# Design Exercise – UniStudy (Part 2)

## From Requirements to Design

# Recap – Where We Are

In the previous exercise we produced:

- **Actors**: Student, Teacher
- **Use Case Diagram** (overview level)
- **User Requirements** (3)
- **System Requirements** (2 per user requirement) / **Non-Functional Requirements**

*Now we move from what the system shall do*
*to how the system is structured and behaves.*

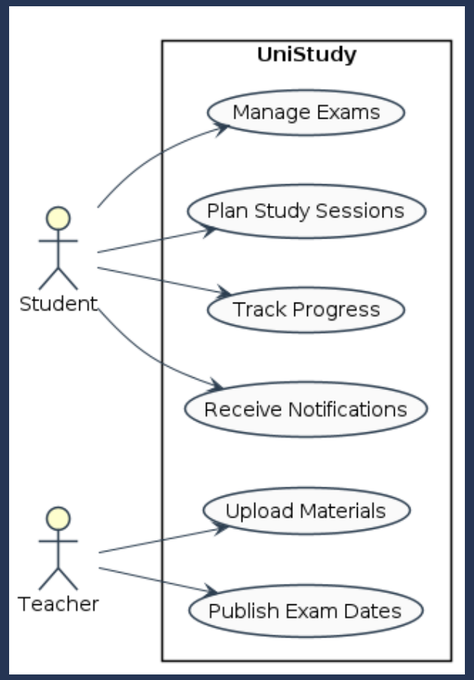
**UniStudy** is a web-based platform that allows:

- **Students** to plan study sessions, track progress, access materials, receive notifications.
- **Teachers** to upload teaching materials and publish exam dates.

Each study session is associated with a specific **exam**.
Each exam is taught by one **teacher** and may have multiple **teaching materials** attached. Students belong to a university and can enrol in multiple exams.

# Input – Use Case Diagram

The starting point of this exercise is the use case diagram produced previously:



# Input – User Requirements

1. **UR1** – The student shall be able to plan study sessions for each exam.
2. **UR2** – The student shall be able to track their study progress.
3. **UR3** – The teacher shall be able to upload teaching materials.

*These three requirements are enough to drive the design of classes,*

# Exercise Overview

We will design the **static and dynamic structure** of UniStudy.

Steps:

- Identify the main classes of the domain
- Draw an **overview Class Diagram** (names + associations + multiplicities)
- Draw **two specification Class Diagrams** (attributes and operations)

# Exercise Overview

- Choose one scenario and draw a **Sequence Diagram**
- Draw an **overview Activity Diagram** of the main workflow
- Draw a **State Diagram** for one significant class

# Modelling Guidelines

For this exercise we stay at the **design level**, not implementation.

- No data types (no `int`, `String`, `Date`)
- No visibility modifiers (`+`, `–`, `#`)
- No getters/setters
- No technological details (no DB, no REST endpoints)
- Operation names describe **what** the object does, not **how**

# Modelling Guidelines

Think in terms of **responsibilities**, not code.

Solution diagrams in this document are rendered with **PlantUML**.

Both the PNG outputs and the `.puml` sources are provided (see the `diagrams/` folder) so that you can study, modify and reproduce them.

# Step 1 – Identify the Classes

## Question

Re-read the system description and the use case diagram.

**Which entities of the domain deserve to be a class?**

*Tip: look for nouns that have a state and a behaviour.*
*Discard nouns that are just attributes (e.g. "name", "date").*

# Step 1 – Classes

- Student
- Teacher
- Exam
- StudySession
- TeachingMaterial
- Notification
- ProgressReport

# Step 2 – Overview Class Diagram

## Task

Draw the **overview class diagram**.

Constraints:

- Only **class names**
- Lines between associated classes
- Each association has a **name** (verb or short phrase)
- Each end has a **multiplicity** (`1`, `0..1`, `1..*`, `0..*`)

# Step 2 – Hints

Ask yourself, for each pair of classes:

- Is there a meaningful link between them?
- How many instances on each side?
- What is the **verb** that describes the link?

Examples of verbs: *plans*, *enrols in*, *teaches*, *contains*, *generates*, *attached to*.

