Unit 2
Design

Software Analysis and Design Project

Universidad Autónoma de Madrid

Index

Object Orientation

- The fundamental blocks in the structured programming paradigm (C, Pascal) are operations.
  - Functions and data are separate from each other.
  - This is useful for small scale systems (≈5000 LOC)

- The object oriented paradigm focuses equally on both data and operations.
  - Objects: Single units that agglutinate state information and operations.
  - “structs (C-style)+functions”.

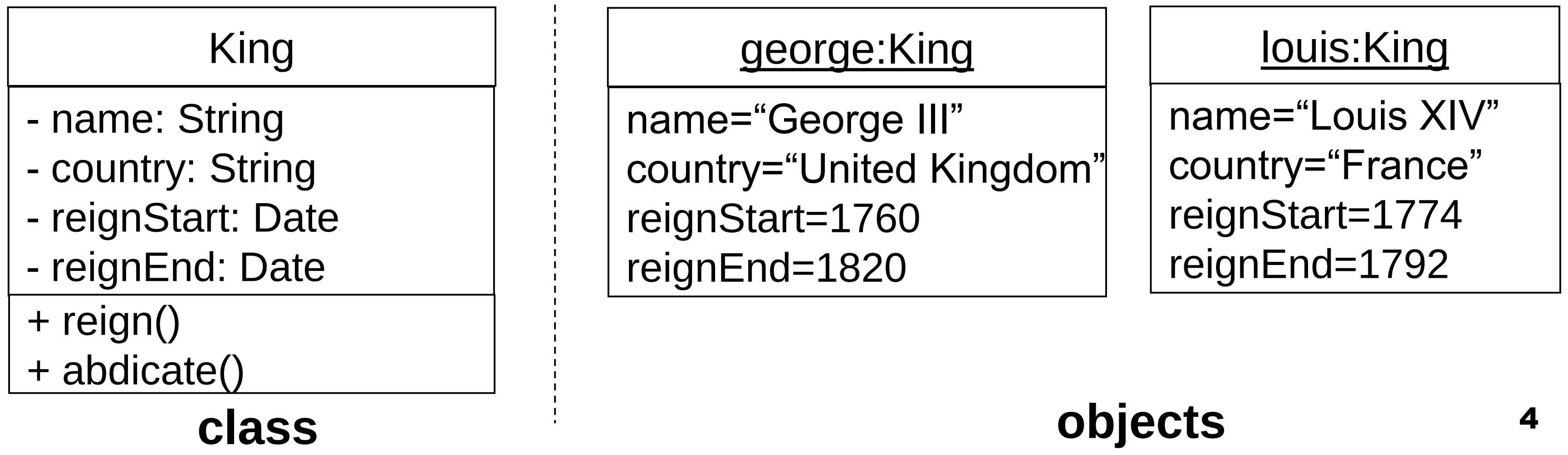
Object Orientation

Classes and Objects

name: George III
country: United Kingdom
reignStart: 1760
reignEnd: 1820

name: Louis XIV
country: France
reignStart: 1774
reignEnd: 1792

- All kings have common aspects.
- We can represent those aspects in a class.
- Specific kings would be instances (objects) of the class.



Object Orientation

Classes and Objects

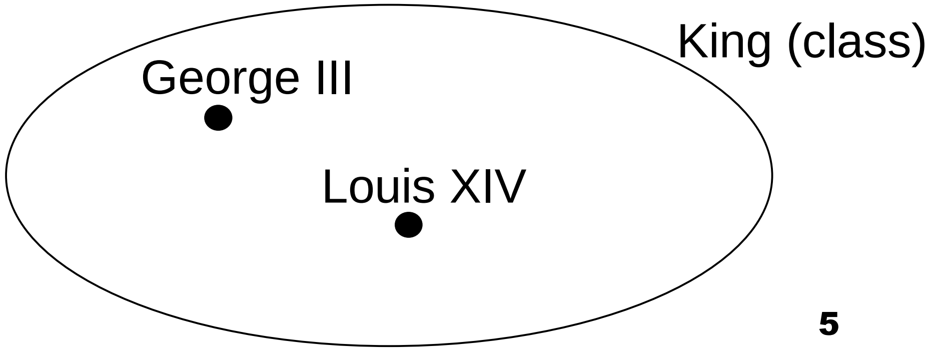
Class

- Declares ***properties*** (attributes) of all its instances.
- Declares ***operations*** (methods) that can be applied to all its instances.

