COMP433 · SOFTWARE ENGINEERING · CHAPTER 4

UML: Class Diagrams

From the behaviour we have already modelled to the system's static structure: the classes, what they hold, and how they connect, read straight out of the requirements.

Move with the arrow keys or the buttons below.

Agenda

Orientation

1. Where class diagrams sit among the UML views
2. A complete class diagram *at a glance*
3. What they capture; three design approaches
4. Reading a class out of the use case and activity

Notation

5. Anatomy of a class box
6. The visibility markers, and how they really work
7. Analysis classes vs design classes

Building and connecting

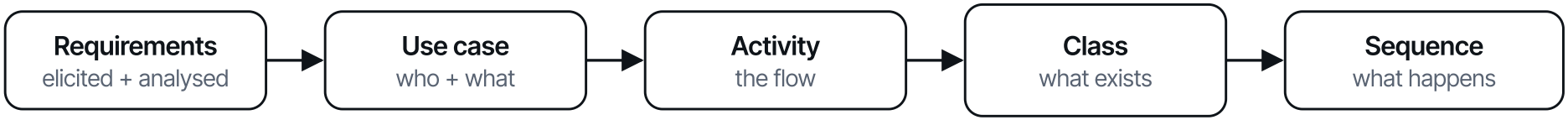
8. Finding classes: noun-verb analysis and judgement calls
9. Relationships: association, multiplicity, generalisation, aggregation, association classes
10. Dependency, data types and enumerations

Putting it to work

11. Good analysis classes; object diagrams
12. **Back to back**: one system as use case, activity and class
13. Pitfalls, and how to build one

Where we are: from behaviour to structure

We have built the two **behavioural** views: the **use case diagram** (*who* uses the system and *what* they can do) and the **activity diagram** (*how* one use case flows). The class diagram is the first **structural** view: the fixed cast of objects the behaviour acts on.



Behaviour (done)

- **Use case:** the tasks and the actors.
- **Activity:** the order of steps inside a task.

Structure (here)

- **Class:** the objects, their data and operations, and how they relate, the long-lived skeleton the behaviour runs on.
- Next: **sequence**, how those objects message one another over time.

LINK The two views are not separate work: the **actors and entities** of the use case diagram, the **data** a use case reads or writes, and the **operations** it needs all reappear here as classes, attributes and methods.

What a class diagram is, and when to draw it

A class diagram is used when developing an **object-oriented system model** to show the **classes** in a system and the **associations** between them. An object class is a general definition of one kind of system object; an association is a link saying two classes are related. In early analysis, objects represent things in the real world: a patient, a prescription, a doctor, a book.

What it captures

- The **classes**: each kind of object the system knows about.
- Their **attributes** (data) and **operations** (behaviour).
- The **associations**, with multiplicity, that connect them.

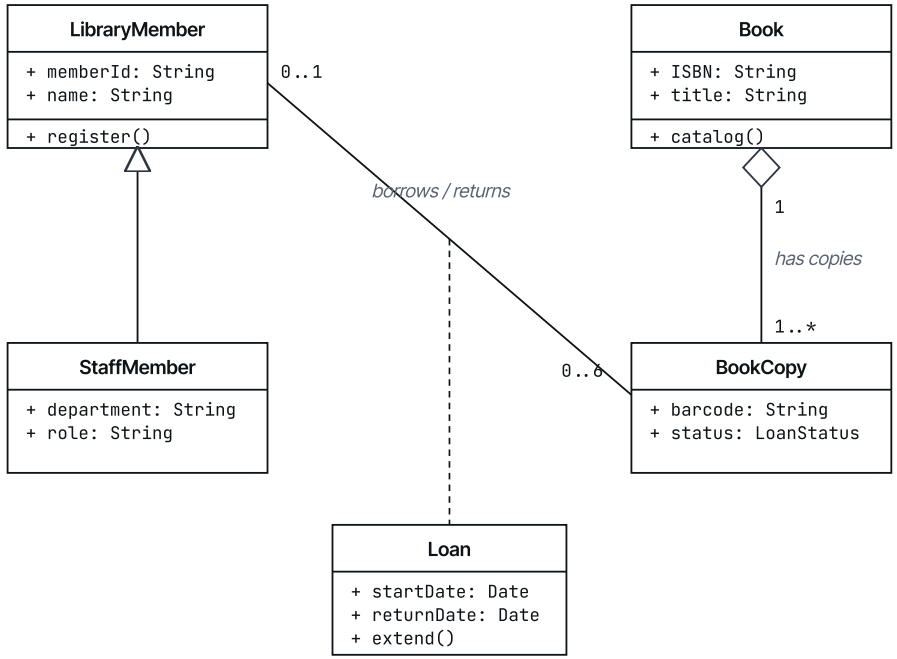
What it does not capture

- The **order** in which operations run (sequence / activity).
- An object's **lifecycle** over time (state machine).
- Concurrency, data flow, or where the code is deployed.

WHY IT MATTERS It is the **longest-lived diagram** on most projects: the static structure changes far more slowly than any one flow through it, so it is the artefact teams maintain and design against.

A class diagram at a glance

Before any details, here is a complete class diagram for a small library. Do not decode every symbol yet, the rest of the deck explains each labelled part. Keep returning to this picture: everything that follows is a piece of it.



WHAT YOU ARE LOOKING AT

Four **classes** (each a box of name / attributes / operations); a **generalisation** (StaffMember *is a* LibraryMember, the hollow triangle); an **aggregation** (a Book *has* copies, the diamond); an **association** with **multiplicity** (a member borrows 0..6 copies); and an **association class** (Loan, holding the borrow dates). Each one gets its own slide next.

Where the classes come from: three design approaches

Before any class is drawn you decide how to *decompose* the system. Adel's notes name three classic strategies; analysis-level class modelling usually blends them, but it helps to know which one you are leaning on.

Top-down

Begin with the whole system, split it into subsystems, then into classes. Strong when the high-level architecture is already understood; the risk is inventing classes the requirements never asked for.

Bottom-up

Begin with concrete domain things, a book, a copy, a member, and compose them upward. This is exactly what **noun-verb analysis** does, and it keeps the model anchored to the requirements text.