# Step 2 – Solution (textual)

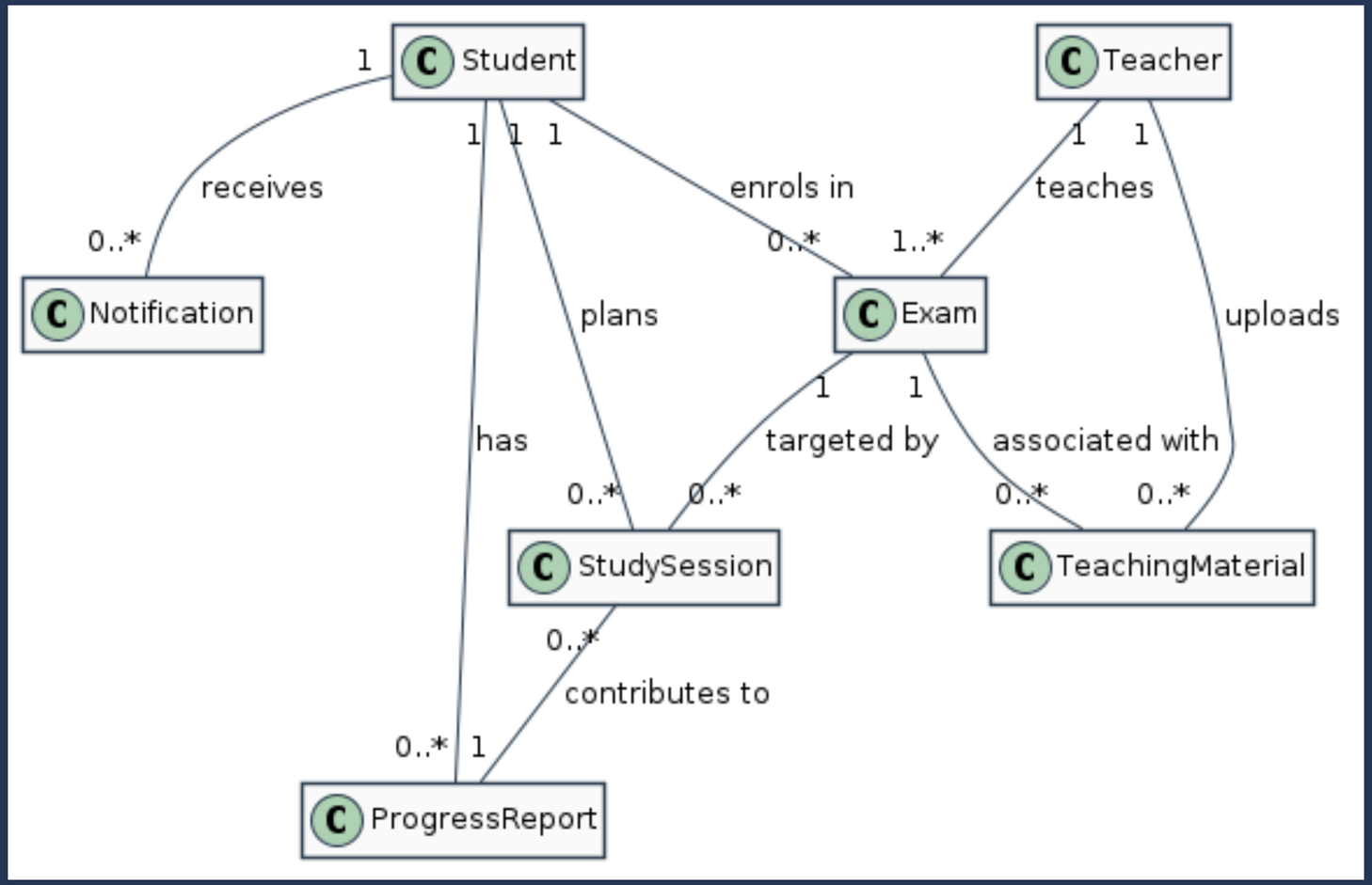
```
Student   1 ── enrols in ──   0..*   Exam
Student   1 ── plans ──        0..*   StudySession
Student   1 ── receives ──     0..*   Notification
Student   1 ── has ──          0..*   ProgressReport

Teacher   1 ── teaches ──      1..*   Exam
Teacher   1 ── uploads ──      0..*   TeachingMaterial

Exam      1 ── targeted by ──  0..*   StudySession
Exam      1 ── associated with- 0..*   TeachingMaterial

StudySession  0..* ── contributes to ── 1  ProgressReport
```

