Software Engineering with UML

Tewfik Ziadi
ziadi@ece.fr

Organisation of the course (1/4)

- Week 1 : Introduction to SE with UML
  - Part 1 : Course
  - Part 2 : Exercises/ Lab work
- Week 2 : Requirement analysis: Use case diagrams
  - Part 1 : Course
  - Part 2 : Exercises/ Lab work
- Week 3 : Structural modeling class diagrams + design pattern
  - Part 1 : Course
  - Part 2 : Exercises/ Lab work
- Week 4: Structural modeling : Lab work and work on project.
  - Part 1 : Exercises/ Lab work
  - Part 2 : Work on the project.

•

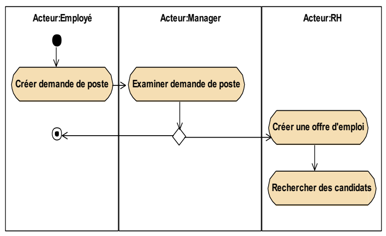
Organisation of the course (2/4)

- <u>Week 5</u> : Sequence diagrams + state machines
  - Part 1 : Course
  - Part 2 : Exercises/ Lab work
- **<u>Week 6</u> : Sequence diagrams + state machines: Lab work**
  - **Part 1 : Exercises/ Lab work**
  - **Part 2 : Work on the project**
- <u>Week 7</u> : Code generation & reverse engineering
  - Part 1: Course
  - Part 2: Exercises/ Lab work
- <u>Week 8</u> : Software Testing
  - Part 1 : Course
  - Part 2 : Exercises/ Lab works

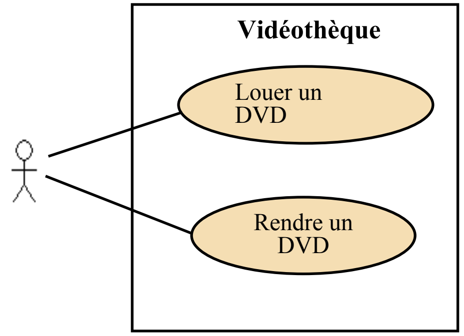
Organisation of the course (3/4)

- <u>Week 9</u> : Software Product Lines
  - Part 1 : Course
  - Part 2 : Exercises/ Lab work

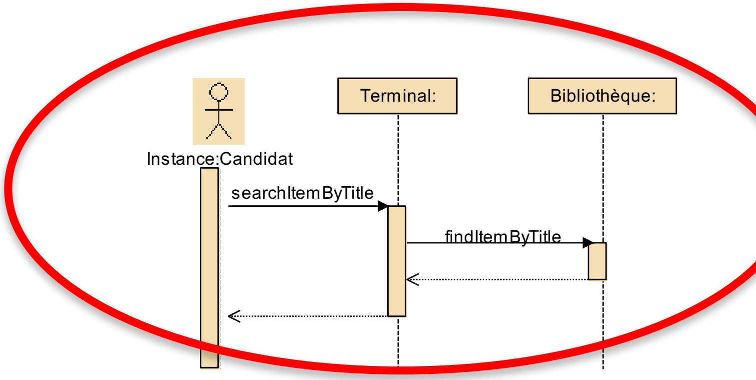
The UML Is a Language for Documenting



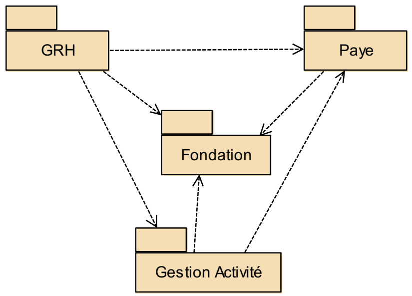
Activity diagrams



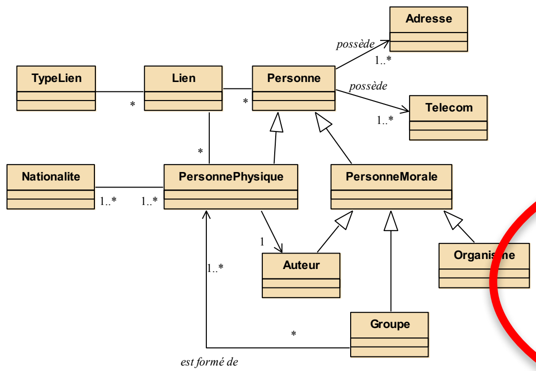
Use case diagrams



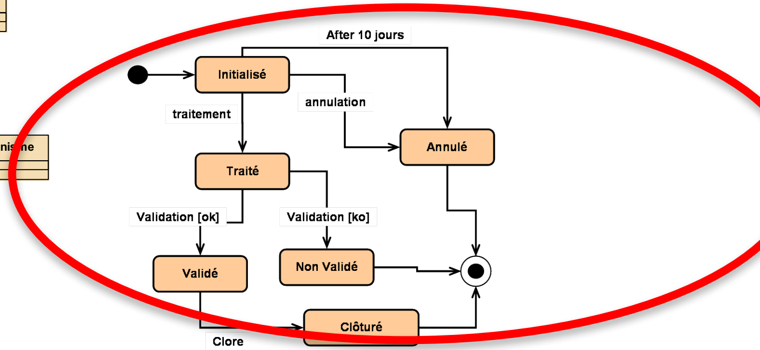
Sequence diagrams



Package Diagrams



Class Diagrams



State machine diagrams

Modeling systems with the UML

Behavior modeling with UML Sequence diagrams

Objectives and outcomes

- The objective of behavior modeling is to study/specify the behavior of the objects in a system.

- <u>Reminder</u>: An OO system is a system made up of objects that work together to realize some desirable functionality.
  - We have the desired functionality -> use cases
  - We have the object structure -> classes

Objectives and outcomes

- We will study two complimentary behavior models:
  - **Sequence diagrams** specify how objects work together to realize a single functionality ➔ **Inter-Object view**
  - State Machines specify the global behavior (participation in all functionalities) of a single object ➔ **Intra-Object View**

Objectives and outcomes

- At the logical level, the behavior models allow us to:
  - Complete the structural model by finding the methods of our classes.
  - Validate the structural model by making sure that all required attributes and (navigable) associations are present.

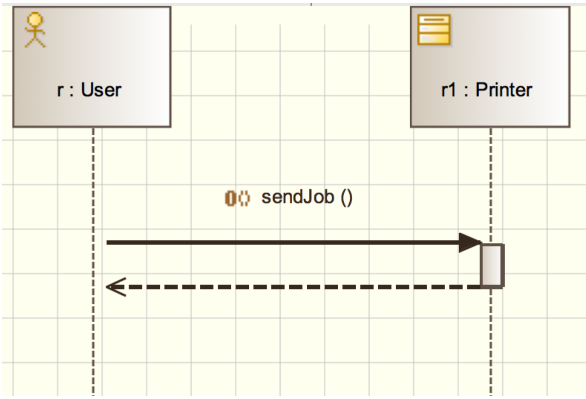
- At the physical level,
  - Sequence diagrams define a specification for our algorithms.
  - State Machines can be used to generate executable code.

Sequence diagrams

- A sequence diagrams (also called "interaction diagram") shows a sequence of messages exchanged by the objects of a system.
- We generally use a sequence diagram to specify the realization of a single course of action in a use case.
  - Helps us find the methods of our classes
  - Helps us validate that the logical data structure is sufficient to realize the functionality.

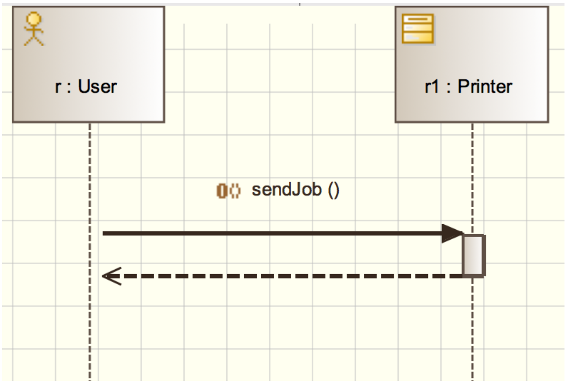
- Key concepts of a sequence diagram
  - Participant: entities participating in the interaction
  - Messages: Communications between objects.