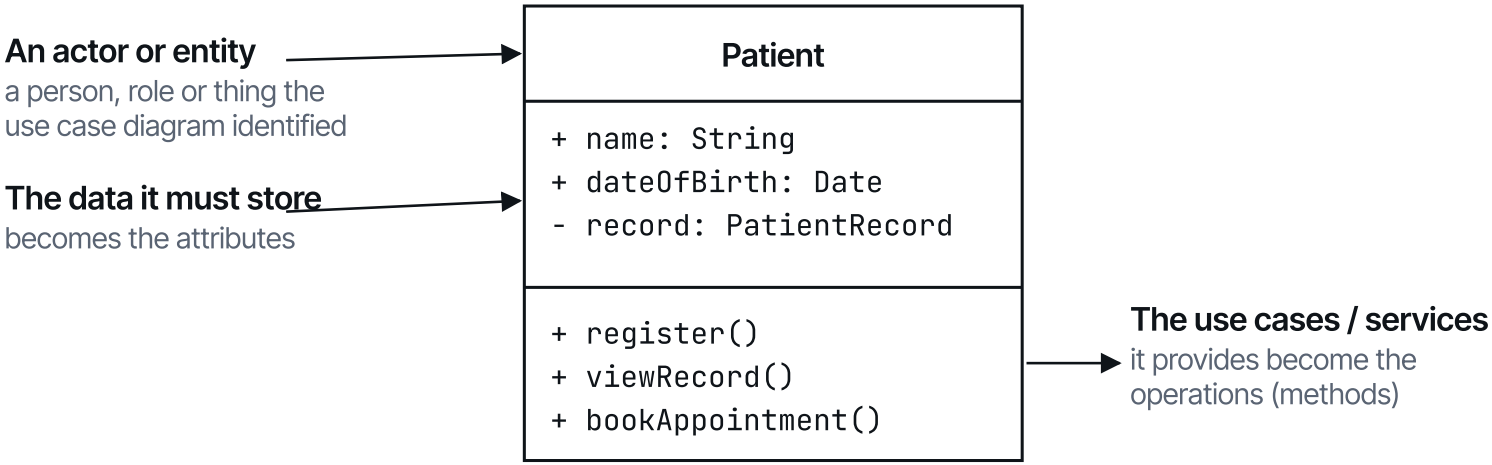
Middle-out

Begin with the few central concepts you are sure of, then work outward in both directions. Common in practice when one or two entities (Patient, Loan) clearly anchor the domain.

IN ANALYSIS We work mostly **bottom-up** from the requirements, using the noun-verb method on the next slides, and sanity-check it against a top-down view of the whole. The goal is classes *justified by the text*, not by the developer's imagination.

Reading a class out of the other diagrams

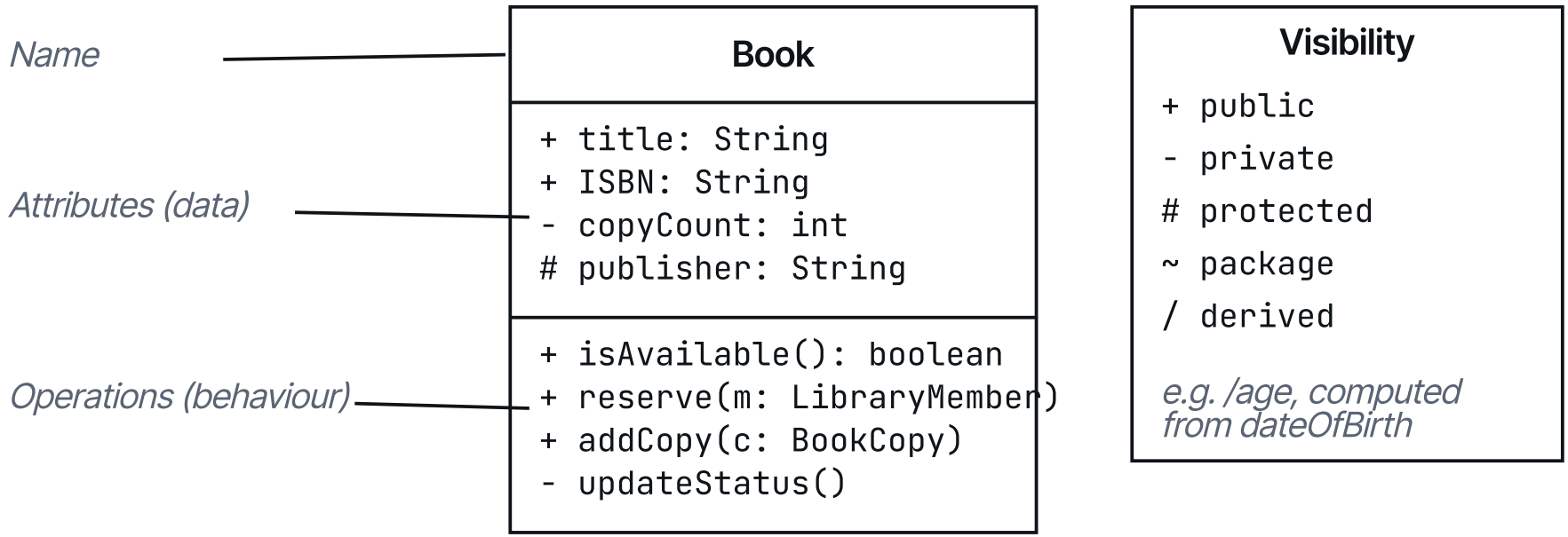
A single class box has three compartments, and each maps to something you have already produced. This is the bridge from the behavioural views to the structural one.



TRACEABILITY Name compartment ← an actor/entity. Attributes ← the data the use cases touch. Operations ← the use cases that object performs. Every part of the class is justified by an earlier model.

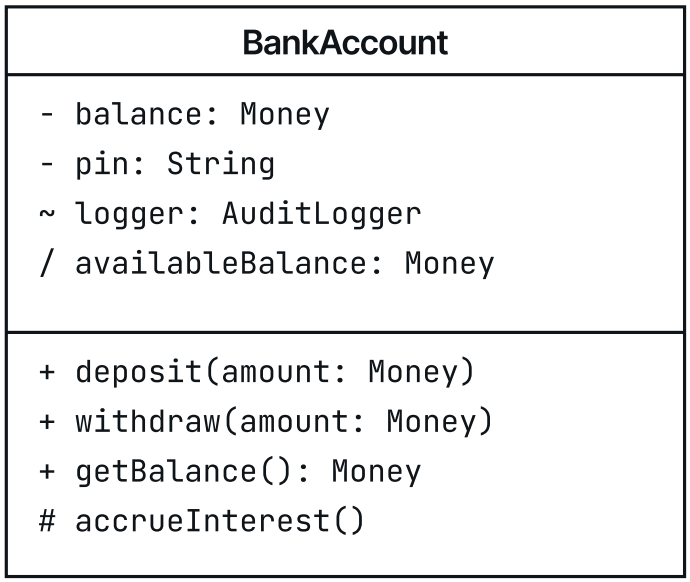
Anatomy of a class box, in full notation

Three compartments: **name** (top), **attributes** (middle), **operations** (bottom). Every attribute and operation carries a **visibility** marker; attributes and parameters carry a **type** after a colon.



The visibility markers

The marker sits before the member name and records a design decision: *who is allowed to touch this member*. One `BankAccount` class shows all five at once.



