undrawn:
                print(f"   {name}:" + (f" drawn but not in the code {gone}" if gone else
    print("   (an empty report would mean the operations match exactly)")
```

```
diagram: 8 classes | code: 9 classes
   Junior: in the code but not drawn ['fine_per_day', 'loan_limit']
   Member: in the code but not drawn ['open_loans']
   Refusal: in the code, not in the diagram (methods: none)
   Standard: in the code but not drawn ['fine_per_day', 'loan_limit']
   (an empty report would mean the operations match exactly)
```

It found no wrong operations, but it found that my chapter 6 diagram **omitted** some: `Member.open_loans` is not drawn, and the `Junior` and `Standard` overrides are not drawn (a deliberate convention, as above), plus `Refusal`, which I left out as noise. A check can list the differences; deciding which are deliberate is still a human job. The check is more useful when a change breaks something. Here someone renames `check_can_be_lent` to `ensure_lendable` in the code and does not touch the diagram:

```
Copy: drawn but not in the code ['check_can_be_lent']; in the code but not drawn ['ensur
```

That is the signal you want in a build. Run it with the tests, and it fails the day the name changes, not the day someone is misled by the diagram.

WHAT THIS CHECK CANNOT CATCH

It compares names. It will not notice that a drawn operation's meaning changed, that a multiplicity is now wrong, or that an arrow's direction no longer matches the dependencies. For arrows, use the generated diagram or a dependency check; for behaviour, use tests.

Usual Mistakes

MISTAKE	FIX
Editing generated code	Generation gap: subclass in a hand-written file, or generate once and own the result
Expecting reverse engineering to recover the design	It recovers structure, not intent: no multiplicities, roles, constraints, or reasons
Trusting a tool's picture without reading its legend	pyreverse's `*` marked empty methods "abstract"; check against the code
A diagram nobody checks	Date it, generate it, or put a drift check in the build
Treating a skeleton as progress	Behaviour is the work; a generated skeleton is boilerplate, not design
Leaving the diagram silent on what the code needs	If a concrete subclass must implement an operation, say so, or the next reader (or generator) will not know

MISTAKES WHILE PREPARING THIS CHAPTER

- **The generator had the plural bug from chapter 4.** Its first output named the list of copies `copys`, the same "add an s" mistake I made in chapter 4's sentences, and I only saw it by reading the generated code. I had not carried the lesson into new code. It now maps a trailing "y" to "ies" and converts CamelCase to snake_case (`job_parts`). Takeaway: when you have made a mistake in one tool, grep your next tool for it.
- **I assumed pyreverse would draw the collection.** I expected an arrow from `Member` to `Loan`. It drew only the text of the field type. I would not have noticed if I had not compared the picture with the code line by line. Takeaway: reverse-engineered diagrams are claims too; check them.
- **A refactoring script that broke its own file.** I turned `roundtrip.py` into a function with a line-by-line text transformation and it failed with `IndentationError`. Rewriting the file by hand took less time than the script had saved. Takeaway: restructure code with an editor, not a string-munging script, and run it straight away.
- **The first design left a hole I did not see until the tests ran.** The diagram drew the two concrete policies and no operations in them; the generated classes could not be created. It is in the chapter as a finding, but it was a surprise to me, and a good argument for generating code from a design early.

Hands-On Exercises

EXERCISE 1

Run the generator on the bike shop design in the solution file (an enumeration, an interface with an implementation, a composition and two associations). Leave the generated file alone and finish the behaviour by subclassing in a second file, with tests, including one that shows a required link cannot be omitted.

View solution

EXERCISE 2

Reverse-engineer the bike shop code from exercise 1 with pyreverse and compare the relationships with the design. List what was lost and what was gained, and say which losses are the tool's limits and which are limits of the code itself.

View solution

EXERCISE 3

Extend the drift idea from operations to attributes: compare the attributes in a class diagram with the fields in the code (allowing for the link fields the generator adds), show that an untouched generated file has no drift, and that a hand edit (renaming one field, adding another) is reported.

View solution

CHAPTER 10 QUICK REFERENCE

- Class to dataclass; interface to Protocol (structural) or ABC; abstract to ABC; enumeration to Enum; generalisation to subclass; association to a field by multiplicity (1 required, 0..1 optional, many a list); composition = the whole creates the parts; constraints are code you must write
- Forward generation gives structure and boilerplate, no behaviour; a diagram may omit what code needs (inherited operations the subclass must implement); never edit generated code, use the generation gap (subclass elsewhere)
- pyreverse: shows classes, methods, types, inheritance; misses collections as associations, structural realisation, properties; marked an empty method abstract; a round trip is lossy

- Model-driven generation as "the model is the program" faded; narrow generation (schemas, interfaces, ORM) survived (general knowledge)
- Keep diagrams honest: draw less and date it, generate it, or check it (conformance, dependency rules, a drift check on names); a check compares names, not meaning
- Java mapping given here was not compiled

COMING NEXT

Object-Oriented Design with UML 11 (Patterns, Principles and Design Review): showing Strategy, Observer, State, Factory and Composite on a class diagram (linking to *Design Patterns*, not re-teaching), principles on diagrams, a design review checklist, spotting a god class, a cyclic dependency and an anemic model on a diagram, and refactoring the library on the diagram first and then in code.

Object-Oriented Design with UML

Course 1 · Chapter 11 · Patterns, Principles and Design Review

By now you can draw the library six ways. This chapter is about **judging** a design from its diagrams: recognising a known shape (a design pattern), spotting a bad one (a god class, a cycle, an anemic model), and changing a design safely, on paper first. It finishes with a real change to the library, made in the order a careful team would work: review the current design, redraw it, then change the code and check that nothing broke.

WHAT WAS RUN, AND WHAT WAS NOT

Run here (Python 3.14, Mermaid 12.1.0): the library from chapter 9 refactored for two new requirements (the 10 old tests pass unchanged, and 13 more were added); five pattern diagrams each **checked against the code** by a script, which I also tampered with to prove it can fail; three smell detectors I wrote, run on the course's own code and on a deliberately flawed module; and a dependency check on a package cycle and its fix. **Not re-read:** the pattern catalogue (the "Gang of Four" book) is from general knowledge; patterns are explained, with their trade-offs, in *Design Patterns* (State is its chapter 8), and SOLID in *Clean Code, SOLID & Refactoring* 6 and 7. This chapter shows the *diagrams*, and does not re-teach the patterns. The detectors' thresholds are my own and arbitrary; they are prompts for a review, not verdicts.

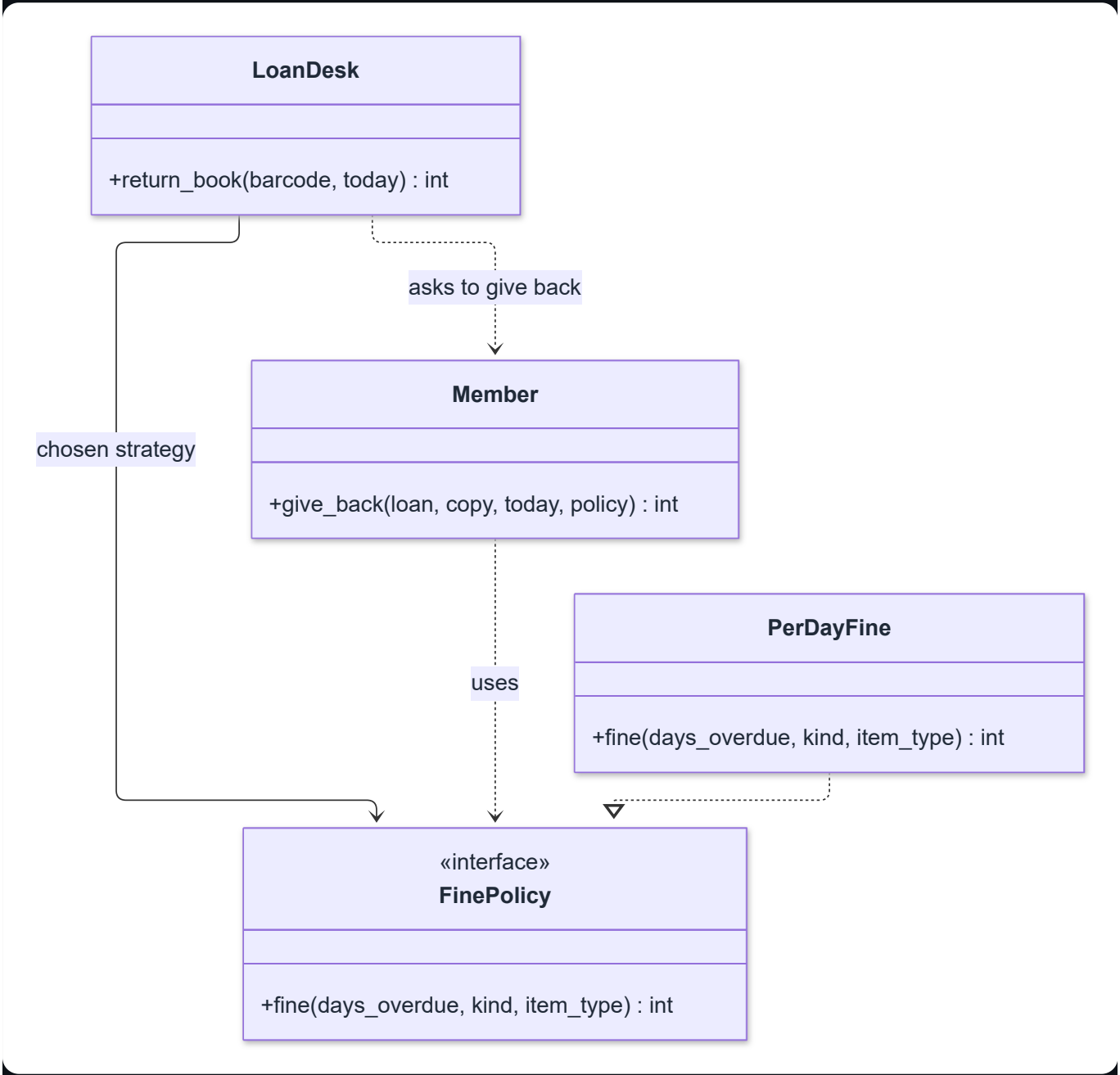
Five Patterns, Drawn and Checked

A pattern is a recurring arrangement of classes that solves a recurring problem. On a diagram it is a *shape*: if you can recognise the shape, you know the intent without reading the code. For each pattern below there is a diagram using the library's own names, the code it describes, and a note on where the diagram and code agree.

PATTERN	PROBLEM IT SOLVES	IN THE LIBRARY
Strategy	A rule that varies, to be chosen at run time	`FinePolicy` (how a fine is worked out), added in this chapter
Observer	Tell interested parties when something happens, without the source knowing them	`LoanDesk` publishes events to `Subscriber`s
State	Behaviour that depends on the object's state	The loan lifecycle of chapter 8, one class per state

PATTERN	PROBLEM IT SOLVES	IN THE LIBRARY
Factory	One place that decides which class to create	`KindFactory` creating the right `MemberKind`
Composite	Treat one thing and a group of things the same way	`Holdings`: a `Single` copy or a `Shelf` of them

Strategy



The shape: a client that holds (or is given) an interface, and several classes that realise it. `Member.give_back` uses a `FinePolicy` without knowing which; `PerDayFine` is one.

```python
"""How a fine is worked out. A Strategy: the rule can be replaced without touching Membe
from __future__ import annotations

from typing import TYPE_CHECKING, Protocol, runtime_checkable

if TYPE_CHECKING:                                   # only for the type hints: no import
    from library.domain.model import MemberKind

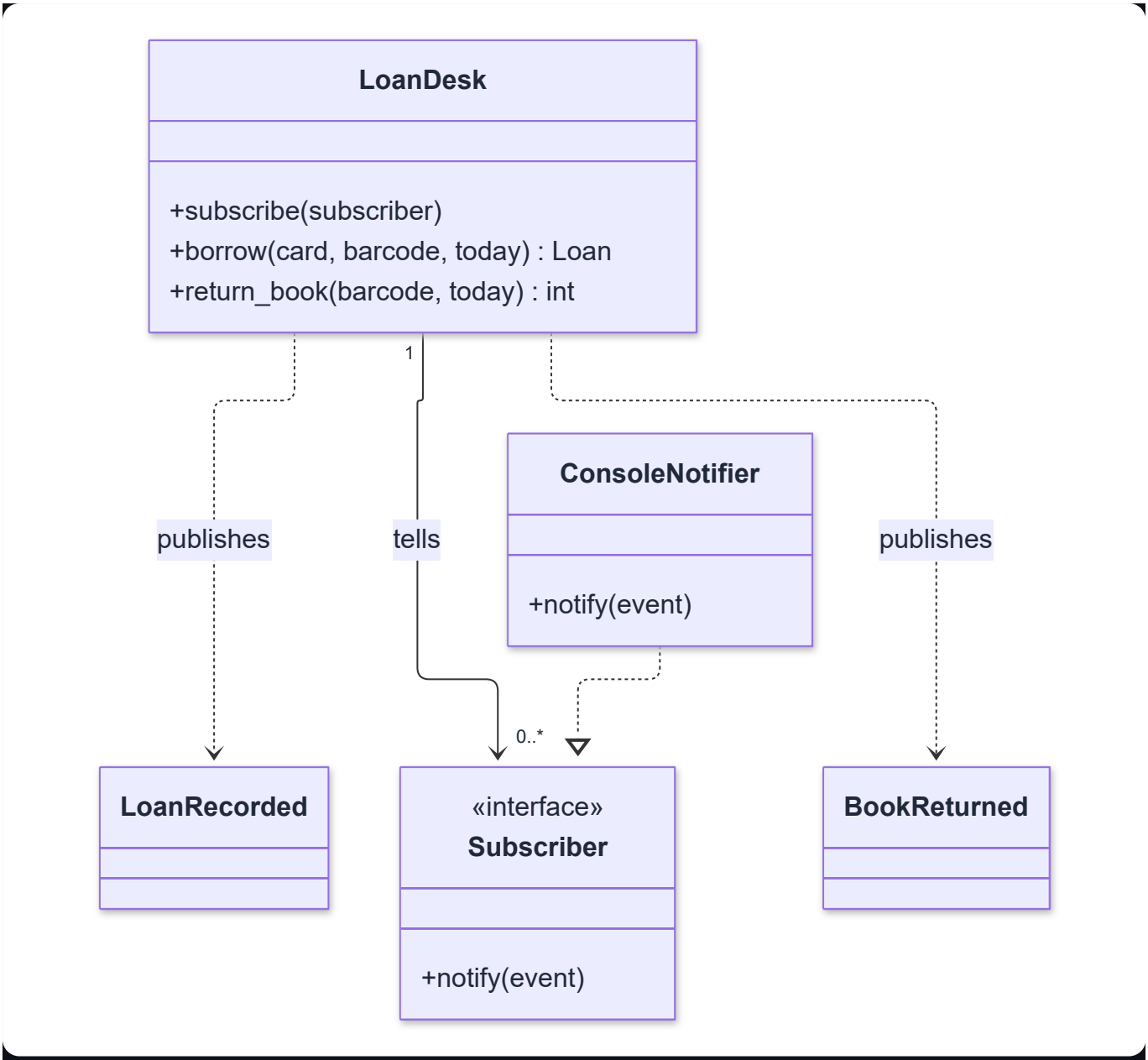
ITEM_MULTIPLIER = {"book": 1, "dvd": 2}


@runtime_checkable
class FinePolicy(Protocol):
    def fine(self, days_overdue: int, kind: MemberKind, item_type: str) -> int: ...


class PerDayFine:
    """Days overdue times the member kind's daily rate times the item type's multiplier

