Unified Modeling Language (UML)

Models help engineers think and communicate. This chapter surveys UML and practises the core diagrams on ITC Club Hub; the Software Engineering course later covers each diagram type in depth.

UML 2.5.1 (OMG, 2017) | Mermaid · PlantUML · draw.io | Diagram to C++20 mapping | Case study: ITC Club Hub

AGENDA

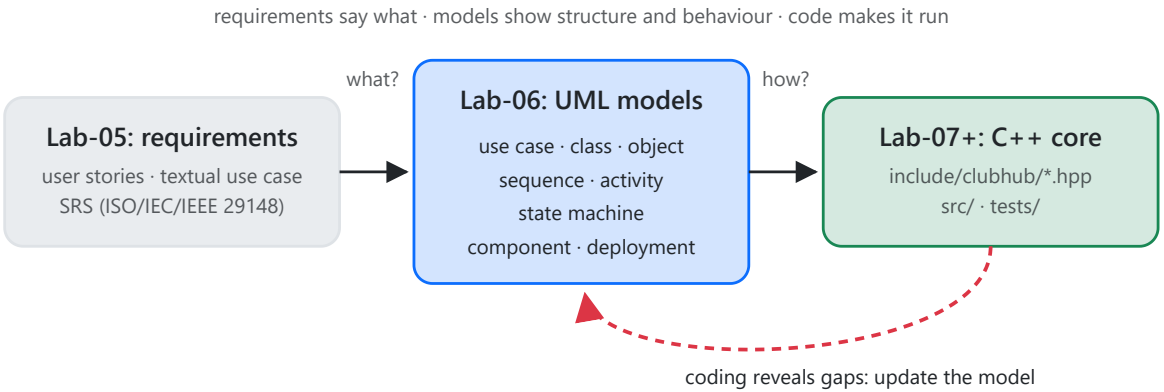
What we will cover

1. Why model: abstraction, communication, documentation and their limits
2. UML overview: structure diagrams and behaviour diagrams
3. Use case diagrams: actors, use cases, system boundary, include and extend
4. Class diagrams: classes, attributes, operations, associations, multiplicity, generalization, and how they map to C++ classes
5. Object diagrams as snapshots of a class diagram
6. Sequence diagrams: lifelines, messages, activations, fragments
7. Activity diagrams: actions, decisions, forks and joins, swimlanes
8. State machine diagrams: states, events, transitions
9. Component and deployment diagrams at overview level
10. Modelling tools (draw.io, PlantUML, Mermaid) and diagram quality guidelines
11. Chapter Quiz
12. Lab-06

Click any item to jump directly to that topic.

From requirements to models to code

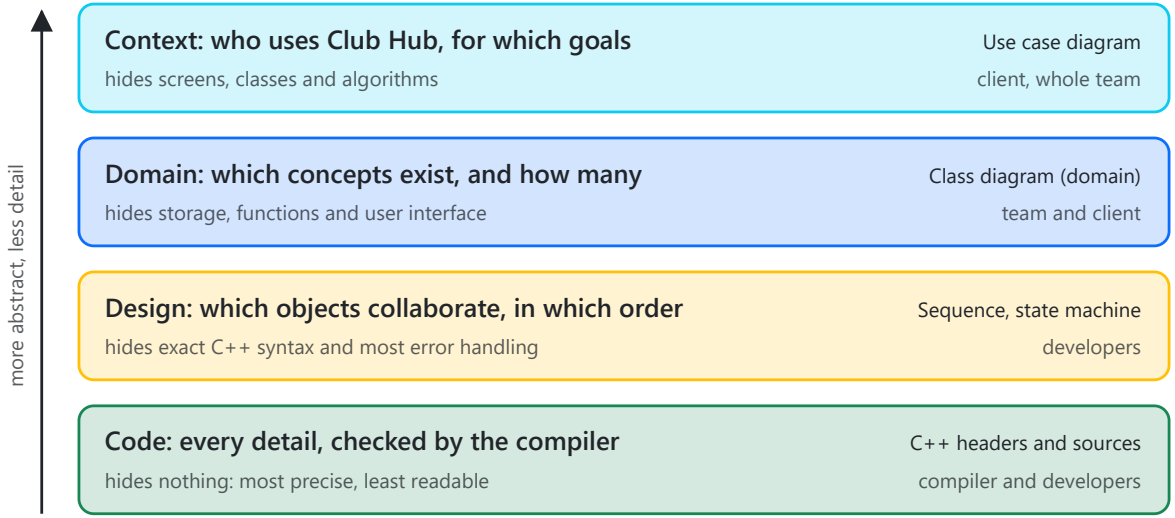
- Lab-05 gave your team **user stories**, a **textual use case** and a short **SRS** for ITC Club Hub: words that say *what* the system must do.
- Before writing C++ in Sprint 1, we draw **models**: pictures that show the system's structure (which classes, how many) and behaviour (who calls whom, in which order, which states).
- **UML**, the Unified Modeling Language, is the standard notation for these pictures, maintained by the Object Management Group (OMG). Current version: **UML 2.5.1 (2017)**.
- "Unified" because it merged the methods of Booch, Rumbaugh and Jacobson in the late 1990s; OMG adopted it in 1997.
- This is a **survey**: each diagram type with its core notation and one Club Hub example. The Software Engineering course covers each diagram type in depth.



A model is a deliberate simplification

- **Abstraction**: leaving out detail on purpose so that one question can be answered. A metro map hides streets so you can plan a route.
- **Communication**: a diagram on the whiteboard lets the Product Owner, the client and the Developers agree before code exists.
- **Documentation**: a new team member reads the class diagram in ten minutes instead of the code in two days.
- **Analysis**: drawing forces decisions ("can a registration exist without an event?") and exposes gaps early, when a change costs a pen stroke.
- Every model has a *purpose* and an *audience*; choose the abstraction level for them.

SWEBOK Guide v4.0, Software Engineering Models and Methods; UML 2.5.1.