Object

- Includes values for the attributes declared in the class.
- Reacts to invocations of the methods declared in the class, using the values of its attributes as state

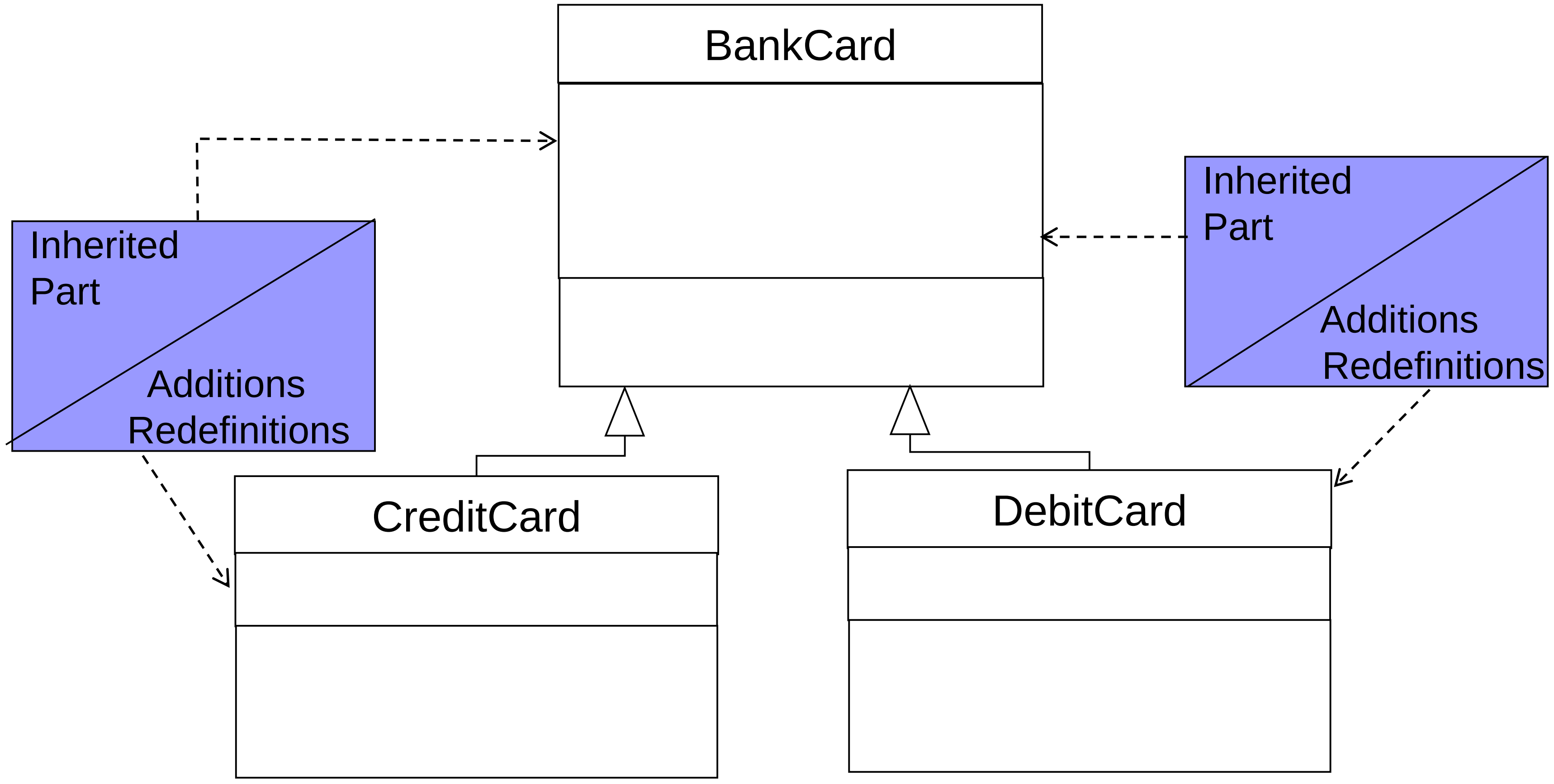
In terms of sets, we can understand a class as the definition of the set of all its instances



