# Software Engineering - Introduction to Behavior Modeling in UML

L.EIC-ES-2021-22

**J. Pascoal Faria, Ademar Aguiar**

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

# Agenda

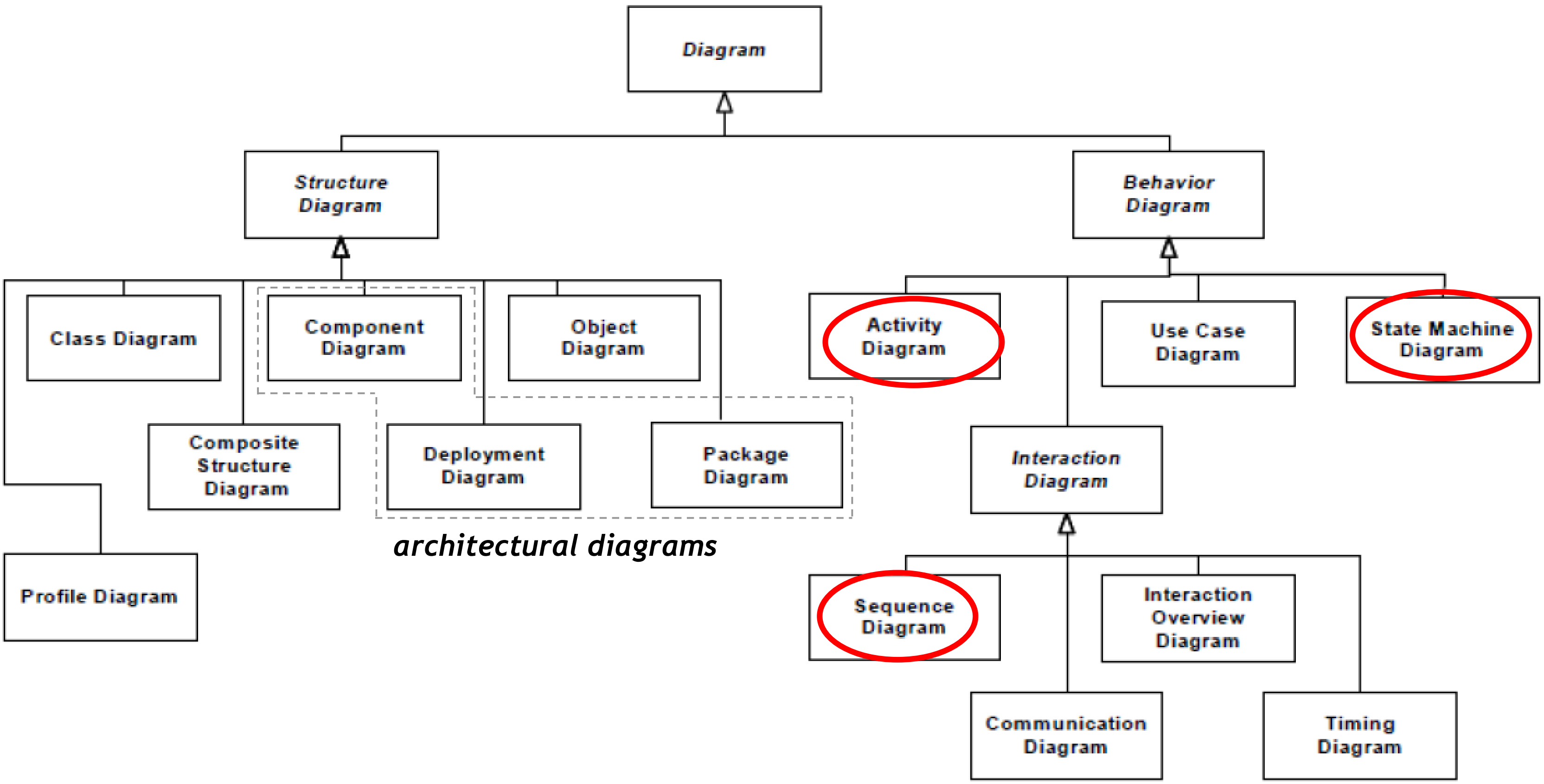
- Introduction
- Sequence diagrams
- State machine diagrams
- Activity diagrams
- Exercises