Worked example: one rule in words, in UML and in C++

In words (Lab-05, BR-02): "A student cannot register twice for the same event."

In the class diagram: `Student "1" -- "0..*" Registration` allows many registrations per student, so multiplicity cannot express the rule. UML needs a *constraint* in braces, attached as a note: `{a student has at most one active Registration per Event}`.

In C++: only code actually enforces it.

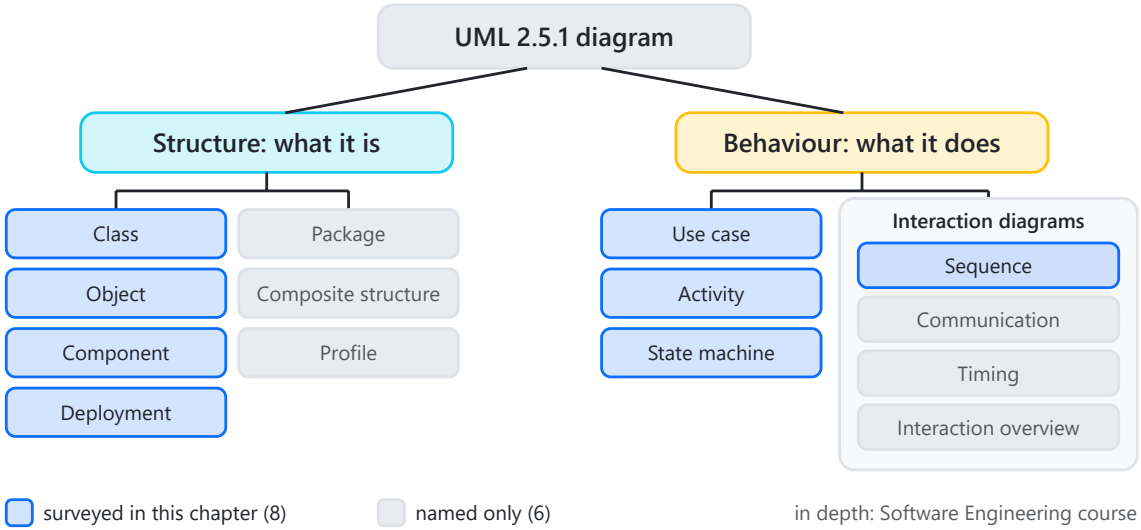
```cpp
bool alreadyRegistered(const std::vector<Registration>& regs,
                       const std::string& studentId, int eventId)
{
    return std::ranges::any_of(regs, [&](const Registration& r) {
        return r.studentId() == studentId
            && r.eventId() == eventId
            && r.status() != RegistrationStatus::Cancelled;
    });
}
```

Mode	What it means	Where
Sketch	Quick, partial, for a conversation; thrown away or kept as a photo	Most teams, this course
Blueprint	Complete and precise enough for another team to code from	Contracts, safety-critical work
Programming language	The model is executed or generates code (model-driven engineering)	Specialised tools, embedded systems

Modes after Fowler, *UML Distilled* (3rd ed., 2003).

⚠ **Limits of models.** They do not run, so nothing checks them; they go stale when code changes; they cannot show every rule (see BR-02); and drawing costs time. Model enough to answer a question, keep the source next to the code, and delete diagrams nobody maintains.

Fourteen diagram types in two families



- **Structure diagrams** show the static parts of a system: classes, objects, components, nodes. They are true at any moment.
- **Behaviour diagrams** show what happens over time: goals, flows, messages, state changes.
- **Interaction diagrams** are a sub-family of behaviour diagrams focused on messages between participants. The sequence diagram is by far the most used.
- You rarely need all 14. Most teams use 4 to 6: use case, class, sequence, activity, state machine and a deployment or component overview.

Source: UML 2.5.1 (OMG, 2017), Annex A. A *diagram* is only a view: the same class can appear in many diagrams of one *model*.

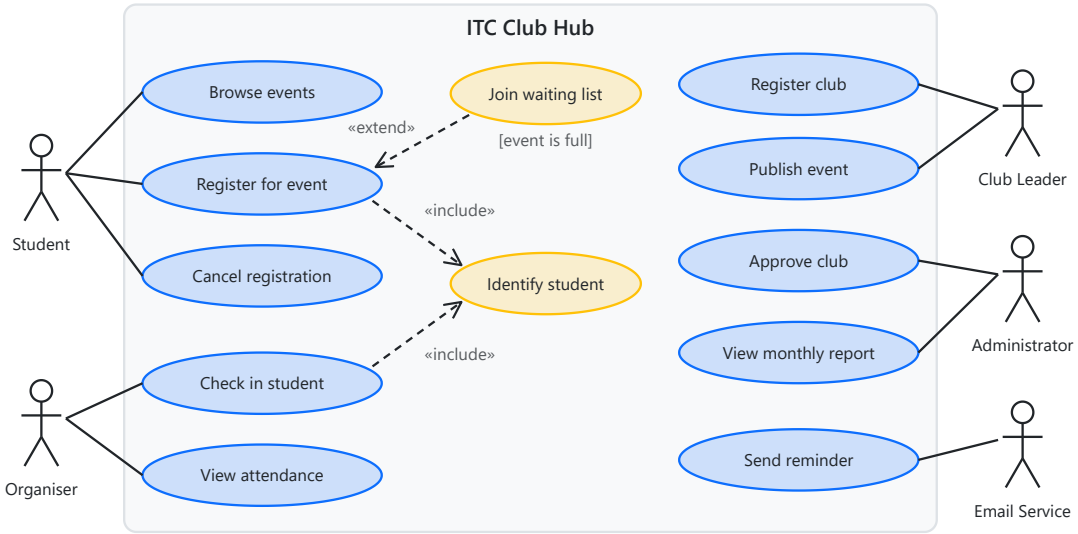
Shared notation: the vocabulary of lines and arrowheads

notation (A left, B right)	name	reads as	typical C++
A —— B	Association	A is linked to B	`int bId_; or B&`
A ——> B	Directed association	A knows B, not the reverse	`member in A only`
A ——▷ B	Generalization	A is a kind of B	`class A : public B`
A - - -▷ B	Realization	A implements interface B	`override pure virtuals`
A - - -> B	Dependency	A uses B briefly	`B as parameter`
A ◇—— B	Aggregation	A has B; B lives on alone	`non-owning id or ptr`
A ◆—— B	Composition	A owns B; B dies with A	`std::vector<B> member`

