UML: Sequence Diagrams

From the static structure to the system in motion: who sends which message to whom, and in what order, for one scenario of one use case.

Move with the arrow keys or the buttons below.

Agenda

Orientation

1. From structure to interaction: where sequence diagrams fit
2. A sequence diagram *at a glance*
3. What they capture, and when to draw one
4. Where participants and messages come from

Notation

5. Lifelines, activations, and the message kinds
6. One scenario, or all of them (instance vs generic)
7. Combined fragments: opt, alt, loop, par, ref

Worked examples

8. ATM “Deposit Cash”, step by step (interactive)
9. **Back to back**: the Library “Borrow a copy” scenario

Putting it to work

10. How to build one; sanity checks
11. Pitfalls, and where sequence diagrams earn their place
12. The four views together

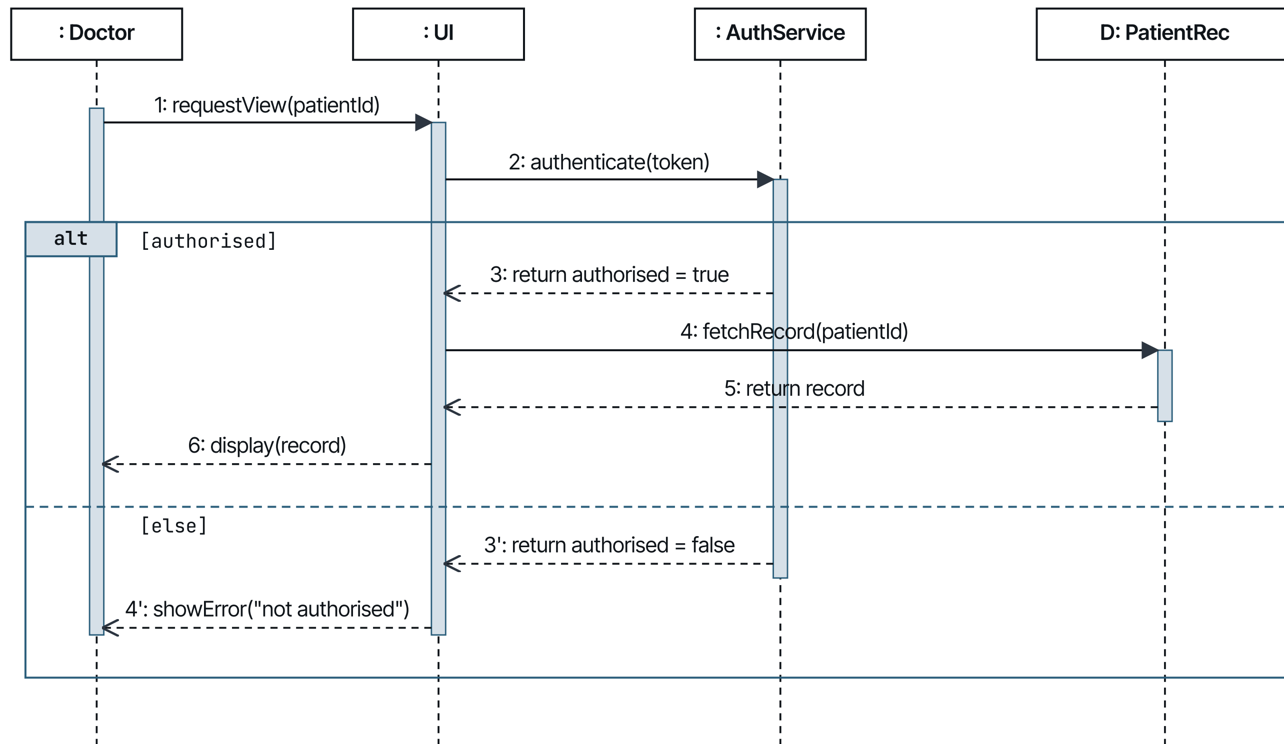
Where we are: from structure to interaction

The class diagram told us *what exists*: the cast of objects, their data and operations. A **sequence diagram** shows *what happens*: those objects exchanging messages, in time order, to carry out one scenario.

THE PAIRING Class and sequence are two views of the same objects. The class diagram is the system *at rest*; the sequence diagram is the same cast *in motion* for a single run. The lifelines you draw here are instances of the classes you drew there, and the messages are their operations.

A sequence diagram at a glance

Before the details, a complete sequence diagram for one scenario: a doctor viewing a patient record, where access depends on authorisation. Do not decode every mark yet, the rest of the deck explains each labelled part.