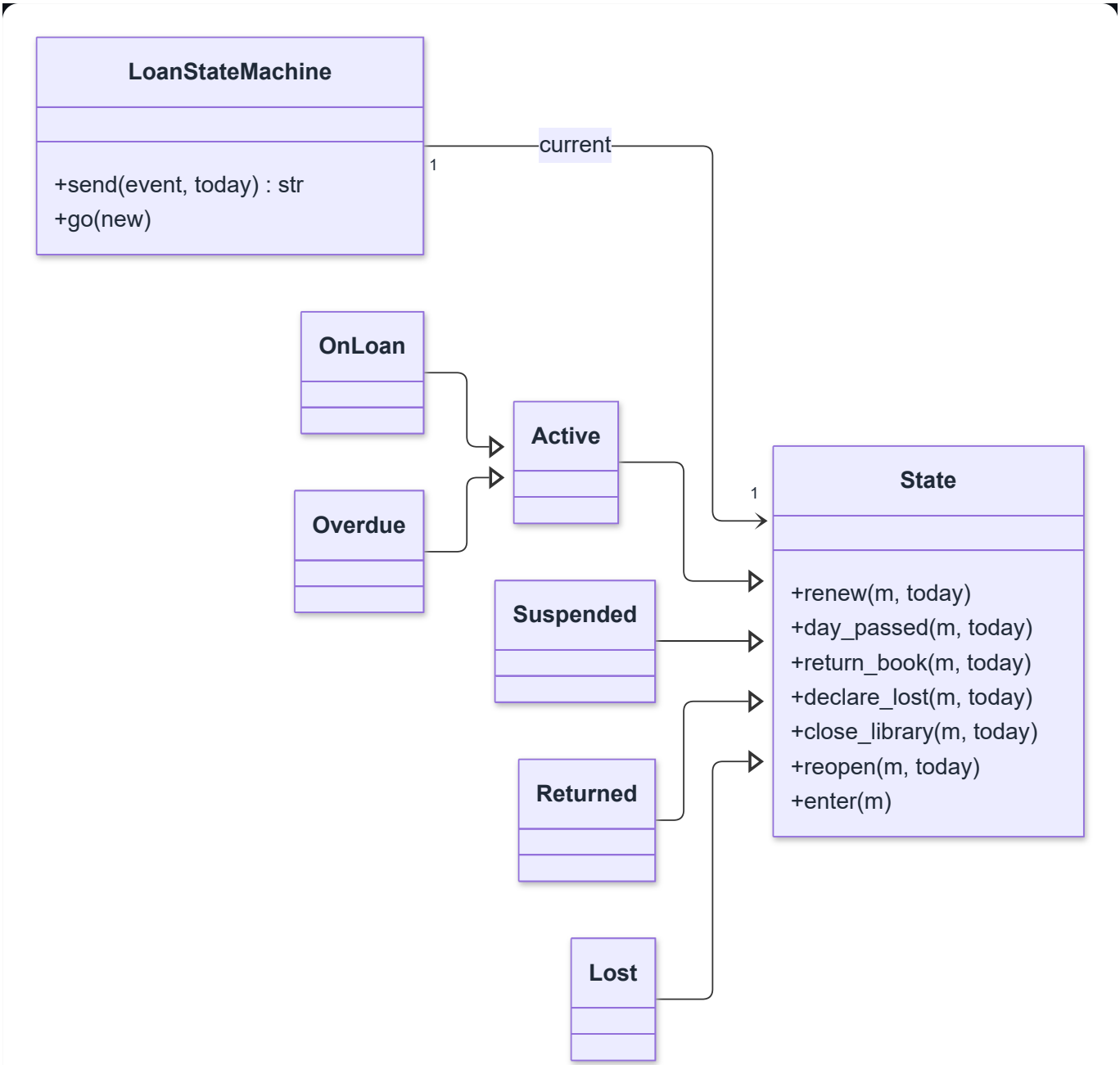
    def fine(self, days_overdue: int, kind: MemberKind, item_type: str) -> int:
        return max(0, days_overdue) * kind.fine_per_day() * ITEM_MULTIPLIER[item_type]
```

Observer



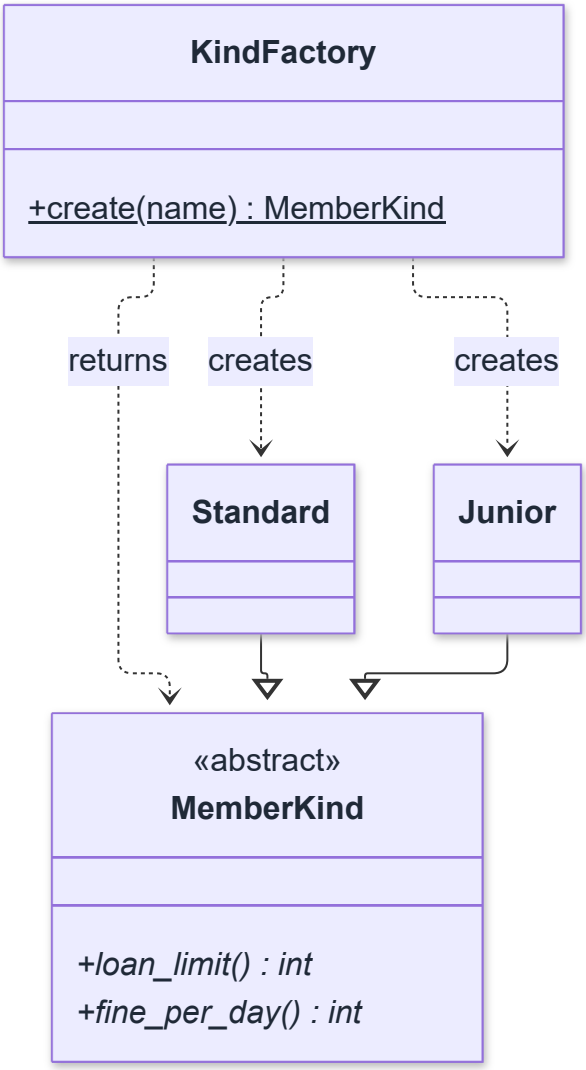
The shape: a subject with a list of subscribers behind an interface, and events it publishes. The desk knows `Subscriber`, never `ConsoleNotifier`.

State

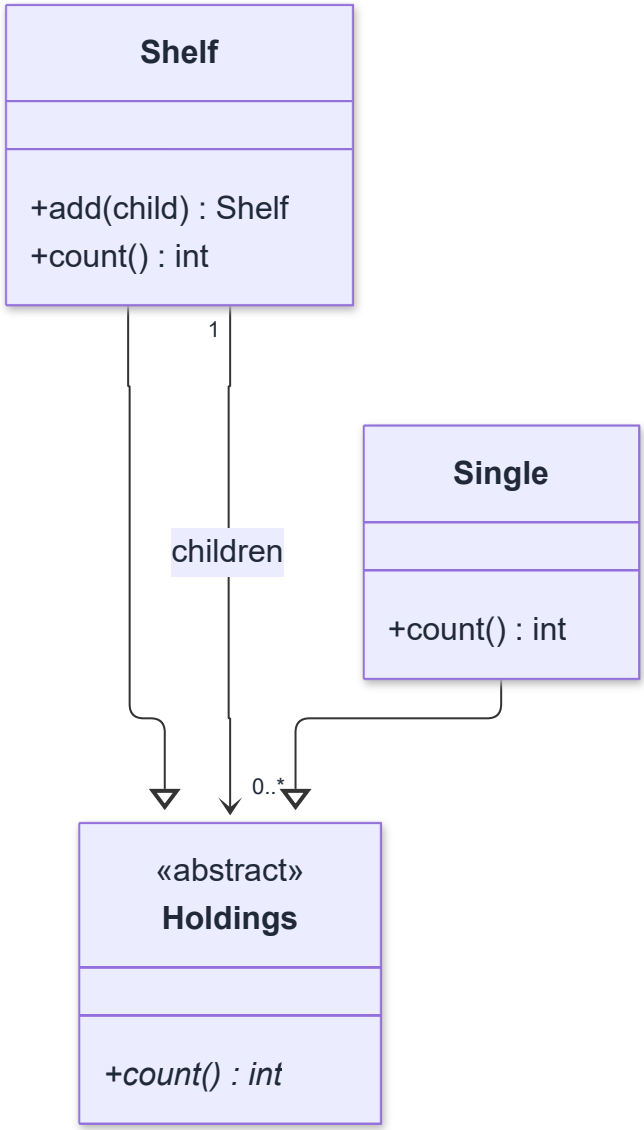


The shape: a context that points at a current state object, and one class per state. This is exactly the code of chapter 8 (`loan_state_pattern.py`); the diagram is checked against it. `Active` is the composite state from chapter 8's diagram, here as a shared parent class.

Factory and Composite



Factory: callers ask `KindFactory.create("junior")` and get a `MemberKind`. Only the factory names `Standard` and `Junior`.



Composite: a `Shelf` is a `Holdings` and holds `Holdings` (single copies or other shelves), so `count()` recurses.

```
"""Two small patterns the library does not otherwise need, written so every diagram in t
from abc import ABC, abstractmethod

from library.domain import Copy, Junior, MemberKind, Refusal, Standard


class KindFactory:
    """Factory: one place that turns a name into the right MemberKind, so callers do not

    registry = {"standard": Standard, "junior": Junior}

    @classmethod
    def create(cls, name: str) -> MemberKind:
        if name not in cls.registry:
            raise Refusal(f"unknown member kind {name!r}")
```

```
        return cls.registry[name]()


class Holdings(ABC):
    """Composite: a single copy and a whole shelf of things can both be asked how many c

    @abstractmethod
    def count(self) -> int: ...


class Single(Holdings):
    def __init__(self, copy: Copy):
        self.copy = copy

    def count(self) -> int:
        return 1


class Shelf(Holdings):
    def __init__(self, label: str):
        self.label = label
        self.children: list[Holdings] = []

    def add(self, child: Holdings) -> "Shelf":
        self.children.append(child)
        return self

    def count(self) -> int:
        return sum(child.count() for child in self.children)
```

Are the Diagrams True?

A pattern diagram is only worth having if it matches the code. The check below reads each diagram and verifies, against the source files, that every drawn operation and attribute exists, that every drawn inheritance is real, and that every drawn realisation is real (by actually testing, at run time, that the class fits the interface):

```
"""Check each pattern diagram against the code it describes: operations and attributes e

Names only: this cannot tell whether the pattern is applied well, just that the diagram
"""
import ast
```

```
import importlib
import pathlib
import re

HERE = pathlib.Path(__file__).parent
CHECKS = {
    "strategy.mmd": ["library/domain/fines.py", "library/domain/model.py", "library/appl
    "observer.mmd": ["library/application/desk.py", "library/application/ports.py", "lib
    "state.mmd": ["loan_state_pattern.py"],
    "factory.mmd": ["patterns_extra.py", "library/domain/model.py"],
    "composite.mmd": ["patterns_extra.py"],
    "before.mmd": ["library_before/domain/model.py", "library_before/application/desk.py
    "after.mmd": ["library/domain/fines.py", "library/domain/model.py", "library/applica
                  "library/infrastructure/notifier.py"],
}
MODULES = ["library.domain", "library.application", "library.infrastructure", "patterns_


def diagram(path: pathlib.Path):
    ops, attrs, inherits, realises, current = {}, {}, [], [], None
    for line in path.read_text(encoding="utf8").splitlines():
        c = re.match(r"\s*class (\w+)", line)
        if c:
            current = c.group(1)
            ops.setdefault(current, set())
            attrs.setdefault(current, set())
            if line.rstrip().endswith("}") is False and "{" not in line:
                current = None if False else current
            continue
        o = re.match(r"\s*\+(\w+)\(.*\)", line)
        if o and current:
            ops[current].add(o.group(1))
            continue
        a = re.match(r"\s*\+(\w+):", line)
        if a and current:
            attrs[current].add(a.group(1))
            continue
        i = re.match(r"\s*(\w+) --\|> (\w+)", line)
        if i:
            inherits.append((i.group(1), i.group(2)))
        r = re.match(r"\s*(\w+) \.\.\|> (\w+)", line)
        if r:
            realises.append((r.group(1), r.group(2)))
    return ops, attrs, inherits, realises
```

```
def code(paths):
    methods, fields, bases = {}, {}, {}
    for rel in paths:
        for node in ast.parse((HERE / rel).read_text(encoding="utf8")).body:
            if isinstance(node, ast.ClassDef):
                methods.setdefault(node.name, set()).update(f.name for f in node.body if
                fields.setdefault(node.name, set()).update(s.target.id for s in node.body
                bases[node.name] = {b.id for b in node.bases if isinstance(b, ast.Name)}
    return methods, fields, bases


def find_class(name: str):
    for module in MODULES:
        mod = importlib.import_module(module)
        if hasattr(mod, name):
            return getattr(mod, name)
    try:                                                # classes not exported by a p
        for dotted in ("library.domain.model", "library.domain.fines", "library.applicati
            mod = importlib.import_module(dotted)
            if hasattr(mod, name):
                return getattr(mod, name)
    except ImportError:
        pass
    return None


def main() -> int:
    problems = 0
    for name, files in CHECKS.items():
        ops, attrs, inherits, realises = diagram(HERE / "diagrams" / name)
        methods, fields, bases = code(files)
        issues = []
        for cls, wanted in ops.items():
            if cls in methods:
                issues += [f"{cls}.{op}() is drawn but not in the code" for op in sorted
        for cls, wanted in attrs.items():
            if wanted and cls in fields:
                issues += [f"{cls}.{a} is drawn but is not a field in the code" for a in
        for sub, base in inherits:
            if base not in bases.get(sub, set()):
                issues.append(f"{sub} --|> {base} is drawn, but {sub} does not inherit fr
        for impl, iface in realises:
            a, b = find_class(impl), find_class(iface)
            try:
```

```
                ok = a is not None and b is not None and issubclass(a, b)
            except TypeError:
                ok = False
            if not ok:
                issues.append(f"{impl} ..|> {iface} is drawn, but {impl} does not fit {i
        print(f"{name:<14} {'OK' if not issues else str(len(issues)) + ' problem(s)'}")
        for i in issues:
            print("   ", i)
        problems += len(issues)
    return 1 if problems else 0


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

```
strategy.mmd   OK
observer.mmd   OK
state.mmd      OK
factory.mmd    OK
composite.mmd  OK
before.mmd     OK
after.mmd      OK
```

All seven diagrams (the five patterns plus the library's before and after, below) pass. **A check that passes first time deserves suspicion**, so here it is on three deliberately tampered diagrams: an operation renamed in the strategy diagram, a wrong interface in the observer diagram, and a wrong parent in the state diagram:

```
strategy.mmd   1 problem(s)
    FinePolicy.calculate() is drawn but not in the code
observer.mmd   1 problem(s)
    ConsoleNotifier ..|> FinePolicy is drawn, but ConsoleNotifier does not fit FinePolicy
state.mmd      1 problem(s)
    OnLoan --|> State is drawn, but OnLoan does not inherit from State in the code
(exit status 1)
```

Each lie is caught and named. What the check cannot say is whether the *pattern is used well*: it checks names, as in chapter 10.

Principles You Can See on a Diagram

SOLID is covered properly in *Clean Code, SOLID & Refactoring* 6 and 7. What this course adds is that several of its ideas show up as *shapes*:

- **Dependency inversion: interfaces are owned by the client, and arrows point at them.** In the after diagram below, `LoanDesk` has three arrows, all to interfaces (`Catalogue`, `Subscriber`, `FinePolicy`), and every concrete class has a dotted arrow *into* an interface. A concrete class with an arrow pointing out of the desk would be a sign that high-level code depends on low-level detail. Chapter 9's dependency check makes the package-level version of this a test.
- **Open for extension, closed for change: a new box that touches no old box.** Adding a subscriber is a new class; nothing existing is edited (Exercise 2 proves it by fingerprinting the library's files before and after).
- **Cohesion: clusters.** Classes that change together should be close together and linked; classes that do not should not be. In chapter 9's package diagram, the clusters are the packages. Inside a class, the equivalent is "do its methods use the same data?", which the god class detector below measures.
- **Coupling: count the arrows leaving a box.** A box with arrows to everything has too many reasons to change. A box with none (`Copy`, in the domain) is easy to understand and test.

A Design Review Checklist

A review is a short, structured look at a design (diagrams plus a few key files) by someone other than its author, done *before* the cost of change is high. Questions, each with what to look for on a diagram and, where one exists, an automatic check from this course:

QUESTION	LOOK FOR ON THE DIAGRAM	AUTOMATIC CHECK
Does each class have one job?	A box with many unrelated operations; a name ending in Manager, Handler or Util	God class detector (below)
Do the dependencies point the right way?	A high-level box with arrows to concrete low-level boxes	Package rules (chapter 9)
Are there cycles?	Arrows that return to where they started	Package cycle check (chapter 9); mutual references (below)

QUESTION	LOOK FOR ON THE DIAGRAM	AUTOMATIC CHECK
Where does the behaviour live?	Boxes with fields and no operations, beside one box with all the operations and all the arrows	Anemic detector (below); reach-in count (chapter 6)
What varies, and is it isolated?	`if kind == ...` chains; one concrete class per variation with no interface	Search for type-tag comparisons (chapter 6)
What will the next likely change touch?	Trace the change across the boxes: how many must be edited?	Diff statistics after a trial change (this chapter)
What if something fails or runs twice?	Calls to outside things with no failure path (the e-mail server, a second desk)	Tests for failure paths (the broken subscriber below)
Do the diagrams still match the code?	Operation or class names that do not exist	Diagram check (above), drift check (chapter 10)

Finding a God Class, a Cycle and an Anemic Model

Three common problems, and their shape on a diagram:

- **God class:** one big box with arrows out to nearly everything, and a long list of operations that fall into several unrelated groups (lending, e-mail, reports, files, search).
- **Cyclic dependency:** a loop of arrows, typically between packages, so that neither side can be understood or tested alone (chapter 9 showed one stop the program importing).
- **Anemic model:** chapter 6's picture: several boxes with fields and no operations, and one service box with every operation and an arrow to each of them.