- Two axes in a sequence diagram:
  - Horizontal: which object/participant is acting
  - Vertical: time (forward in time)

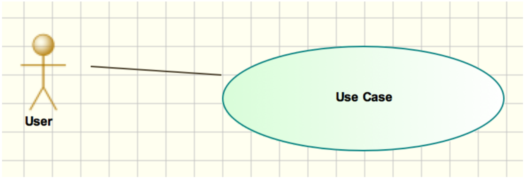


Participant: Objects

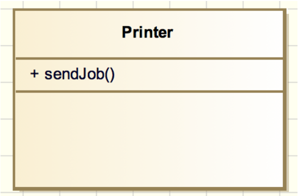
- Two kinds of participants: Objects or Actors
- An object is shown as a box at the top of a dashed vertical line called its lifeline.
  - The box contains a label of the form instanceName:className.
  - The lifeline shows the object's life during the interaction
  - objects may be created and destroyed during the interaction (see later)
- A participant may be also an instance of an actor defined in use case diagrams



Use Case



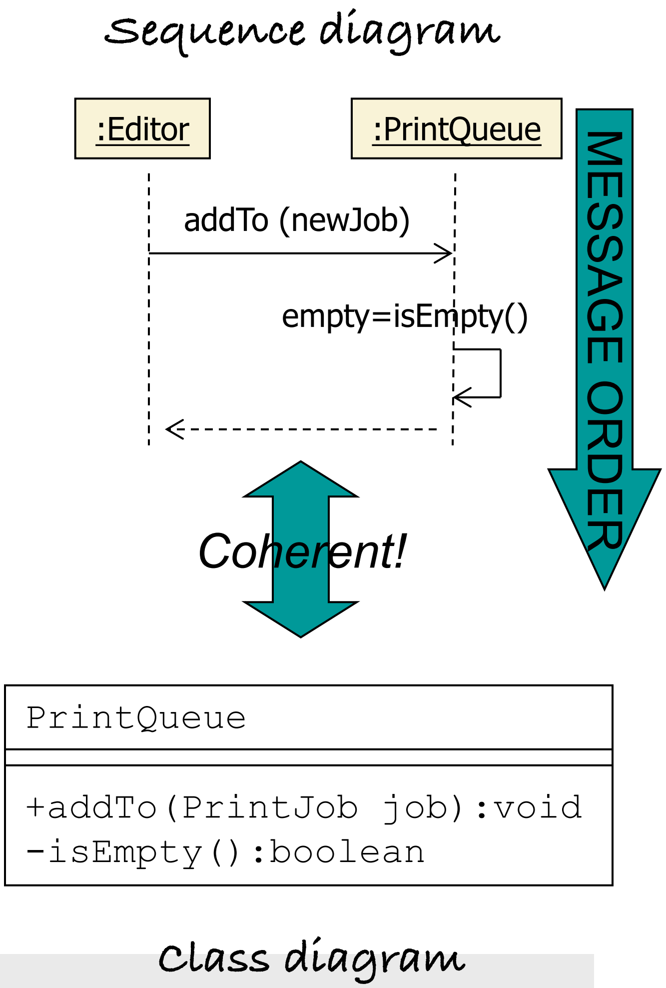
Class



Messages

We use messages to ask objects to execute services, store some data, or retrieve some data (or a combination of all three). We can also think of messages as "delegating work".

- A message is shown by an **arrow** between the lifelines (or activations) of two objects.
- The message is declared as an operation in the receiving object's class.
- A message must specify :
  - the operation name
  - the message arguments
  - a return value
- The order in which messages are sent are shown from top to bottom on the page.
- A self call, shown by a message returning to the sending object, indicates that an object calls one of its own methods



Messages

- UML defines three basic message types

Simple message : Message execution model unspecified.

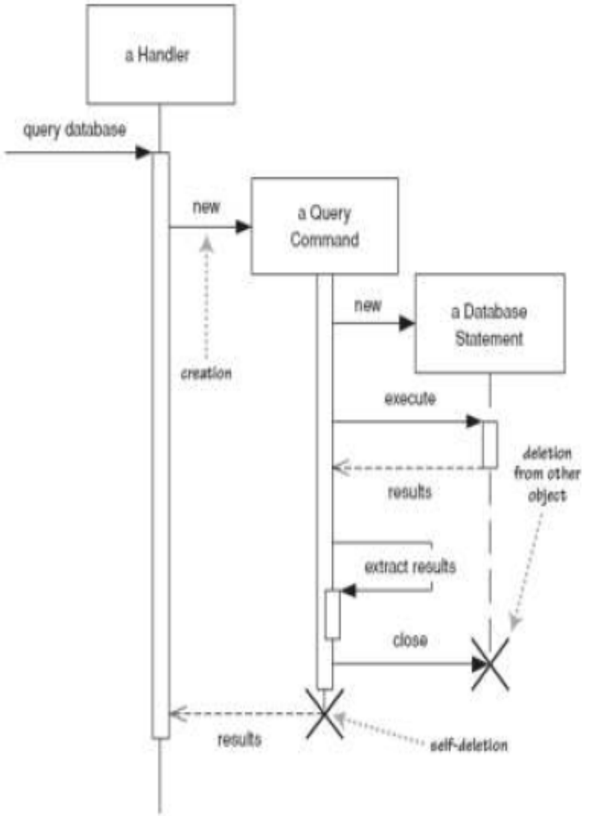
Operation call : A synchronous message that blocks the caller while the called method executes.

Signal : An asynchronous message that does not block the caller. The caller continues to execute concurrently with the called method.

Return : Shows the return of control following a prior message

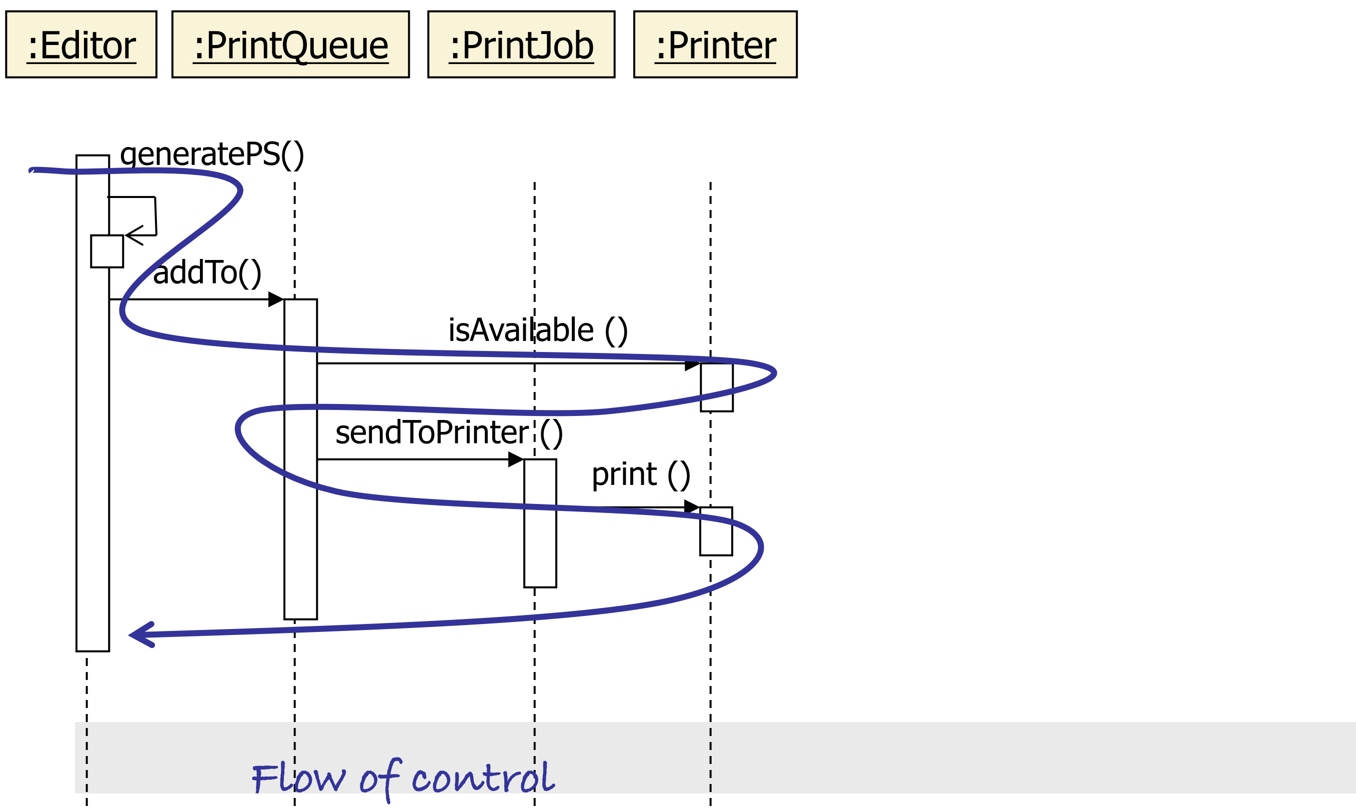
Creation and Deletion Messages

- **Creation Message**: arrow with 'new' written above it
  - an object is created
- Deletion Message: an X at bottom of object's lifeline
  - Java doesn't explicitly delete objects; they fall out of scope and are garbage collected