Why each marker

+	**public**	any other class's code may use it – *deposit(), getBalance()*
-	**private**	only code inside this class – *balance, pin*
#	**protected**	this class and its subclasses – *accrueInterest()*
~	**package**	classes in the same package – *logger*
/	**derived**	computed, not stored – *availableBalance = balance – holds*

HOW TO CHOOSE, AND A REAL-LIFE ANALOGY

Default to **private data, public operations**: the outside calls methods, never reaches into a field. Think of a house, **public** is the *doorbell* anyone may ring; **private** your *diary*; **protected** a *spare key the family (subclasses) share*; **package** a *key the other flats in the building hold*; **derived** your *age*, computed from the birth date, not stored. Keep it light at analysis level; tighten at design level.

How visibility actually works

Two clarifications make the markers click. Both are exactly how Java, C#, C++ and UML define visibility.

1. It governs code, not people

Visibility says which **code** (which classes) may use a member; the *compiler* enforces it. Not about end-users: a `Customer` is a separate class, limited to an account's **public** methods.

2. The unit is the class, not the object

`private` means "any code *inside this class*, on **any** object of it", not just `this`. So one account can read another's private balance, if that code lives in `BankAccount`.

The tangible example

```
class BankAccount {
  - balance: Money                          // private to the class

  + transferTo(other: BankAccount, amt) {
       this.balance  = this.balance  - amt
       other.balance = other.balance + amt    // LEGAL: 'other' is a BankAccount too
  }
}

class StatementPrinter {                     // a different class
  + print(a: BankAccount) {
       a.getBalance()    // OK, getBalance() is public
       a.balance         // COMPILER ERROR, balance is private to BankAccount
  }
}
```

THE ANSWER "Can one object touch another's private data?" **Yes if they are the same class** (the transfer between two `BankAccount`s); **no** for any other class, which sees only the public methods.

Analysis classes vs design classes

An **analysis class** abstracts one or more classes (and sometimes subsystems) of the eventual design. It models the problem domain, not the solution.

An analysis class

- Focuses on **functional requirements**.
- Defines **responsibilities**: cohesive subsets of behaviour (the use cases or services it provides to others).
- Defines **attributes** at a domain level.
- Expresses the **relationships** it takes part in.

A design class adds

- Full method **signatures** and parameter / return **types**.
- Tightened **visibility**.
- Infrastructure classes (persistence, controllers, factories) and design patterns.
- Navigability and implementation detail.

IN THIS COURSE Phase 3 expects an **analysis-level** class diagram drawn from your requirements. Reserve the design-level detail (signatures, patterns, infrastructure) for later.

Finding the classes: data-driven noun-verb analysis

The standard recipe (Shlaer and Mellor, late 1980s): identify the data first, group it into classes, then attach behaviour. It keeps the model grounded in the requirements rather than the developer's imagination.

1. Read the requirements and **underline every noun and noun phrase**; collect them as candidate classes.
2. **Discard** the inappropriate candidates (next slide), promoting some to attributes.
3. Identify every **verb and verb phrase**; these are candidate operations.
4. **Assign** each operation to the class responsible for it.

TWO OUTPUTS Nouns become **classes and attributes**; verbs become **operations**. The noun-verb pairings hint at the **associations** between classes.

Which nouns to keep, which to discard

Discard a candidate noun if it is

- **Redundant**, names the same thing as another (member / borrower).
- **Omnipotent**, a vague "manager" or "system" that would do everything.
- **Vague**, "information", "data", "item".
- **Meta-language**, words about the requirements ("feature", "constraint").
- **Outside the system scope**.
- **Really an attribute**, "title", "ISBN" belong to Book.

Good classes tend to be (Booch; Shlaer & Mellor)

- **Tangible / real-world things**: book, copy, course, room.
- **Roles**: library member, student, doctor, member of staff.
- **Events**: arrival, departure, request, payment.
- **Interactions**: meeting, consultation, treatment.

Worked example: the Library, noun analysis

Adel's running example. Underline the nouns in the requirement text, then sift them.

The library contains books and journals. It may have several copies of a given book. Some of the books are for short-term loans only. All other books may be borrowed by any library member for three weeks. Members of the library can normally borrow up to six items at a time, but members of staff may borrow up to 12 items at one time. Only members of staff may borrow journals. The system must keep track of when books and journals are borrowed and returned.

Candidate classes (kept)

`Book`, `BookCopy`, `Journal`, `LibraryMember`, `StaffMember`

Discarded / demoted

Library (the system itself), **item** (vague), **three weeks / six / 12** (attributes or rules, not classes).

Which nouns become classes, and the judgement calls

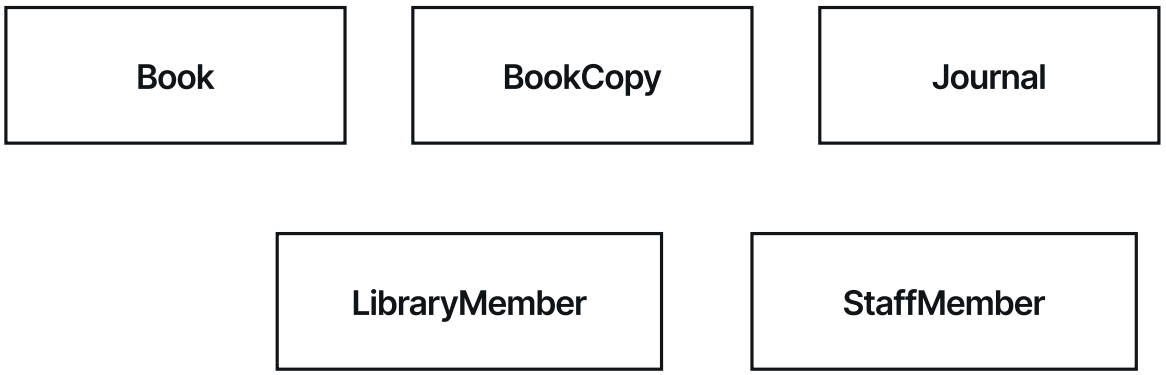
Noun-verb analysis gives candidates, not answers. For each one you argue *keep*, *reject*, or *it depends*, and the reasoning matters more than the verdict. Reasonable modellers differ on the rows marked "debatable".