Object Orientation

Inheritance

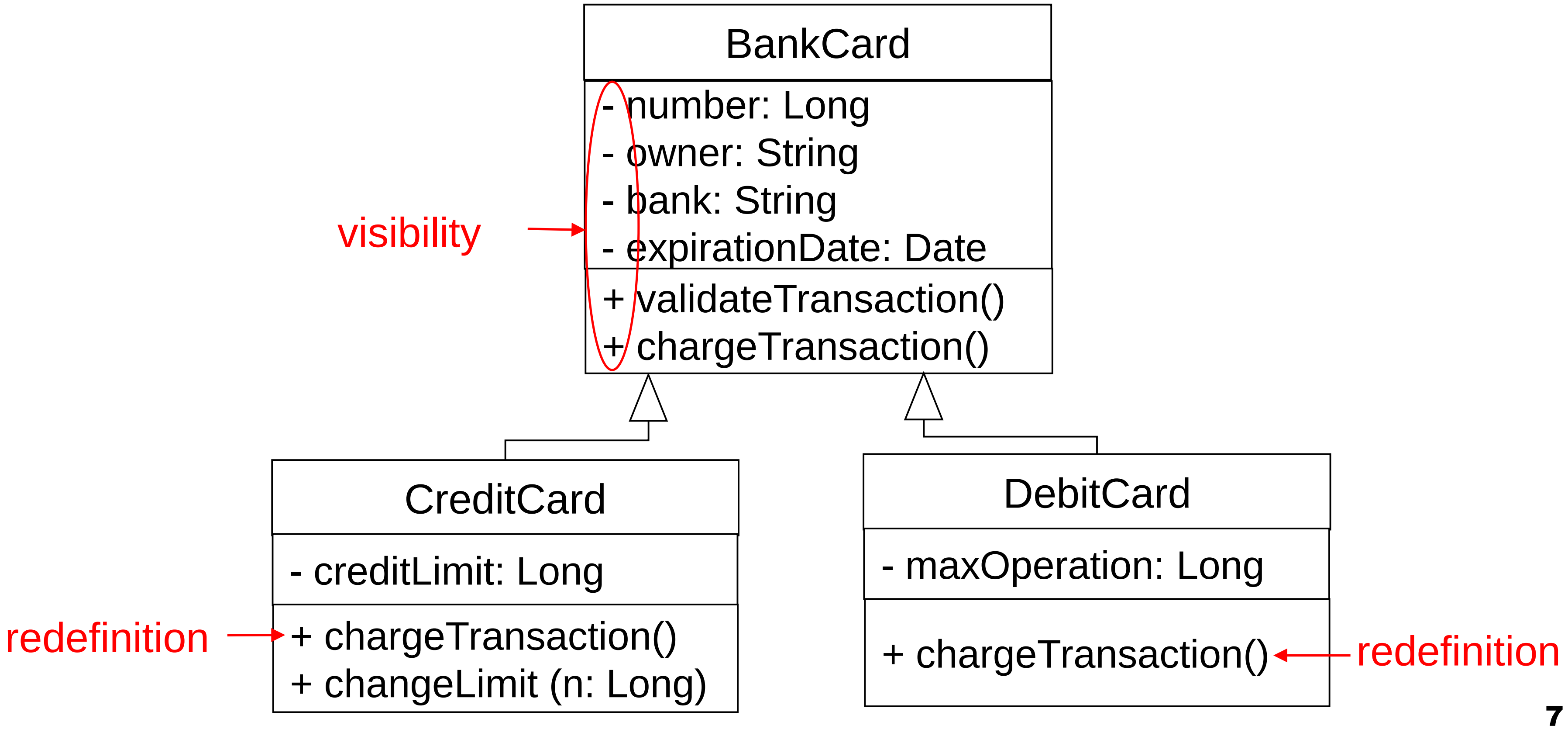
- Hierarchies of class specialization.
- Inheritance of properties and operation.