# Introduction

# Motivation

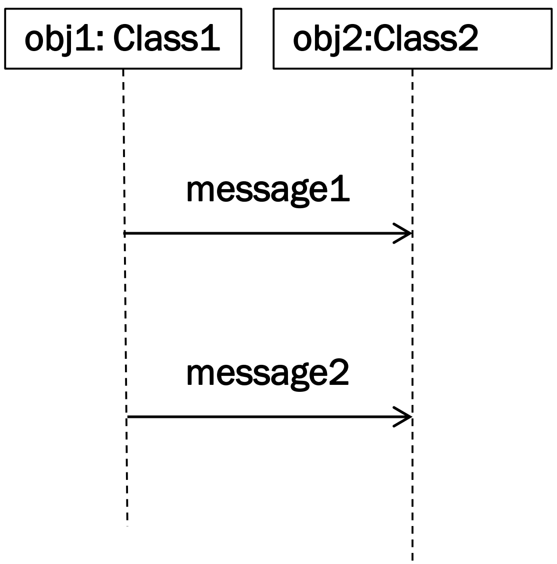
- Models are used in all fields of Engineering to tackle complexity through abstraction
- UML is the modeling standard in Software Engineering
- So far, we have studied UML diagrams for
  - Use case modeling (system functionality)
  - Domain modeling (domain concepts and entities)
  - Architecture modeling (physical and logical architecture)
- Here, we briefly study three types of UML behavioral diagrams for describing *how a system works*
  - Sequence diagrams
  - State machine diagrams
  - Activity diagrams

# UML diagrams



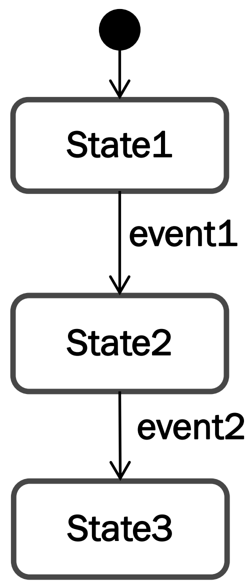
# Behavior modeling

Sequence diagram



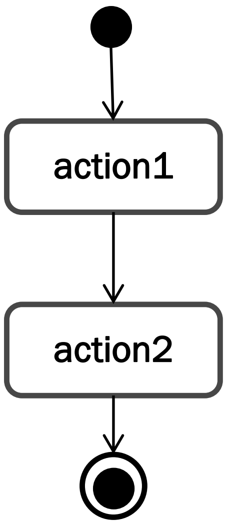

Emphasizes interactions (between objects, systems, actors, components, etc.)