CANDIDATE	DECISION	WHY, AND THE ALTERNATIVE YOU REJECTED
`book`	class	The *work* itself (ISBN, title). Why not merge it with its copies? One title has many physical copies, and you lend a copy, not the title – merging the two loses that distinction.
`copy`	class	The *physical item* on the shelf (barcode, status). Keep only `Book` and you cannot say which of its three copies is on loan; `BookCopy` is the fix.
`journal`	class (debatable)	Kept separate because journals follow different loan rules (staff only). **Alternative:** let `Book` and `Journal` both specialise an abstract `LibraryItem` holding the shared loan behaviour – tidier if more item types arrive.
`item`	reject, but…	Too vague to be a class itself, yet it is the *hint* for that `LibraryItem` superclass. A rejected noun can still shape the model.
`loan` / borrowing	association class	It looks like a verb, but it carries its own data (start and return dates) belonging to the *member-copy pairing*, not to either alone – so it becomes the `Loan` association class.
`member`, `staff`	class + generalisation	Staff *is-a* member, so generalisation. **Alternative:** if one person can be both at once, model a `Role` rather than a subclass – a genuine design fork.
`library`	reject	The *system boundary*, not a domain object – unless the requirements track several libraries, when it becomes a class after all.
three weeks, six, 12	reject	Loan period and borrowing limits: *attributes and constraints* (multiplicities), not classes.

THE TEACHING POINT A good class list is **defended, not asserted**. Be ready to say why `BookCopy` earns its place and `library` does not, and to name the alternative you considered – the `LibraryItem` superclass, the `Role` class – and why you set it aside.

First-cut analysis class model

The kept nouns become boxes. At this stage they are just named classes; the verb analysis then fills in operations, and the associations come next.



Verbs (borrow, return, keep track) become the operations next.

VERB ANALYSIS **borrow**, **return**, **keep track of** become operations such as `borrow()` / `return()` on the member classes and `recordLoan()` on the system. Assign each verb to the class responsible for it.

Relationships between classes

Relationships are connections between modelling elements: they describe how objects work together and act as a sanity check on the model. Four kinds carry almost all the weight.

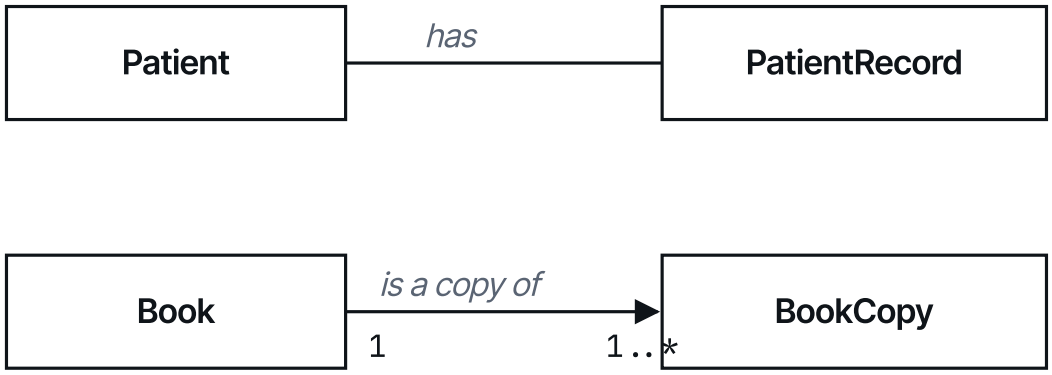
RELATIONSHIP	MEANING	NOTATION
Association	A persistent link: one class holds, uses or refers to another.	plain line, with multiplicity at each end
Generalisation	One class is a more specific kind of another and inherits it (`is-a`).	solid line, hollow triangle pointing at the parent
Aggregation / composition	Part-of: one class is built out of others (`part-of`).	diamond at the whole, hollow (aggregation) or filled (composition)
Dependency	A transient "uses" link: a change in one may affect the other.	dashed line with an open arrowhead

LINKS VS ASSOCIATIONS

An **association** is a relationship between *classes*; a **link** is an instance of an association between two specific *objects*. `Adel:LibraryMember` linked to `copy42:BookCopy` is one link of the member-borrows-copy association.

Associations: direction and naming

Name an association with a verb phrase and read it in a direction. Navigability (the arrowhead) says which way access and flow can travel.



bidirectional
access flows both ways

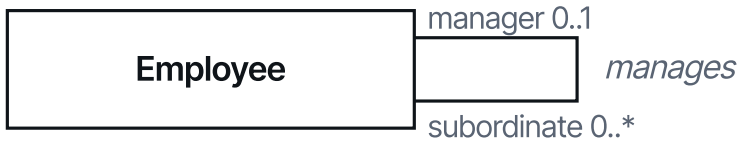
unidirectional
a Book reaches its copies, not the other way

READING DIRECTION A plain association line is **bidirectional**. An **arrowhead** makes it unidirectional: the tail class can reach the head class, but not vice versa. Multiplicity sits at each end.

Associations in detail: roles, reflexive links, and instances

An association line can say more than “these two are related”. It can name the **role** each end plays, and a class can even be associated with **itself**.

Role names
each end can be labelled with the role that class plays in the link



Reflexive (self) association
a class related to itself: one Employee manages many, each managed by one

ASSOCIATION VS LINK

An **association** connects *classes* – the type level. A **link** is one *instance* of it between two specific objects (Adel: “links instantiate associations”). The class diagram shows the associations; the **object diagram** (later) shows the links, e.g. `sara:Employee` manages `omar:Employee`.

Multiplicity (cardinality)

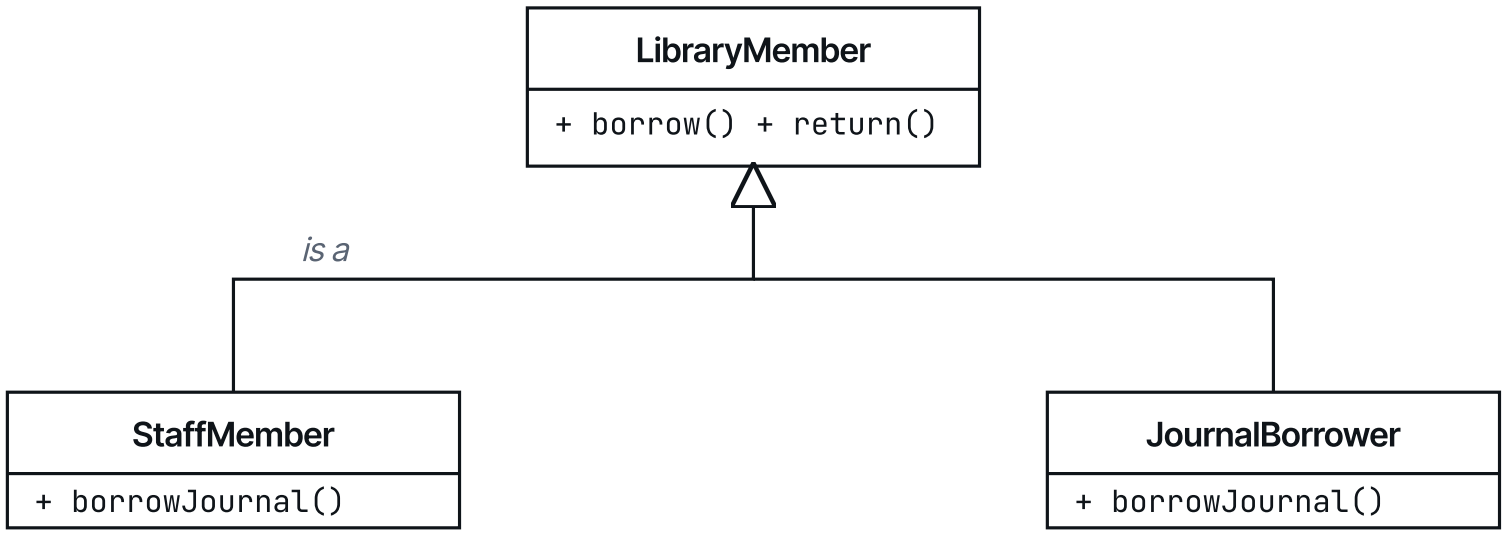
Multiplicity on an association end answers: how many instances of this class can take part in the relationship at one time?