Detectors can point at candidates. These are mine, small, and honest about being crude (they measure: for a god class, how many *unrelated groups* a class's methods fall into, linking two methods when they use the same field or call each other, a measure called LCOM4 as far as I remember; for mutual references, which classes name each other; for anemia, how many classes hold only data and how often a service reaches into them from outside):

```
"""Three design smells, found by counting. My own simple detectors, written for this cha

  god class     a class whose methods split into unrelated groups (LCOM4 > 1: methods are
                other; each separate group is a separate job), together with a large numb
  mutual refs   two classes that each name the other (not always wrong: a two-way associa
```

```
  anemic       classes with several fields and no behaviour, beside a class that reads a

    py smells.py module.py [module.py ...]
"""
import ast
import pathlib
import sys

GOD_METHODS = 8          # arbitrary: this many methods in one class is "a lot" for this


def classes_of(path: str):
    tree = ast.parse(pathlib.Path(path).read_text(encoding="utf8"))
    return {n.name: n for n in tree.body if isinstance(n, ast.ClassDef)}


def fields_of(cls: ast.ClassDef) -> set[str]:
    out = {s.target.id for s in cls.body if isinstance(s, ast.AnnAssign) and isinstance(
    for fn in cls.body:
        if isinstance(fn, ast.FunctionDef) and fn.name == "__init__":
            out |= {n.attr for n in ast.walk(fn) if isinstance(n, ast.Attribute) and isi
                    and isinstance(n.ctx, ast.Store)}
    return out


def lcom4(cls: ast.ClassDef) -> tuple[int, list[list[str]]]:
    """Connected groups of methods, linking two methods when they touch the same field o
    own = fields_of(cls)
    methods = {f.name: f for f in cls.body if isinstance(f, ast.FunctionDef) and not f.na
    uses, calls = {}, {}
    for name, fn in methods.items():
        uses[name] = {n.attr for n in ast.walk(fn) if isinstance(n, ast.Attribute) and is
        calls[name] = {n.func.attr for n in ast.walk(fn) if isinstance(n, ast.Call) and
                       and isinstance(n.func.value, ast.Name) and n.func.value.id == "se
    parent = {m: m for m in methods}

    def find(x):
        while parent[x] != x:
            parent[x] = parent[parent[x]]
            x = parent[x]
        return x

    names = list(methods)
    for i, a in enumerate(names):
        for b in names[i + 1:]:
```

```
            if uses[a] & uses[b] or b in calls[a] or a in calls[b]:
                parent[find(a)] = find(b)
    groups: dict[str, list[str]] = {}
    for m in names:
        groups.setdefault(find(m), []).append(m)
    return len(groups), list(groups.values())


def named_classes(cls: ast.ClassDef, all_names: set[str]) -> set[str]:
    found = set()
    for n in ast.walk(cls):
        if isinstance(n, ast.Name) and n.id in all_names and n.id != cls.name:
            found.add(n.id)
        if isinstance(n, ast.Constant) and isinstance(n.value, str):          # forward r
            for name in all_names:
                if name != cls.name and name in n.value.replace("|", " ").split():
                    found.add(name)
    return found


def analyse(path: str) -> list[str]:
    classes = classes_of(path)
    report = []
    for name, node in classes.items():
        methods = [f for f in node.body if isinstance(f, ast.FunctionDef) and not f.name
        count, groups = lcom4(node)
        if count > 1 and len(methods) >= GOD_METHODS:
            report.append(f"GOD CLASS  {name}: {len(methods)} methods in {count} unrelat
    refs = {n: named_classes(c, set(classes)) for n, c in classes.items()}
    seen = set()
    for a in refs:
        for b in refs[a]:
            if a in refs.get(b, set()) and (b, a) not in seen:
                seen.add((a, b))
                report.append(f"MUTUAL     {a} and {b} each name the other")
    data_only = [n for n, c in classes.items() if len(fields_of(c)) >= 2 and not [f for
    if data_only:
        biggest = max(classes, key=lambda n: len([f for f in classes[n].body if isinstan
        field_names = {f for n in data_only for f in fields_of(classes[n])}
        poking = sum(1 for n in ast.walk(classes[biggest]) if isinstance(n, ast.Attribut
                     and n.attr in field_names)
        report.append(f"ANEMIC?    {len(data_only)} of {len(classes)} classes hold only
                      f"{biggest} reads or writes their fields {poking} times from outsi
    return report
```

```
if __name__ == "__main__":
    for path in sys.argv[1:]:
        print(path)
        for line in analyse(path) or ["(nothing flagged)"]:
            print("   " + line)
```

The flawed module they are run on (a manager that does everything, two classes that reference each other, two data-only classes):

```
"""A deliberately flawed library module, for the design review exercises: a god class, m
from dataclasses import dataclass, field
from datetime import date, timedelta


@dataclass
class Copy:
    barcode: str
    on_loan: bool = False


@dataclass
class Fine:
    card: str
    pence: int
    paid: bool = False


@dataclass
class Member:
    card: str
    email: str
    loans: list = field(default_factory=list)

    def record_loan(self, loan: "Loan") -> None:
        self.loans.append(loan)
        loan.member = self                                  # the Member reaches into th


@dataclass
class Loan:
    barcode: str
    due_on: date
```

```
    member: "Member | None" = None

    def overdue_fine(self, today: date) -> int:
        owner: Member = self.member                          # ... and the Loan reaches
        return max(0, (today - self.due_on).days) * (5 if owner.card.startswith("J") else

class LibraryManager:
    """Does everything: lending, fines, e-mail, reports, files, and searching."""

    def __init__(self, smtp_host: str, data_file: str):
        self.members: dict[str, Member] = {}
        self.copies: dict[str, Copy] = {}
        self.fines: list[Fine] = []
        self.smtp_host = smtp_host
        self.sent: list[str] = []
        self.data_file = data_file
        self.report_lines: list[str] = []
        self.catalogue_index: dict[str, list[str]] = {}

    # lending
    def borrow(self, card: str, barcode: str, today: date) -> Loan:
        member, copy = self.members[card], self.copies[barcode]
        if copy.on_loan:
            raise ValueError("already on loan")
        loan = Loan(barcode, today + timedelta(days=21))
        member.record_loan(loan)
        copy.on_loan = True
        return loan

    def return_book(self, card: str, barcode: str, today: date) -> int:
        member = self.members[card]
        loan = next(l for l in member.loans if l.barcode == barcode)
        fine = loan.overdue_fine(today)
        if fine:
            self.fines.append(Fine(card, fine))
            self.send_email(member.email, f"You owe {fine}p")
        self.copies[barcode].on_loan = False
        member.loans.remove(loan)
        return fine

    # e-mail
    def send_email(self, to: str, text: str) -> None:
        self.sent.append(f"{self.smtp_host}: to {to}: {text}")
```

```
    # reporting
    def build_report(self) -> list[str]:
        self.report_lines = [f"{f.card}: {f.pence}p" for f in self.fines if not f.paid]
        return self.report_lines

    def print_report(self) -> str:
        return "\n".join(self.report_lines)

    # persistence
    def save(self) -> str:
        return f"saved {len(self.members)} members to {self.data_file}"

    def load(self) -> str:
        return f"loaded from {self.data_file}"

    # searching
    def index_title(self, title: str, barcode: str) -> None:
        self.catalogue_index.setdefault(title.lower(), []).append(barcode)

    def search(self, word: str) -> list[str]:
        return self.catalogue_index.get(word.lower(), [])
```

The results on the course's own earlier code and the flawed module:

```
library_a.py
   ANEMIC?    3 of 5 classes hold only data (Member, Copy, Loan); LibraryService reads o
library_b.py
   (nothing flagged)
flawed_library.py
   GOD CLASS  LibraryManager: 9 methods in 2 unrelated groups: {borrow, return_book, sen
   MUTUAL     Member and Loan each name the other
   ANEMIC?    2 of 5 classes hold only data (Copy, Fine); LibraryManager reads or writes
```

- **Chapter 6's version A is flagged** as anemic (3 of its 5 classes hold only data and the service reaches into them 22 times), and **version B is not flagged**. The detector reproduces the judgement of chapter 6 from the code alone.
- **The flawed module** gets all three flags: the god class, the mutual reference between `Member` and `Loan`, and an anemic suspicion for `Copy` and `Fine`.

- **The god-class verdict is weaker than it looks.** A reader sees five jobs in `LibraryManager` (lending, e-mail, reporting, persistence, searching). The detector found only *two* groups, because the jobs share fields (the member list, the fines) and one method calls another. Exercise 1 shows this. A detector's "one group" does not clear a class, and its "two" is a floor, not a count.
- **The mutual reference is not necessarily wrong.** A member and its loans may legitimately know each other. Here the flaw is how: each reaches into the other's data (`loan.member = self`; `owner.card.startswith("J")` to choose a fine rate: a rule hiding in the wrong class, with a type test in disguise). The detector asks the question; the review answers it.

The same code scanned file by file in the current library finds nothing to flag:

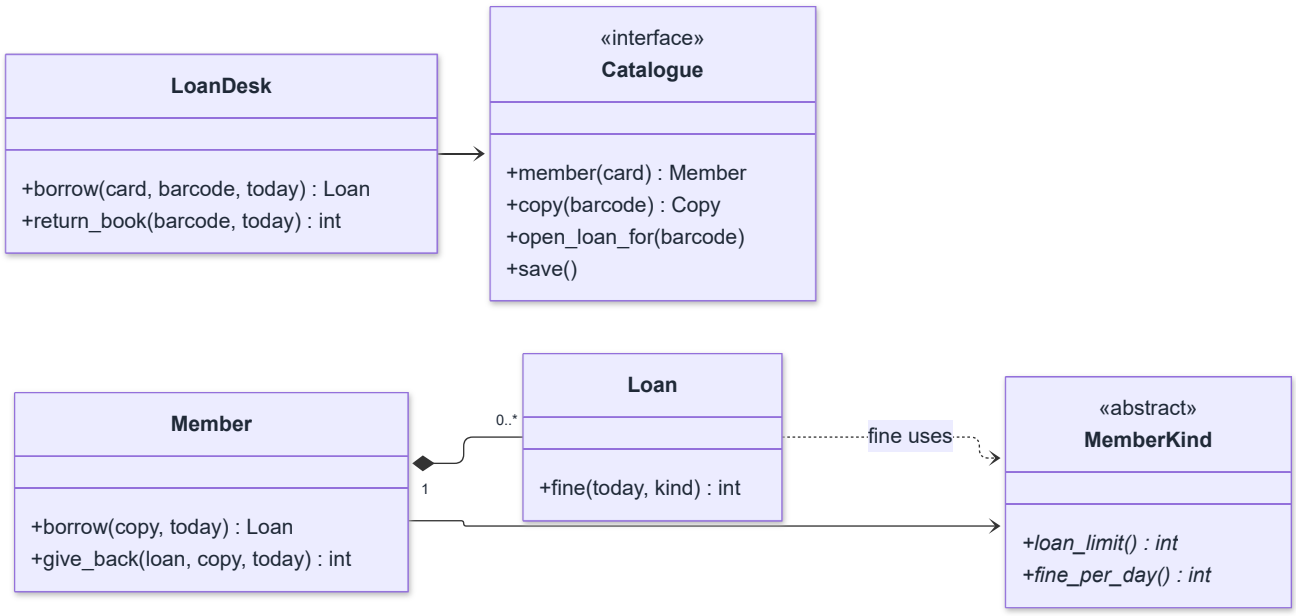
```
library/domain/model.py
   (nothing flagged)
library/domain/fines.py
   (nothing flagged)
library/application/desk.py
   (nothing flagged)
library/infrastructure/jsonfile.py
   (nothing flagged)
library/infrastructure/memory.py
   (nothing flagged)
```

And a package-level cycle, on a tiny tree where `billing` and `members` import each other, caught by chapter 9's checker:

```
analysed 3 files, found 2 cross-package imports
package dependencies found in the code (from -> to : first place):
   billing         -> members         library\billing\__init__.py:1  (library.members)
   members         -> billing         library\members\__init__.py:1  (library.billing)
R1 direction:
R2 no cycles:
   VIOLATION cycle: billing -> members -> billing
R3 public API only:
result: 1 violation(s)
```

Refactoring the Library: Diagram First, Then Code

Two requirements arrive. **One:** a DVD that comes back late should cost double a book. **Two:** the member should be sent a notice whenever a loan is recorded and when a late return adds a fine. This is the chapter-9 library:



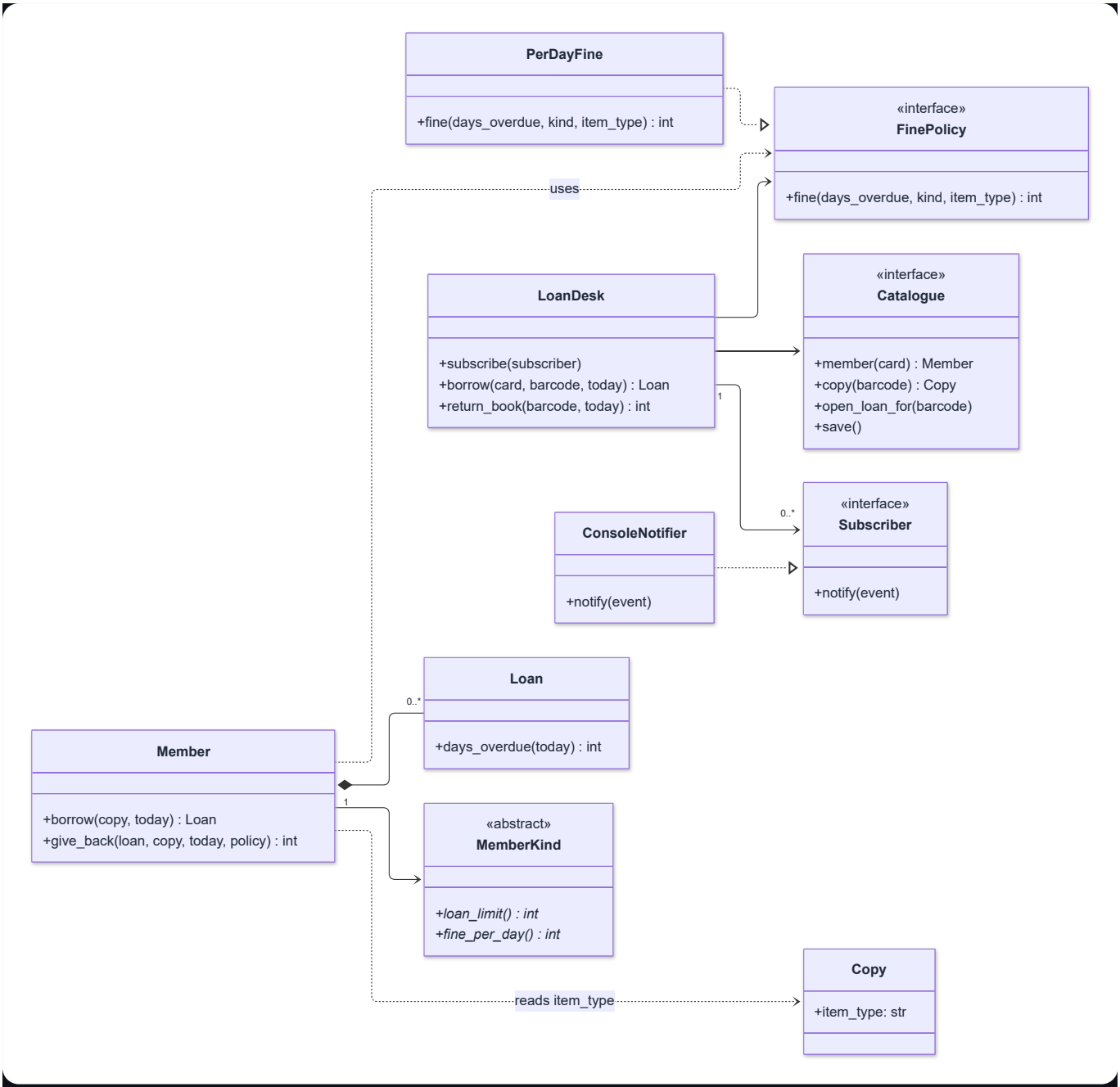
The library's core before the change (checked against the code: all drawn operations exist).