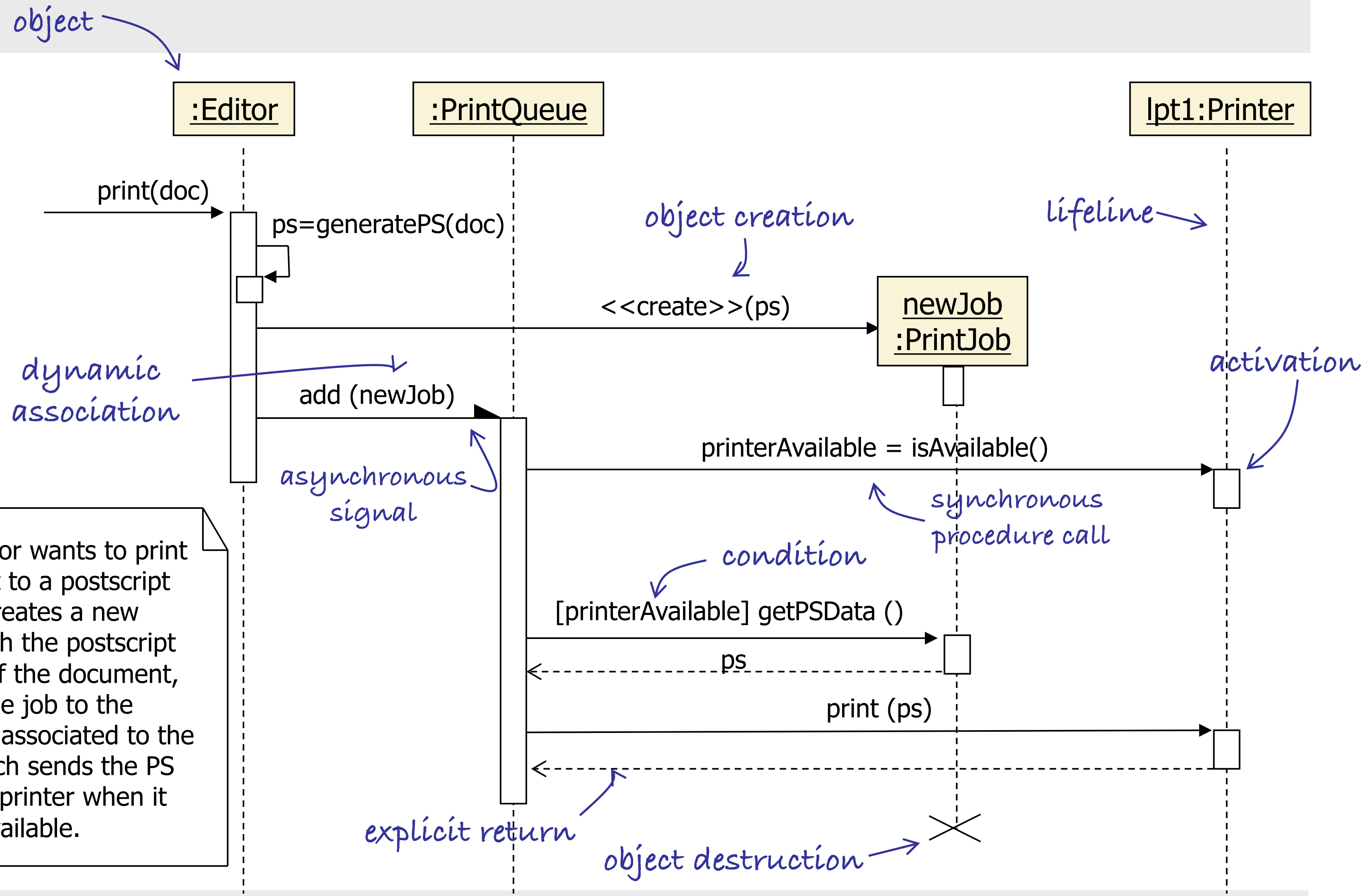


Activation

- **Activation:** thick box over object's life line
  - Either: that object is running its code or it is on the stack waiting for another object's method



Sequence diagram example

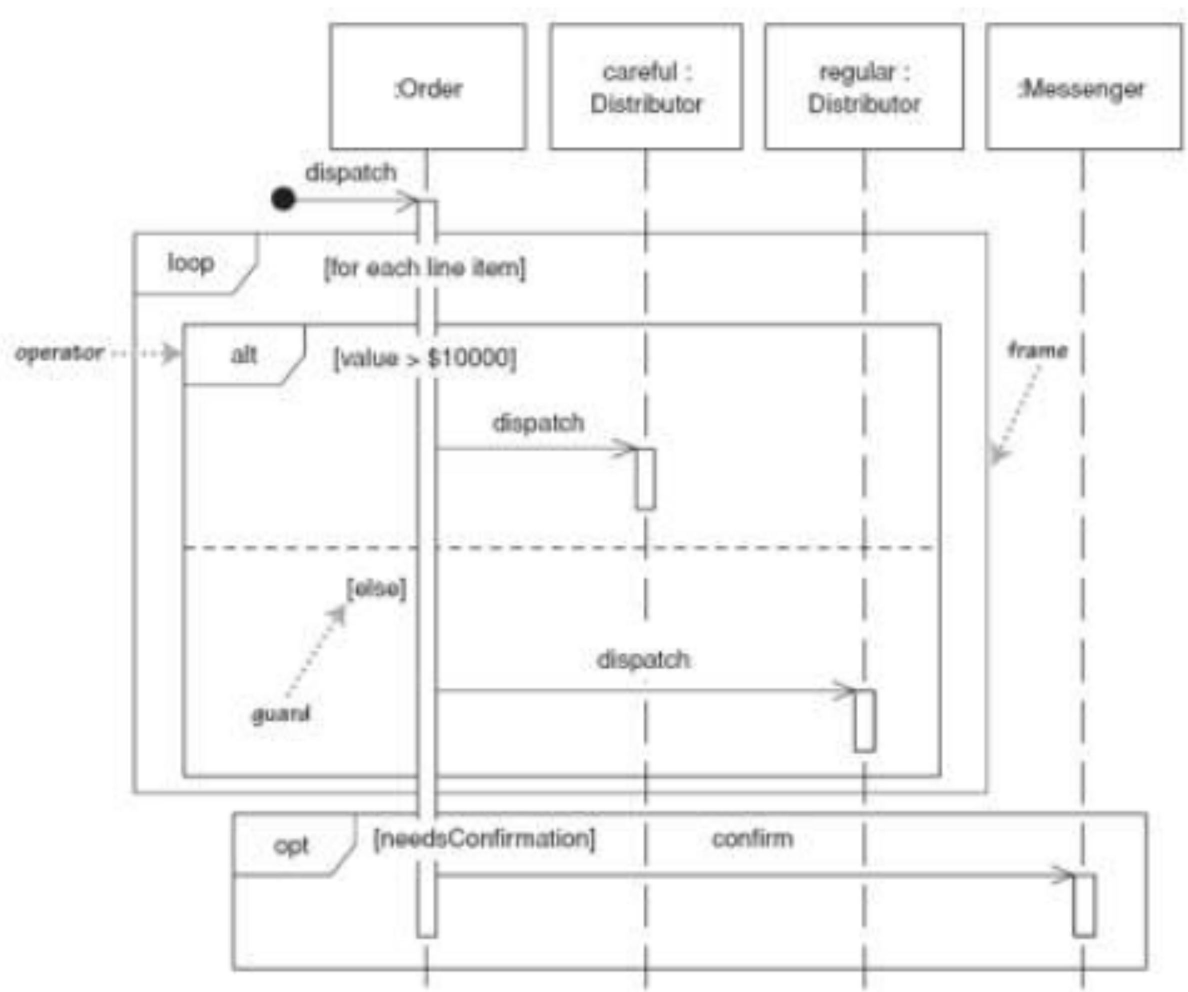


An text editor wants to print a document to a postscript printer. It creates a new PrintJob with the postscript rendering of the document, and adds the job to the PrintQueue associated to the printer, which sends the PS data to the printer when it becomes available.

Combined Fragments; opt, alt, loop

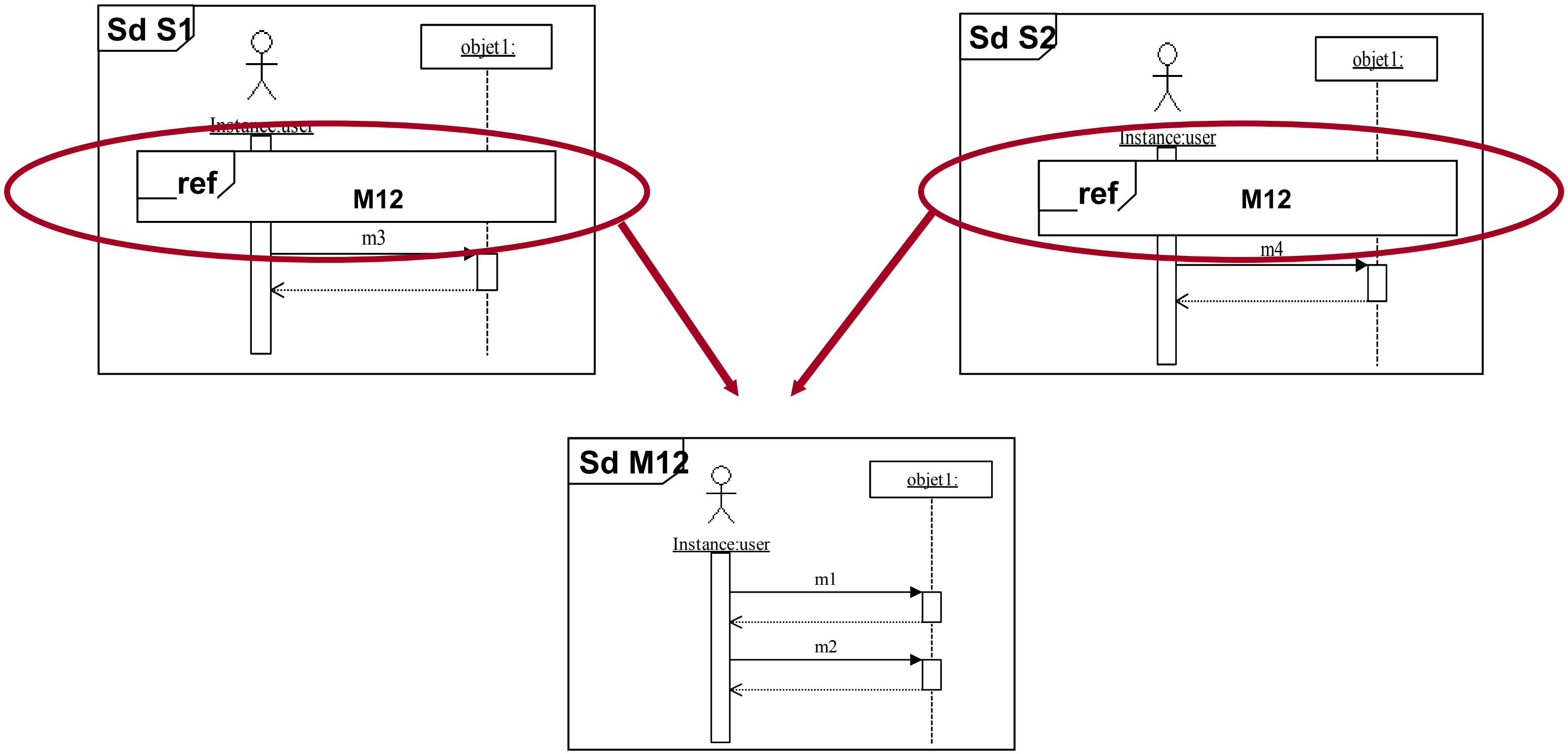
- frame: box around part of a sequence diagram to indicate selection or loop
  - **if** -> (opt) [condition]
  - **if/else** -> (alt) [condition], separated by horizontal dashed line
  - **loop** -> (loop) [condition or items to loop over]