- The **arrowhead shape** carries the meaning: open arrow, hollow triangle and diamond are three different things.
- Solid line = structural link; **dashed** line = weaker (implements, uses, includes).
- Arrows point from the element that *depends* or *specialises* to the one it depends on: child → parent, user → used.
- **Stereotypes** in guillemets add meaning to any element: `«interface»`, `«include»`, `«enumeration»`.
- A **note** (a box with a folded corner) adds free text or a `{constraint}`.

⚠ Reversed generalization arrows are the most common mistake in student diagrams. The triangle always touches the *parent*.

Who uses Club Hub, for which goals



- **Actor**: a role outside the system (person or system), never a person's name. **Use case**: a goal, named verb + object. **System boundary**: the rectangle; actors stay outside.
- Email Service is a **secondary actor**: a system that Club Hub uses to deliver the reminder.

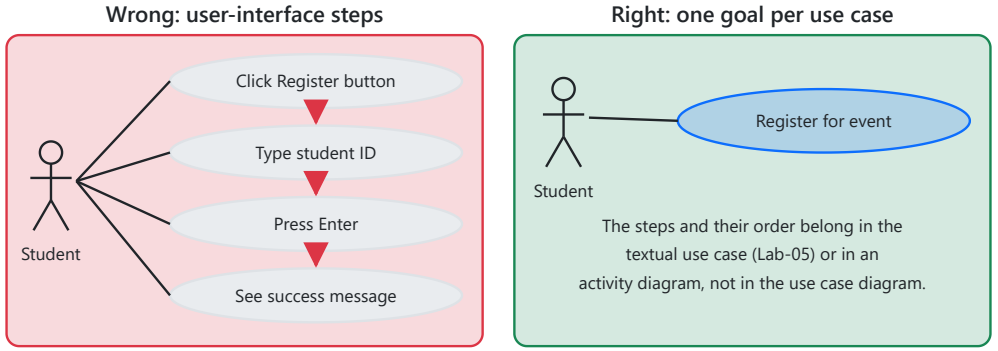
```
@startuml
left to right direction
actor Student
actor Organiser
actor "Club Leader" as Leader
actor Administrator as Admin
actor "Email Service" as Mail
rectangle "ITC Club Hub" {
  usecase "Browse events" as UC1
  usecase "Register for event" as UC2
  usecase "Cancel registration" as UC3
  usecase "Check in student" as UC4
  usecase "View attendance" as UC5
  usecase "Identify student" as UC6
  usecase "Join waiting list" as UC7
  usecase "Register club" as UC8
  usecase "Publish event" as UC9
  usecase "Approve club" as UC10
  usecase "View monthly report" as UC11
  usecase "Send reminder" as UC12
}
Student -- UC1
Student -- UC2
Student -- UC3
Organiser -- UC4
Organiser -- UC5
UC8 -- Leader
UC9 -- Leader
UC10 -- Admin
UC11 -- Admin
UC12 -- Mail
UC2 ..> UC6 : <<include>>
UC4 ..> UC6 : <<include>>
UC7 ..> UC2 : <<extend>>
@enduml
```

«include», «extend», and what does not belong in the diagram

	«include»	«extend»
Meaning	The base *always* performs the included steps	The extension *sometimes* adds steps, under a condition
Arrow	base → included Register for event → Identify student	extension → base Join waiting list → Register for event
Use it for	Steps shared by two or more use cases	Optional or exceptional behaviour, often a Should feature
C++ analogy	A helper function every caller calls	An `if` branch that runs only when the condition holds

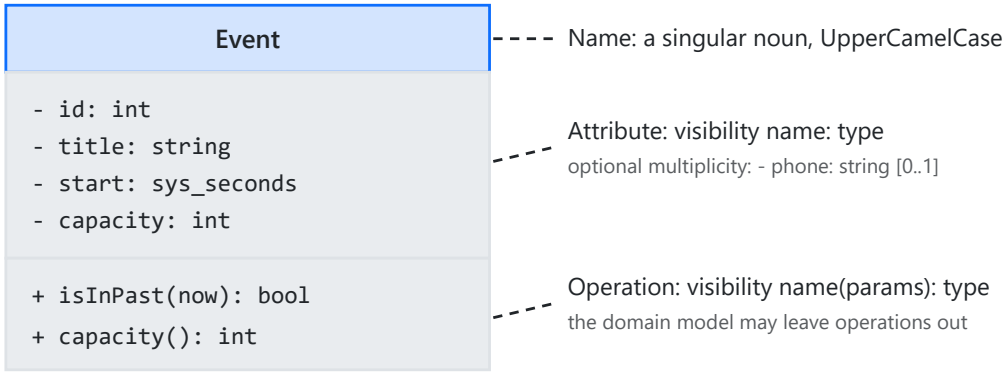
- Each ellipse corresponds to one **textual use case** from Lab-05 (main flow and alternative flows). The diagram is the table of contents; the text holds the steps.
- Use «include» and «extend» sparingly: one or two per diagram. If unsure, leave them out.

UML 2.5.1, clause 18 (Use Cases).



⊗ **Common mistake:** a use case diagram shows *who* and *what*, never *how* or *in which order*. Arrows between use cases are not a flow chart.