The Review

Applying the checklist's "what will the next change touch?" question to requirement one: the fine is worked out in `Loan.fine(today, kind)`, which knows only the member's kind. A copy's type (book or DVD) lives nowhere, and a loan holds its copy as a barcode, not as an object. To vary by item type I would have to edit `Loan`, `Member.give_back` and `Copy` and keep them consistent, and the next rule would edit all three again. That is a fine *rule* with no home of its own. For requirement two: the obvious place to send a notice is inside `LoanDesk.borrow`. That would make the desk (high-level code) depend on e-mail or printing (low-level detail), would make every desk test send messages, and would let a broken mail server stop a loan being recorded. It would also be the first step towards the god class above.

The Redrawn Design

Before touching code, the new shape, as a diagram: the fine rule becomes a **Strategy** (`FinePolicy`, with the item type on `Copy` passed to it), and the notices become an **Observer** arrangement (`Subscriber` and `ConsoleNotifier`). Both interfaces are owned by the code that uses them.



After. `LoanDesk` depends on three interfaces; each new class points into one of them. (This diagram is also checked against the code: see the seven-diagram check above.)

The Code

Now the change. The unchanged parts of the package are the safety net: chapter 9's 10 tests, run before and after without being edited. The changes:

```
"""Things that happen at the desk, announced to whoever has subscribed (an Observer arra
from dataclasses import dataclass
from datetime import date


@dataclass(frozen=True)
```

```
class LoanRecorded:
    card: str
    barcode: str
    due_on: date


@dataclass(frozen=True)
class BookReturned:
    card: str
    barcode: str
    fine_pence: int
```

```
"""The interfaces the application REQUIRES. Defined here, in the application, not beside
from typing import Protocol, runtime_checkable

from library.domain import Copy, Loan, Member


@runtime_checkable
class Catalogue(Protocol):
    def member(self, card: str) -> Member: ...

    def copy(self, barcode: str) -> Copy: ...

    def open_loan_for(self, barcode: str) -> tuple[Member, Loan]: ...

    def save(self) -> None: ...


@runtime_checkable
class Subscriber(Protocol):
    """Anything that wants to hear about desk events (an email sender, a log, a counter)

    def notify(self, event: object) -> None: ...
```

```
"""The use cases of the lending desk. Depends on the domain and on the interfaces it requ
from datetime import date

from library.application.events import BookReturned, LoanRecorded
from library.application.ports import Catalogue, Subscriber
from library.domain import FinePolicy, Loan
```

```
class LoanDesk:
    def __init__(self, catalogue: Catalogue, subscribers: tuple[Subscriber, ...] = (), f
        self.catalogue = catalogue
        self.subscribers = list(subscribers)
        self.fine_policy = fine_policy
        self.errors: list[str] = []              # a subscriber that failed: recorded, n

    def subscribe(self, subscriber: Subscriber) -> None:
        self.subscribers.append(subscriber)

    def borrow(self, card: str, barcode: str, today: date) -> Loan:
        loan = self.catalogue.member(card).borrow(self.catalogue.copy(barcode), today)
        self.catalogue.save()
        self._publish(LoanRecorded(card, barcode, loan.due_on))
        return loan

    def return_book(self, barcode: str, today: date) -> int:
        member, loan = self.catalogue.open_loan_for(barcode)
        fine = member.give_back(loan, self.catalogue.copy(barcode), today, self.fine_poli
        self.catalogue.save()
        self._publish(BookReturned(member.card, barcode, fine))
        return fine

    def _publish(self, event: object) -> None:
        for subscriber in self.subscribers:
            try:
                subscriber.notify(event)
            except Exception as e:      # noqa: BLE001 - one broken subscriber must not
                self.errors.append(f"{type(subscriber).__name__} failed on {type(event).
```

```
"""A subscriber that turns desk events into messages (printed here; an email sender woul
from library.application import BookReturned, LoanRecorded


class ConsoleNotifier:
    def __init__(self, write=print):
        self.write = write

    def notify(self, event: object) -> None:
        if isinstance(event, LoanRecorded):
            self.write(f"notice to {event.card}: {event.barcode} is due back on {event.d
```

```
        elif isinstance(event, BookReturned) and event.fine_pence:
            self.write(f"notice to {event.card}: a fine of {event.fine_pence}p was added
```

Three decisions in the desk are worth reading twice. **A subscriber's failure never undoes a loan**: the loan is saved first, then published, and each notification is isolated, so a broken subscriber is recorded and the others still hear. **The desk publishes after saving**, so subscribers never hear about a loan that was not recorded. And **the old data files still load**: files without an item type load as books (`item_type` defaults to `"book"`).

What the Change Touched

file	status	lines added	lines removed
application/__init__.py	changed	3	2
application/desk.py	changed	21	5
application/events.py	NEW	17	0
application/ports.py	changed	8	1
domain/__init__.py	changed	2	1
domain/fines.py	NEW	21	0
domain/model.py	changed	7	4
infrastructure/__init__.py	changed	3	2
infrastructure/jsonfile.py	changed	3	2
infrastructure/notifier.py	NEW	13	0
ui/cli.py	changed	2	2
TOTAL		100	19

Three new files and eight edited, 100 lines added and 19 removed. The domain's `model.py`, where the fine rule used to live, changed by only seven lines added and four removed. Not one line of the member, loan or catalogue behaviour changed except the fine's calculation moving out. Whether that is "a small change" is a judgement; the point is that it is *countable*, and that the same question can be asked of the next change.

Does It Still Work, and Is the Architecture Intact?

```
"""Tests for the refactored library: the strategy, the subscribers, and the promise that
import json
import os
import tempfile
import unittest
from datetime import date
```

```
from library.application import BookReturned, LoanDesk, LoanRecorded, Subscriber
from library.domain import Copy, FinePolicy, Member, PerDayFine
from library.infrastructure import ConsoleNotifier, InMemoryCatalogue, JsonCatalogue

TODAY = date(2026, 10, 10)
LATE = date(2026, 11, 3)                  # a loan taken on TODAY falls due on 2026-10-3


def catalogue():
    return InMemoryCatalogue([Member("M1", date(2027, 1, 1), "standard")], [Copy("B1"), (


class Recorder:
    def __init__(self):
        self.events = []

    def notify(self, event):
        self.events.append(event)


class Broken:
    def notify(self, event):
        raise RuntimeError("the mail server is down")


class Strategy(unittest.TestCase):
    def test_a_dvd_costs_double_for_the_same_lateness(self):
        desk = LoanDesk(catalogue())
        desk.borrow("M1", "B1", TODAY)
        desk.borrow("M1", "D1", TODAY)
        self.assertEqual(desk.return_book("B1", LATE), 30)
        # M1 now has a fine, but returning is still allowed: only borrowing is blocked
        self.assertEqual(desk.return_book("D1", LATE), 60)

    def test_the_default_policy_satisfies_the_interface(self):
        self.assertIsInstance(PerDayFine(), FinePolicy)

    def test_a_different_policy_can_be_plugged_in_without_changing_anything_else(self):
        class Amnesty:
            def fine(self, days_overdue, kind, item_type):
                return 0
        desk = LoanDesk(catalogue(), fine_policy=Amnesty())
        desk.borrow("M1", "B1", TODAY)
        self.assertEqual(desk.return_book("B1", LATE), 0)
```

```
class Observer(unittest.TestCase):
    def test_subscribers_hear_events_in_order(self):
        rec = Recorder()
        desk = LoanDesk(catalogue(), subscribers=(rec,))
        desk.borrow("M1", "B1", TODAY)
        desk.return_book("B1", LATE)
        self.assertEqual(rec.events, [LoanRecorded("M1", "B1", date(2026, 10, 31)), Book

    def test_a_broken_subscriber_neither_undoes_the_loan_nor_silences_the_others(self):
        rec = Recorder()
        desk = LoanDesk(catalogue(), subscribers=(Broken(), rec))
        loan = desk.borrow("M1", "B1", TODAY)
        self.assertEqual(loan.barcode, "B1")                                    # t
        self.assertEqual(len(rec.events), 1)                                    # t
        self.assertEqual(desk.errors, ["Broken failed on LoanRecorded: RuntimeError"])

    def test_the_desk_does_not_know_who_is_listening(self):
        self.assertIsInstance(Recorder(), Subscriber)
        self.assertIsInstance(ConsoleNotifier(), Subscriber)

    def test_console_notifier_messages(self):
        lines = []
        desk = LoanDesk(catalogue(), subscribers=(ConsoleNotifier(lines.append),))
        desk.borrow("M1", "B1", TODAY)
        desk.return_book("B1", date(2026, 10, 31))                               # 
        self.assertEqual(lines, ["notice to M1: B1 is due back on 2026-10-31"])


class Compatibility(unittest.TestCase):
    def test_a_data_file_from_before_item_types_still_loads_as_books(self):
        old = {"members": [{"card": "M1", "expires_on": "2027-01-01", "kind": "standard"
               "copies": [{"barcode": "B1", "on_loan": False}]}
        with tempfile.TemporaryDirectory() as d:
            path = os.path.join(d, "old.json")
            with open(path, "w", encoding="utf8") as f:
                json.dump(old, f)
            self.assertEqual(JsonCatalogue(path).copy("B1").item_type, "book")

    def test_item_type_survives_a_save_and_load(self):
        with tempfile.TemporaryDirectory() as d:
            path = os.path.join(d, "new.json")
            c = JsonCatalogue(path)
            c.add(copy=Copy("D1", item_type="dvd"))
```

```
            c.save()
            self.assertEqual(JsonCatalogue(path).copy("D1").item_type, "dvd")


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

```
Ran 23 tests in 0.223s

OK
```

All 23 tests pass: chapter 9's 10 (unmodified), the 9 new behaviour tests, the 2 architecture tests, and 2 for the extra patterns. The dependency rules from chapter 9 are unchanged and still clean (the new `infrastructure -> application` edge is the notifier importing the events, which the rules allow). And running the command-line program shows the observer at work:

```
$ py -m library.ui.cli --data library.json init
initialised: library.json
$ ... borrow M1 B1 --today 2026-10-10
notice to M1: B1 is due back on 2026-10-31
loan recorded: B1 due 2026-10-31
$ ... return B1 --today 2026-11-03
notice to M1: a fine of 30p was added for B1
returned; fine 30p
```

Usual Mistakes

MISTAKE	FIX
Reaching for a pattern before there is a problem	Wait for the second variation or the second listener; the diagram of the simple version is better than a pattern with one member
Naming a class after its pattern ("StrategyManager")	Name it for its job (`FinePolicy`); the diagram shows the shape
Treating a smell detector's silence as a clean bill of health	It gave two groups for five jobs; read the class

MISTAKE	FIX
A pattern diagram that is wishful	Check it against the code, and tamper with the check once to see it fail
Refactoring with no safety net	Run the old tests before and after, unedited; add new ones for the new behaviour
Observers that can break the subject	Isolate each notification and record failures
Reviewing the code but not the change	Count what a plausible next change would touch

MISTAKES WHILE PREPARING THIS CHAPTER

- **My first Strategy created an import cycle.** The default policy lived in `fines.py`, which needed `MemberKind` from `model.py`, while `model.py` needed `PerDayFine`. I first hid the cycle with an import inside a method and a comment explaining why: a warning sign I had written down myself. The proper fix is for `fines.py` to import `MemberKind` only for type hints (under `TYPE_CHECKING`), so nothing is imported at run time and there is no cycle. Takeaway: a comment that explains why an import is in the wrong place is a design problem, not a solution.
- **A scripted edit put a comment inside an expression.** Patching `jsonfile.py` with a text-replacement script inserted my trailing comment before a closing parenthesis, and the file stopped parsing (`SyntaxError: '(' was never closed`). Running the tests straight away found it in seconds. This is the same kind of slip as in chapters 7 and 10: use the editor's replace for code edits, and run the code immediately.
- **The god class detector under-counted.** I wrote the flawed module with five jobs and expected the detector to find five. It found two. I kept the result in the chapter and added Exercise 1 on it. Takeaway: validate a detector on a case where you already know the answer, and report the gap.
- **An over-strict guard in my own tool, found by a new exercise.** Chapter 9's dependency checker reported "nothing was analysed" when a fixed tree had no imports *between* packages at all, which is a legitimate state. Exercise 3 hit it. I changed the guard to fail only when no files are found, corrected chapter 9's script and its text, and re-ran its tests. Takeaway: a guard against "checking nothing" must not mistake "nothing to report" for "nothing checked".
- **A check that passed first time.** The diagram check said every diagram was fine on its first run. That is the moment to distrust it, so I wrote the tampered-diagram demonstration. It is in the chapter because the first pass alone proves nothing.

Hands-On Exercises

EXERCISE 1

Run the smell detector on the flawed module, then list the jobs you see in `LibraryManager` yourself and compare with what the detector reports. Explain the difference, and say how you would split the class (name the new classes and the interfaces between them).

View solution

EXERCISE 2

Add a third subscriber to the refactored desk: a counter of how many loans fall due on each day. Write it without editing the library package, run it alongside the console notifier, and prove the library's files did not change.

View solution

EXERCISE 3

Build a two-package tree where `billing` and `members` import each other and show the dependency checker reporting the cycle. Remove the cycle by having `members` declare the interface it needs and `billing` provide it, show the checker clean, and show the fixed code running.

View solution

CHAPTER 11 QUICK REFERENCE

- Patterns are shapes on a diagram: Strategy (client + interface + implementations), Observer (subject + subscriber interface), State (context + one class per state), Factory (one creator), Composite (a node that contains nodes); this chapter shows the diagrams, *Design Patterns* teaches the patterns
- Check pattern diagrams against the code (operations, inheritance, realisation), and tamper with the check once to see it fail
- Principles as shapes: interfaces owned by the client with arrows pointing at them; new behaviour as a new box that touches no old box; clusters for cohesion; few arrows out of a box
- Review questions: one job per class? dependencies pointing the right way? cycles? where does behaviour live? what varies? what will the next change touch? what if it fails? do the diagrams match the code?

- Smell detectors point, people decide: god class (methods in unrelated groups, a floor on the count), mutual references (legitimate or not), anemic (data-only classes beside a service that reaches in)
- Refactor on the diagram first, change the code with the old tests as the safety net, then count what the change touched and check the architecture rules
- An observer must not be able to break the subject: save first, publish after, isolate and record failures

COMING NEXT

Object-Oriented Design with UML 12 (Capstone): design a new system, a workshop-booking service, from requirements to use cases, domain model, class design, key sequence and state diagrams and a component view, all as diagram-as-code files, implement and run the core in Python, and produce a short design document. Course complete, with pointers to *Design Patterns*, *Software Architecture Fundamentals* and *Clean Code, SOLID & Refactoring*.

Object-Oriented Design with UML

Course 1 · Chapter 12 · Capstone: Designing a System from Requirements

Eleven chapters taught the pieces on one running example, the library. This chapter uses them on a **different system**, so that what you practise is the method and not the example: a booking service for a community centre's workshops. The order is the order of the course. Requirements, then use cases, a domain model, a class design, the key behaviours (a sequence diagram and a state machine), a component view, then code that runs, tests and checks that tie the diagrams to the code, and a short design document that says what was decided and what was not.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Everything here was run on this machine (Python 3.14, PlantUML 1.2026.8, Mermaid 12.1.0): the service (11 files in three layers), **23 passing tests**, three deliberately broken versions of it that the tests catch, a story that exercises every use case, a check that the class and sequence diagrams match the code (and fail on tampered copies), and the dependency rules from chapter 9. **Not built:** a user interface, a real card service, real e-mail, persistence, and anything beyond one process. The design document lists them as gaps. **The system is invented**: the centre, its refund rule and its questions are mine, labelled as assumptions where I filled a gap.

Step 1: Requirements, as Words, With the Questions They Raise

The input is what the centre's manager said, written down as she said it. Underneath, the questions the words do not answer. Each unanswered question is either asked or *assumed* in writing, so the assumption can be challenged later.