State machine diagram




Emphasizes states & transitions of an object or system

Activity diagram



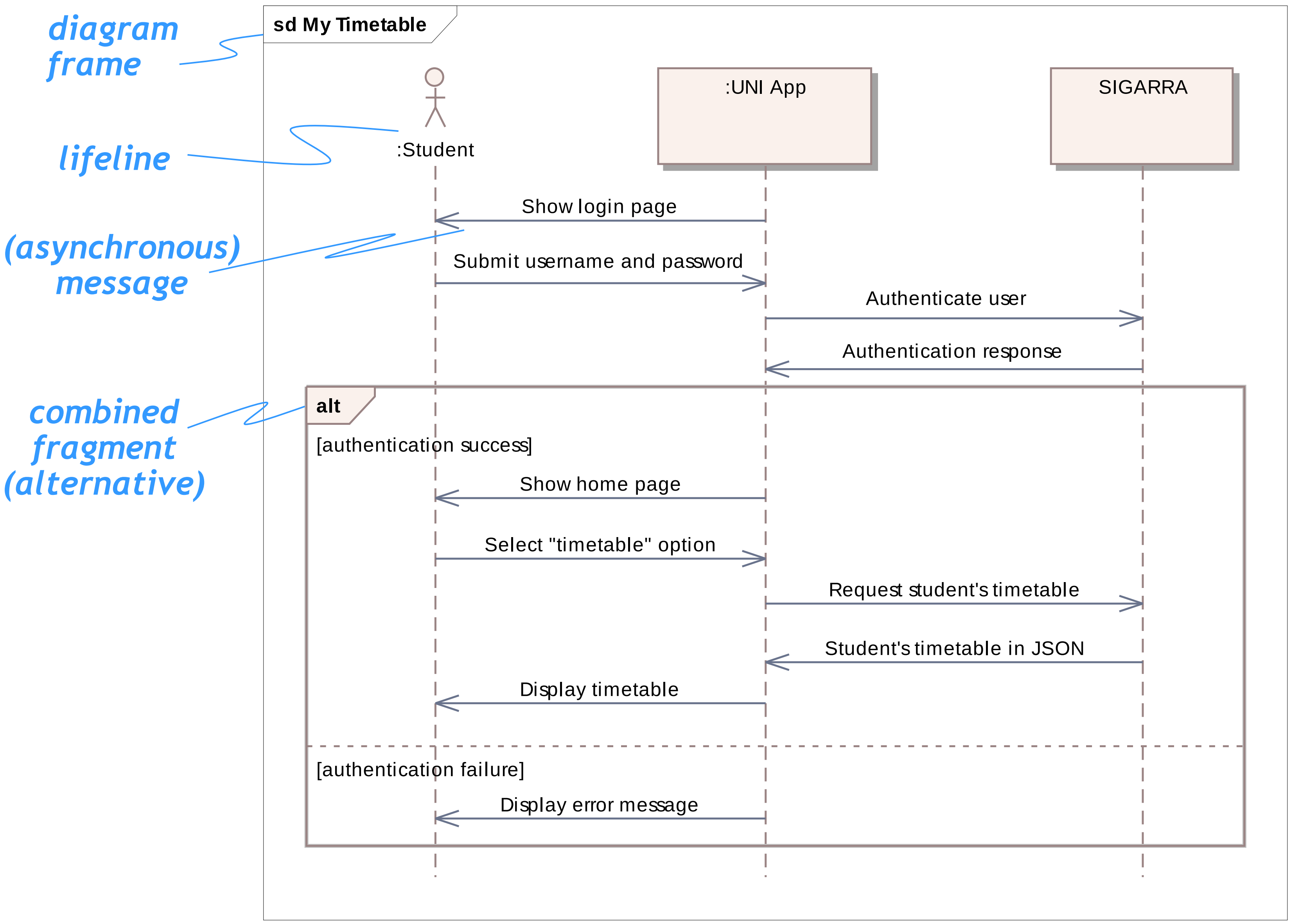

Emphasizes processing steps

# Sequence diagrams

# Sequence diagrams

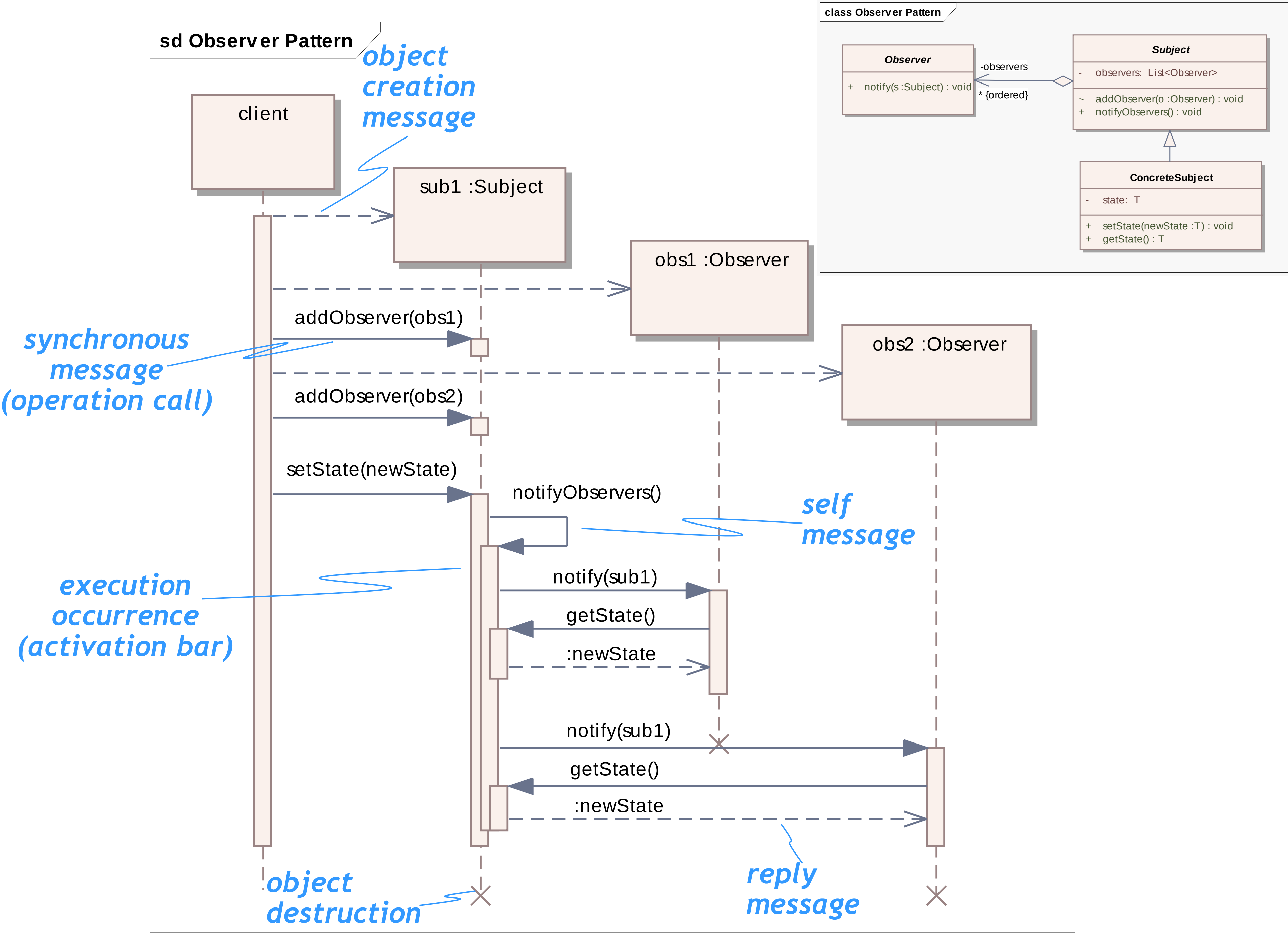
- Show the interaction (messages exchanged along time) between a set of participants in a given context
- Context may be:
  - Use case (e.g., get student timetable)
  - Mechanism (e.g., observer pattern)
  - Scenario
  - Operation of a class
  - Etc.
- Participants may be:
  - Actors
  - Systems
  - Components
  - Objects
  - Etc.