WHAT YOU ARE LOOKING AT

Four **participants** across the top, each with a dashed **lifeline** and a grey **activation bar** when busy; **messages** as arrows (solid = a call, dashed = a return); time running **top to bottom**; and an **alt fragment** splitting the flow on a guard, `[authorised]` versus `[else]`. Each part gets its own slide next.

What a sequence diagram is, and when to draw it

A sequence diagram models the **interactions between the actors and objects** of a system as a **time-oriented** view: it shows the order of messages exchanged during *one particular use case, or one instance of it*. Participants are listed across the top with a dashed line dropping from each; interactions are annotated arrows; the diagram is read **left to right and top to bottom**.

What it captures

- The **order** of messages for one scenario.
- Which participant **sends** and which **receives** each message, and the arguments.
- When each participant is **active** (the activation bars).
- Conditions and repetition, via fragments.

What it does not capture

- The system's **structure** (use a class diagram).
- Every possible scenario at once, one diagram is one story.
- An object's full **lifecycle** across all interactions (use a state machine).

WHEN TO DRAW ONE

Draw a sequence diagram for a scenario whose **call order is not obvious**, or where the **collaboration between several objects** needs pinning down: a checkout, an authorisation, a booking. It is how you catch a forgotten return, a missing error path, or a message sent to an object that has no such operation, before any code exists.

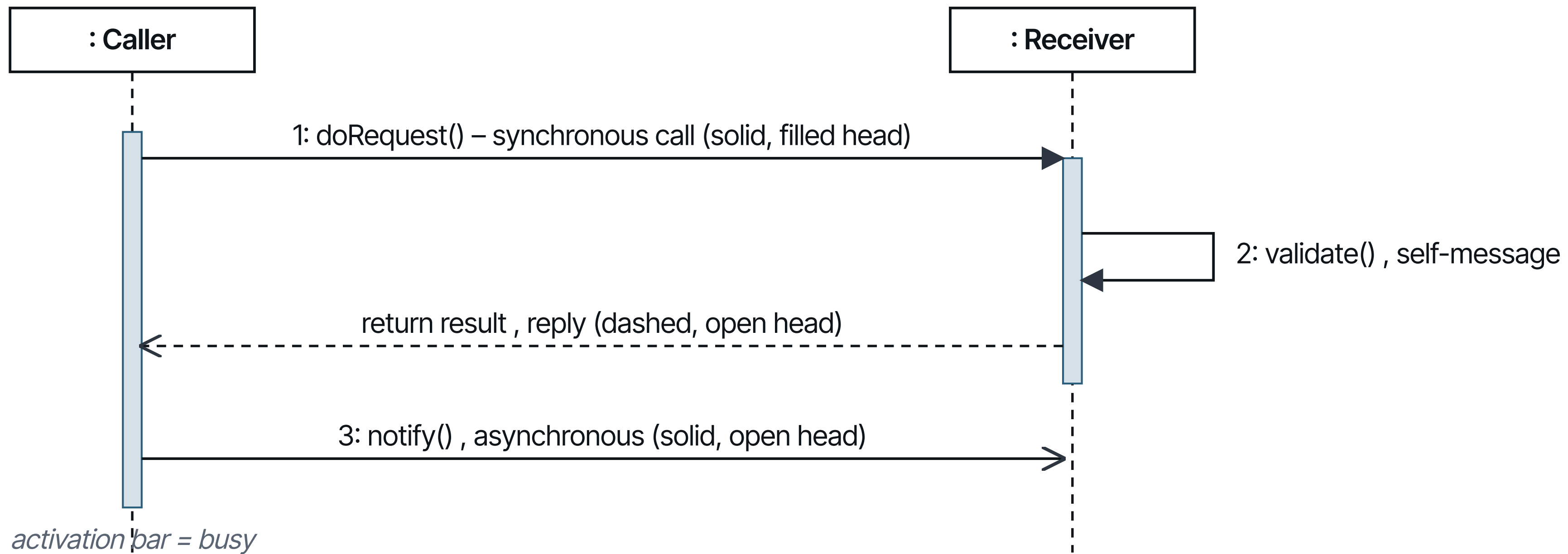
Where the participants and messages come from

A sequence diagram does not invent anything. It *realises* one scenario of a use case, using the classes from the class diagram, calling their operations.