NOTATION	MEANING
`1`	exactly one
`0..1`	optional: zero or one
`*` or `0..*`	zero or more
`1..*`	one or more
`m..n`	at least m, at most n (e.g. `0..6`, `3..7`)
`m`	exactly m (e.g. `5`, a chessboard has `64` squares)

READ IT BOTH WAYS In the Library: a `LibraryMember` borrows `0..6` copies; a `BookCopy` is borrowed by `0..1` member at a time. Multiplicity is where business rules ("up to six items") enter the static model.

Generalisation (inheritance)

When one class is a more specific kind of another, draw a **generalisation**: a solid line with a hollow triangle pointing at the parent. The child inherits the parent's attributes, operations and associations, and adds its own.



WHICH WAY THE TRIANGLE POINTS

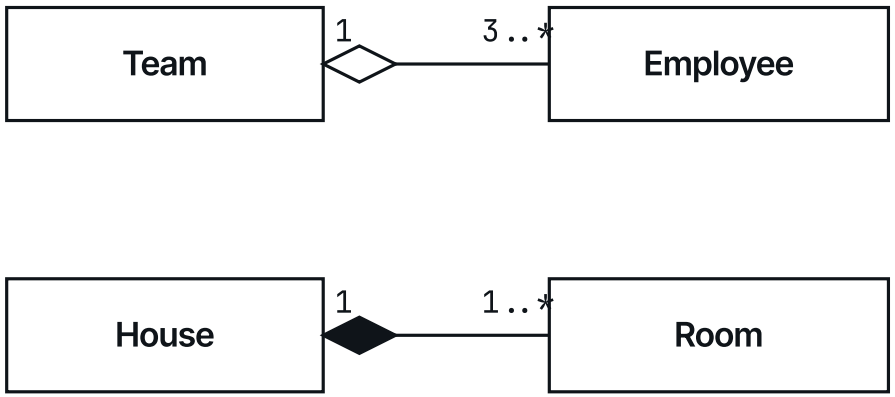
The triangle always points at the **general** (parent) class. Use generalisation only for a genuine `is-a`: a StaffMember *is a* LibraryMember.

EXAMPLE: RIGHT VS WRONG

Wrong: "`Car` is-a `Engine`". A car is not a kind of engine; it *has* one, so that is an **association** (`Car` has `Engine`), not inheritance. **Right:** "`ElectricCar` is-a `Car`". The test is **substitutability**: anywhere the code expects a `LibraryMember` you can hand it a `StaffMember` and nothing breaks. If the substitution would be absurd (an `Engine` where a `Car` is expected), it is not generalisation.

Aggregation and composition (part / whole)

Both say one class is built out of others. They differ in the **lifetime** of the part: the diamond sits at the *whole*.



Aggregation (hollow diamond)
weak part-of: an Employee outlives the Team

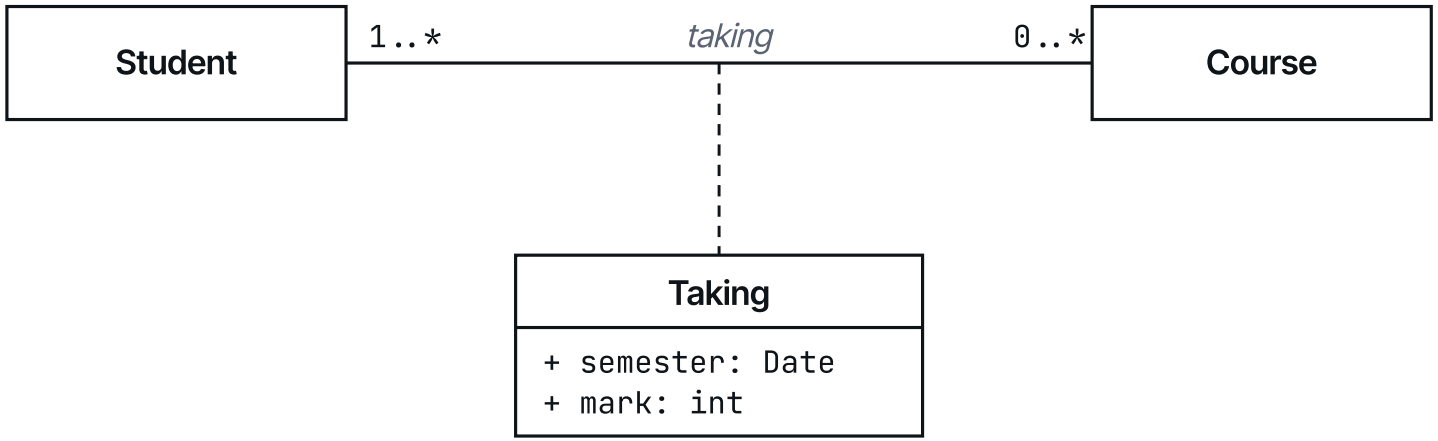
Composition (filled diamond)
strong part-of: destroy the House and its Rooms go too

PRACTITIONER NOTE Prefer **composition** where it fits: aggregation is semantically weak and many tools and reviewers cannot tell it apart from a plain association. Composition has real lifetime bite.

ADEL'S EXAMPLES A **Course** is part of a **Programme** (aggregation, and a course may sit in more than one programme). An **Airplane** is made up of **Assemblies**, each made up of **Components** – aggregation can *recurse*. A **chessboard** is composed of exactly **64 Squares** that cannot exist without it – composition, multiplicity `64` on the part end.

Association classes

When an attribute belongs to the *relationship* rather than to either class, put it on an **association class**: a class box joined to the association line by a dashed link, sharing the association's name.