```
WORKSHOP BOOKING SERVICE: REQUIREMENTS (as the community centre's manager described them

The centre runs workshops (pottery, coding for beginners, bookbinding). Each workshop is

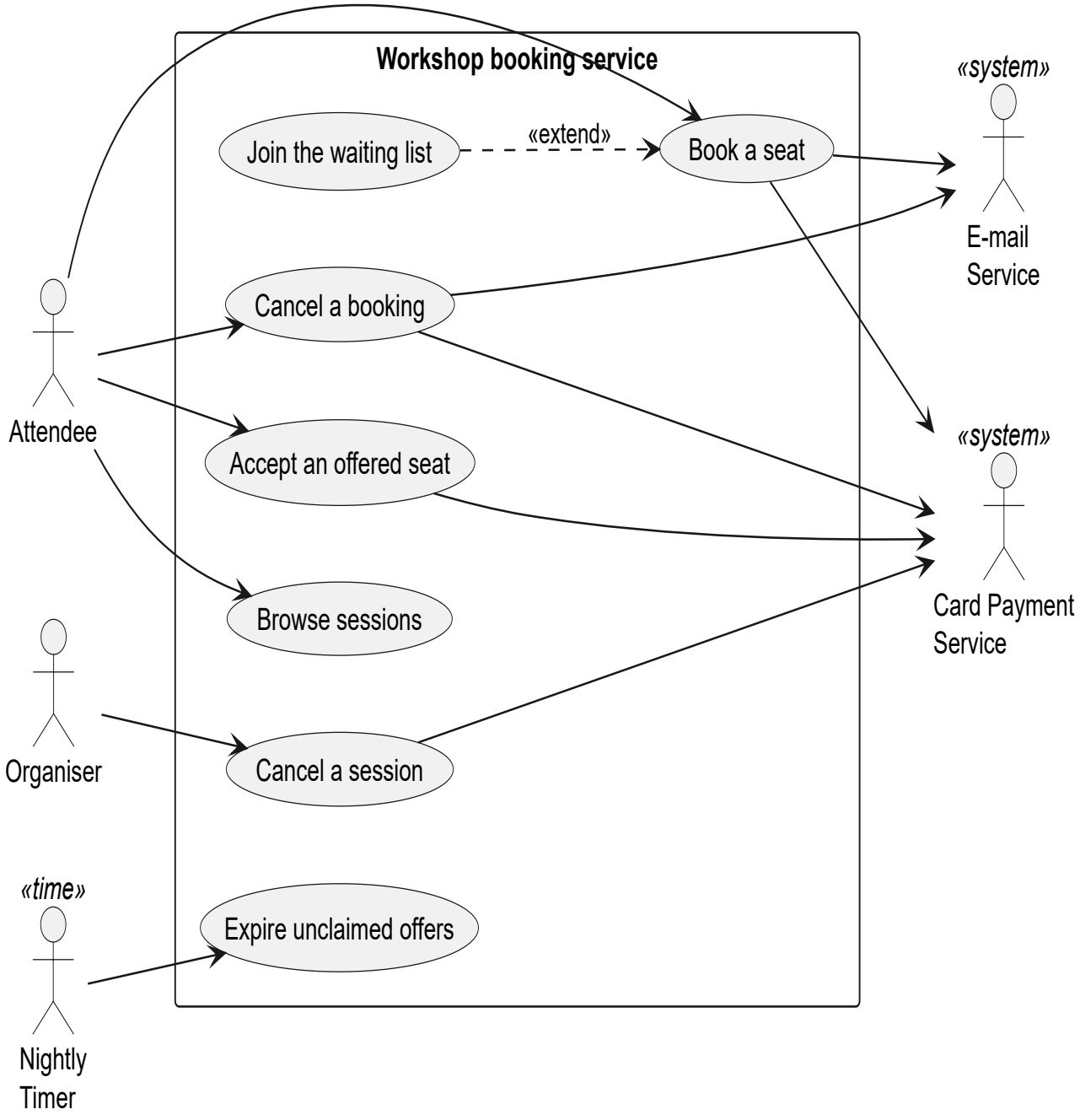
R1  People can see the upcoming sessions and how many seats are left.
R2  A person books a seat on a session and pays the session's fee by card at the time of
R3  If a session is full, the person can join its waiting list instead. They are not cha
R4  A person can cancel their booking. If the session is at least a week away, they get
R5  When a seat becomes free and people are waiting, the person who has waited longest i
R6  The organiser can cancel a whole session. Everybody who paid gets everything back, h
R7  People should be told by e-mail when their booking is confirmed, when they are put o
```

```
Not stated, so asked (open questions for the manager; assumptions made so that work coul
Q1  Can one person book several seats for friends?                                  ASSU
Q2  If the card is declined, is the person put on the waiting list?                  ASS
Q3  What if the organiser changes a session's date?                                  NOT
Q4  Is the one-week refund rule measured from the booking date or the session date? ASSU
Q5  Do people on the waiting list pay when they join?                                ASS
Q6  Does the waiting-list offer go to the next person at once if the first does not take
```

Note how much of the work is already in the questions. "A week away" (Q4), what happens to someone whose card is declined (Q2) and what a waiting person pays and when (Q5) all change the design, and none was in the manager's words.

Step 2: Use Cases

Actors first (chapter 3): the attendee, the organiser, and three things that are not people: the card service, the e-mail service, and a nightly timer that expires offers. The scope boundary is the service itself.



PlantUML. "Join the waiting list" extends "Book a seat" (the arrow points at the base, as chapter 3 said it must). This diagram is already at the edge of readable: the lines to the two outside services cross. A list of use cases would carry the same facts.

```
@startuml
scale 1.3
left to right direction

actor Attendee
```

```
actor Organiser
actor "Card Payment\nService" as Pay <<system>>
actor "Nightly\nTimer" as Timer <<time>>
actor "E-mail\nService" as Mail <<system>>

rectangle "Workshop booking service" {
  usecase "Browse sessions" as Browse
  usecase "Book a seat" as Book
  usecase "Join the waiting list" as Wait
  usecase "Cancel a booking" as Cancel
  usecase "Accept an offered seat" as Accept
  usecase "Cancel a session" as CancelSession
  usecase "Expire unclaimed offers" as Expire
}

Attendee --> Browse
Attendee --> Book
Attendee --> Cancel
Attendee --> Accept
Organiser --> CancelSession
Timer --> Expire
Book --> Pay
Accept --> Pay
Cancel --> Pay
CancelSession --> Pay
Wait ..> Book : <<extend>>
Book --> Mail
Cancel --> Mail
@enduml
```

The descriptions carry the substance. Two are written in full. Look at the guarantees and at the extensions: every "what if" from the requirements ended up as a numbered case.

```
USE CASE: Book a Seat
Scope:            Workshop booking service (black box)
Goal level:       User goal
Primary actor:    Attendee
Supporting actors: Card Payment Service, E-mail Service
Stakeholders:     Attendee (wants a seat, charged once, told clearly)
                  Organiser (wants full sessions and no double-charged or double-booked
Precondition:     The session exists and has not started or been cancelled.
Trigger:          The Attendee asks to book a seat on a session.
Success guarantee: The Attendee holds a confirmed seat and has paid the fee once; an e-m
```

```
Minimal guarantee: Nobody is charged for a booking that did not succeed, and no seat is l

MAIN SUCCESS SCENARIO
 1. The Attendee chooses a session and asks to book.
 2. The system checks the session is open and the Attendee has no live booking for it.
 3. The system checks a seat is free and holds it for the Attendee.
 4. The system charges the fee to the Attendee's card through the Card Payment Service.
 5. The system confirms the booking.
 6. The system tells the Attendee, by e-mail, that the booking is confirmed.

EXTENSIONS
 2a. The session has been cancelled, has started, or the Attendee already has a live boo
     2a1. The system refuses, saying why. Use case ends.
 3a. No seat is free:
     3a1. The system puts the Attendee on the waiting list (no charge) and e-mails them.
 4a. The payment is declined:
     4a1. The system gives up the held seat and records the booking as declined. Nothing
 6a. The e-mail cannot be sent:
     6a1. The booking stands; the failure is recorded. Use case ends.

OPEN QUESTIONS: see requirements Q1, Q2, Q5.
```

```
USE CASE: Cancel a Booking
Scope:            Workshop booking service (black box)
Goal level:       User goal
Primary actor:    Attendee
Supporting actors: Card Payment Service, E-mail Service
Stakeholders:     Attendee (wants as much money back as the rules allow, and no surprise
                  Organiser (wants a freed seat filled)
Precondition:     The Attendee has a booking that is Confirmed, Waitlisted or Offered.
Trigger:          The Attendee asks to cancel the booking.
Success guarantee: The booking is cancelled, any refund due is paid, and a freed seat is
Minimal guarantee: If the refund cannot be paid, the Attendee is told and the failure is

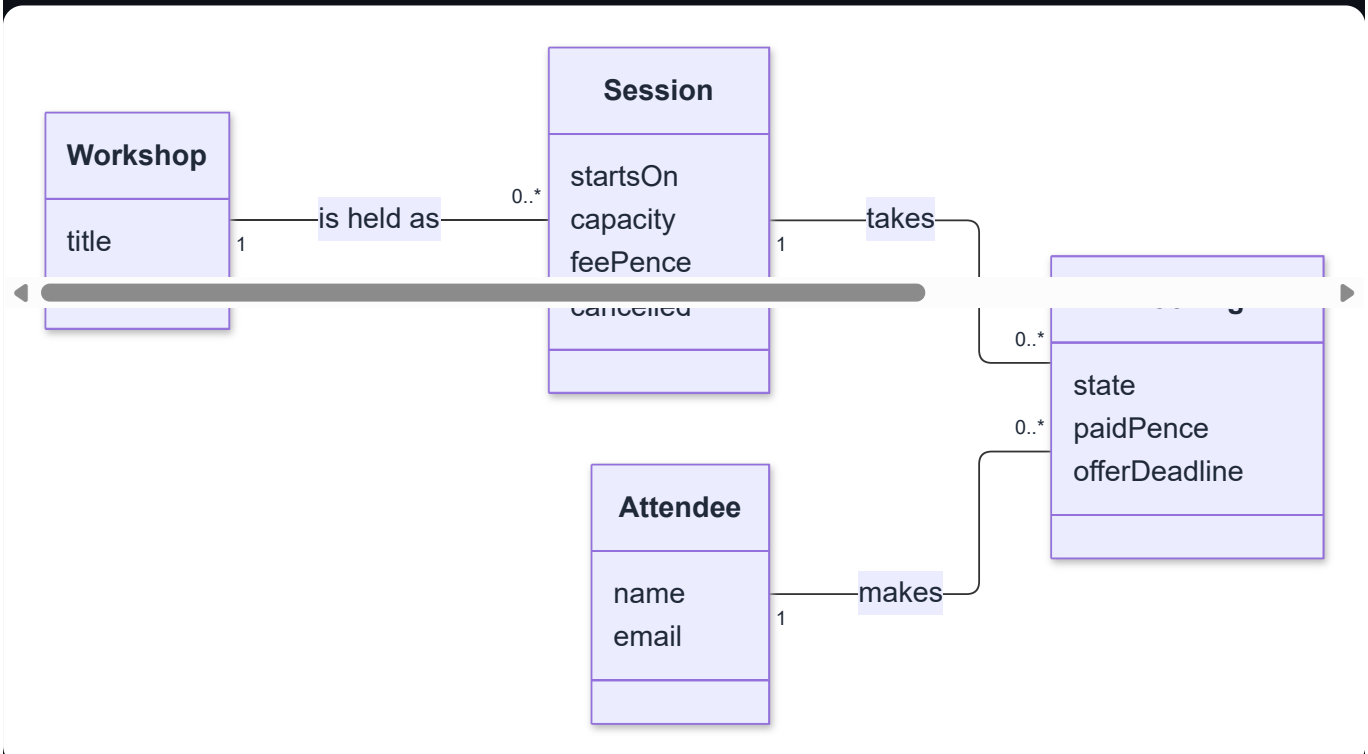
MAIN SUCCESS SCENARIO
 1. The Attendee asks to cancel a booking.
 2. The system finds the booking and its session.
 3. The system works out the refund: the full fee if the session is at least 7 days away
 4. The system cancels the booking.
 5. The system asks the Card Payment Service to refund the amount.
 6. The system offers the freed seat to the person who has waited longest, if anyone is w
 7. The system tells the Attendee, by e-mail, about the cancellation and the refund.
```

```
  EXTENSIONS
   2a. The booking is already cancelled, expired or declined:
       2a1. The system refuses: there is nothing to cancel. Use case ends.
   3a. The booking was only on the waiting list, or an unpaid offer:
       3a1. The refund is zero; continue at step 4.
   5a. The refund amount is zero:
       5a1. The system does not contact the payment service. Continue at step 6.
   5b. The payment service fails to refund:
       5b1. NOT YET DESIGNED: see the design document's open decisions.
```

Writing extension 5b of the second use case ("the payment service fails to refund") produced the first real finding. I did not know what the service should do, so the description says **NOT YET DESIGNED** instead of pretending. It comes back below: the code, written without an answer, does something harmful there.

Step 3: The Domain Model and a Glossary

The concepts and their links, as in chapter 4: written once as data, producing the diagram and the sentences to read aloud to the manager. The glossary fixes the words, including one it forbids.



Mermaid, generated from `domain_model.py`. Four concepts, three associations. The `Booking` has a `state`, which Step 5 expands.

```
Each Workshop is held as zero or more Sessions.
   ...and each Session is a session of exactly one Workshop.
Each Session takes zero or more Bookings.
   ...and each Booking is for exactly one Session.
Each Attendee makes zero or more Bookings.
   ...and each Booking is made by exactly one Attendee.
```

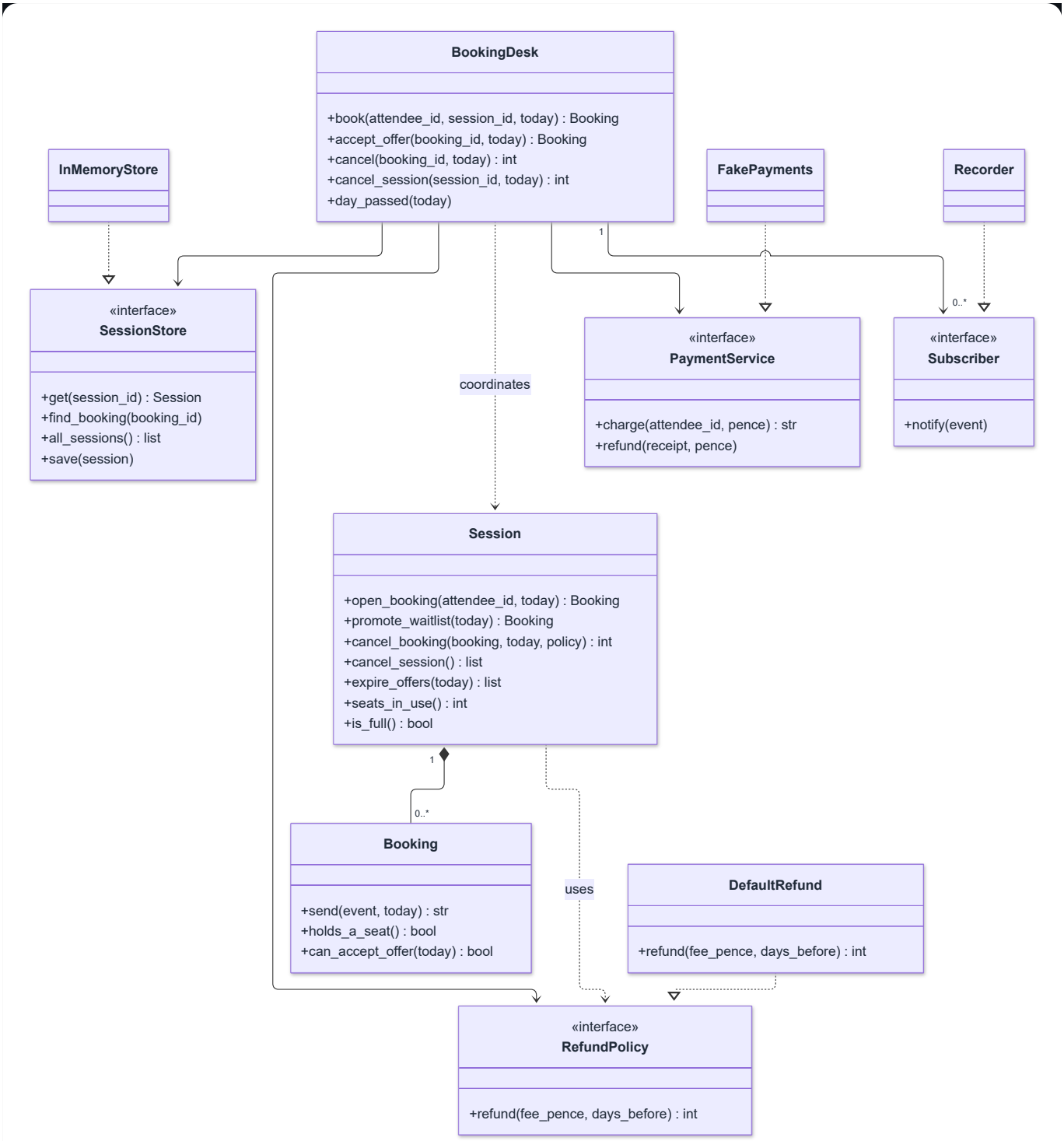
```
GLOSSARY: Workshop booking service                                  (draft 2026-10-10: t