# Step 2 – Solution (diagram)



# Step 3 – Specification Class Diagrams

Pick **two classes** from the overview diagram and draw the **specification class diagram**:

- **Attributes** (state)
- **Operations** (behaviour)

*Stay at the specification level: no types, no implementation, no getters/setters.*

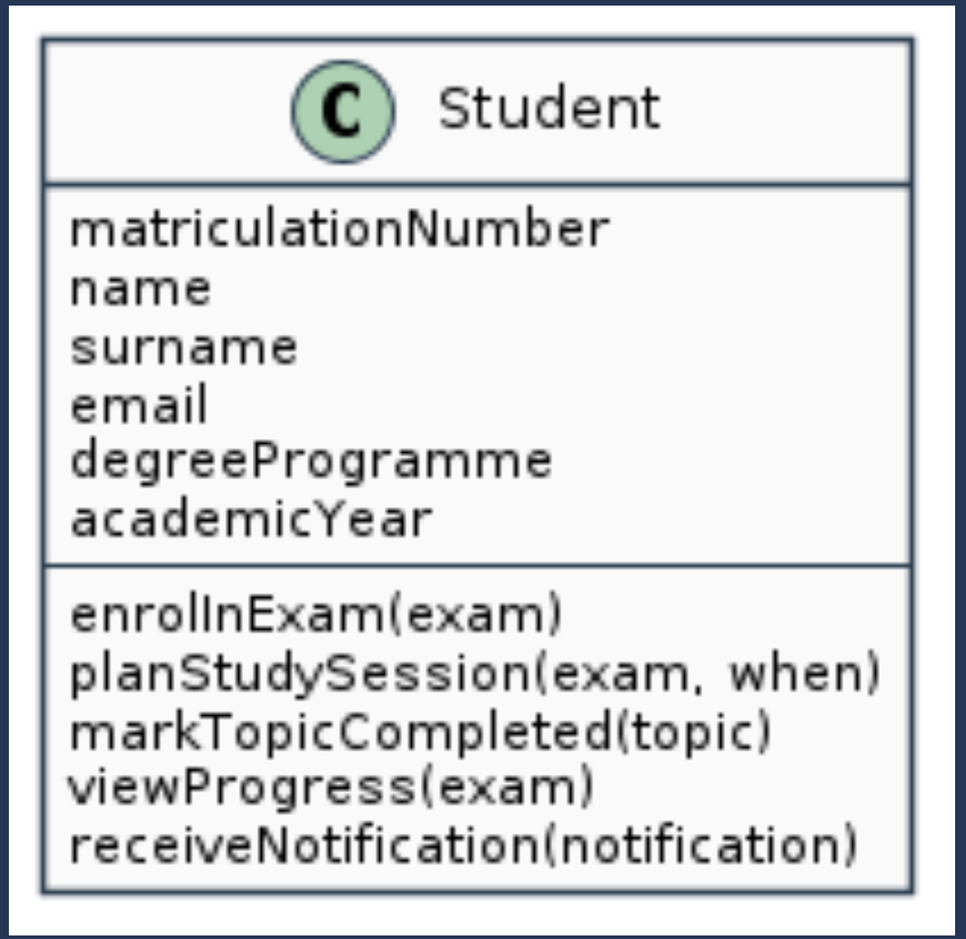
For this exercise we choose: **Student** and **StudySession**.

# Step 3 – Specification: `Student` Question

What information does a Student **hold**?
What does a Student **do** in the system?

(Think in terms of responsibilities towards the rest of the model.)