Anatomy of a class, and how to read multiplicity

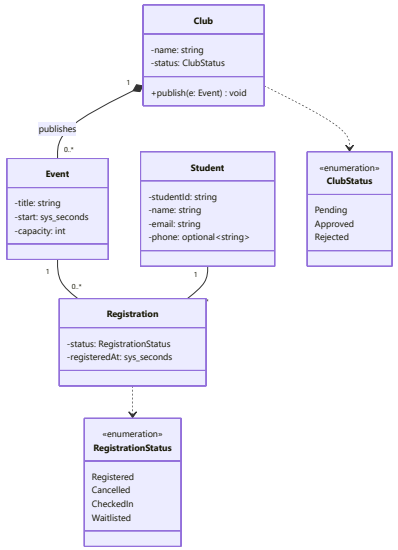


Visibility: + public · - private · # protected · ~ package

Multiplicity	Reads as	C++ member
`1`	exactly one	`T` or its id
`0..1`	zero or one (optional)	`std::optional<T>`
`0..*` or `*`	any number, possibly none	`std::vector<T>`
`1..*`	at least one	`std::vector<T>` + a check
`2..5`	between two and five	container + a check

- Multiplicity is written at the **far end** of the association: `Club "1" -- "0..*" Event` reads "one club publishes zero or more events; each event belongs to exactly one club".
- A **domain** class diagram (analysis) shows concepts and attributes; a **design** class diagram adds operations, types and interfaces.

The Club Hub domain model in Mermaid



Some attributes (location, description, ids) are omitted to fit the slide; Lab-06 Task 2 asks for all of them.

```
classDiagram
  class Club {
    -name: string
    -status: ClubStatus
    +publish(e: Event) void
  }
  class Event {
    -title: string
    -start: sys_seconds
    -capacity: int
  }
  class Student {
    -studentId: string
    -name: string
    -email: string
    -phone: optional~string~
  }
  class Registration {
    -status: RegistrationStatus
    -registeredAt: sys_seconds
  }
  class ClubStatus {
    <<enumeration>>
    Pending
    Approved
    Rejected
  }
  class RegistrationStatus {
    <<enumeration>>
    Registered
    Cancelled
    CheckedIn
    Waitlisted
  }
  Club "1" *-- "0..*" Event : publishes
  Event "1" -- "0..*" Registration
  Student "1" -- "0..*" Registration
  Club ..> ClubStatus
  Registration ..> RegistrationStatus
```

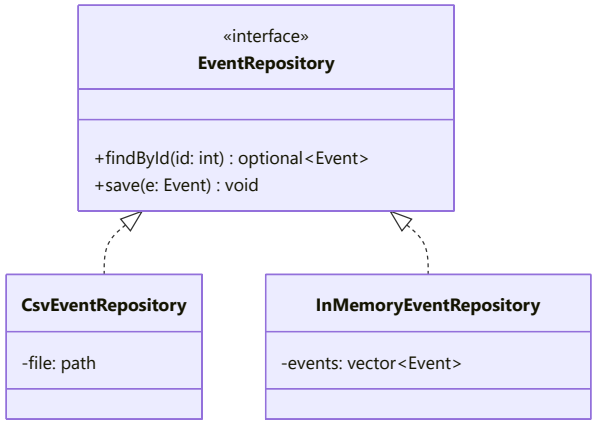
From the class diagram to C++ headers

UML element	C++20
Class	`class` in its own header in `include/clubhub/`
`-` attribute / `+` operation	`private:` data member / `public:` member function
Enumeration	`enum class`
Composition `*--` with `0..*`	member container: `std::vector<Event> events_;`
Association to a class stored elsewhere	the other object's **id** (`int eventId_`); a reference or pointer only when the other object surely outlives this one
Multiplicity `0..1`	`std::optional<T>`
Generalization / realization	`public` inheritance, `virtual` + `override`

Why ids? Registrations and events are loaded from different CSV files. A `const Event&` inside `Registration` would dangle as soon as the event list is reloaded; an id never does.

```cpp
// include/clubhub/club.hpp
#pragma once
#include <string>
#include <vector>
#include "clubhub/event.hpp"

namespace clubhub {

// «enumeration» ClubStatus
enum class ClubStatus { Pending, Approved, Rejected };

class Club {
public:
    explicit Club(std::string name) : name_{std::move(name)} {}

    void approve() { status_ = ClubStatus::Approved; }
    // only an approved club may publish (business rule)
    void publish(Event e);

    const std::string& name() const { return name_; }
    ClubStatus status() const { return status_; }
    const std::vector<Event>& events() const { return events_; }

private:
    std::string name_;
    ClubStatus status_ = ClubStatus::Pending;
    // composition "1" *-- "0..*": the club owns its events
    std::vector<Event> events_;
};

} // namespace clubhub
```

Generalization in practice, and a reading exercise



Exercise: read `Event "1" -- "0..*" Registration`

1. Can an event have no registrations yet?
2. Can a registration exist without an event?
3. Does `0..*` say that registrations stop at the capacity?

Answers: 1. Yes, the lower bound is 0. 2. No, each registration has exactly one event. 3. No: `*` is unbounded. The capacity limit is a business rule: a `{constraint}` note in UML, an `if` in `registerStudent`.

```cpp
// include/clubhub/event_repository.hpp
// «interface» -> abstract class with pure virtual functions
class EventRepository {
public:
    virtual ~EventRepository() = default;
    virtual std::optional<Event> findById(int id) const = 0;
    virtual void save(const Event& e) = 0;
};

// realization (dashed line, hollow triangle) -> public inheritance
class CsvEventRepository : public EventRepository {
public:
    explicit CsvEventRepository(std::filesystem::path file)
        : file_{std::move(file)} {}
    std::optional<Event> findById(int id) const override;
    void save(const Event& e) override;
private:
    std::filesystem::path file_;
};
```

- **Generalization** (solid line, hollow triangle): "is a kind of". **Realization** (dashed): "implements this interface". In C++ both become `public` inheritance.
- `InMemoryEventRepository` lets unit tests run without files (Chapter 09). Prefer small interfaces over deep inheritance trees.