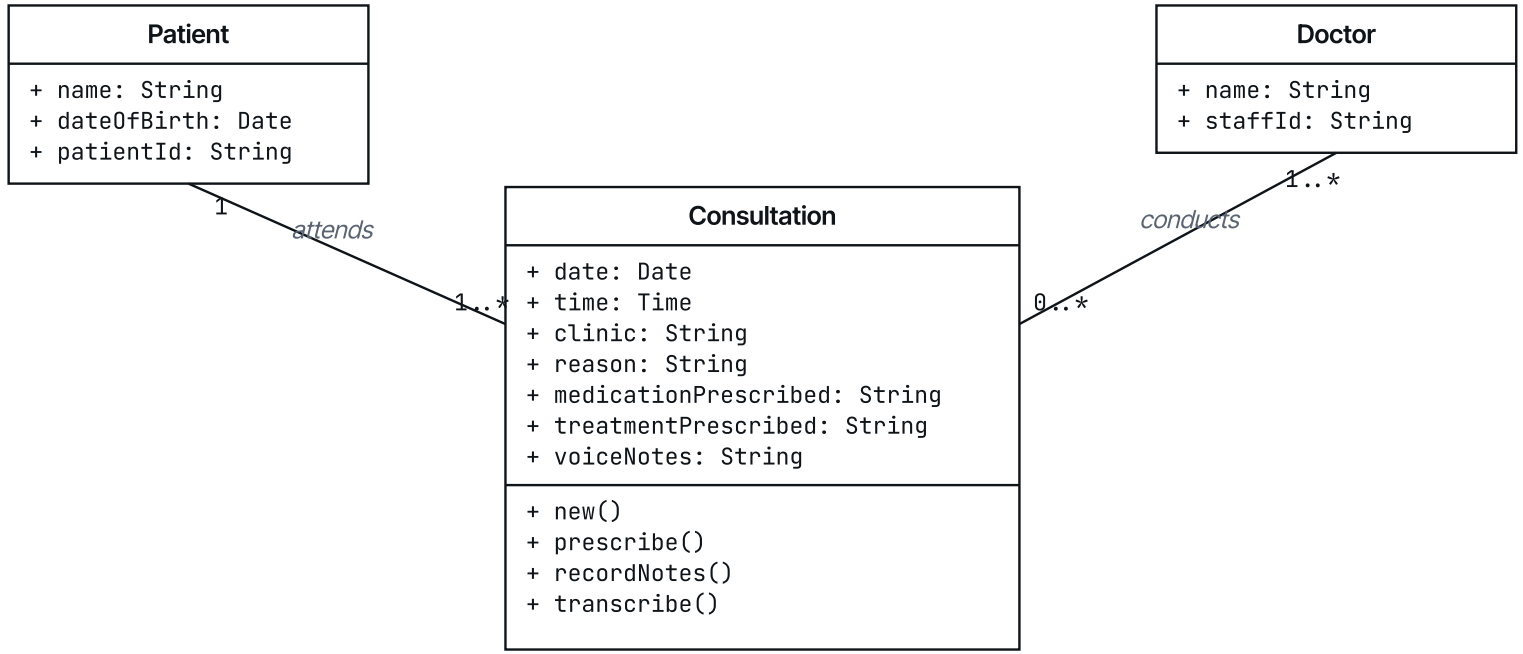


WHERE IT BELONGS

`mark` is not a property of the Student (they have many marks) nor of the Course (it has many) but of the specific *Student-takes-Course* pairing. A **ternary** association class does the same for a three-way relationship (e.g. Doctor-Patient-Treatment).

A second domain: the MHC-PMS (patient records)

Adel's other running example, from Sommerville's mental-health-care patient management system. It shows a different shape: one **central, attribute-rich class**, `Consultation`, that anchors the whole model.

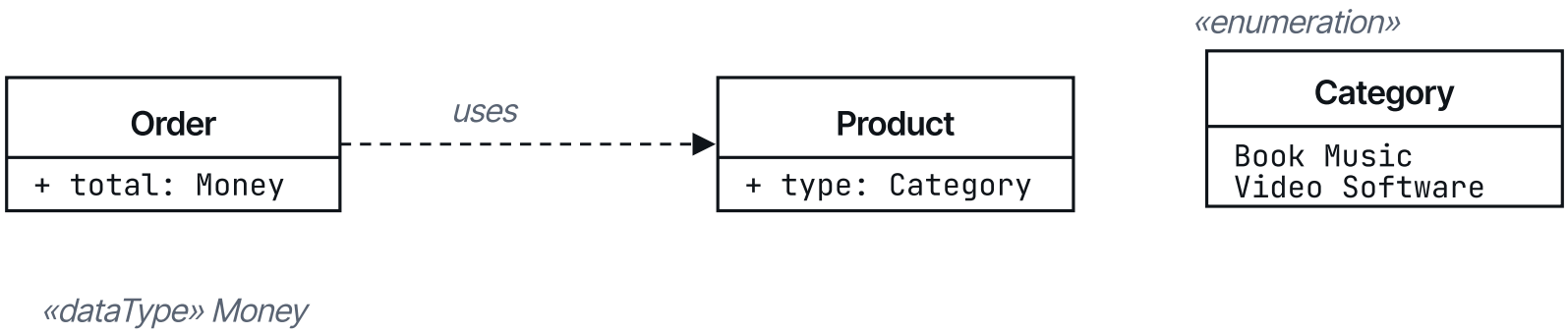


WHAT TO NOTICE

The most important class in a model is often not the actor but a central *event* or *interaction*. `Consultation`'s **attributes** are the data one consultation records; its **operations** are what a doctor does during one (`prescribe`, `recordNotes`). The two associations read straight from the sentence "a doctor conducts a consultation that a patient attends".

Dependency, data types and enumerations

Two more notations from richer diagrams. A **dependency** (dashed open arrow) is a transient "uses" link, weaker than an association. **Stereotypes** in guillemets label a class's kind.



ASSOCIATION VS DEPENDENCY

If an Order *holds* Products over time, that is an **association** (solid line). If a method merely *takes a Product as an argument* or calls it once, that is a **dependency** (dashed). Dependency is the weakest of the relationships.

WHAT «ENUMERATION» AND «DATATYPE» MEAN

An **enumeration** is a type whose value must be one of a *fixed, named list*, exactly like an `enum` in code: a `Category` is one of `Book`, `Music`, `Video` or `Software`, and nothing else. A **data type** is a type with *no identity of its own*, defined purely by its value: two `Money` amounts of £5 are interchangeable, whereas two `Order` objects are distinct even with identical fields. Both are written with a **stereotype** in guillemets, «like this», to flag that the box is not an ordinary domain class.

What makes a good analysis class

A short checklist that reviews look for. Most weak class models fail on cohesion or coupling.

- **Its name reflects its intent.** `Prescription` tells you what it is; `Manager` or `Helper` tells you nothing.
- **It is a crisp abstraction** of one specific element of the problem domain, not several.
- **It has a small, defined set of responsibilities**, one reason to change.
- **High cohesion:** it holds only the attributes and operations that genuinely belong together; unrelated ones lower cohesion.
- **Low coupling:** it has only the associations it truly needs; accidental links raise coupling and spread change.