# Step 3 – Specification: `Student`

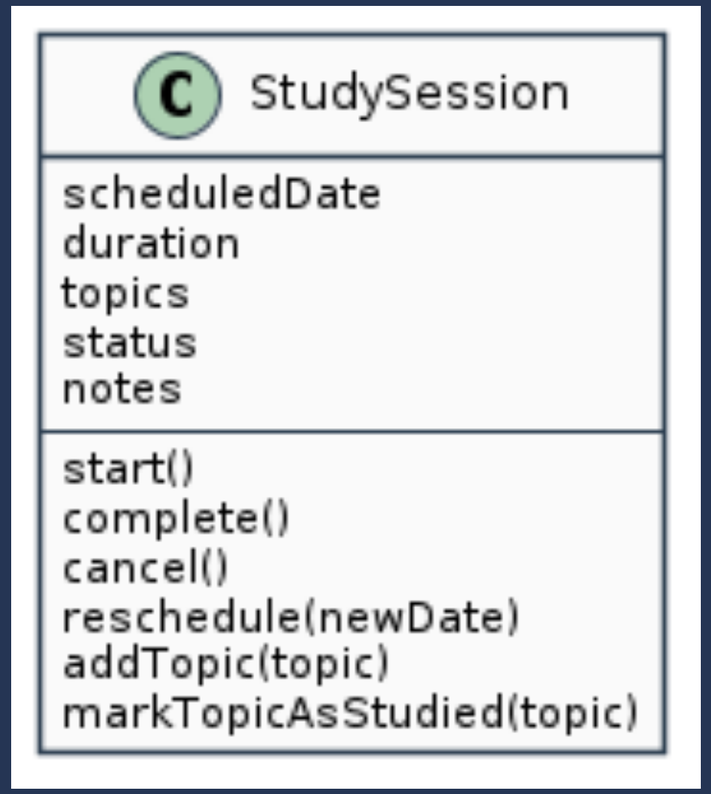


# Step 3 – Specification:
# `StudySession`
# Question

A `StudySession` is **scheduled by a student** for a **specific exam**, on a **specific date and time**, possibly on **specific topics**.

What is its state and its behaviour?

# Step 3 – Specification:
## `StudySession`



# Step 3 – Reflection

Observe:

- Attributes are the **memory** of the object.
- Operations are the **services** the object offers to others.
- The two together define the **contract** of the class.

# Step 4 – Sequence Diagram

## Task

Choose **one scenario** from the use case diagram and draw a **sequence diagram** that shows the **interaction between objects** that realises it.

For this exercise the chosen scenario is:

*"The student plans a new study session for an exam."*

# Step 4 – Scenario Description

**Precondition**: the student is logged in and has at least one exam in their plan.

**Main flow**:

- The student opens the planning view.
- The student selects an exam.
- The student inserts date, duration and topics.

- The system validates the request (no overlapping sessions).
- The system creates and stores the session.
- The system confirms the operation to the student.