# Example 1: Get student timetable

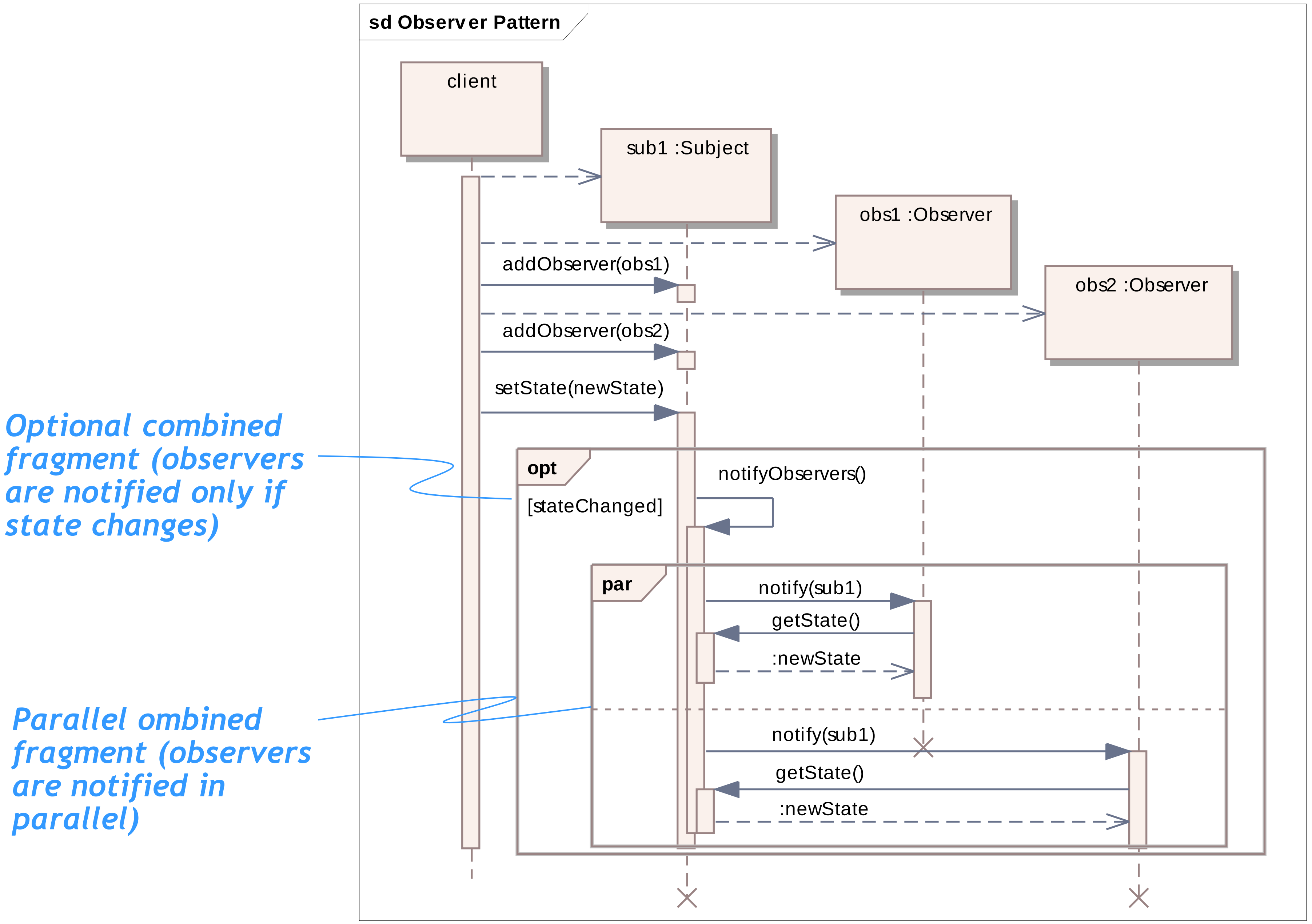


# Example 2: Observer pattern

- The **observer pattern** is a software design pattern in which an object, named the **subject**, maintains a list of its dependents, called **observers**, and notifies them automatically of any state changes, usually by calling one of their methods. [source: GoF book]
- Example:
  - Subject: car park occupancy
  - Observers: students interested in parking their cars
- Next slides show two possible designs:
  - Sequential notification
  - Optional, parallel, notification



sd Observer Pattern
object creation message
client
sub1 :Subject
obs1 :Observer
obs2 :Observer
addObserver(obs1)
synchronous message (operation call)
addObserver(obs2)
setState(newState)
notifyObservers()
self message
execution occurrence (activation bar)
notify(sub1)
getState()
:newState
notify(sub1)
getState()
:newState
reply message
object destruction
class Observer Pattern
Observer
+ notify(s :Subject) : void
-observers
* {ordered}
Subject
- observers: List<Observer>
~ addObserver(o :Observer) : void
+ notifyObservers() : void
ConcreteSubject
- state: T
+ setState(newState :T) : void
+ getState() : T