Worked example: `registration.hpp`, compiled in your head line by line

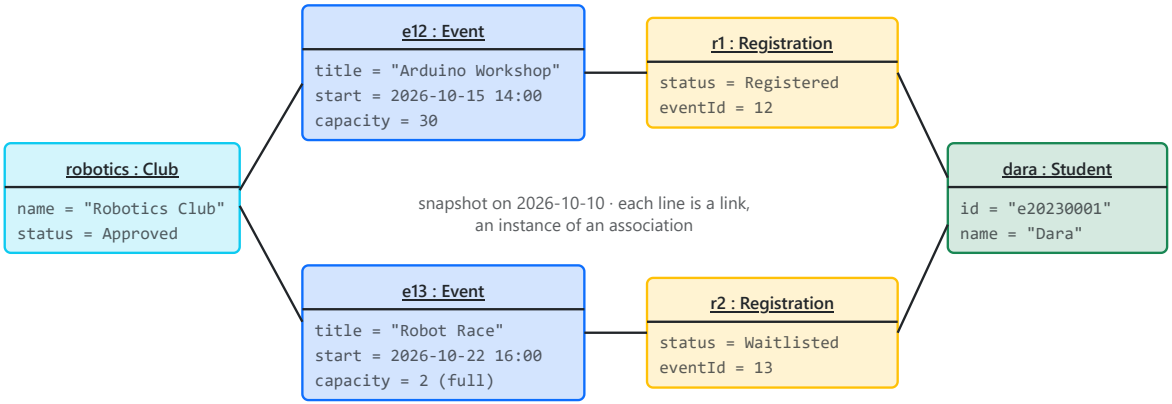
```cpp
// (1) one class, one header
#pragma once
#include <chrono>
#include <string>

namespace clubhub {

// (2) «enumeration» -> enum class
enum class RegistrationStatus { Registered, Cancelled, CheckedIn, Waitlisted };

class Registration {
public:
    // (3) every "1" end must be supplied when the object is created
    Registration(int id, int eventId, std::string studentId,
                 std::chrono::sys_seconds registeredAt);
    // (4) + operations -> public member functions
    void cancel(std::chrono::sys_seconds now,
                std::chrono::sys_seconds eventStart);
    void checkIn();
    int eventId() const { return eventId_; }
    const std::string& studentId() const { return studentId_; }
    RegistrationStatus status() const { return status_; }
private:
    // (5) - attributes -> private data members
    int id_;
    // (6) association to Event ("1" end) -> the event's id
    int eventId_;
    // (7) association to Student ("1" end) -> the student's id
    std::string studentId_;
    RegistrationStatus status_ = RegistrationStatus::Registered;
    std::chrono::sys_seconds registeredAt_;
};

} // namespace clubhub
```

#	What the compiler sees	Diagram element
1	`#pragma once`: the header is read once per translation unit	Class `Registration`
2	A scoped enum: `RegistrationStatus::Cancelled`, no implicit int conversion	`«enumeration»` box
3	No default constructor: you cannot create a registration without event and student	The two `"1"` ends
4	Declarations only; bodies go to `src/registration.cpp`	Operations compartment
5	Only member functions of the class can read or write these	`-` visibility
6, 7	Plain values: copying a `Registration` is safe	Associations

Where did `0..*` go? The "many" end (an event's registrations) is not a member here: the `RegistrationRepository` answers "all registrations of event 12". Navigability decides which side stores the link.

`Student` follows the same pattern; its `0..1` phone becomes `std::optional<std::string> phone_;`.

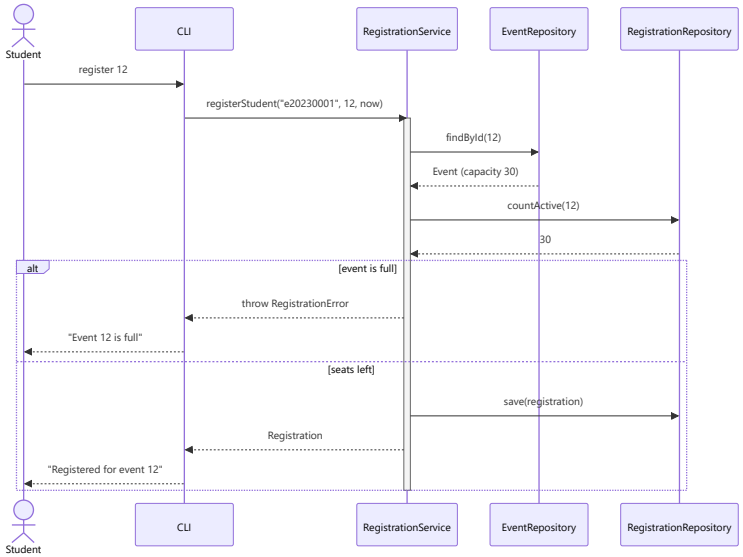
An object diagram is a snapshot: concrete objects at one moment



- Box title `name : Class`, **underlined**; attribute *values* instead of types; no multiplicities (a link joins exactly two objects).
- Use it to **test a class diagram** with a real example: if a situation from the requirements cannot be drawn, the class diagram is wrong.
- A snapshot is also a ready-made **test fixture**:

```
using namespace std::chrono;
Club robotics{"Robotics Club"};
robotics.approve();
robotics.publish(Event{12, "Arduino Workshop",
                       sys_days{2026y / October / 15} + 14h,
                       30});
Registration r1{1, 12, "e20230001",
                sys_days{2026y / October / 1} + 9h};
```

"Register for an event" as a sequence of messages



Element	Drawn as
Lifeline	A participant (object or actor) with a dashed line going down: time flows downwards
Synchronous message	Solid line, filled arrowhead: the caller waits
Return	Dashed line back to the caller
Activation	Thin bar on a lifeline: that object is executing
Fragment	A frame: `alt` (if/else), `opt` (if), `loop`, `par`

One sequence diagram = **one scenario** of one use case. Draw the main success path first, then add `alt` fragments for the alternative flows of the textual use case. UML 2.5.1, clause 17.