**Postcondition**: a new `StudySession` exists, linked to the selected `Exam` and to the `Student`.

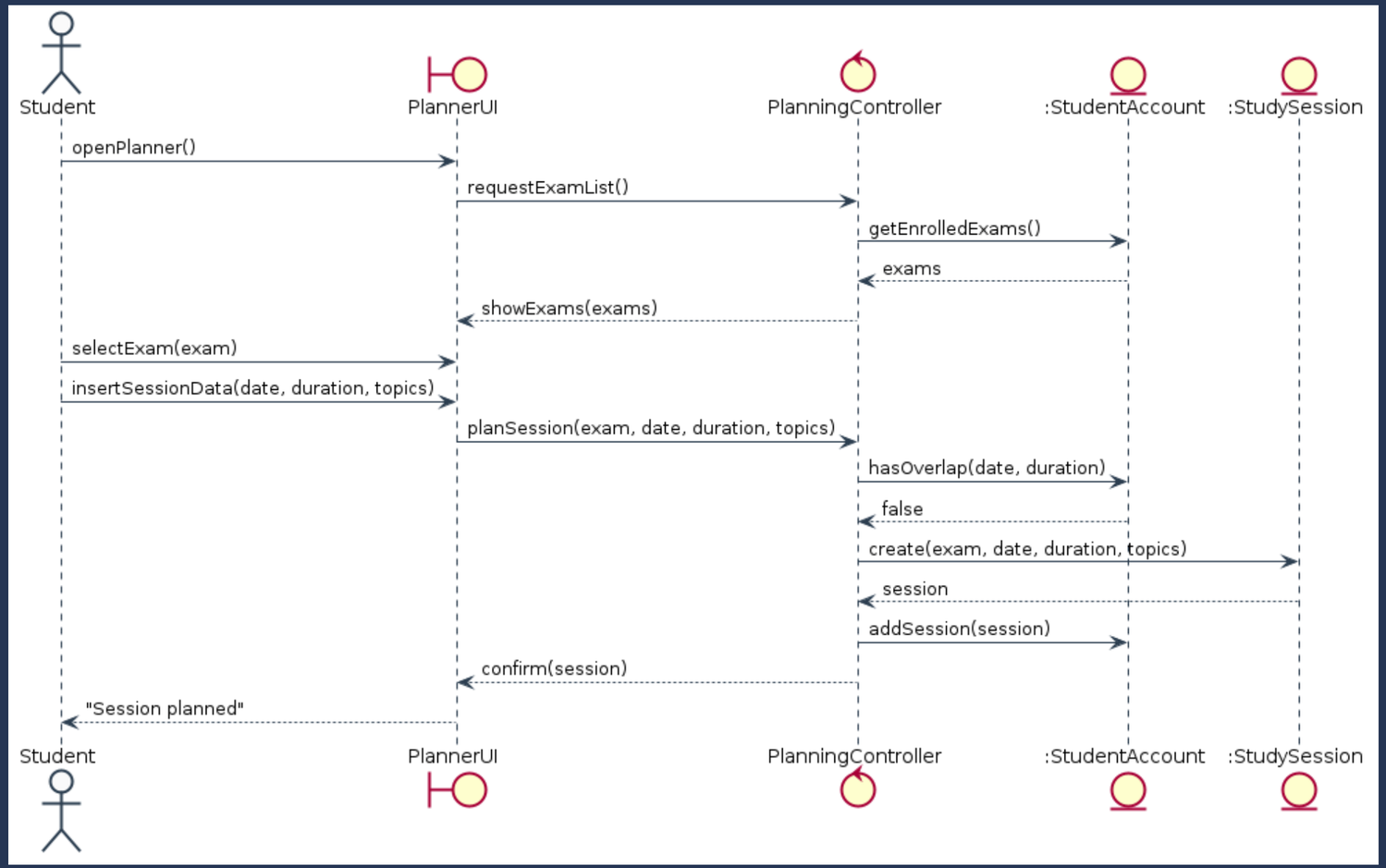
# Step 4 – Hints

Participants you will need on the lifelines:

- `:Student` (actor)
- `:PlannerUI` (boundary)
- `:PlanningController` (control)
- `:StudentAccount` (entity)
- `:StudySession` (entity)

*You don't need to invent everything. Stay close to the scenario.*

# Step 4 – Solution



# Step 5 – Activity Diagram

Draw an **overview activity diagram** of the main workflow of a student using UniStudy.

Goal: show the **flow of activities** from the moment the student plans their study to the moment they review their progress.

Use the following constructs:
- **Initial node** and **final node**
- **Actions** (verb + object)
- **Decision nodes** (rhombus) with guard conditions