Workshop     A subject the centre teaches ("Pottery for beginners"). It is held as one o
Session      One occasion on which a Workshop is held: a date, a limited number of seats
Attendee     A person who books, joins a waiting list for, or is offered a seat. Identif
Booking      An Attendee's claim on a Session, from the moment they ask until it is conf
             It has a state (see the state diagram). "Reservation" is NOT used: it means
Seat         One unit of a Session's capacity. A Booking holds a seat while it is Reques
Waiting list The Bookings of a full Session that are Waitlisted, in the order they were
Offer        A freed Seat held for the Waitlisted Booking that has waited longest, for o
Refund       Money returned to an Attendee: the full fee when a Booking is cancelled at
             when the Organiser cancels the Session; otherwise none.
Organiser    A role played by centre staff. An actor in the use cases; not part of the d
```

The glossary's note on "Reservation" is chapter 4's lesson applied: in the library a "reservation" was a thing in its own right; here the word would mean "a booking that has not been paid for", which this system never has, so it is banned rather than allowed to mean two things.

Step 4: Class Design and Responsibilities

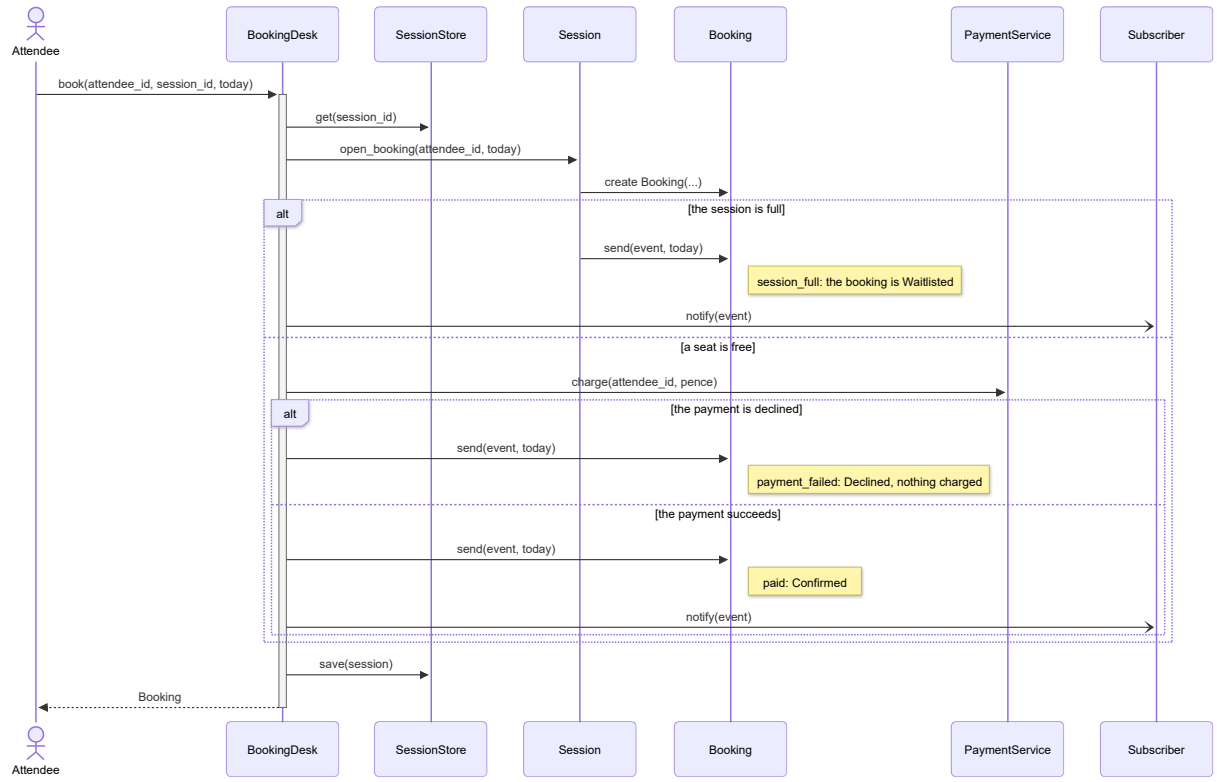
Using chapter 6's questions (who has the information? who creates? who coordinates?): the `Session` holds the capacity and the bookings, so it decides who gets a seat and creates the bookings; the `Booking` knows its own state; a `BookingDesk` coordinates a use case across the session, the payment service and the notifications and holds no rule. Everything outside is behind an interface that the application declares (chapter 9), and the refund rule is a Strategy (chapter 11).



Mermaid. The desk depends on interfaces; each provider points into one. The `Session` owns its bookings (composition). **This diagram is checked against the code** (Step 8).

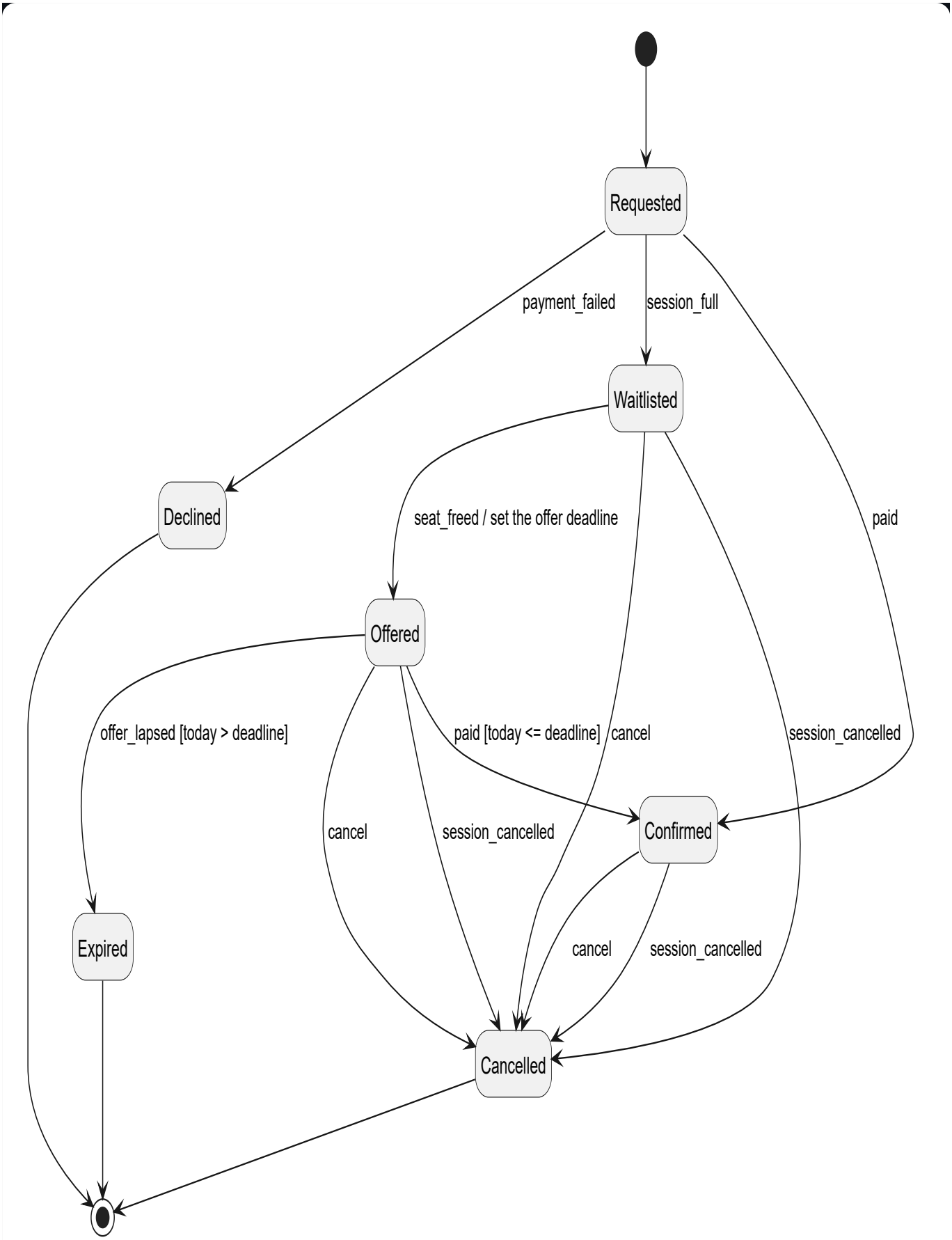
Step 5: Behaviour: One Sequence, One State Machine

The booking's central scenario is "Book a seat". The sequence diagram (chapter 7) draws the three outcomes in one picture with `alt` fragments: the session is full, the payment is declined, the payment succeeds.



Mermaid. Drawn by hand from the design, then checked against traces of the real code in Step 8.

The booking also has a *life*, which is a state machine (chapter 8): it starts Requested, can be Waitlisted, Offered, Confirmed, and ends Declined, Expired or Cancelled. As in chapter 8, the table of transitions is the source, and the diagram is generated from it.



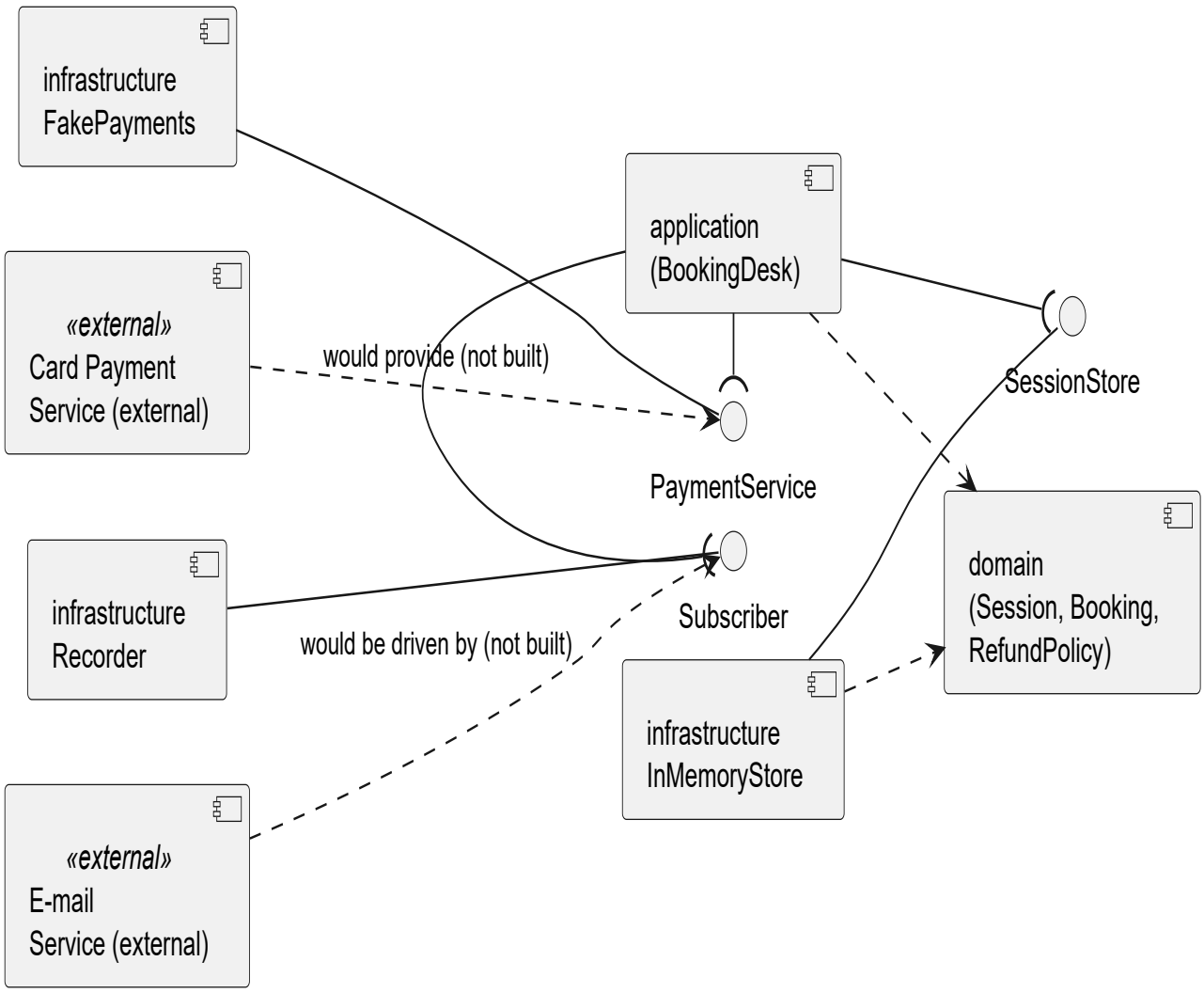
PlantUML, generated from `TRANSITIONS` in `workshops/domain/booking.py`.

And the matrix, computed by driving a real booking into every state and sending every event (`(stays)` means the event was ignored or its guard was false):

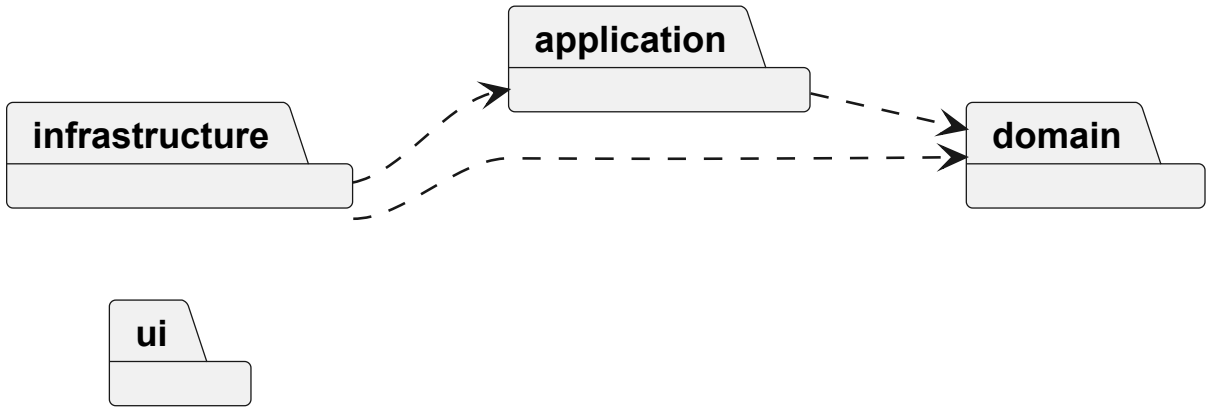
	paid	payment_failed	session_full	seat_freed
Requested	Confirmed	Declined	Waitlisted	illegal
Waitlisted	illegal	illegal	illegal	Offered
Offered	Confirmed	illegal	illegal	illegal
Confirmed	illegal	illegal	illegal	illegal
Declined	illegal	illegal	illegal	illegal
Expired	illegal	illegal	illegal	illegal
Cancelled	illegal	illegal	illegal	illegal

Read the matrix as design review. Terminal states accept nothing. A confirmed booking can only be cancelled (by the attendee or by the organiser). Two cells deserve a question: a booking that is still `Requested` cannot be cancelled by anyone, and cannot be cancelled by the organiser's session cancellation. With synchronous payment a booking is Requested for the length of one call, so this cannot happen today. If payment ever becomes asynchronous it will, which is recorded as a design decision (D8) and not left as a surprise.

Step 6: Structure



PlantUML. The application requires three interfaces and the infrastructure provides them; the two external services are drawn as "would provide" and "would be driven by", because they were not built.



PlantUML, generated from the real imports by chapter 9's checker, adapted to this package. Everything points at the domain.

```
analysed 11 files, found 4 cross-package imports
package dependencies found in the code (from -> to : first place):
   application     -> domain          workshops\application\desk.py:7  (workshops.domain
   infrastructure  -> application     workshops\infrastructure\memory.py:2  (workshops.a
   infrastructure  -> domain          workshops\infrastructure\memory.py:3  (workshops.d
R1 direction:
R2 no cycles:
R3 public API only:
result: clean
```

Step 7: The Code

Three layers, with the dependency rules of chapter 9. The domain holds the rules and imports nothing. These are its key files: the booking and its state machine, the session, and the refund strategy.