Combined Fragments; opt, alt, loop



Combined Fragments: Ref

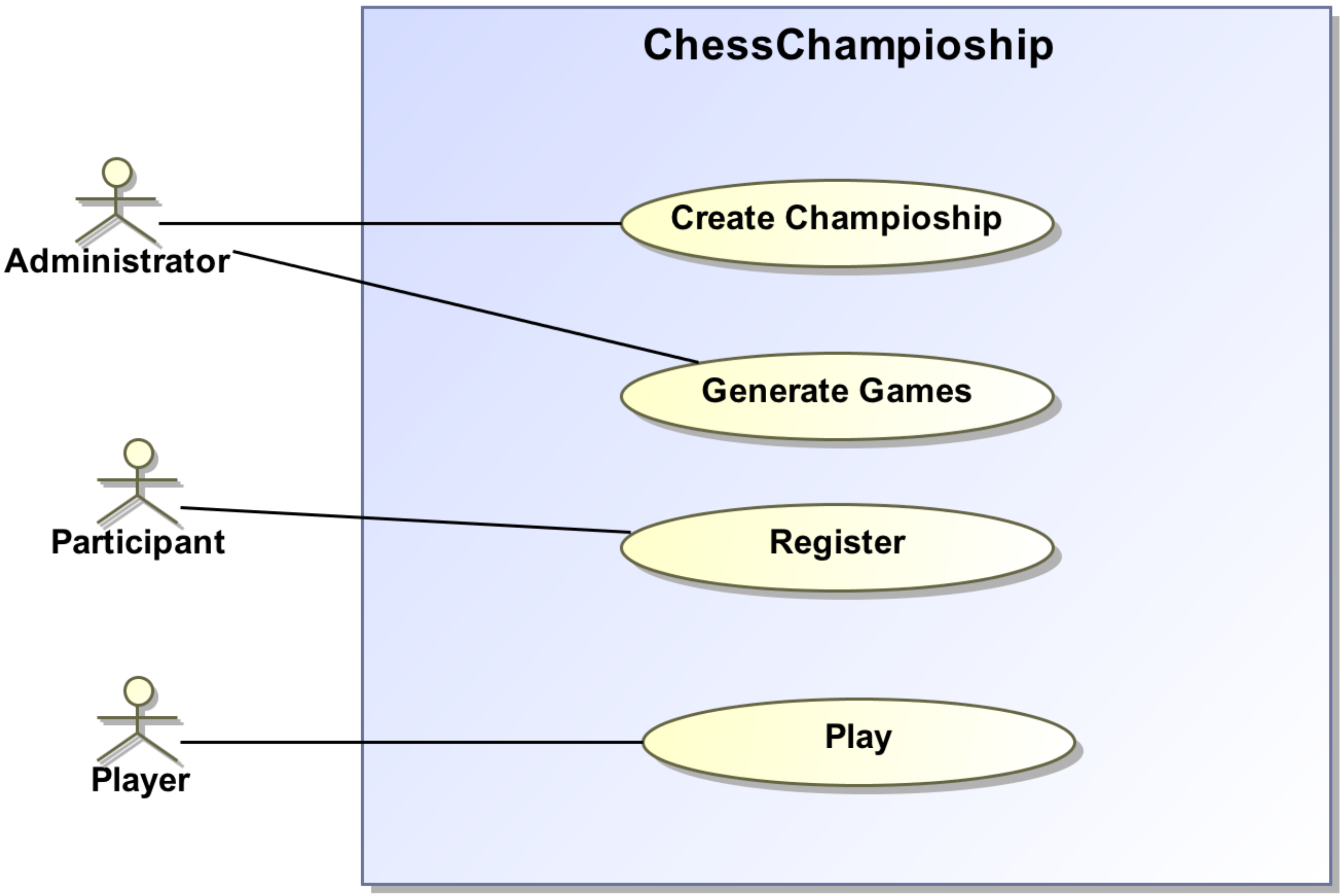
- Example **ref operator**: allows to refer to another sequence diagram.



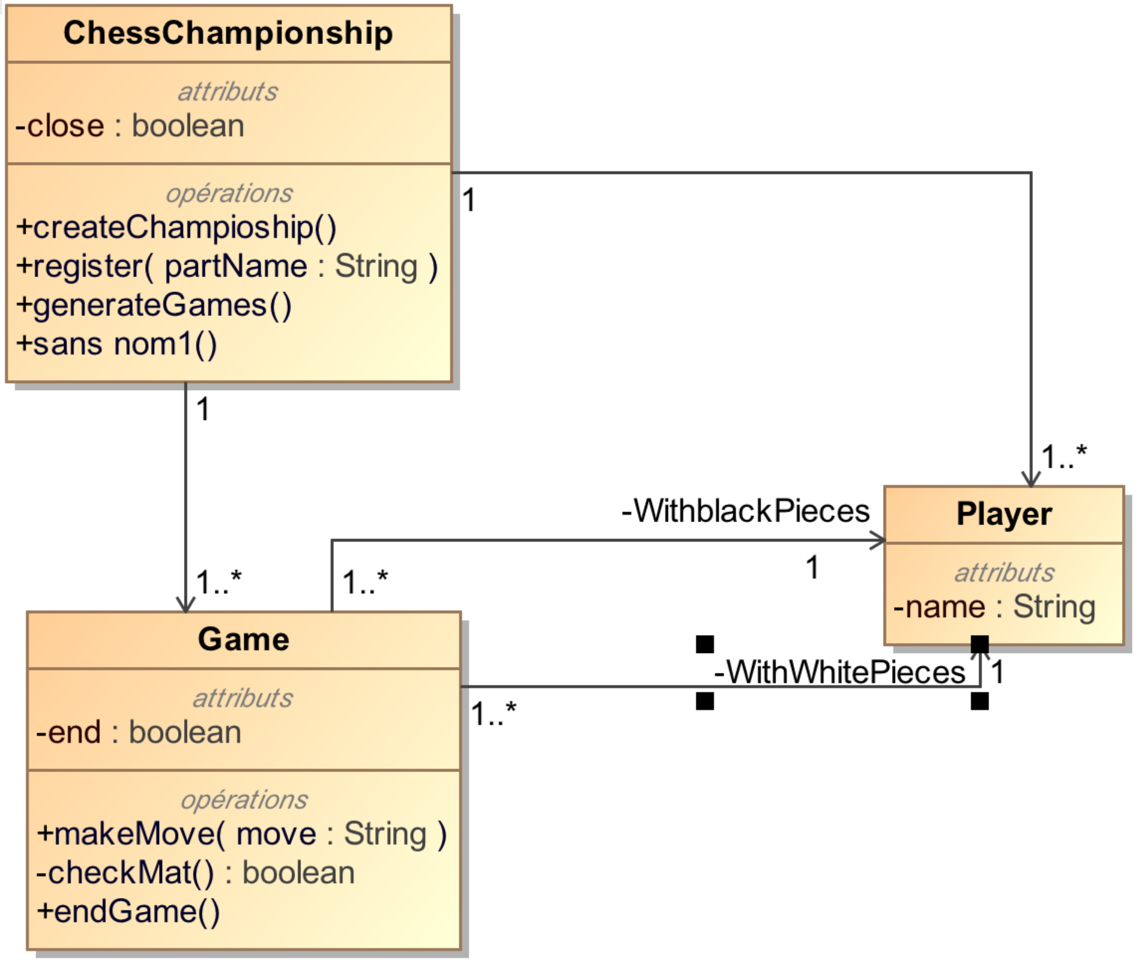
Example: Chess Championship

- A simple application to manage an online chess championship
- Administrator
  - Create a championship
  - Generate games
- Participant
  - Register
- Player
  - Play a game