# Step 5 – Scenario for the Activity

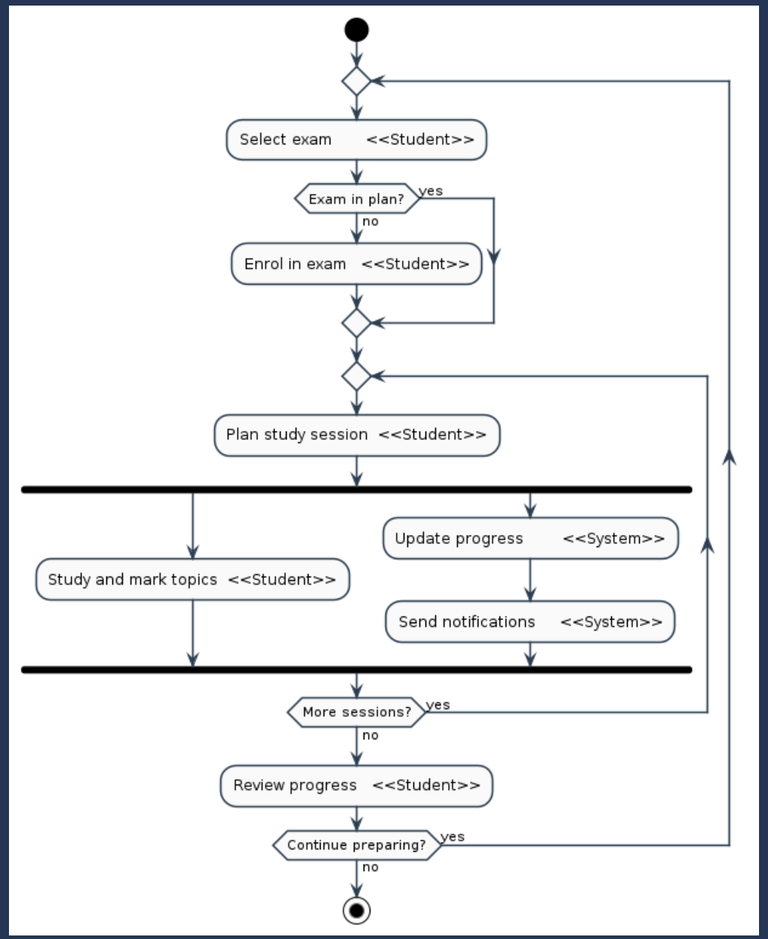
1. The student selects an exam to prepare for.
2. The student plans one or more study sessions.
3. For each session, the student marks topics as completed.
4. In parallel, the system updates progress and may send notifications.
5. When the exam approaches, the student reviews the overall progress.
6. The cycle ends when the student decides to stop preparing.

# Step 5 – Hints

- Where is the **first decision** in the flow? (e.g. is the exam already in the plan?)
- Which activities can happen **in parallel** between Student and System?
- Where does the flow **loop back**? (e.g. after completing one session, you may plan another)
- What is the **exit condition** of the workflow?

*Stay at overview level: do not model every click.*

# Step 5 – Solution



# Step 5 – Reflection

Discussion points:

- The **fork/join** captures that progress update is a **system-side activity** that runs alongside the student's work.
- The **two decision nodes** (more sessions? continue preparing?) define two nested loops: short-term (within an exam) and long-term (across exams).
- The diagram is **overview**: a refined version could explode "Study and mark topics" into a sub-activity diagram.

# Step 6 – State Diagram

Choose **one class** whose instances have a meaningful **lifecycle** and draw its **state diagram**.

For this exercise we choose: **StudySession**.

*Why this class? In Step 3 we introduced* `status` *as an attribute and described its operations as state transitions (*`start`*,* `complete`*,* `cancel`*,* `reschedule`*).*
*Now we model that lifecycle explicitly.*

# Step 6 – Identifying States

## Question

What are the **states** in the life of a `StudySession`?

Hints:

- A session is **born** when the student plans it.
- It becomes **active** when the student starts studying.
- It can be **finished** successfully or **abandoned**.
- It can also be **moved** to a different date before it starts.

# Step 6 – Solution: States

States of `StudySession`:

- **Planned** – created, scheduled in the future, not yet started
- **Ongoing** – the student is currently studying
- **Completed** – the session ended and topics were marked as studied
- **Cancelled** – the student gave up the session before completing it
- **Missed** – the scheduled time passed without the session