```
"""A booking, and its life as a table-driven state machine. The diagram of the booking's
from dataclasses import dataclass
from datetime import date, timedelta

OFFER_DAYS = 1                      # a member offered a freed seat has this many days


class Refusal(Exception):
    """A business rule said no. The message is for the person."""


class IllegalTransition(Exception):
    pass


@dataclass(frozen=True)
class Transition:
    source: str
    event: str
    target: str
    guard: str = ""
    action: str = ""
```

```
TRANSITIONS = [
    Transition("Requested", "paid", "Confirmed"),
    Transition("Requested", "payment_failed", "Declined"),
    Transition("Requested", "session_full", "Waitlisted"),
    Transition("Waitlisted", "seat_freed", "Offered", action="set the offer deadline"),
    Transition("Offered", "paid", "Confirmed", guard="today <= deadline"),
    Transition("Offered", "offer_lapsed", "Expired", guard="today > deadline"),
    Transition("Confirmed", "cancel", "Cancelled"),
    Transition("Waitlisted", "cancel", "Cancelled"),
    Transition("Offered", "cancel", "Cancelled"),
    Transition("Confirmed", "session_cancelled", "Cancelled"),
    Transition("Waitlisted", "session_cancelled", "Cancelled"),
    Transition("Offered", "session_cancelled", "Cancelled"),
]
STATES = ["Requested", "Waitlisted", "Offered", "Confirmed", "Declined", "Expired", "Can
EVENTS = ["paid", "payment_failed", "session_full", "seat_freed", "offer_lapsed", "cance
IGNORED = {(s, "offer_lapsed") for s in STATES if s != "Offered"}      # a daily tick tha
HOLDS_A_SEAT = {"Requested", "Offered", "Confirmed"}


@dataclass
class Booking:
    id: str
    attendee_id: str
    session_id: str
    state: str = "Requested"
    offer_deadline: date | None = None
    receipt: str | None = None
    paid_pence: int = 0

    def send(self, event: str, today: date | None = None) -> str:
        if (self.state, event) in IGNORED:
            return self.state
        options = [t for t in TRANSITIONS if t.source == self.state and t.event == event
        if not options:
            raise IllegalTransition(f"{event} is not allowed for a booking that is {self
        t = options[0]
        if t.guard == "today <= deadline" and not today <= self.offer_deadline:
            raise IllegalTransition("the offer has lapsed")
        if t.guard == "today > deadline" and not today > self.offer_deadline:
            return self.state                                                  # not y
        if t.action == "set the offer deadline":
            self.offer_deadline = today + timedelta(days=OFFER_DAYS)
        self.state = t.target
        return self.state
```

```
    def holds_a_seat(self) -> bool:
        return self.state in HOLDS_A_SEAT

    def can_accept_offer(self, today: date) -> bool:
        return self.state == "Offered" and today <= self.offer_deadline


def to_plantuml() -> str:
    lines = ["@startuml", "scale 1.2", "hide empty description", "[*] --> Requested"]
    for t in TRANSITIONS:
        label = t.event + (f" [{t.guard}]" if t.guard else "") + (f" / {t.action}" if t.
        lines.append(f"{t.source} --> {t.target} : {label}")
    lines += [f"{s} --> [*]" for s in ("Declined", "Expired", "Cancelled")]
    lines.append("@enduml")
    return "\n".join(lines) + "\n"
```

```
"""A session of a workshop: its capacity, its fee, and its bookings. The session decides
from dataclasses import dataclass, field
from datetime import date
from itertools import count

from workshops.domain.booking import Booking, Refusal
from workshops.domain.refunds import RefundPolicy

_ids = count(1)


@dataclass
class Session:
    id: str
    workshop: str
    starts_on: date
    capacity: int
    fee_pence: int
    bookings: list[Booking] = field(default_factory=list)
    cancelled: bool = False

    def seats_in_use(self) -> int:
        return sum(1 for b in self.bookings if b.holds_a_seat())

    def is_full(self) -> bool:
        return self.seats_in_use() >= self.capacity
```

```
    def find(self, booking_id: str) -> Booking:
        for b in self.bookings:
            if b.id == booking_id:
                return b
        raise Refusal("no such booking")

    def open_booking(self, attendee_id: str, today: date) -> Booking:
        """Create a booking. It holds a seat while it awaits payment, or joins the waiti
        if self.cancelled:
            raise Refusal("this session has been cancelled")
        if today >= self.starts_on:
            raise Refusal("this session has already started")
        if any(b.attendee_id == attendee_id and b.state in ("Requested", "Waitlisted", "
            raise Refusal("already booked, or waiting, for this session")
        full = self.is_full()
        booking = Booking(f"{self.id}-B{next(_ids)}", attendee_id, self.id)
        self.bookings.append(booking)
        if full:
            booking.send("session_full")
        return booking

    def promote_waitlist(self, today: date) -> Booking | None:
        """If a seat is free and someone is waiting, offer it to whoever has waited long
        if self.cancelled or self.is_full():
            return None
        for b in self.bookings:                                  # bookings are kept in
            if b.state == "Waitlisted":
                b.send("seat_freed", today)
                return b
        return None

    def cancel_booking(self, booking: Booking, today: date, policy: RefundPolicy) -> int
        """Cancel one booking and return the refund owed."""
        was = booking.state
        booking.send("cancel")
        if was != "Confirmed":
            return 0
        return policy.refund(booking.paid_pence, (self.starts_on - today).days)

    def cancel_session(self) -> list[tuple[Booking, int]]:
        """Cancel the whole session: everyone who had paid is owed everything back."""
        self.cancelled = True
        out = []
        for b in self.bookings:
```

```
            if b.state in ("Waitlisted", "Offered", "Confirmed"):
                owed = b.paid_pence if b.state == "Confirmed" else 0
                b.send("session_cancelled")
                out.append((b, owed))
        return out

    def expire_offers(self, today: date) -> list[Booking]:
        lapsed = []
        for b in self.bookings:
            if b.state == "Offered" and today > b.offer_deadline:
                b.send("offer_lapsed", today)
                lapsed.append(b)
        return lapsed
```

```
"""How much of the fee is returned when an attendee cancels. A Strategy, so the organiser
from typing import Protocol, runtime_checkable


@runtime_checkable
class RefundPolicy(Protocol):
    def refund(self, fee_pence: int, days_before: int) -> int: ...


class DefaultRefund:
    """Everything back if the session is at least `full_days` away, nothing after that."

    def __init__(self, full_days: int = 7):
        self.full_days = full_days

    def refund(self, fee_pence: int, days_before: int) -> int:
        return fee_pence if days_before >= self.full_days else 0
```

The application layer: the interfaces it requires, and the desk that coordinates the use cases.

```
"""The interfaces the application REQUIRES. Declared here; the code that provides them l
from typing import Protocol, runtime_checkable

from workshops.domain import Booking, Session


class PaymentDeclined(Exception):
```

```
    """The payment service said no (card declined, service unavailable)."""


@runtime_checkable
class SessionStore(Protocol):
    def get(self, session_id: str) -> Session: ...

    def find_booking(self, booking_id: str) -> tuple[Session, Booking]: ...

    def all_sessions(self) -> list[Session]: ...

    def save(self, session: Session) -> None: ...


@runtime_checkable
class PaymentService(Protocol):
    def charge(self, attendee_id: str, pence: int) -> str: ...      # returns a receipt;

    def refund(self, receipt: str, pence: int) -> None: ...


@runtime_checkable
class Subscriber(Protocol):
    def notify(self, event: object) -> None: ...
```

```
"""The use cases of the booking desk. It coordinates; the rules live in Session and Book
from datetime import date

from workshops.application.events import (AddedToWaitlist, BookingCancelled, BookingConf
                                          SessionCancelled)
from workshops.application.ports import PaymentDeclined, PaymentService, SessionStore, S
from workshops.domain import Booking, DefaultRefund, RefundPolicy, Refusal


class BookingDesk:
    def __init__(self, store: SessionStore, payments: PaymentService, subscribers: tuple
                 refund_policy: RefundPolicy | None = None):
        self.store, self.payments = store, payments
        self.subscribers = list(subscribers)
        self.refund_policy = refund_policy or DefaultRefund()
        self.errors: list[str] = []

    # --- use case: book a seat ---
```

```
    def book(self, attendee_id: str, session_id: str, today: date) -> Booking:
        session = self.store.get(session_id)
        booking = session.open_booking(attendee_id, today)
        if booking.state == "Waitlisted":
            self._publish(AddedToWaitlist(booking.id, attendee_id, session_id))
        else:
            try:
                booking.receipt = self.payments.charge(attendee_id, session.fee_pence)
            except PaymentDeclined:
                booking.send("payment_failed")
                self._publish(BookingDeclined(booking.id, attendee_id))
                # no waiting list to serve here: with synchronous payment, nobody can be
            else:
                booking.paid_pence = session.fee_pence
                booking.send("paid", today)
                self._publish(BookingConfirmed(booking.id, attendee_id, session_id))
        self.store.save(session)
        return booking

    # --- use case: pay for an offered seat ---
    def accept_offer(self, booking_id: str, today: date) -> Booking:
        session, booking = self.store.find_booking(booking_id)
        if not booking.can_accept_offer(today):                    # decide BEFORE takin
            raise Refusal("there is no open offer for this booking")
        try:
            booking.receipt = self.payments.charge(booking.attendee_id, session.fee_penc
        except PaymentDeclined:
            raise Refusal("the payment was declined; the offer stays open until its dead
        booking.paid_pence = session.fee_pence
        booking.send("paid", today)
        self._publish(BookingConfirmed(booking.id, booking.attendee_id, session.id))
        self.store.save(session)
        return booking

    # --- use case: cancel a booking ---
    def cancel(self, booking_id: str, today: date) -> int:
        session, booking = self.store.find_booking(booking_id)
        receipt = booking.receipt
        refund = session.cancel_booking(booking, today, self.refund_policy)
        if refund:
            self.payments.refund(receipt, refund)
        self._publish(BookingCancelled(booking.id, booking.attendee_id, refund))
        self._offer_freed_seat(session, today)
        self.store.save(session)
        return refund
```

```
    # --- use case: the organiser cancels a session ---
    def cancel_session(self, session_id: str, today: date) -> int:
        session = self.store.get(session_id)
        total = 0
        for booking, owed in session.cancel_session():
            if owed:
                self.payments.refund(booking.receipt, owed)
                total += owed
        self._publish(SessionCancelled(session_id, total))
        self.store.save(session)
        return total

    # --- the nightly clock tick ---
    def day_passed(self, today: date) -> None:
        for session in self.store.all_sessions():
            for lapsed in session.expire_offers(today):
                self._publish(OfferExpired(lapsed.id, lapsed.attendee_id))
            self._offer_freed_seat(session, today)
            self.store.save(session)

    def _offer_freed_seat(self, session, today: date) -> None:
        while (offered := session.promote_waitlist(today)) is not None:
            self._publish(OfferMade(offered.id, offered.attendee_id, offered.offer_deadl

    def _publish(self, event: object) -> None:
        for s in self.subscribers:
            try:
                s.notify(event)
            except Exception as e:      # noqa: BLE001 - a broken subscriber must not un
                self.errors.append(f"{type(s).__name__} failed on {type(event).__name__}
```

(The infrastructure, `workshops/infrastructure/memory.py`, holds an in-memory store, a payment fake that can be told to decline particular people, and a recorder that stands in for the e-mail service. The event classes are plain dataclasses. All of it is in the project files.) A few lines carry decisions from the design document: the desk decides whether an offer is still open *before* it takes money (`accept_offer`), a failing subscriber is recorded and never undoes a booking (`_publish`), and the comment in `book` says why a declined payment needs no waiting-list handling.

Step 8: Tests, Mutants and Checks

The tests cover the state machine (every state and event pair is defined, ignored or illegal), the session's rules (one seat per person, a seat held from the request, the longest waiter first, the refund boundary) and every desk use case including the failure paths. They pass:

```
Ran 23 tests in 0.250s

OK
```

Passing means little until the tests are shown to fail on broken code. Three plausible bugs, each introduced on purpose:

```
unmodified: 0 of 21 tests fail []
refund boundary off by one: 1 of 21 tests fail ['SessionRules.test_refund_depends_on_how_
newest waiter is offered the seat: 2 of 21 tests fail ['Desk.test_an_unclaimed_offer_exp
money taken before the offer is checked: 1 of 21 tests fail ['Desk.test_a_lapsed_offer_i
restored: 0 of 21 tests fail []
```

Each was caught: an off-by-one on the refund boundary (exactly seven days must still refund), the newest waiter offered the seat instead of the oldest, and money taken before the offer was checked (the guarantee in the "Book a seat" use case that nobody is charged for something that did not succeed).

Do the Diagrams Match the Code?

The class diagram is checked for drawn operations, attributes and realisations (chapter 11's check). The sequence diagram is checked against *three traced runs* of the real desk (seat free and paid; session full; payment declined), using chapter 7's tracer, extended to trace a whole package: every call the code makes must be drawn, and every call drawn must be made in at least one run.

```
"""Check the capstone's design diagrams against the code.

1. the class diagram: every drawn operation and attribute exists, every drawn realisatio
2. the sequence diagram: run three scenarios (seat free and paid, session full, payment
   every call the code makes (at the diagram's level of detail) is drawn, and every call
```

```
Names only, and the sequence check does not test the ORDER of calls across branches: it s
"""
import pathlib
import re
import sys
from datetime import date

import diagram_check as dc
from tracetool import to_mermaid, trace
from workshops.application import BookingDesk
from workshops.domain import Session
from workshops.infrastructure import FakePayments, InMemoryStore, Recorder

HERE = pathlib.Path(__file__).parent
dc.HERE = HERE
dc.MODULES = ["workshops.domain", "workshops.application", "workshops.infrastructure"]
dc.CHECKS = {"design.class.mmd": ["workshops/domain/session.py", "workshops/domain/booki
                                  "workshops/application/desk.py", "workshops/applicatio

PROVIDERS = {"InMemoryStore": "SessionStore", "FakePayments": "PaymentService", "Recorde
HIDDEN = {"is_full", "seats_in_use", "_publish", "holds_a_seat", "can_accept_offer"}
ACTOR = "Attendee"
TODAY = date(2026, 11, 1)


def drawn_calls() -> set[tuple[str, str, str]]:
    out = set()
    for line in (HERE / "diagrams" / "book.sequence.mmd").read_text(encoding="utf8").spli
        m = re.match(r"^\s*(\w+)(?:->>|-\))\+?(\w+):\s*(.+)$", line)
        if m:
            s, r, label = m.groups()
            op = "create" if label.startswith("create ") else label.split("(")[0]
            out.add((s, r, op))
    return out


def traced_calls(scenario) -> set[tuple[str, str, str]]:
    desk, session = scenario()
    events, _, _ = trace("workshops", desk.book, "ann", "S1", TODAY, actor=ACTOR)
    out = set()
    for line in to_mermaid(events, actor=ACTOR).splitlines():
        m = re.match(r"^\s*(\w+)->>(\w+):\s*(.+)$", line)
        if not m:
            continue
        s, r, label = m.groups()
```

```
        op = "create" if label.startswith("create ") else label.split("(")[0]
        if op in HIDDEN or (op == "create" and r != "Booking"):          # the events th
            continue
        out.add((PROVIDERS.get(s, s), PROVIDERS.get(r, r), op))
    return out


def seat_free():
    s = Session("S1", "Pottery", date(2026, 12, 1), 2, 2500)
    return BookingDesk(InMemoryStore([s]), FakePayments(), (Recorder(),)), s


def full():
    s = Session("S1", "Pottery", date(2026, 12, 1), 1, 2500)
    desk = BookingDesk(InMemoryStore([s]), FakePayments(), (Recorder(),))
    desk.book("other", "S1", TODAY)
    return desk, s


def declined():
    s = Session("S1", "Pottery", date(2026, 12, 1), 2, 2500)
    return BookingDesk(InMemoryStore([s]), FakePayments({"ann"}), (Recorder(),)), s


def main() -> int:
    problems = 0
    print("class diagram against the code:")
    for line in _class_check():
        print("  " + line)
    drawn = drawn_calls()
    seen: set = set()
    for name, scenario in (("seat free, paid", seat_free), ("session full", full), ("pay
        calls = traced_calls(scenario)
        seen |= calls
        undrawn = sorted(calls - drawn)
        print(f"sequence diagram, scenario '{name}': {len(calls)} calls traced; not draw
        problems += len(undrawn)
    phantom = sorted(drawn - seen)
    print(f"sequence diagram, drawn but never made in any scenario: {phantom or 'none'}"
    return 1 if problems + len(phantom) else 0


def _class_check() -> list[str]:
    import contextlib
    import io
```

```
    buf = io.StringIO()
    with contextlib.redirect_stdout(buf):
        dc.main()
    return buf.getvalue().splitlines()


if __name__ == "__main__":
    sys.exit(main())
```

```
class diagram against the code:
  design.class.mmd OK
sequence diagram, scenario 'seat free, paid': 8 calls traced; not drawn: none
sequence diagram, scenario 'session full': 7 calls traced; not drawn: none
sequence diagram, scenario 'payment declined': 8 calls traced; not drawn: none
sequence diagram, drawn but never made in any scenario: none
```

It passed on the first run, which is not evidence on its own, so here it is on a copy where I removed a drawn call, added a call the code never makes, and renamed an operation in the class diagram:

```
class diagram against the code:
  design.class.mmd 1 problem(s)
      Session.start_booking() is drawn but not in the code
sequence diagram, scenario 'seat free, paid': 8 calls traced; not drawn: [('BookingDesk'
sequence diagram, scenario 'session full': 7 calls traced; not drawn: [('BookingDesk', '
sequence diagram, scenario 'payment declined': 8 calls traced; not drawn: [('BookingDesk
sequence diagram, drawn but never made in any scenario: [('BookingDesk', 'Session', 'ref
(exit status 1)
```

Each tamper was found. What it does not check: the order of calls across the alternatives, and anything about the meaning of a message.

The Story

All the use cases end to end, with the e-mails the subscriber would send. Two people book a two-seat pottery session and a third is waitlisted; a card is declined on another session; the first booker cancels twenty days ahead (refunded in full), the freed seat is offered to the waiting person, who does not take it (the offer expires at the nightly run); a second cancellation three

days before the session earns no refund; the organiser cancels a session; a booking on that session is refused.