Object Orientation

Inheritance

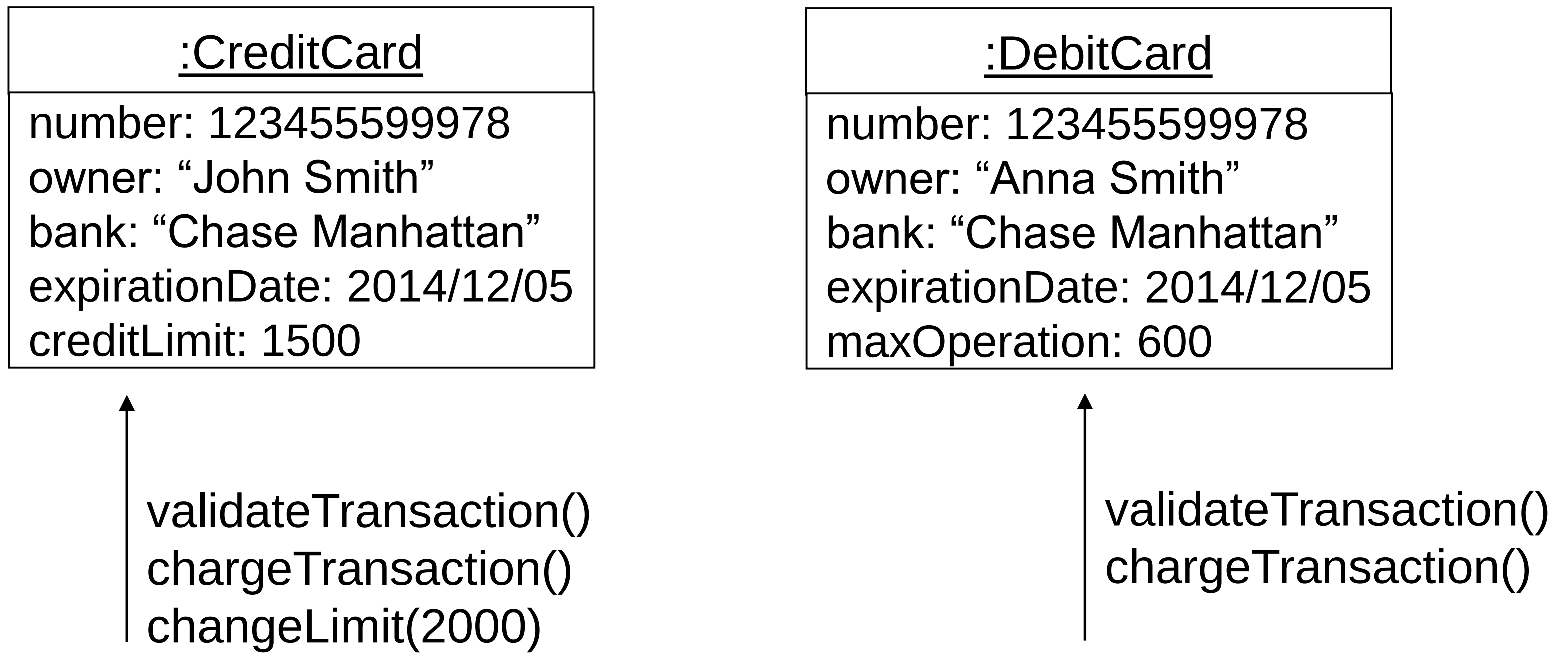
- Hierarchies of class specializations.
- Inheritance of properties and operation.