Example: Chess Championship



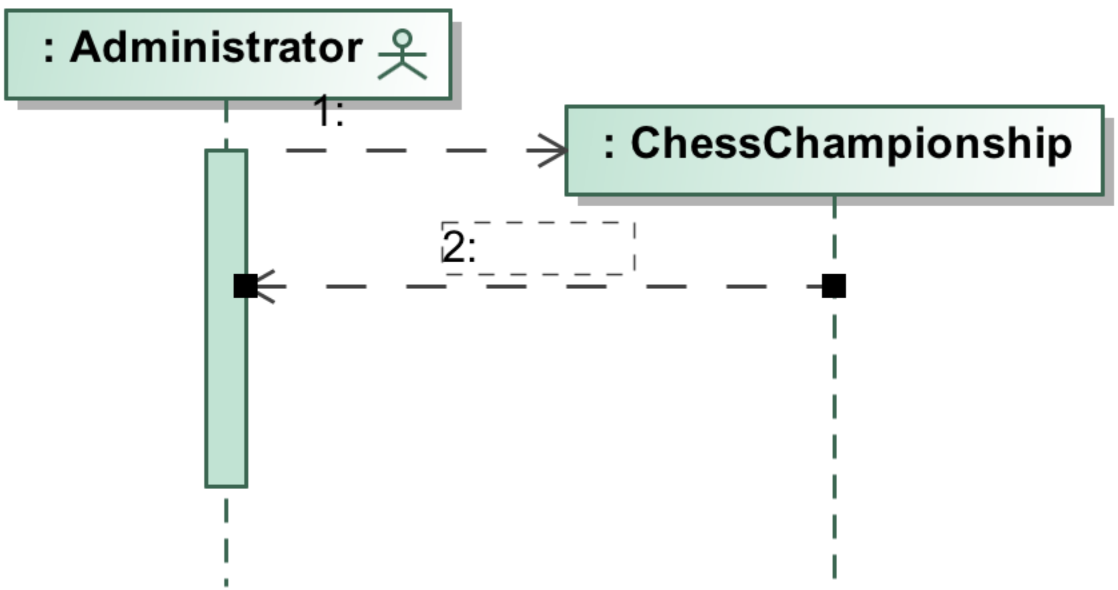
Example: Chess Championship



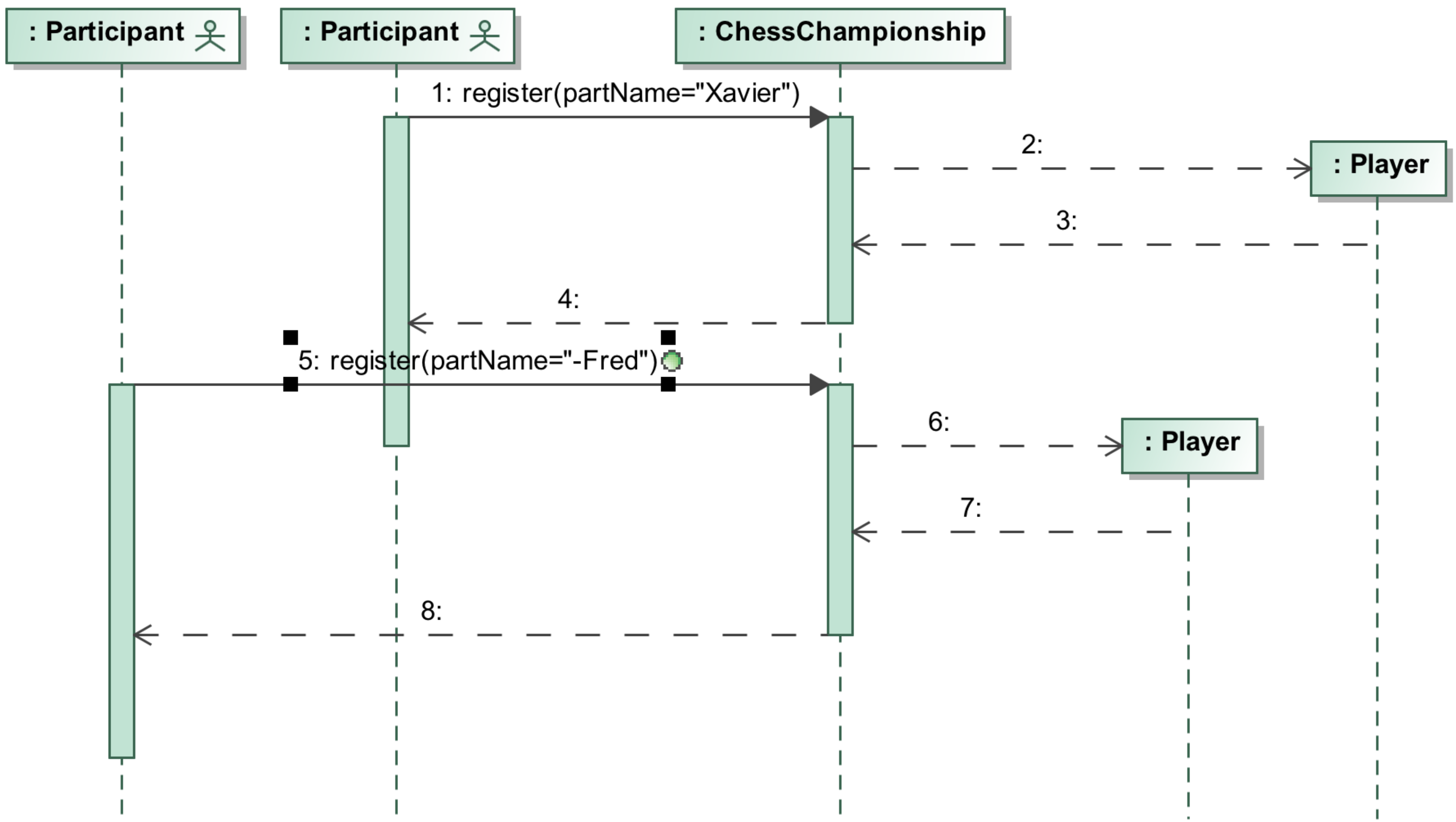
Example: Chess Championship

- How to specify the behavior of this application using sequence diagrams?
- Idea :
  - At least, one sequence diagram per use case.
  - The sequence diagrams must be coherent with the class diagram.