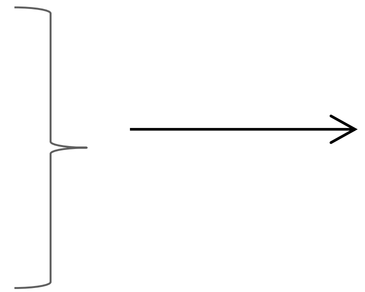


sd Observer Pattern
client
sub1 :Subject
obs1 :Observer
obs2 :Observer
addObserver(obs1)
addObserver(obs2)
setState(newState)
opt
[stateChanged]
notifyObservers()
par
notify(sub1)
getState()
:newState
notify(sub1)
getState()
:newState
Optional combined fragment (observers are notified only if state changes)
Parallel ombined fragment (observers are notified in parallel)

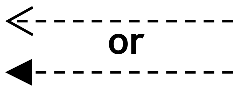
# Types of messages

- synchCall: synchronous operation call (the caller is blocked until the operation returns)

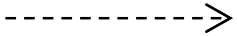
- asynchCall: asynchronous operation call (the caller is not blocked)
- asynchSignal: asynchronous sending and reception of as instance of a signal



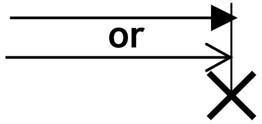
- reply: return from an operation call




- createMessage: object creation message (e.g., constructor call)



- deleteMessage: message causing object destruction (e.g., destructor call)



# Combined fragments

- Combined fragments allow creating structured sequence diagrams, with alternatives, repetition, parallelism, etc.
- A combined fragment has an interaction operator (determ- ining its semantics) and one or more interaction operands:
  - opt - Optional execution of the (single) operand, depending on a guard condition
  - alt - Alternative execution of the (multiple) operands, depending on their respective guard conditions
  - loop - Repeated execution of the (single) operand, depending on an iteration expression
  - par - Parallel execution of the operands
  - Other: seq, strict, critical, neg, assert, consider, ignore

# State machine diagrams

# State machine diagrams

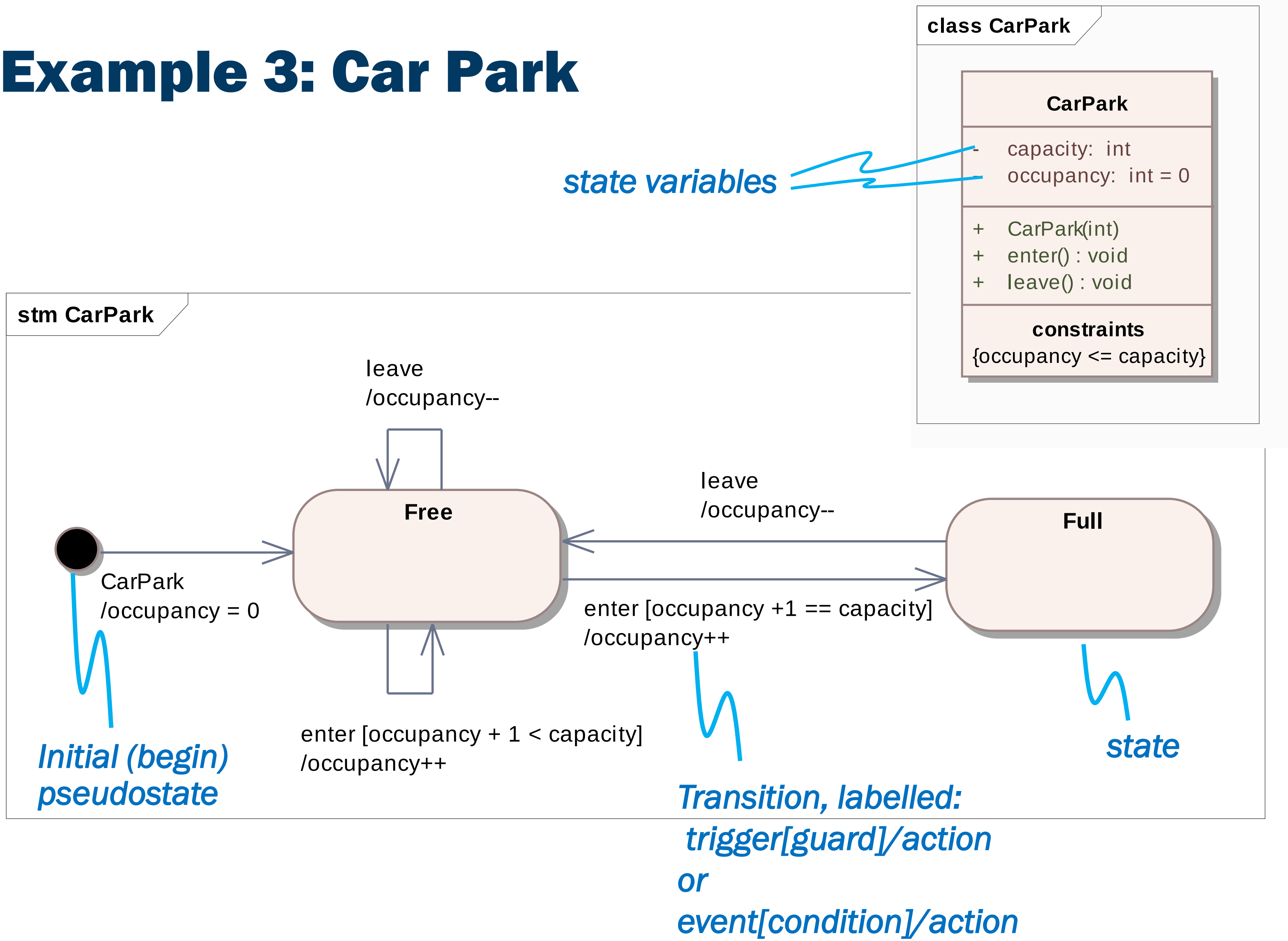
- Useful for modeling the lifecycle of objects or systems with a discrete event-driven behavior, showing:
  - Possible states of the object or system (finite)
  - Transitions between states (usually instantaneous)
  - Events that trigger transitions
  - (Opt.) Actions taken by object/system in response to an event
  - (Opt.) Activities performed by object/system while in a state
- Applicable in different contexts, such as:
  - Behavior of an interactive component (e.g., Button)
  - Navigation map of a user interface (e.g., a Site Map)
  - Behavior of a computer controlled system (e.g., Traffic Lights)
  - Lifecycle of a business object (e.g., a Car Rental)