The same scenario in C++: every message is a call

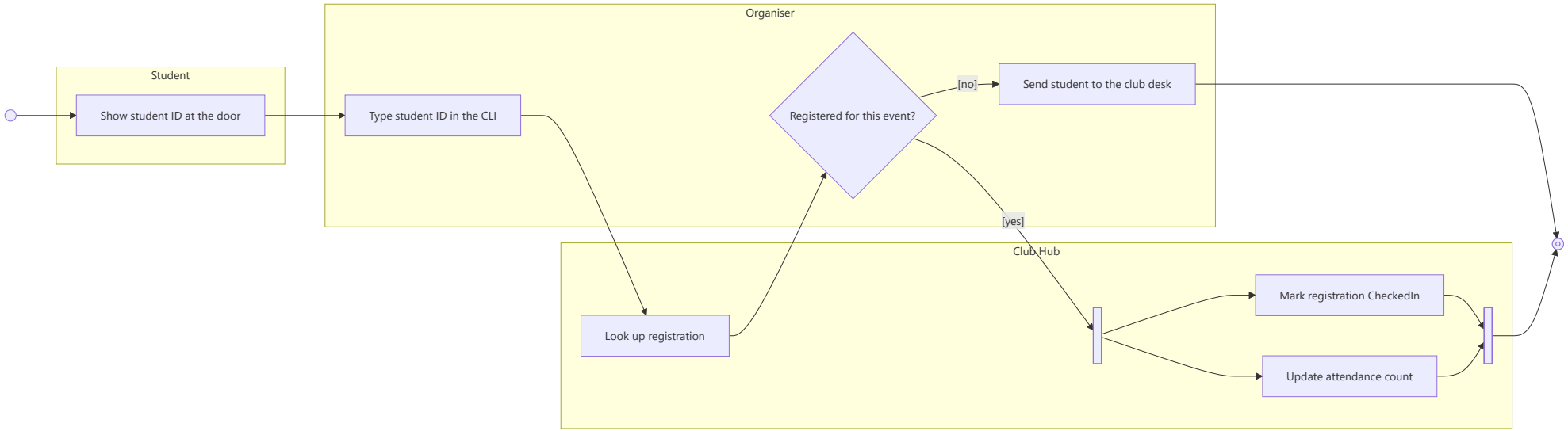
```cpp
// src/registration_service.cpp
Registration RegistrationService::registerStudent(
    const std::string& studentId, int eventId,
    std::chrono::sys_seconds now)
{
    // message findById(12) to the EventRepository lifeline
    const std::optional<Event> event = events_.findById(eventId);
    if (!event) {
        throw RegistrationError{"no such event"};
    }
    // message countActive(12) to the RegistrationRepository lifeline
    const int active = registrations_.countActive(eventId);

    // alt [event is full]
    if (active >= event->capacity()) {
        throw RegistrationError{"event is full"};
    }
    // else [seats left]: save(registration), then return it
    Registration r{registrations_.nextId(), eventId, studentId, now};
    registrations_.save(r);
    return r;
}
```

```cpp
// include/clubhub/registration_service.hpp
class RegistrationService {
public:
    RegistrationService(EventRepository& events,
                        RegistrationRepository& regs)
        : events_{events}, registrations_{regs} {}
    Registration registerStudent(
        const std::string& studentId, int eventId,
        std::chrono::sys_seconds now);
private:
    EventRepository& events_;
    RegistrationRepository& registrations_;
};
```

Diagram	Code
Lifeline `ER`, `RR`	Members `events_`, `registrations_`
Message + return	Function call + return value
`alt` fragment	`if` / `throw`
Activation bar	Duration of `registerStudent`

⚠ The "no such event", past-event and duplicate checks are not in the diagram. Lab-06 Task 4 adds the "already registered" branch so diagram and code agree.

Check-in at the door as an activity diagram

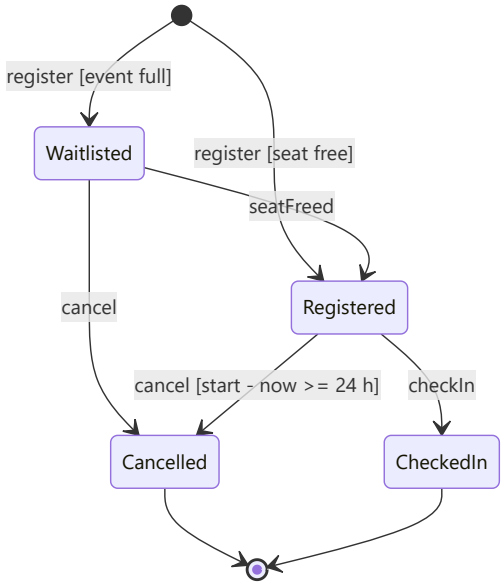


Element	Symbol	Meaning
Initial / final node	● / ◉	Where the flow starts and ends
Action	Rounded box	One step, verb first
Decision / merge	Diamond	Branch on a guard in `[brackets]`; guards must not overlap
Fork / join	Thick bar	Fork: flows run in parallel. Join: waits for *all* of them
Swimlane (partition)	Column or band	Who performs the actions

- Activity diagrams model **workflows**: a business process, a use case's steps, or an algorithm.
- Mermaid has no true swimlanes; labelled subgraphs are a common approximation. draw.io and PlantUML draw real partitions.

UML 2.5.1, clause 15 (Activities).

The life of one `Registration`



- **State**: a condition an object stays in for a while (rounded box, named with an adjective or past participle: *Registered*, not *Registering*).
- **Event (trigger)**: something that happens: `cancel`, `checkIn`, `seatFreed`.
- **Guard** in brackets: the transition fires only if it is true.
- **Transition**: arrow *event [guard] / action* from one state to the next. Any event without an arrow from the current state is **illegal**.