THE TWO FAILURE MODES

Most weak models fail one of two ways: an **omnipotent** class (the "manager" that does everything) has **low cohesion**; a class wired to almost every other one has **high coupling** and spreads change. The next slide shows the cohesion failure concretely.

Cohesion: high versus low

Shown, not told. The class on the left does four unrelated jobs; the one on the right does one, with the rest moved to their own classes.

Low cohesion (bad)

```
Patient
  + name, dateOfBirth
  + register()
  + sendReminderEmail()    // messaging
  + generatePdfReport()    // reporting
  + connectToDatabase()    // persistence
  + calculateInsurance()   // billing
```

Four unrelated jobs in one class – **four reasons to change**. Touch the email server and you risk breaking billing.

High cohesion (good)

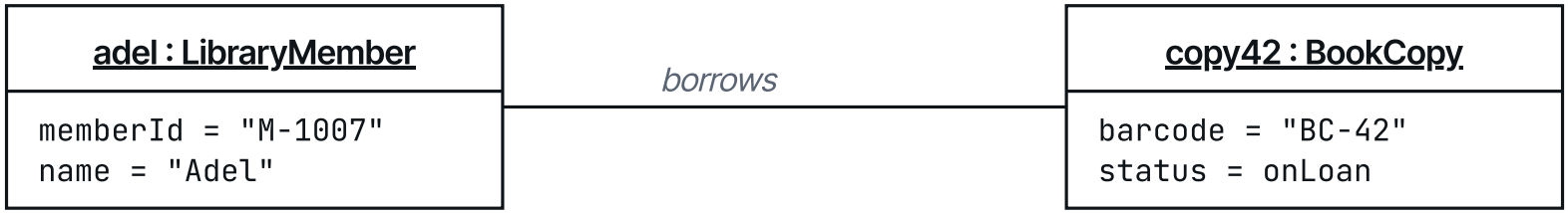
```
Patient
  + name, dateOfBirth
  + register()
  + updateContact()

// the rest moved out, to:
NotificationService, ReportGenerator,
PatientRepository, InsuranceCalculator
```

One job: **be a patient**. Each other concern lives in its own class. One reason to change.

THE TEST Count the **reasons to change**. The left `Patient` changes whenever email, reporting, persistence *or* billing changes, four reasons. The right one changes only when what it means to *be a patient* changes. Fewer reasons to change is higher cohesion.

Object diagrams: a snapshot of instances

Where a class diagram shows the *types*, an **object diagram** shows specific **instances** and the **links** between them at one moment, a snapshot. Object names are underlined, in the form `name : Class`.



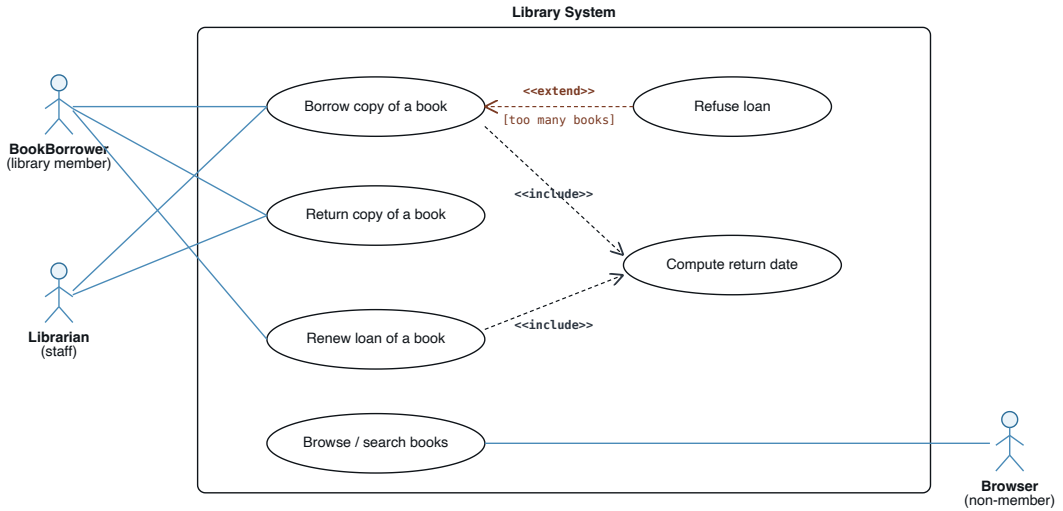
WHY DRAW ONE Object diagrams **validate the class model**: building a concrete snapshot of real data quickly exposes a missing class, a wrong multiplicity, or an attribute that has nowhere to live.

Back to back: one system, three views (1 of 4)

The three diagrams are not separate exercises. Take a single description and watch the **same words** drive the use case diagram, then the activity diagram, then the class diagram.

The Library. A library holds *books* and *journals*, with several *copies* of each book. A *library member* may *borrow* a copy for three weeks, up to six items; *staff* may borrow more, and journals. A *librarian issues*, *returns* and *renews* loans; the system *computes the return date* and *refuses* a loan past the limit. Anyone may *browse* the catalogue.

View 1 – the use case diagram: *who*, and *what*

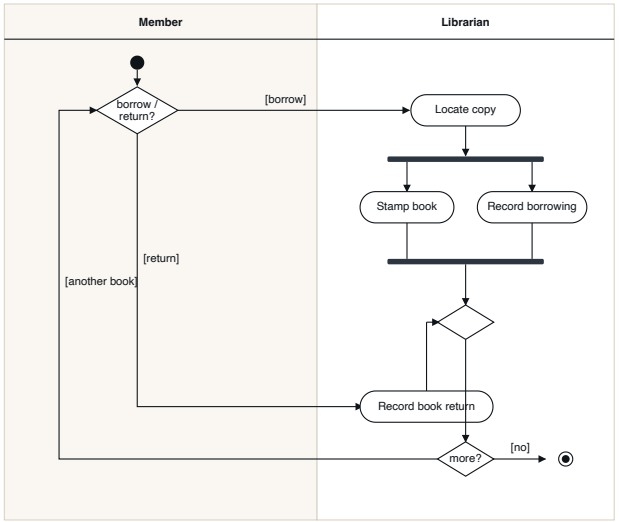


FROM THE TEXT

The **roles** (“member”, “librarian”, anyone browsing) become **actors**; the things they *do* (“borrow”, “return”, “renew”, “browse”) become **use cases**; “compute the return date”, always part of borrowing, is an `<<include>>`; “refuse a loan when too many”, a conditional, is an `<<extend>>`.

Back to back: one system, three views (2 of 4)

Pick one use case, **Borrow copy of a book**, and unpack *how it flows*. Each action is a verb the system or an actor performs; the swimlanes record who does each.