```
"""One story through the running service: booking, waiting list, cancellation, offer, ex
from datetime import date, timedelta

from workshops.application import BookingDesk
from workshops.domain import Refusal, Session
from workshops.infrastructure import FakePayments, InMemoryStore


class Printer:
    """The stand-in for the e-mail service: prints what it would send."""

    def __init__(self):
        self.today = None

    def notify(self, event):
        print(f"   [{self.today}] e-mail: {type(event).__name__} {', '.join(f'{k}={v}' f

printer = Printer()
pottery = Session("S1", "Pottery for beginners", date(2026, 12, 1), 2, 2500)
coding = Session("S2", "Coding for beginners", date(2026, 11, 20), 5, 4000)
payments = FakePayments(decline={"dee"})
desk = BookingDesk(InMemoryStore([pottery, coding]), payments, (printer,))
d = date(2026, 11, 1)


def step(day, text):
    printer.today = day
    print(f"{day}  {text}")


step(d, "ann and bo book the pottery session (2 seats), cy asks too")
ann = desk.book("ann", "S1", d)
bo = desk.book("bo", "S1", d)
cy = desk.book("cy", "S1", d)
step(d, f"dee's card is declined on the coding session; the state is {desk.book('dee', '
step(d, f"seats in use on pottery: {pottery.seats_in_use()} of {pottery.capacity}; cy is

d2 = d + timedelta(days=10)
step(d2, "ann cancels, 20 days before the session")
step(d2, f"refund: {desk.cancel(ann.id, d2)}p; cy is now {cy.state} (deadline {cy.offer_
```

```
d3 = d2 + timedelta(days=3)
step(d3, "three days later cy has not paid: the nightly job runs")
desk.day_passed(d3)
step(d3, f"cy is {cy.state}")

d4 = date(2026, 11, 28)
step(d4, "bo cancels three days before the session")
step(d4, f"refund: {desk.cancel(bo.id, d4)}p")

step(d4, "the organiser cancels the coding session")
step(d4, f"refunds paid: {desk.cancel_session('S2', d4)}p (no confirmed bookings there, d

step(d4, "an attempt to book a cancelled session")
try:
    desk.book("eve", "S2", d4)
except Refusal as e:
    print(f"   refused: {e}")
print()
print("payments charged:", payments.charges)
print("payments refunded:", payments.refunds)
```

```
2026-11-01  ann and bo book the pottery session (2 seats), cy asks too
   [2026-11-01] e-mail: BookingConfirmed attendee_id=ann, session_id=S1
   [2026-11-01] e-mail: BookingConfirmed attendee_id=bo, session_id=S1
   [2026-11-01] e-mail: AddedToWaitlist attendee_id=cy, session_id=S1
   [2026-11-01] e-mail: BookingDeclined attendee_id=dee
2026-11-01  dee's card is declined on the coding session; the state is Declined
2026-11-01  seats in use on pottery: 2 of 2; cy is Waitlisted
2026-11-11  ann cancels, 20 days before the session
   [2026-11-11] e-mail: BookingCancelled attendee_id=ann, refund_pence=2500
   [2026-11-11] e-mail: OfferMade attendee_id=cy, deadline=2026-11-12
2026-11-11  refund: 2500p; cy is now Offered (deadline 2026-11-12)
2026-11-14  three days later cy has not paid: the nightly job runs
   [2026-11-14] e-mail: OfferExpired attendee_id=cy
2026-11-14  cy is Expired
2026-11-28  bo cancels three days before the session
   [2026-11-28] e-mail: BookingCancelled attendee_id=bo, refund_pence=0
2026-11-28  refund: 0p
2026-11-28  the organiser cancels the coding session
   [2026-11-28] e-mail: SessionCancelled session_id=S2, refunds_pence=0
2026-11-28  refunds paid: 0p (no confirmed bookings there, dee was declined)
2026-11-28  an attempt to book a cancelled session
```

```
    refused: this session has been cancelled

payments charged: [('ann', 2500, 'R1'), ('bo', 2500, 'R2')]
payments refunded: [('R1', 2500)]
```

Step 9: A Review, and the Gap It Found

Chapter 11's checklist, applied to the finished design. The architecture rules are clean; the state machine's matrix has no unexplained cell; the smell detector flags nothing in the session or the desk. It flags one thing in `booking.py`, "anemic?", for the `Transition` class, which is a false alarm: a row of a table is data by design. Reading the code found what the detectors did not (the exercises build on these):

- **The desk sets fields on a booking from outside** (`receipt` and `paid_pence`, four assignments). A `mark_paid` on the booking would put that rule with the data.
- **Cancelling a session ignores a booking that is still Requested.** Harmless with synchronous payment, wrong the day payment is not.
- **A question for the manager nobody asked:** booking on the day of the session is refused ("already started"). Is that right? It is an assumption.
- **What if the payment service fails to refund?** This is the "NOT YET DESIGNED" extension from the use case. Here is what the code does today:

```
"""Use case 'Cancel a booking', extension 5b ('the payment service fails to refund') is n
from datetime import date

from workshops.application import BookingDesk
from workshops.domain import Session
from workshops.infrastructure import FakePayments, InMemoryStore, Recorder

TODAY = date(2026, 11, 1)


class RefundsFail(FakePayments):
    def refund(self, receipt, pence):
        raise ConnectionError("payment service unreachable")


session = Session("S1", "Pottery", date(2026, 12, 1), 1, 2500)
rec = Recorder()
```

```
desk = BookingDesk(InMemoryStore([session]), RefundsFail(), (rec,))
ann = desk.book("ann", "S1", TODAY)
bo = desk.book("bo", "S1", TODAY)                       # waits
rec.events.clear()
try:
    desk.cancel(ann.id, TODAY)
except ConnectionError as e:
    print(f"desk.cancel raised: {type(e).__name__}: {e}")
print(f"ann's booking is now: {ann.state}   (paid {ann.paid_pence}p, refunded: nothing)"
print(f"bo (waiting) is now : {bo.state}   (the freed seat was never offered)")
print(f"events published    : {rec.events or 'none'}")
```

```
desk.cancel raised: ConnectionError: payment service unreachable
ann's booking is now: Cancelled   (paid 2500p, refunded: nothing)
bo (waiting) is now : Waitlisted   (the freed seat was never offered)
events published    : none
```

That is a real defect, found by writing a use case honestly and then asking what the code does where the use case is silent. The attendee's booking is cancelled, money is owed and nothing records it, and the person waiting does not get their offer until the next nightly run. The design document lists it as gap G1 with a proposal, and Exercise 1 closes it, as a subclass that does not touch the package.

Step 10: The Design Document

A short document, two pages as text, for the next person (who may be you in six months): what the system is for, where each artefact lives, the decisions with their reasons and how each is checked, the evidence of quality, and the gaps. This is the whole of it.

```
WORKSHOP BOOKING SERVICE: DESIGN DOCUMENT                                  version 1, 20

1. PURPOSE AND SCOPE
   Let people book, cancel and wait for seats on a community centre's workshop sessions,
   In scope: booking, waiting list, offers, cancellation with refund, cancelling a sessi
   several seats per booking (Q1), real card and e-mail providers, a user interface, per

2. WHERE THINGS ARE
   requirements.txt           the manager's words, with questions and assumptions
```

```
   usecases.puml, *.usecase.txt  who wants what; "Book a seat" and "Cancel a booking" in
   glossary.txt, domain.mmd      the vocabulary and the concepts (generated from domain_mo
   design.class.mmd              classes, responsibilities, interfaces
   book.sequence.mmd             "Book a seat" with its alternatives
   booking.state.puml            the life of a booking (generated from the transition tabl
   components.puml, packages.puml  the pieces and the direction of dependency (packages
   workshops/                    the code: domain, application, infrastructure; 23 tests;

3. DECISIONS
   D1  Three layers, dependencies pointing inward: infrastructure -> application -> doma
       Why: the rules (who gets a seat, what is refunded) must be testable without a car
       Checked by: deps.py, run as a test. Consequence: every outside thing is behind an
   D2  The Session decides who gets a seat; BookingDesk only coordinates.
       Why: the session holds the capacity and the bookings (Information Expert). Conseq
   D3  A booking is a table-driven state machine (Requested, Waitlisted, Offered, Confir
       Why: the lifecycle has many "not allowed" cases, and a table makes each one visib
       and event pair is defined, ignored or illegal. Consequence: a new state is new ro
   D4  A booking holds a seat from the moment it is requested until it ends (Requested,
       Why: otherwise two people could be told yes to the last seat. Consequence: the wa
   D5  Refund rule as a Strategy (RefundPolicy). Why: the manager will change it. Conseq
   D6  Notifications as events to subscribers (Observer); a failing subscriber is record
       Why: R7 will grow (text messages, a log), and a mail outage must not stop booking
   D7  The desk decides an offer is still open BEFORE charging for it. Why: charging the
   D8  Payment is synchronous: book() asks the card service and waits. Why: simplest thi
       Consequence: a booking is "Requested" only for the length of one call, and some t
       illegal, which is fine now and would need design if payment became asynchronous.

4. QUALITY EVIDENCE
   23 tests pass (state machine, session rules, every desk use case, the architecture ru
   charge before checking) were each caught by the tests. The class diagram and the sequ
   which also failed, as it should, on tampered copies.

5. KNOWN GAPS AND OPEN DECISIONS (nothing below is built)
   G1  If the payment service fails to REFUND, the booking is already cancelled, the att
       published; the waiting person is only offered the seat at the next nightly run (g
       on the booking, retry it nightly, and tell the attendee. Needs the manager's answ
   G2  One process, in memory. Two desks, or a restart, lose or fork the data (chapter 9
   G3  Q3 (date change), Q1 (several seats) and Q4 (how "a week" is counted) are assumed
   G4  Real card and e-mail providers are not built; FakePayments and Recorder stand in.
       and Subscriber interfaces, and the card provider's real failure modes (timeouts,
   G5  The offer deadline is whole days; "a day to take it" is "until the end of the nex
```

MISTAKES WHILE PREPARING THIS CHAPTER

- **I drew an `extend` arrow backwards.** My first use case diagram had `Book a seat` pointing at `Join the waiting list`. Chapter 3 says the extend arrow points at the *base*, and I still got it wrong on the first try. I only saw it when I looked at the rendered picture. Takeaway: the notation rules you know are not the ones you apply without checking.
- **A test file copied from another project named a file that does not exist here.** I reused chapter 11's architecture test, which tampers with `domain/model.py` to prove the check can fail, and this project has no such file (`FileNotFoundError`). The error was loud, the cause trivial, and the lesson is that a copied test carries assumptions with it.
- **I wrote code for a case that cannot happen.** My first desk, after a declined payment, offered the freed seat to the waiting list. Writing the test for it, I could not construct the situation: someone only waits while the session is full, and with synchronous payment a declined booking never coexists with a waiting person. I deleted the line, replaced the test with one stating the invariant (`test_a_waiting_list_never_coexists_with_a_free_seat`), and recorded in the design document that asynchronous payment would reopen it.
- **The first test for that case was muddled.** It tried to simulate a payment "in flight" by editing a booking's state by hand. A test that has to cheat to reach a case is evidence the case is unreachable, which is what it turned out to be.
- **A shell here-document wrote nothing.** The application and infrastructure files were written with a single heredoc command that the shell refused ("unexpected EOF"), and because the whole command failed, the folders stayed empty. I noticed only when I listed them. The file-writing tool is the reliable way to create source; and after a failed command, check what it did not do.
- **Everything passed first time.** The 21 tests, the architecture rules, the diagram checks: all green on the first run. That is the moment to distrust it. The mutants and the tampered-diagram run are in the chapter for that reason, not for decoration.

What the Method Gave, and What It Did Not

- **It found the questions early** (requirements Q1 to Q6) and made the assumptions visible.
- **It found one real defect** (G1) through a use case extension left honest, and several smaller decisions (D4, D7, D8) while drawing.
- **It did not make the design correct.** The 23 tests and the checks show the code does what the diagrams say, not that the diagrams are what the centre needs. Only the manager can confirm Q1 to Q6, and nothing here replaces asking her.
- **It cost effort that a smaller system would not repay.** For a throwaway script, none of this is warranted (chapter 1). For a service people pay money into, the checks on refunds and offers were cheap compared with one wrong refund.

Hands-On Exercises

EXERCISE 1

Close gap G1 without editing the package: write a desk subclass that, when a refund fails, records the debt, publishes an event, still offers the freed seat, and retries at the nightly run. Show it with a payment service that is down and then comes back.

View solution

EXERCISE 2

Add requirement R1, which the design so far lacks: "see the upcoming sessions and how many seats are left". Write it as a read-only query beside the desk, with tests (cancelled and past sessions excluded, seats left never negative, waitlisted people hold no seat).

View solution

EXERCISE 3

Review the capstone design yourself with the chapter 11 checklist. Back four findings with evidence from the code (run something, do not just assert it): at least one the smell detector reports, one it misses, and one that is a question for the manager.

View solution

CHAPTER 12 QUICK REFERENCE

- The method, in order: requirements and their questions, use cases, domain model and glossary, class design by responsibility, key sequence and state diagrams, components and packages, code, tests and checks, a short design document
- Leave a use case extension honest ("NOT YET DESIGNED") rather than invent an answer; then ask what the code does there: it found gap G1
- Generate what you can (domain diagram from data, state diagram from a table, package diagram from imports); check the rest (class and sequence diagrams against the code)
- Mutants and tampered diagrams prove the tests and checks can fail; a check that passes first time is not yet evidence

- A design document records decisions with reasons and how each is checked, the evidence, and the gaps: what you did not build is part of the design
- Diagrams and tests show the code matches the design, not that the design is right; confirm the assumptions with the people who own them

Course Complete: What You Can Now Do

CHAPTER	YOU CAN NOW...
1 Why Model	Say what analysis and design are, choose sketch mode, and judge how much modelling a problem deserves
2 Diagrams as Code	Write diagrams as text in Mermaid and PlantUML, render them, and keep them beside the code
3 Use Cases	Find actors and goals and write a use case with alternative flows, then turn it into a test
4 Domain Model	Find the concepts of a problem, draw them with multiplicities, keep a glossary, and review it with non-programmers
5 Class Diagrams	Read and draw the full class notation and the six relationships, and know what each becomes in code
6 Responsibilities	Assign responsibilities with CRC cards and GRASP, recognise an anemic model, and measure the difference
7 Sequence Diagrams	Turn a scenario into a sequence diagram, derive operations from it, and check it against a call trace
8 State and Activity	Model an object's life as a tested state machine and a process as an activity diagram with swimlanes
9 Structure Diagrams	Draw components and packages, enforce dependency rules as a test, and ask "what if there are two?" of a deployment
10 UML and Code	Generate a skeleton from a diagram, reverse-engineer code, and keep diagrams honest with checks
11 Patterns and Review	Recognise patterns on a diagram, review a design with a checklist, and refactor on the diagram before the code
12 Capstone	Take a new problem from requirements to a checked, running design with a short document and an honest list of gaps

Where to go next. *Design Patterns* teaches the patterns this course only drew, with their trade-offs, and *Clean Code, SOLID & Refactoring* covers the principles behind chapters 6 and 11 and how to refactor safely. *Software Architecture Fundamentals* takes the structure diagrams of chapter 9 up a level (the C4 model, architecture decisions and the qualities a system must have), and *The Software Development Lifecycle* places requirements, design documents and reviews inside a whole project. Then take something you are building and work through the method on it. Start with the questions: that part costs the least and finds the most.

COURSE COMPLETE

This was the last chapter of *Object-Oriented Design with UML*. A combined PDF of all twelve chapters is built from these pages.