From \ event	seatFreed	cancel	checkIn
Waitlisted	Registered	Cancelled	illegal
Registered	illegal	Cancelled [≥ 24 h]	CheckedIn
Cancelled	illegal	illegal	illegal
CheckedIn	illegal	illegal	illegal

The state-transition table is the same information as the diagram, and it is the input for state-based testing in Chapter 09. UML 2.5.1, clause 14.

The state machine as a `switch` on an `enum class`

```cpp
// src/registration_rules.cpp
enum class Trigger { SeatFreed, Cancel, CheckIn };

// Returns the next state; throws for an arrow that is not in the diagram.
RegistrationStatus next(RegistrationStatus from, Trigger t)
{
    // C++20: write Registered instead of RegistrationStatus::Registered
    using enum RegistrationStatus;
    switch (from) {
    case Waitlisted:
        if (t == Trigger::SeatFreed) return Registered;
        if (t == Trigger::Cancel)    return Cancelled;
        break;
    case Registered:
        if (t == Trigger::Cancel)    return Cancelled;
        if (t == Trigger::CheckIn)   return CheckedIn;
        break;
    case Cancelled:
    case CheckedIn:
        break;   // final states: no arrow leaves them
    }
    throw std::logic_error{"illegal transition from "
                           + std::string{toString(from)}};
}
```

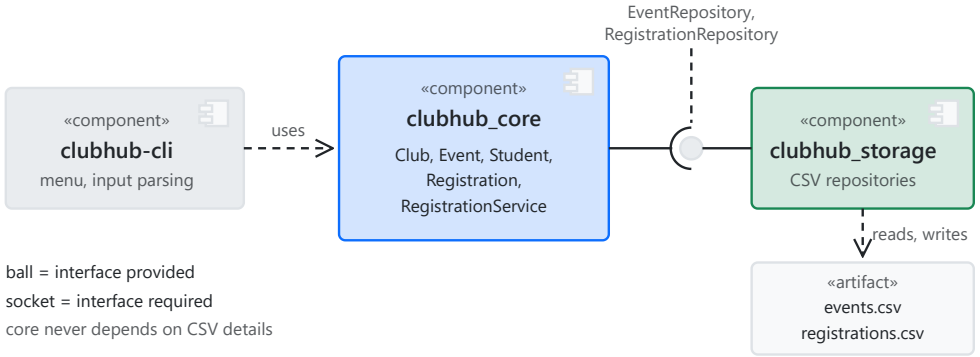
```cpp
// the guard [start - now >= 24 h] lives in the operation
void Registration::cancel(std::chrono::sys_seconds now,
                          std::chrono::sys_seconds eventStart)
{
    using namespace std::chrono_literals;
    if (eventStart - now < 24h) {
        throw RegistrationError{"cancellation closed"};
    }
    status_ = next(status_, Trigger::Cancel);
}
```

Diagram	C++
State	Enumerator of `RegistrationStatus`
Event	Enumerator of `Trigger` / member function
Transition	`return` of the new state
Guard	`if` before the transition
Missing arrow	`throw` (or `std::expected`, C++23, optional)

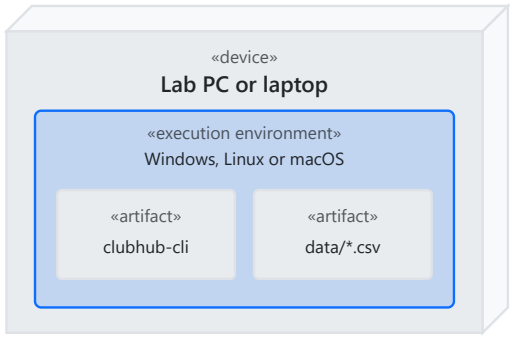
Compile with `-Wall`: GCC and Clang warn when a `switch` on an `enum class` forgets an enumerator, so a new state cannot be silently ignored.

Club Hub from above: its parts and where they run

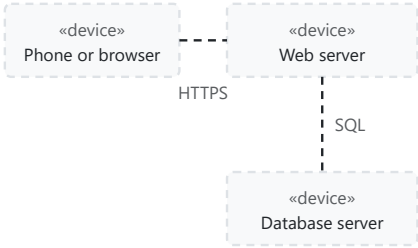
Component view



Deployment view: today



later: Software Engineering course



Component diagram: replaceable parts and their interfaces. In C++ a component is usually a CMake target (`add_library(clubhub_core ...)`, `add_executable(clubhub-cli ...)`).

Deployment diagram: *nodes* (devices, execution environments) and the *artifacts* (executables, files) deployed on them. Both are covered in depth in the Software Engineering course.

Three tools: draw.io, PlantUML, Mermaid

	draw.io (diagrams.net)	PlantUML	Mermaid
Style	Drag and drop editor	Text, rendered by a Java tool or server	Text, rendered in the browser
UML coverage	Everything (free drawing)	All main UML types incl. use case, component, deployment	Class, sequence, state, flowchart; no use case diagram
Layout	Full manual control	Automatic, hints only	Automatic, hints only
In Git	`.drawio` XML: hard to review	`.puml` text: clean diffs	`.mmd` or Markdown: clean diffs
On GitHub	Export PNG or SVG	Export PNG, or a plugin	Rendered directly in ```` ```mermaid ```` blocks
Best for	Posters, whiteboard-style sketches, deployment pictures	Use case and precise UML	Diagrams living next to code and README files

Lab-06 uses PlantUML for the use case diagram and Mermaid for the others. Any of the three is acceptable in the project if the source is committed.

Diagrams as code: the team workflow

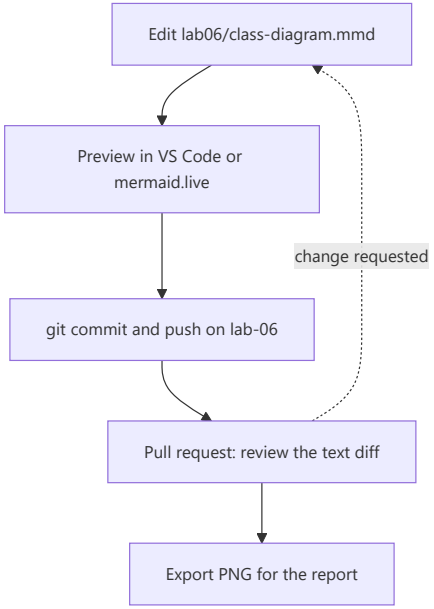


Diagram quality guidelines

1 One purpose, one audience

Give each diagram a title that states the question it answers: "Register for an event: main and full-event paths". If it answers two questions, split it.