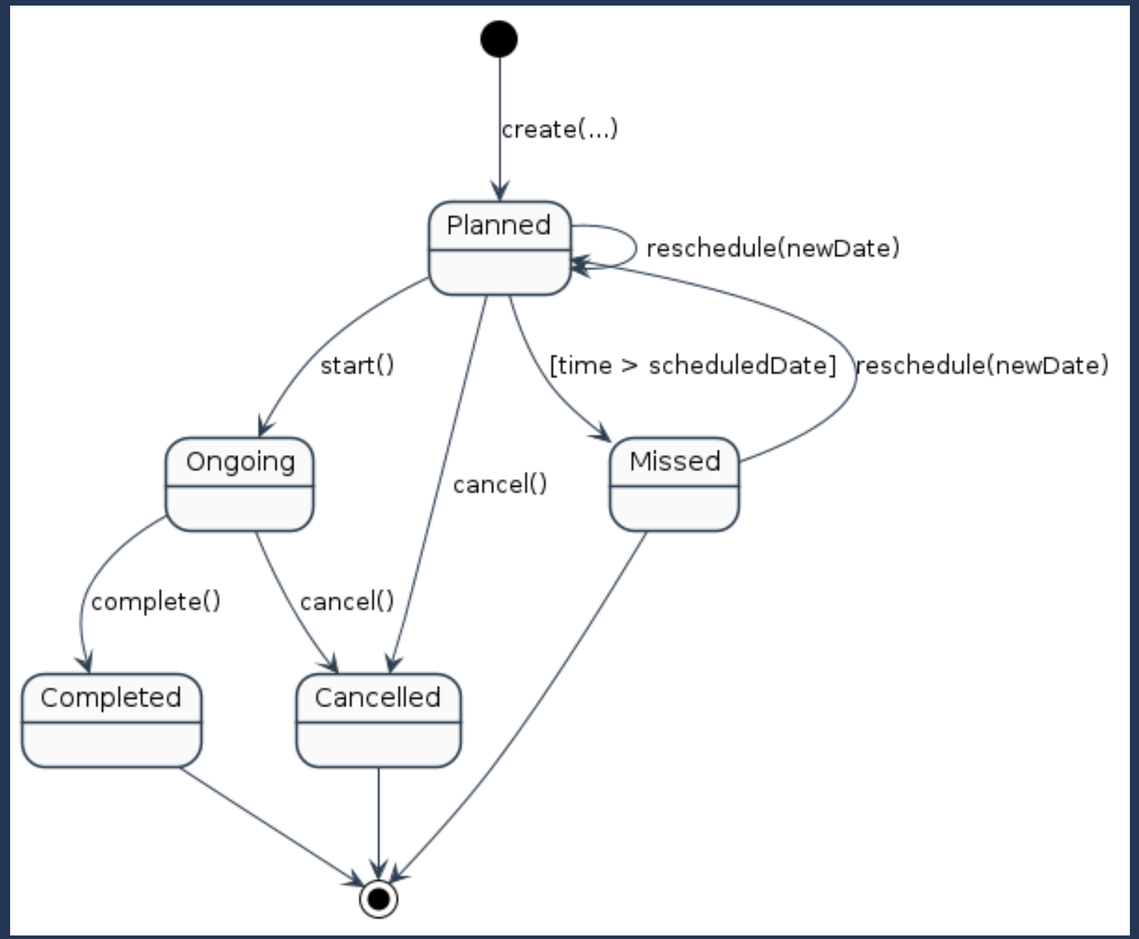
# Step 6 – Identifying Transitions

## Question

For each pair of states, ask: **which event causes the transition?**

The event can be:
- a user action (call to an operation)
- a system event (e.g. time elapsed)
- a guard condition that becomes true

# Step 6 – Solution (diagram)



# Step 6 – Reflection

Discussion points:

- The state diagram makes **explicit** what was implicit in the `status` attribute.
- Operations on `StudySession` correspond to **labels on transitions**: this is a strong cross-check between the specification class diagram and the state diagram.
- The transition `Planned → Missed` is **time-triggered**, not user-triggered.

# Wrap-up

In this exercise we moved from **requirements** to **design**, covering both structure and behaviour:

| View | Diagram | Question answered |
|---|---|---|
| Static | Overview Class Diagram | What entities exist and how are they related? |
| Static | Specification Class Diagrams | What does each class know and do? |
| Dynamic | Sequence Diagram | How do objects collaborate in a scenario? |
| Dynamic | Activity Diagram | What is the workflow of the user? |
| Dynamic | State Diagram | What is the lifecycle of an object? |

*The five views are complementary: each one tells a part of the story.*