IN THE SEQUENCE DIAGRAM	COMES FROM
The **scenario** being drawn	one path through a **use case** (its normal flow, or one alternative)
The **participants** (lifelines)	**actors** and **objects** , the objects are instances of **classes** from the class diagram
The **messages**	the **operations** declared on those classes
The **fragments** (alt, loop)	the **decisions and loops** of the activity diagram for that use case

CONSISTENCY CHECK If a lifeline receives a message `fetchRecord()` , the class it instantiates *must declare* `fetchRecord()` . The sequence diagram is therefore a powerful check on the class diagram: a message with no matching operation means the class model is incomplete.

Notation: lifelines, activations, messages



Lifeline & activation

The box names the participant (`: Class` for an anonymous object, `name : Class` for a named one). The dashed **lifeline** drops from it; the thin **activation bar** marks the stretch of time it is busy handling a message.

Message kinds

Synchronous (filled head): the caller waits for the reply. **Reply / return** (dashed, open head): the answer coming back. **Asynchronous** (open head, solid line): fire and continue, no waiting. A message back to the same lifeline is a **self-message**.

One scenario, or all of them

Adel draws the distinction clearly. A plain sequence diagram is an **instance**: one possible run, with no branching, the "happy path". But a use case has many scenarios, so we often need to show *conditional* behaviour and *iteration* too.

Instance diagram

One concrete sequence of messages, start to finish, no choices. Quick to draw, easy to read, ideal for explaining the normal flow (like the ATM deposit on the next slide).

Generic diagram

Shows the family of runs that can occur, using **combined fragments** (alt, opt, loop) to fold the alternatives and repetitions into one picture, like the authorisation `alt` in the opening diagram.

RULE OF THUMB Draw the **instance** first to nail the normal flow, then promote it to a **generic** diagram only where real branching or looping matters. Do not cram every error path into one diagram, that is what fragments, or separate diagrams, are for.

Combined fragments (UML 2.0)

A **combined fragment** is a boxed region of the diagram with an operator label in the top-left corner. It is how a generic diagram expresses choice and repetition.

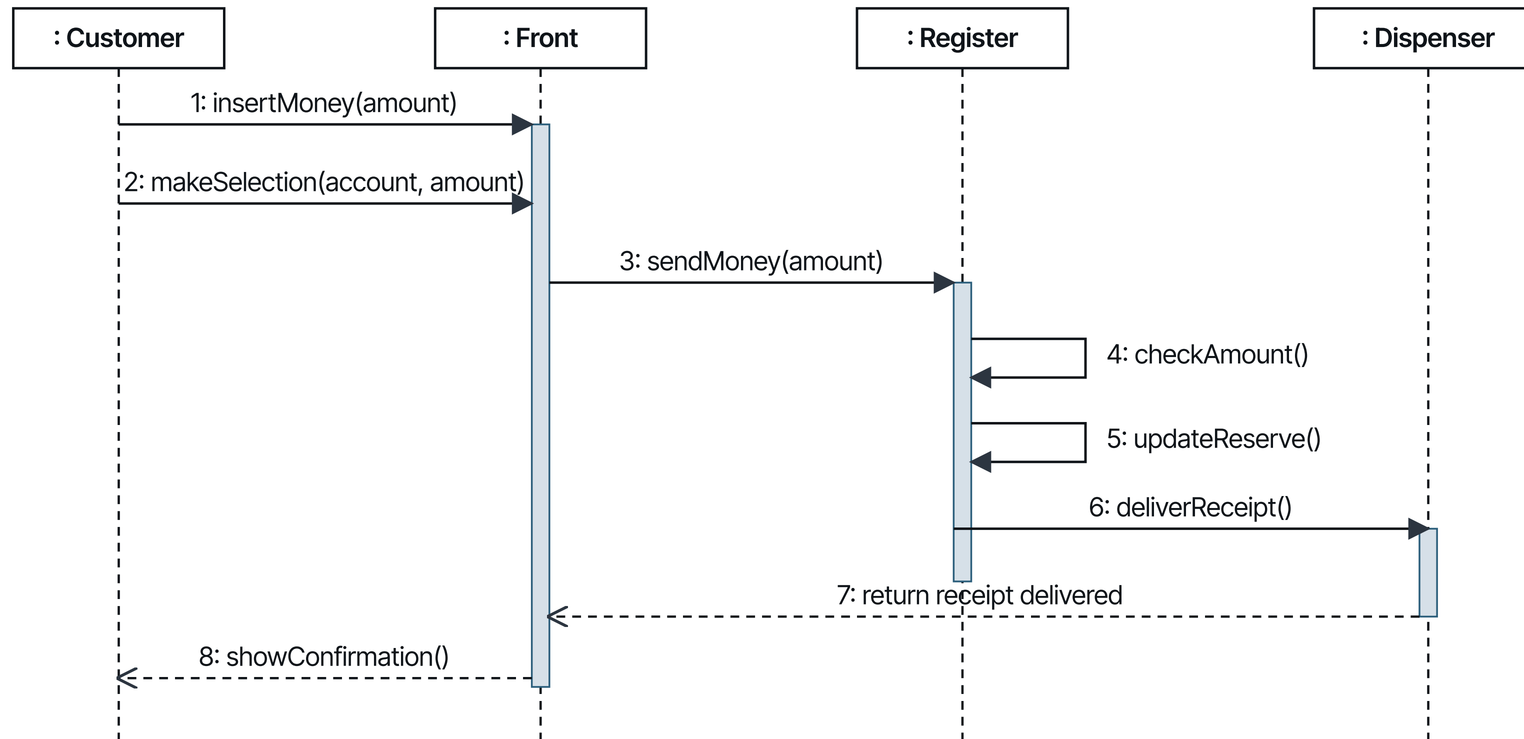
OPERATOR	MEANING
`opt`	**Optional.** The fragment runs only if its guard is true. Equivalent to an `alt` with a single branch.
`alt`	**Alternatives.** Two or more branches separated by dashed lines, each with a guard; exactly one runs per pass.
`loop`	**Loop.** The fragment may run zero or more times; the guard is the iteration condition. `loop(n)` for exactly n times.
`par`	**Parallel.** Two or more branches run concurrently; the order between them is unspecified.
`ref`	**Reference.** Stands for another sequence diagram, so a shared sub-interaction is not redrawn each time.