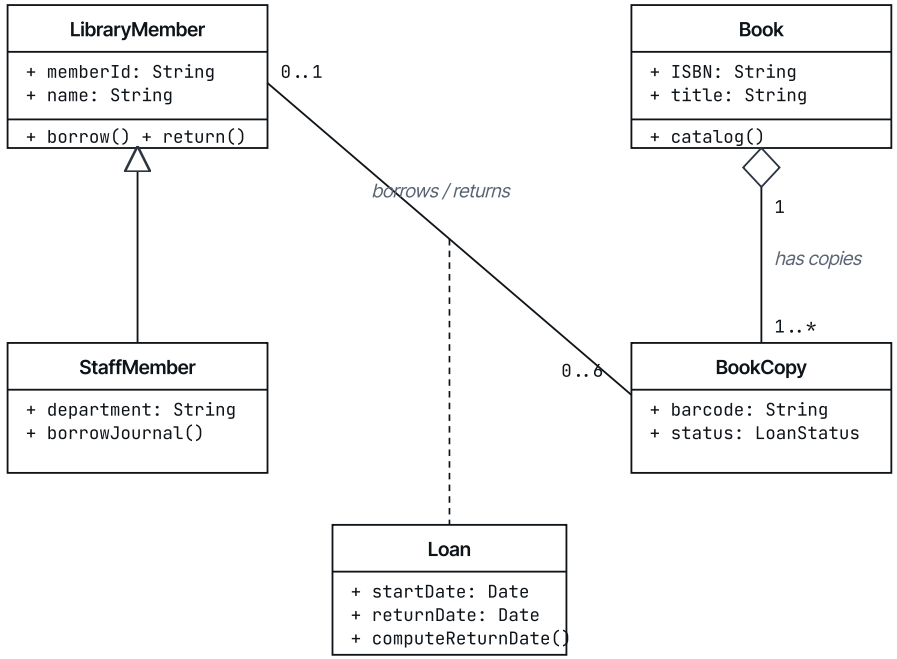


FROM USE CASE TO FLOW, AND HOW TO READ IT

The use case's paths become the activity's **actions, decisions and loops** (locate, stamp, record, the "more?" loop), and those verbs become **operations** next. Reading it: the diamonds use *named* guards (`[borrow]` / `[return]`, not yes/no); the parallel bars are a **fork then join** (stamp and record happen together, the join waits); and the plain unlabelled diamond is a **merge** re-joining the branches, so the decision is closed.

Back to back: one system, three views (3 of 4)

This is the diagram from the opening “at a glance” slide, now read *against the description*. The nouns are classes; the data they hold are attributes; the use cases and activity actions are operations; the noun-verb pairings are the associations.



FROM TEXT TO STRUCTURE

The **nouns** become classes (Book, BookCopy, LibraryMember, StaffMember); the **data** become attributes (ISBN, title, barcode); the **verbs** become operations (borrow, return, computeReturnDate). “Staff are members” is generalisation; “a book has copies” is aggregation; “a member borrows copies” is the association whose dates live on the `Loan` association class; “up to six” is the multiplicity `0..6`.

Back to back: one system, three views (4 of 4)

The traceability map. Every diagram element should be answerable to a phrase in the description – this table is how a reviewer, or you, checks that nothing was invented and nothing was dropped.

PHRASE IN THE DESCRIPTION	USE CASE DIAGRAM	ACTIVITY DIAGRAM	CLASS DIAGRAM
“a library member”, “staff”	`BookBorrower, Librarian (actors)`	`Member, Librarian (swimlanes)`	`LibraryMember, StaffMember (staff is-a member)`
“books”, “copies”, “journals”	`- (subjects of the use cases)`	`- (objects acted on)`	`Book, BookCopy, Journal (Book has 1..* copies)`
“borrow a copy”	`Borrow copy of a book`	`the borrow branch + its actions`	`borrow() operation; member-copy association`
“for three weeks” / “return date”	`Compute return date (<<include>>)`	`action inside the flow`	`Loan.startDate, returnDate, computeReturnDate()`
“up to six items”	`Refuse loan (<<extend>> [too many])`	`a decision guard`	`multiplicity 0..6 on the borrows association`
“browse or search”	`Browse / search books (Browser actor)`	`-`	`a search() operation on the catalogue`

THE DISCIPLINE Read each row as one idea traced across all three views. If a class, use case, or action has no row, ask why it exists; if a description phrase has no entry, ask what you missed. That round trip is what makes the models *trustworthy* rather than decorative, and it is exactly what an examiner checks.

Where class diagrams earn their place

Where they dominate

- All object-oriented code of any real size (Java, C#, Python, TypeScript).
- Domain modelling (Domain-Driven Design).
- ORM mapping (Hibernate, Entity Framework, Django).
- Design patterns, the Gang of Four communicate every pattern as a class diagram.

What they do not solve

- Runtime behaviour, use sequence or state diagrams.
- Data flow, use activity diagrams.
- Concurrency and deployment topology.

A class diagram tells you what exists; nothing about what happens.

Tooling today

- IDEs reverse-engineer them from code.
- PlantUML and Mermaid render them from text in Git workflows.
- Visual Paradigm, Lucidchart, Enterprise Architect do forward and reverse engineering.

Common pitfalls

- **Attributes drawn as classes.** "Title", "ISBN", "status" are attributes of a class, not classes of their own.
- **The omnipotent class.** A "SystemManager" that does everything signals analysis stopped too early; split it by responsibility.
- **Generalisation where association is meant.** Use `is-a` only for genuine specialisation; "has-a" and "part-of" are association and aggregation.
- **Missing multiplicity.** An association end with no multiplicity hides a business rule; always state both ends.
- **Aggregation everywhere.** If lifetime is not tied, it is a plain association; if it is, prefer composition.
- **Mixing analysis and design.** Full method signatures, types and infrastructure classes belong to the design-level diagram, not the first cut.

How to build one, and what comes next

Recipe

1. From the requirements, the nouns and roles become **candidate classes**; discard the inappropriate ones.
2. Attach the data each holds as **attributes**; the verbs it owns as **operations**.
3. Draw the **associations** from the noun-verb pairings; add **multiplicity** at both ends.
4. Add **generalisation** for `is-a`, **aggregation / composition** for `part-of`, **association classes** for relationship attributes.
5. Check each class for cohesion and coupling; validate with an **object diagram**.

Bridge to sequence diagrams

The class diagram tells you what exists; it says nothing about what *happens*. A call to `borrow()` on `LibraryMember` may touch `Book`, `BookCopy`, `Loan` and a persistence store in a specific order, with checks and error returns. That story, messages over time between a fixed cast of objects, is the **sequence diagram**, next.

Sources: A. Taweel, COMP433 Ch.4 lecture notes; Sommerville, *Software Engineering* 10th ed.; Booch et al.; Shlaer & Mellor. Companion: `COMP433_Ch4_UML_System_Modelling_Companion.html`.