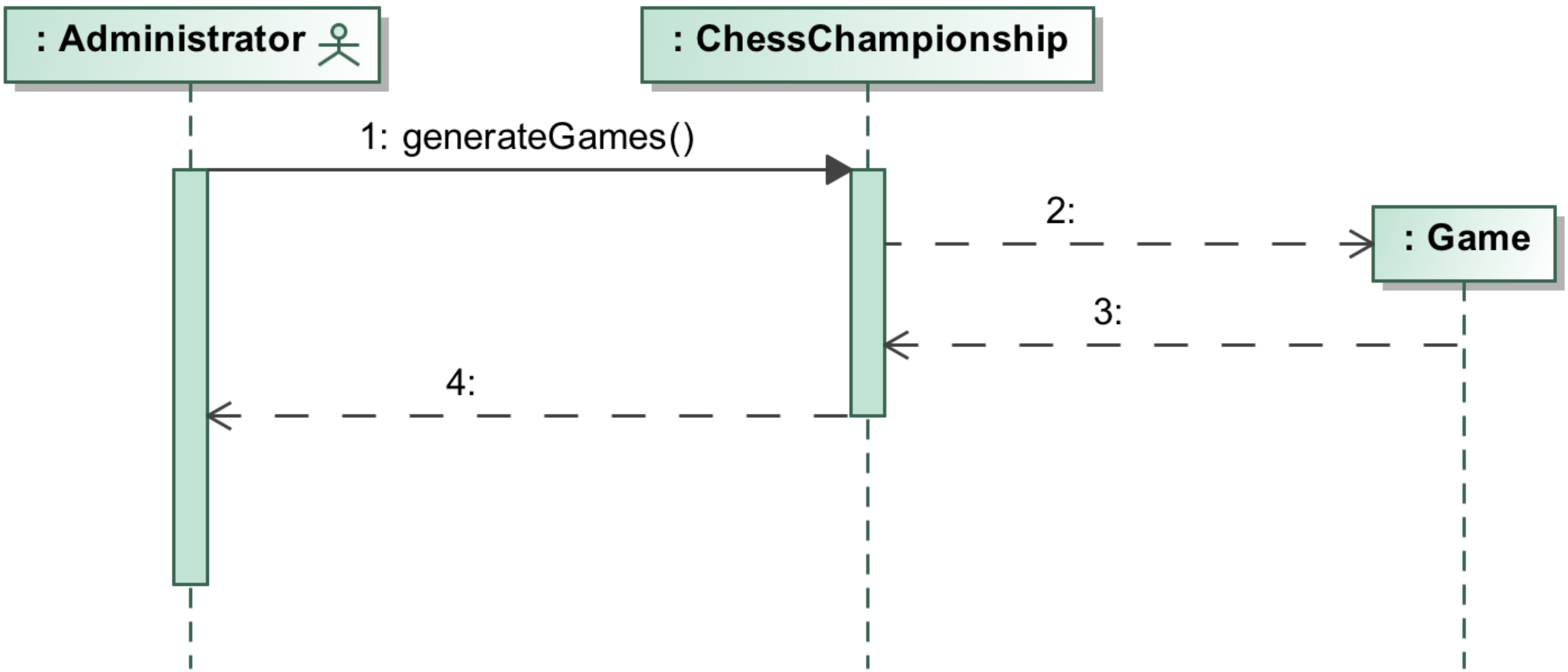
Create a Championship



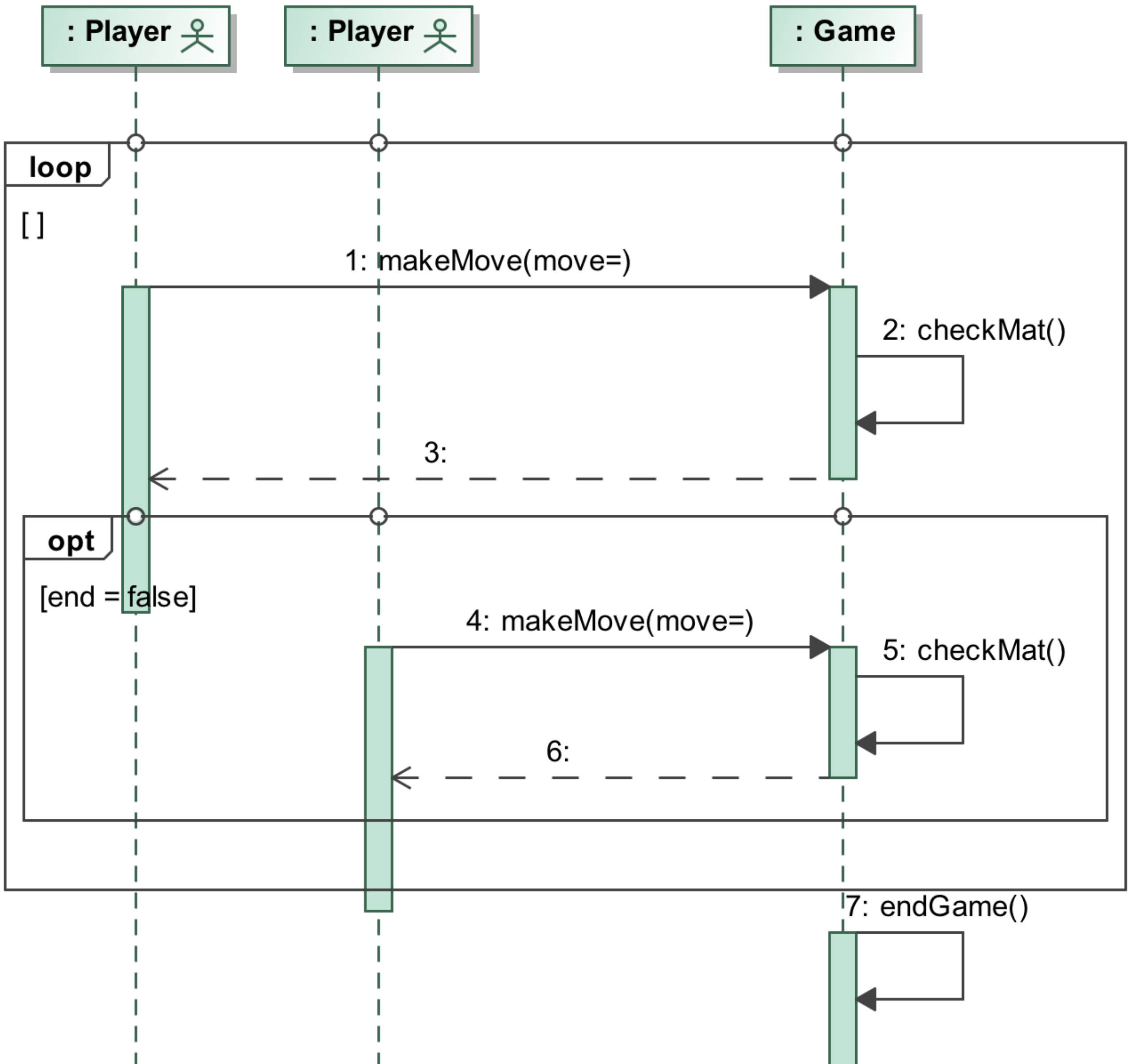
Register two participants



Generate Games



Play a game


: Player
: Player
: Game
loop
[ ]
1: makeMove(move=)
2: checkMat()
3:
opt
[end = false]
4: makeMove(move=)
5: checkMat()
6:
7: endGame()

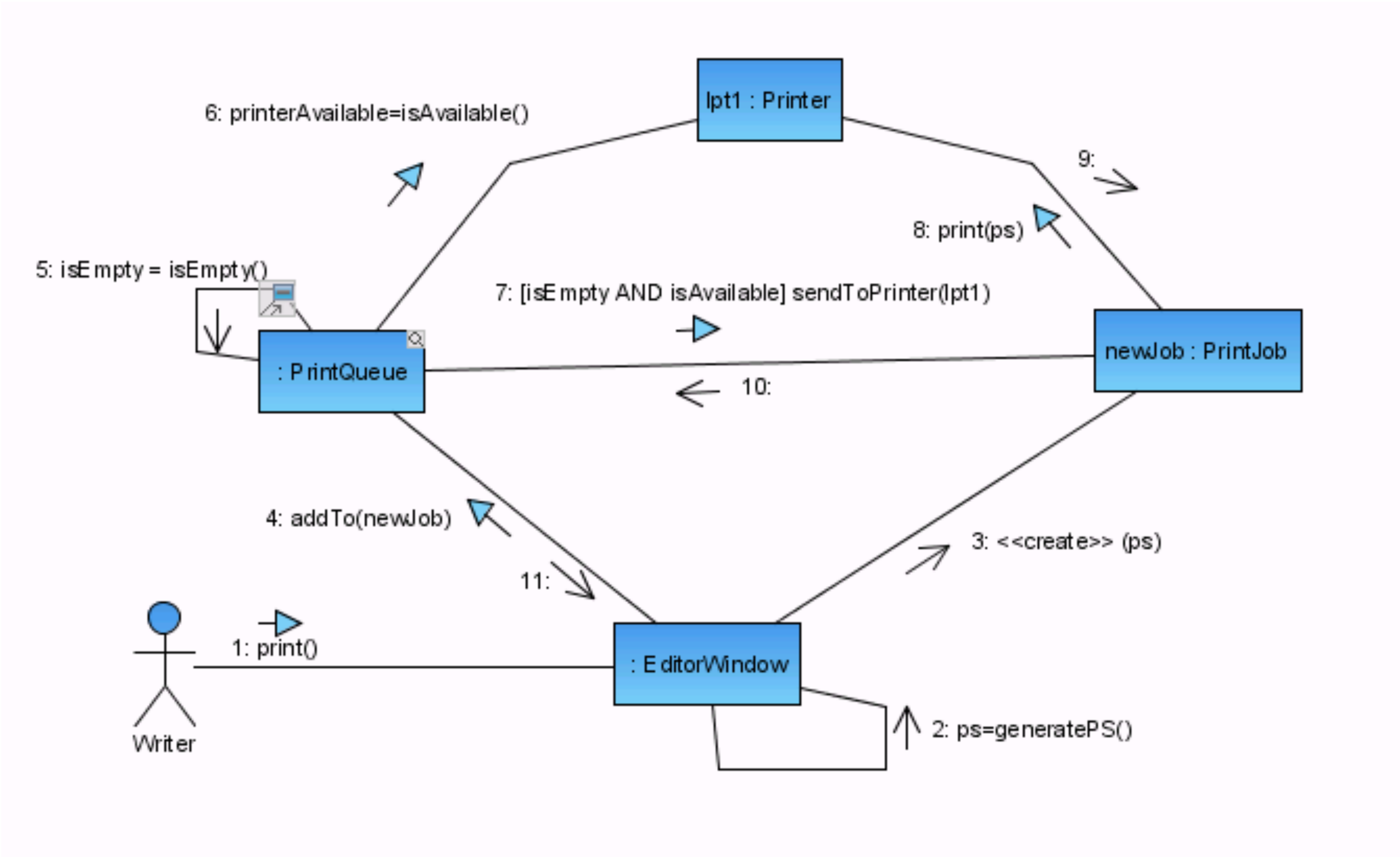
Communication diagrams

An interaction diagram notation emphasizing structure.

Communication diagram

- Collaboration diagrams is an isomorphic (same meaning, different syntax) form of sequence diagrams,
- They place the emphasis first on object structure, then on mesage sequence.
- A collaboration diagram shows both objects and instantiated associations between objects.
- Messages are shown as arrows placed along the instantiated associations.
- Message order is shown by a hierarchical numbering scheme.