SEE IT IN THE OPENING DIAGRAM

The "at a glance" diagram uses an `alt`: the guard `[authorised]` runs the fetch-and-display branch, and `[else]` runs the error branch. Exactly one of the two executes. An `opt` would be the same box with a single guarded branch and no `[else]`.

Worked example: the “Deposit Cash” use case

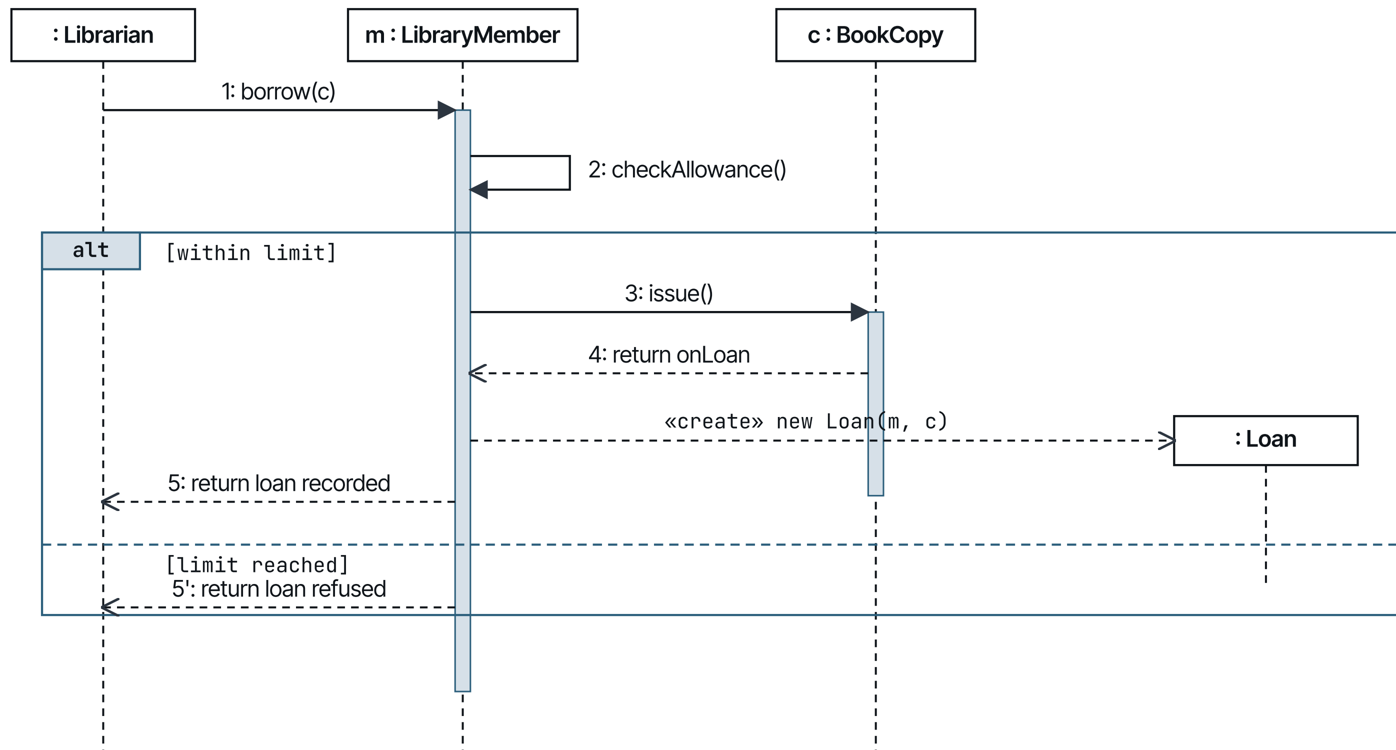
Adel's self-service-machine example, the normal scenario. Three internal objects do the work: the **front** (the customer interface), the **register** (where money is counted and the cash reserve kept), and the **dispenser** (which delivers the receipt).