# Extensions over finite automata

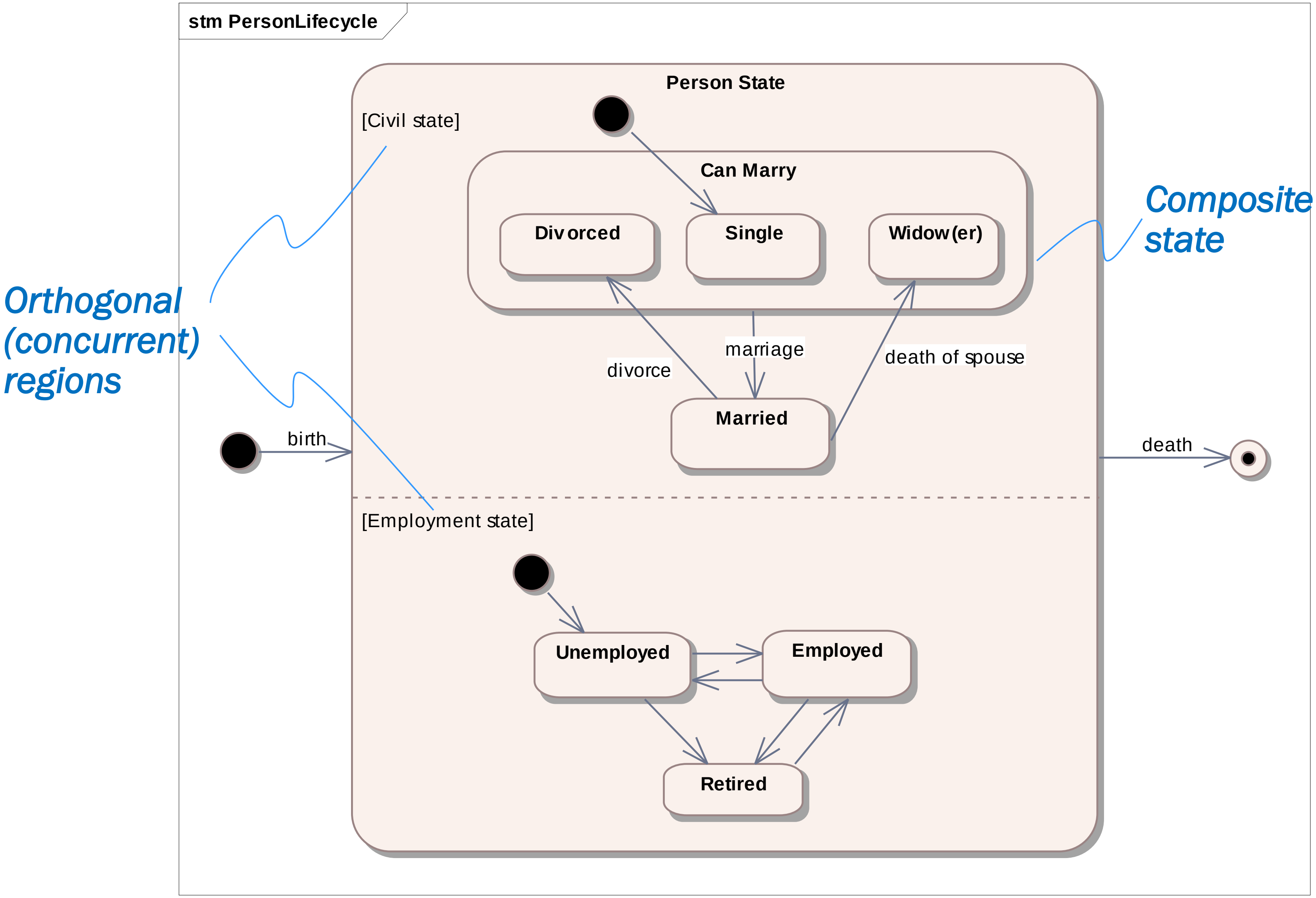
For the sake of scalability, state machine diagrams provide the following extensions as compared to finite automata:

- State variables
  - E.g., an object instance variables (when modeling its lifecycle)
  - The diagram only distinguishes high-level states, where the behavior of the object is significantly different
    - E.g., different operations available, different effects, etc.
- Composite states
  - “Or” composition (of states in the same composite state)
  - Useful to avoid the combinatorial explosion of transitions
- Orthogonal regions (concurrent regions)
  - “And” composition (of states in different orthogonal regions)
  - Useful to avoid the combinatorial explosion of states

# Example 3: Car Park



# Example 4: Civil & Employment State



# Types of events

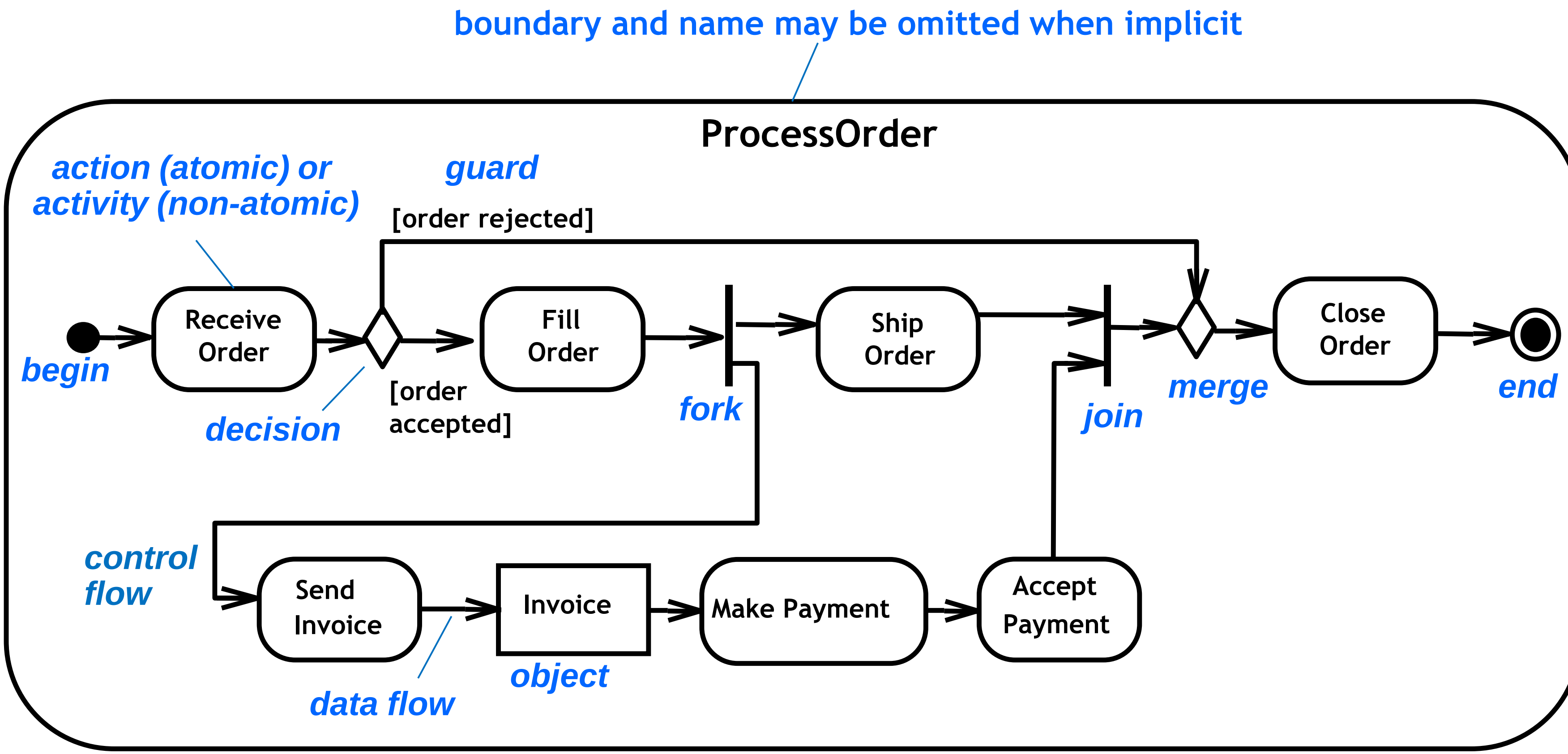
- **Call**: Operation call, usually synchronous
- **Signal**: Symbolic event, modeled as object that is sent asynchronously by an object and received by another object
- **Time events:**
  - after(t) – occurs after *t* time elapsed since entering the source state
  - when(t) – occurs at time instant *t*
- **Change events**
  - when(condition) – occurs when the condition on the internal object/system state becomes true