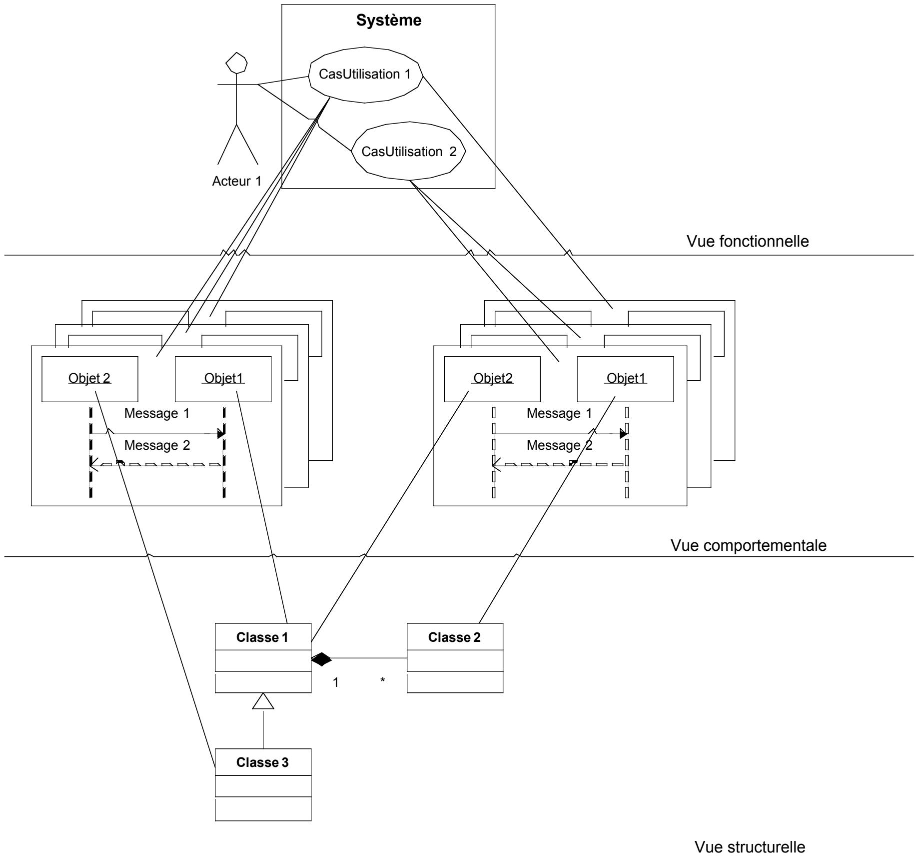
Communication diagram example



Message numbering

- Messages can be simply numbered in ascending order (as in example).
  - Does not show message nesting
- The decimal numbering scheme uses a new decimal point when a message is nested.
  - 1 {1.1, 1.2}, 2, 3 {3.1, 3.2 {3.2.1, 3.2.2}, 3.3}, 4, etc.
- Message order is never as clear as in a sequence diagram.

Use Case, Sequence and Class Diagrams



State machines

Modeling the global behaviour
of **a single object**

(Intra-Object view)

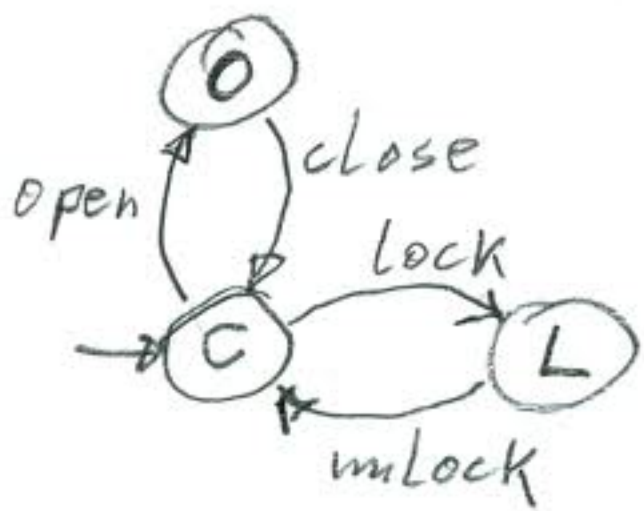
Modeling Behavior

- Sequence diagrams specify the behavior of a collection of objects to realize a specific functionality
  - An Inter-Object View
- But how to specify the behavior a single object?
  - State machine
    - An Intra-Object View

State Machine : Exemple

Door

open()
close()
lock()
unlock()



Two main concepts :

- State
- Transition

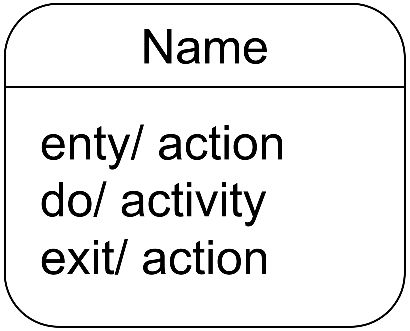
State

- A state is a "stage of existence" during which an object satisfies certain conditions, executes an activity, or waits for an event.
- The state defines how the object reacts to new events in its « life ».
- Examples :

Class	*Possible states*
Human « age »	child, teenager, adult
Washing machine « cycle »	pre-wash, wash, spin, dry
FTP Server « connexion state »	wait for connexion, send data, wait for acknowledgement, resend data ..
Inhabitant “marital state”	single, married, divorced, widowed

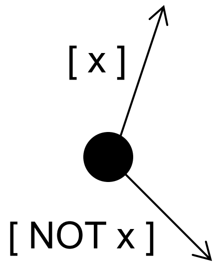
State

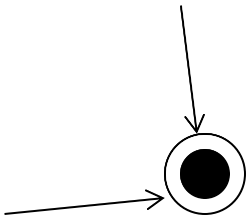
- A state is described by a rounded rectangle containing:
  - **Name**, unique within the class;
  - **Entry** and **Exit** actions, instantaneous processes executed by the object on entering/exiting the state;
  - An **activity**, long-lasting processes executed while the object is in that state.
    - Often has the same name as the state, in which case it can be omitted.



Initial and final states

- Every state diagram has exactly one initial state, and any number of final states.
- The initial state is the "point of creation" of the object.
  - The transition leaving it is taken when the object is instantiated.
  - This transition may contain a guard condition and an action, but not an event.
  - It may have multiple mutually-exclusive outgoing transitions, but no incoming transitions.
- The final state signifies that the object has terminated its execution, and has no further reason to exist.
  - It has no outgoing transitions.





Transition

- A transition describes how an object moves from one state to another.
- Syntax:

don't forget the slash !

event [guard condition] / action

- The transition is taken *when* the **event** is received *if* the **guard condition** is true.
- When it is taken, the **action** is executed by the object.

Transition

- A transition without an event or guard condition is considered to be "automatic". It is taken when the state finishes executing its activity process.
- An event corresponds to a message received from the environment or another object during an interaction. It may contain data parameters which are received by the object.
- The guard condition is a boolean expression based on the object's internal attributes and the event's parameters.
- The action is an instantaneous process executed by the object when the transition is taken.

Execution model

- When an object enters a state, it first executes the **entry** action.
- It then starts the **activity.**
- The activity may be interrupted by an **event** which causes the object to transition to another state if the guard is true.
- If the activity finishes without interruption, an **automatic transition** (if it exists) may triggered.
- Otherwise, it waits in that state until an outgoing transition is triggered by a new event or a guard condition becoming true.
- When the object leaves a state, the **exit** action is executed, followed by the **transition** action.

Timeouts

- Timeouts are a special kind of **event**, which occur *after* a parameterised period of time spent in a state.
- The timeout is reset and restarted every time a state is entered, and canceled whenever the state is exited.