WALKTHROUGH The normal deposit scenario. On screen, use **Next** and **Prev** to step through it message by message; watch each activation bar appear as its participant becomes busy.

Back to back: the Library “Borrow a copy” scenario

The same Library we modelled with a use case, an activity diagram and a class diagram, now in motion for one scenario. The lifelines are objects of the classes `LibraryMember`, `BookCopy` and `Loan`; the messages are their operations; the `alt` is the activity's “limit reached?” decision.



How to build a sequence diagram

Recipe

1. Pick **one scenario** of one use case (start with the normal flow).
2. Place the **participants** across the top: the actor that starts it, then the objects it collaborates with (instances of your classes).
3. Drop a **lifeline** from each.
4. Add **messages** in time order, top to bottom; name each with the **operation** it calls and its arguments.
5. Draw **activation bars** for the stretch each participant is busy, and the **returns** (dashed) where a caller gets its answer.
6. Wrap conditional or repeating parts in **fragments** (`alt`, `opt`, `loop`).

Sanity checks

- Every received message is an **operation that exists** on the receiver's class.
- Every synchronous call has a matching **return**.
- Time only goes **down**: no arrow points back up the page.
- One diagram is **one scenario**; branch with fragments, not by tangling two stories together.

Common pitfalls

- **All scenarios in one instance diagram.** An instance shows one run; use `alt` / `opt` / `loop` (a generic diagram) or separate diagrams for the rest.
- **Forgotten returns.** A synchronous call that never returns hides where control resumes; draw the dashed reply.
- **Messages with no home.** Sending `fetchRecord()` to a lifeline whose class has no such operation, fix the class diagram or the message.
- **Time running upward.** Messages must descend; an arrow back up the page is a modelling error, loops belong in a `loop` fragment.
- **Missing or wrong activations.** The bar should cover exactly the time the participant is processing, no more.
- **Using a sequence diagram for structure.** It shows one interaction over time, not what exists; that is the class diagram's job.

Where sequence diagrams earn their place

Where they dominate

- Designing the **collaboration** between objects or services for a tricky scenario.
- **API and protocol** design: request / response / callback order.
- **Distributed systems**: who calls whom, sync vs async, timeouts.
- Explaining a flow in **code review** or onboarding.

What they do not solve

- The static structure (class diagram).
- A single object's full lifecycle (state machine).
- The branching of one task at the work level (activity diagram).

Tooling today

- PlantUML and Mermaid render them from text in Git.
- IDEs and profilers generate call-sequence traces from running code.
- Visual Paradigm, Enterprise Architect, Lucidchart for full modelling.

Recap, and where this sits

Hold on to

- A sequence diagram is **one scenario, in time order**: participants across the top, time down, messages as arrows.
- **Solid** arrow = call, **dashed** = return, **open head** = asynchronous; activation bars show busy time.
- **Fragments** (alt, opt, loop, par, ref) turn an instance into a generic diagram.
- Lifelines are **objects of your classes**; messages are their **operations** , a live check on the class diagram.

The four views together

Use case (who and what) , activity (how one task flows) , class (what exists) , **sequence** (what happens between objects, over time). Each answers a different question about the same system; together they are the working core of UML for analysis and design.

Sources: A. Taweel, COMP433 Chapter 4 (UML) lecture notes (ATM “Deposit Cash” and MHC-PMS examples); Sommerville, *Software Engineering* 10th ed.; UML 2.x specification (combined fragments). Companion: `COMP433_Ch4_UML_System_Modelling_Companion.html`, section 11.