# Activity diagrams

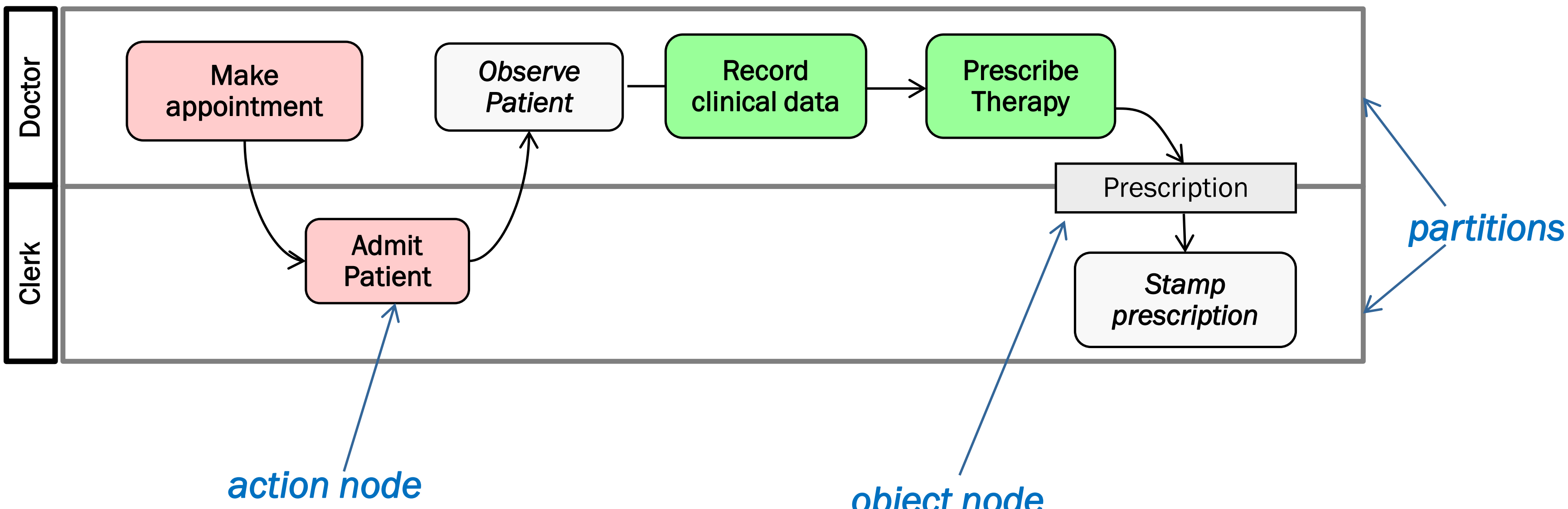
# Activity diagrams

- An activity diagram shows the hierarchical decomposition of an activity (a non-atomic behavior) into a set of subordinate units (ultimately atomic actions), and their coordination using a mixed control and data flow model
- Activity diagrams may be seen as an extension of flowcharts, with features such as:
  - Concurrency - parallel execution
  - Partitions – split activity between participants (objects, business units, actors, etc.)
  - Object nodes and data flow
- May be used to describe
  - Algorithms, business processes (see next examples), etc.

# Example 5: Process Order



# Example 6: Medical consultation




Combines manual (white) and computer-based (colored) activities

# Exercises

- Describe the normal scenario of a use case from your use case model using a UML sequence diagram

- Describe the lifecycle of an object from your domain model using a UML state machine diagram

- Describe a workflow supported by your app using a UML activity diagram
  - nodes may represent the execution of use cases

# References and further info

- www.uml.org – OMG® Unified Modeling Language® (OMG UML®) Version 2.5.1
- Software engineering, 10th edition, Ian Sommerville, Chapter 5 – System Modeling
- http://www.sparxsystems.com.au/resources/uml2_tutorial/
- http://www.agilemodeling.com