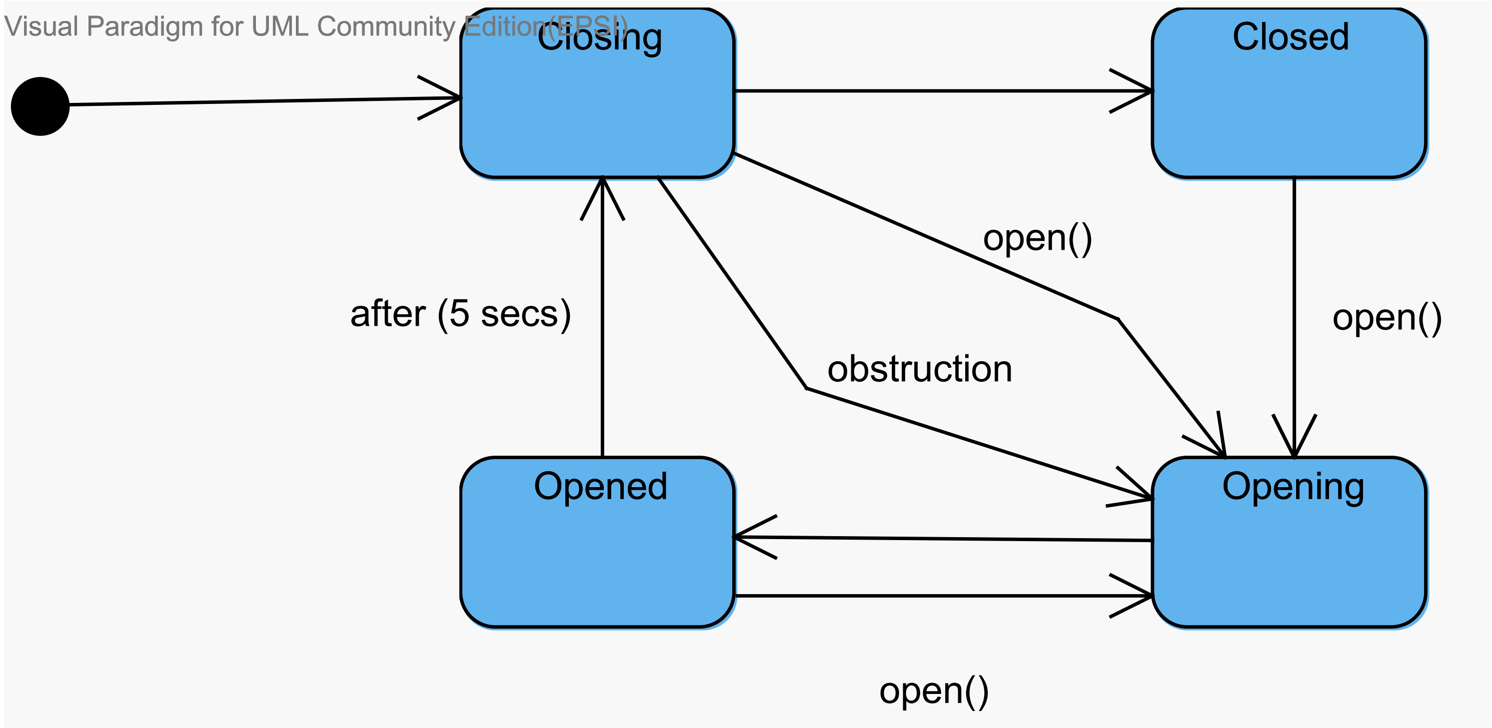
Timeout example

- A door which closes automatically after 5 seconds, and reopens when an obstacle is detected or the open button is pressed.

?

Timeout example

- A door which closes automatically after 5 seconds, and reopens when an obstacle is detected or the open button is pressed.

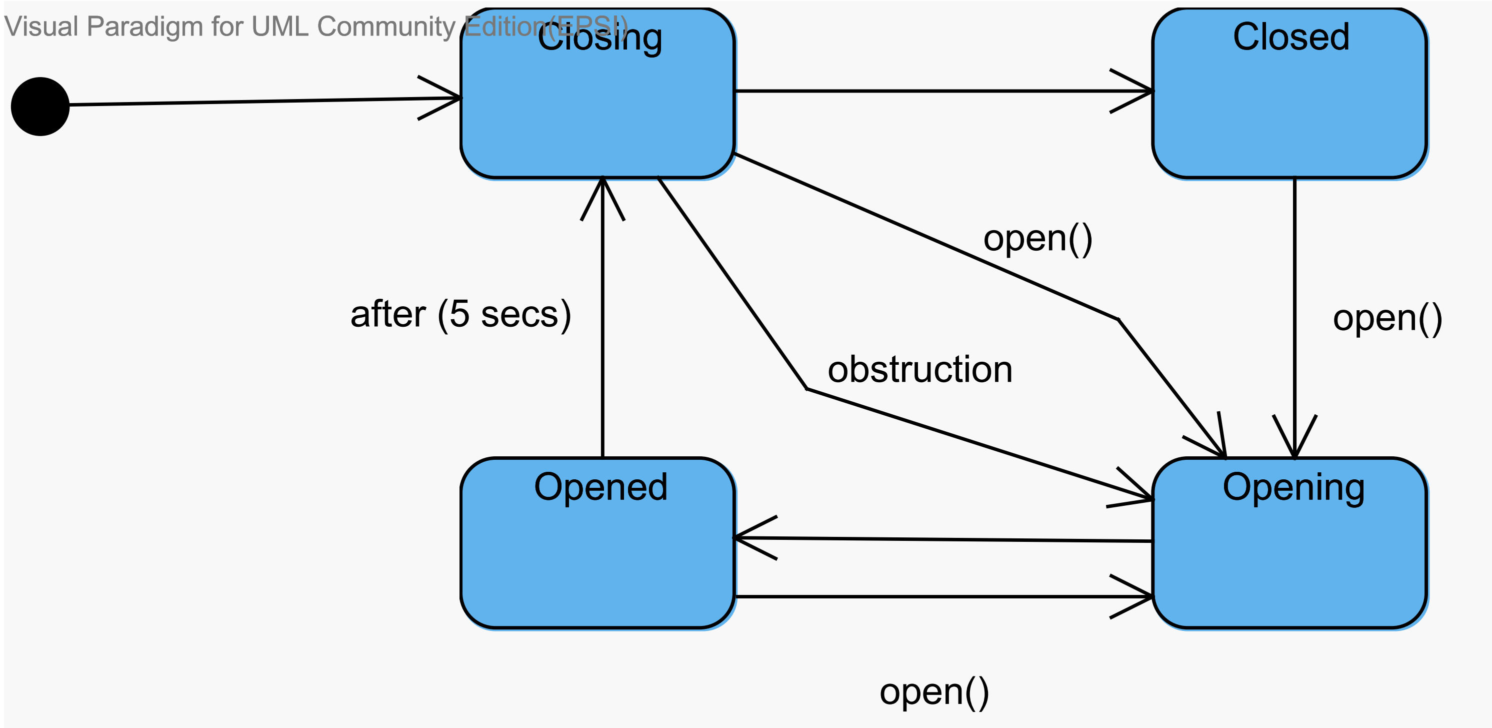


Composite-state

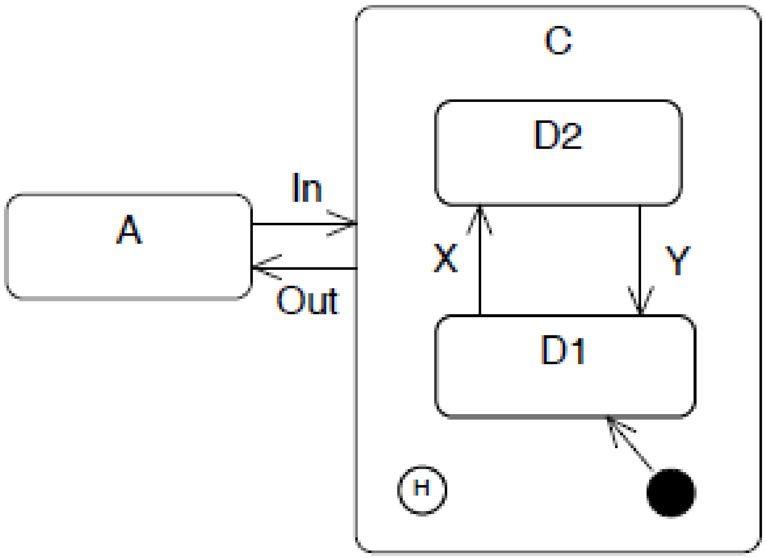
- Composite-states are a mechanism for factorising **actions**, **activities** and outgoing **transitions** that occur in multiple states of the same diagram.
- They can also be used to decompose a complex state and/or abstract the detail of the super-states to a separate diagram.
- When entering a composite-state, the system is placed in one of the sub-states, and normal execution continues.

Timeout example

- What could we factorise?

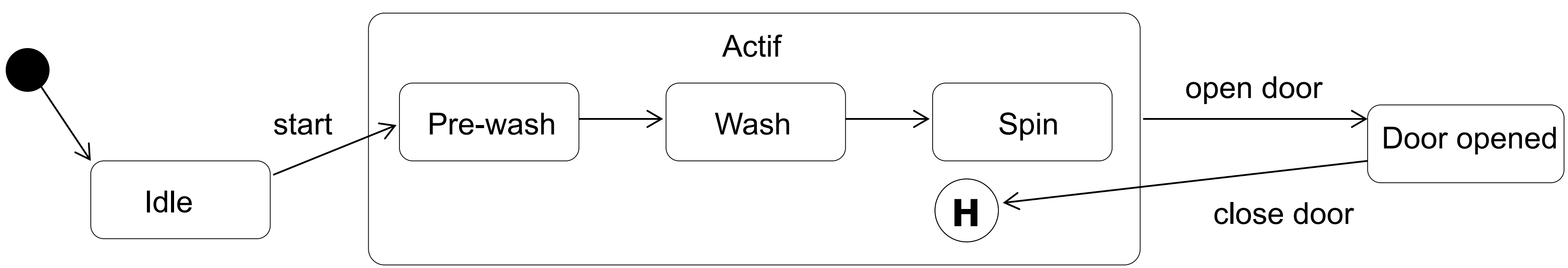


State machines: super-states



Super-state and history

- The **history** construct memorises the last active sub-state of a super-state.
- It enables the system to leave the super-state and later return to the same sub-state.
- Classic example :



Passive objects

- Not all objects have an interesting state.
- Although **all** objects have state, some objects never change state, I.e. they always respond in the same way to a request.
- These are called passive objects (as opposed to reactive objects), and we do not draw state diagrams for such objects.

- Do not use state machines to describe passive objects that always react in the same way (only one state).

Media Library System : TD

- Define sequence diagrams that specify the interaction to realize the functionalities :
  - Borrow a book
  - Return a book
- Define a state machine for the class copy.

Conclusion

- Sequence diagrams
  - An inter-object view
  - To specify the behavior of the system to realize a specific functionality (a use case)
- State machines
  - An intra-object view
  - To specify the complete behavior of an entity (class, sub-system)