2 Stay small

Aim for about 7 to 12 elements. A class diagram with 30 boxes nobody reads; draw one per area (clubs, registrations, reporting) instead.

3 Same names everywhere

Use the glossary and code names: `Registration`, `RegistrationStatus::CheckedIn`, `registerStudent`. No "Signup" in one diagram and "Booking" in another.

4 Correct, standard notation

Arrowheads, dashed lines and diamonds have fixed meanings (slide 7). Label multiplicities on both ends of every association. Put guards in `[brackets]`.

5 Readable layout

Read left to right or top to bottom; avoid crossing lines; align boxes; keep actors outside the boundary; do not rely on colour alone to carry meaning.

6 Versioned with the code

Commit the source (`.puml`, `.mmd`, `.drawio`) next to the exported PNG, with a version and date. When the code changes, update the diagram in the same pull request, or delete it.

Common mistakes in student UML, and the fix

⊗ Arrows reversed

Generalization triangle at the child, «include» pointing from the included use case to the base, «extend» from the base to the extension. **Fix:** triangle touches the parent; «include» base → included; «extend» extension → base.

⊗ UI steps as use cases

"Click Register button", "Enter password". **Fix:** one ellipse per actor goal ("Register for event"); steps go to the textual use case or an activity diagram.

⊗ Attribute and association twice

`Registration` with an attribute `event: Event` *and* a line to `Event`. **Fix:** show a link to another class once, as an association with multiplicities.

⊗ Activities named as states

States called "Registering", "Do check-in". **Fix:** states are conditions (*Registered*, *CheckedIn*); verbs belong to events and actions.

⊗ Only the happy path

A sequence diagram with no full-event branch; an activity diagram with no "not registered" decision. **Fix:** add the alternative flows from the Lab-05 use case as `alt` fragments or decisions.

⊗ Diagram and code disagree

The diagram says `Booking`, the code says `Registration`; an enum value exists in one only. **Fix:** keep a consistency table (Lab-06 Task 3) and check it in every pull request.

Chapter quiz 10 questions

1. Which of these is a *structure* diagram in UML 2.5.1?

- Sequence diagram
- Class diagram
- Activity diagram
- State machine diagram

2. In the Club Hub use case diagram, how is the «extend» arrow between "Join waiting list" and "Register for event" drawn?

- Solid line from Register for event to Join waiting list
- Dashed arrow from Register for event to Join waiting list
- Dashed arrow from Join waiting list to Register for event
- Hollow triangle pointing at Join waiting list

3. What does `Event "1" -- "0..*" Registration` say?

- Every event has at least one registration
- An event has zero or more registrations; each registration belongs to exactly one event
- A registration can belong to several events
- An event can have at most one registration

4. Applied: `class Club { std::vector<Event> events_; };` Which UML relationship does this member implement best?

- Generalization from Club to Event
- Dependency from Club to Event
- Composition: Club "1" *-- "0..*" Event
- Realization of an Event interface

5. Applied: the class diagram shows `-phone: string [0..1]` in `Student`. Which C++ member matches it?

- `std::string phone_;` set to "none" when missing
- `std::optional<std::string> phone_;`
- `std::vector<std::string> phones_;`
- `std::string* phone_ = new std::string;`

6. What is an object diagram?

- A class diagram without operations
- A snapshot of concrete objects and links at one moment, consistent with a class diagram
- A diagram of the objects a sequence diagram creates over time
- Another name for a component diagram

7. In a sequence diagram, what does an `alt` fragment with guards [event is full] and [seats left] mean?

- Both branches run in parallel
- The branch is repeated until the guard is false
- Exactly one branch runs, the one whose guard is true
- The fragment is optional and may be skipped for any reason

8. Applied: with the Registration state machine from this chapter, a CheckedIn registration receives `cancel`. What should `next(CheckedIn, Trigger::Cancel)` do?

- Return Cancelled
- Return Registered
- Return CheckedIn unchanged and say nothing
- Reject it: no transition leaves CheckedIn, so it throws

9. In an activity diagram, what does a *join* bar do?

- Chooses one of several outgoing flows
- Waits until all incoming parallel flows have arrived, then continues
- Ends the whole activity
- Merges two alternative flows without waiting

10. Which tool keeps diagrams as text that GitHub renders directly inside Markdown?

- draw.io
- PlantUML
- Mermaid
- None of them; all need an exported PNG

WRAP-UP

Summary

A model is a deliberate abstraction with a purpose and an audience; it helps thinking and communication but does not run and can go stale.

UML 2.5.1 has 14 diagram types: structure diagrams (what the system is) and behaviour diagrams (what it does).

Use case diagrams show actors, goals and the system boundary; «include» is always performed, «extend» only under a condition.

Class diagrams map to C++: attributes to private members, `0..*` to `std::vector`, `0..1` to `std::optional`, enumerations to `enum class`.

Composition means ownership (a member container); an association to data stored elsewhere is best kept as an id.

Object diagrams are snapshots that test a class diagram and double as test fixtures.

Sequence diagrams show one scenario as messages between lifelines; `alt` becomes `if`/`throw` in code.

Activity diagrams model workflows with decisions, fork/join and swimlanes; state machines model one object's life and map to a `switch`.

Keep diagram sources (PlantUML, Mermaid, draw.io) in Git, small, consistently named and in sync with the code.

Open Lab-06: 5 tasks + 1 challenge

Next chapter: 07 · Agile Software Development Model