Object Orientation

Inheritance

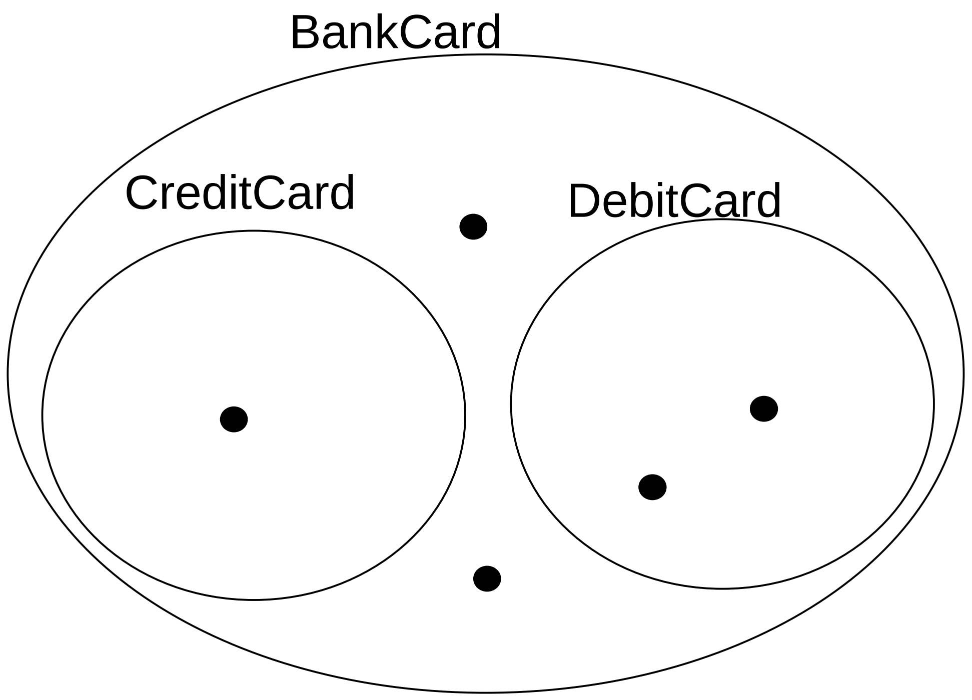
- An object of a subclass inherits the properties of the superclass.
- We can invoke operations defined in the superclass.



Object Orientation

Inheritance

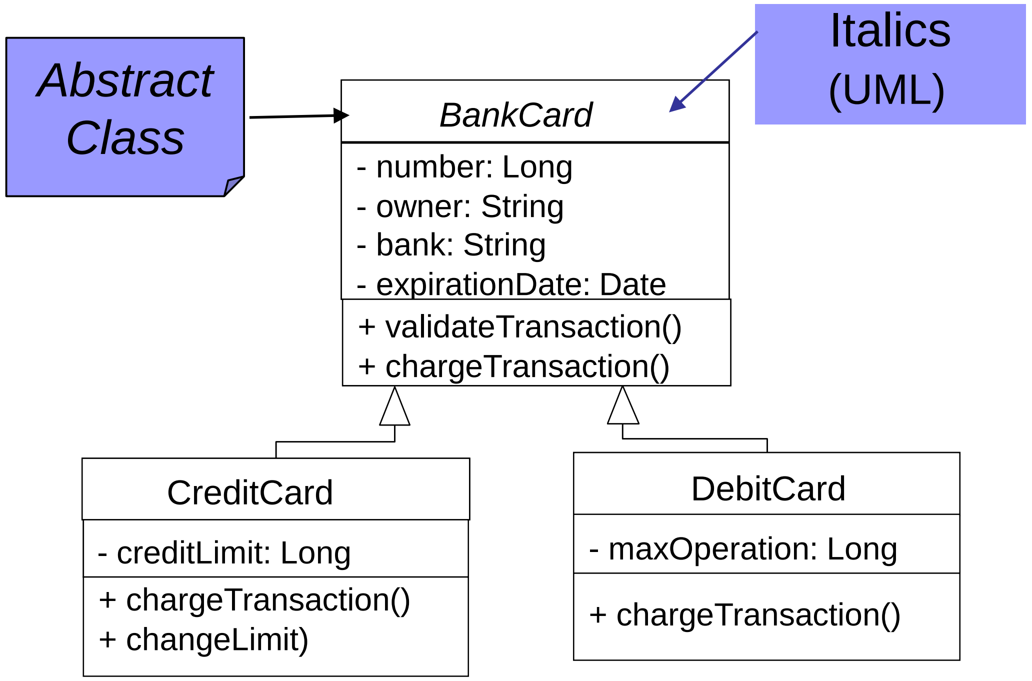
- Safe substitution of supertypes by subtypes.
  - All credit and debit cards are bank cards.
  - There are bank cards that are neither credit nor debit cards (base class is not abstract).
  - No cards are simultaneously credit and debit cards (simple inheritance in this case instead of multiple inheritance).

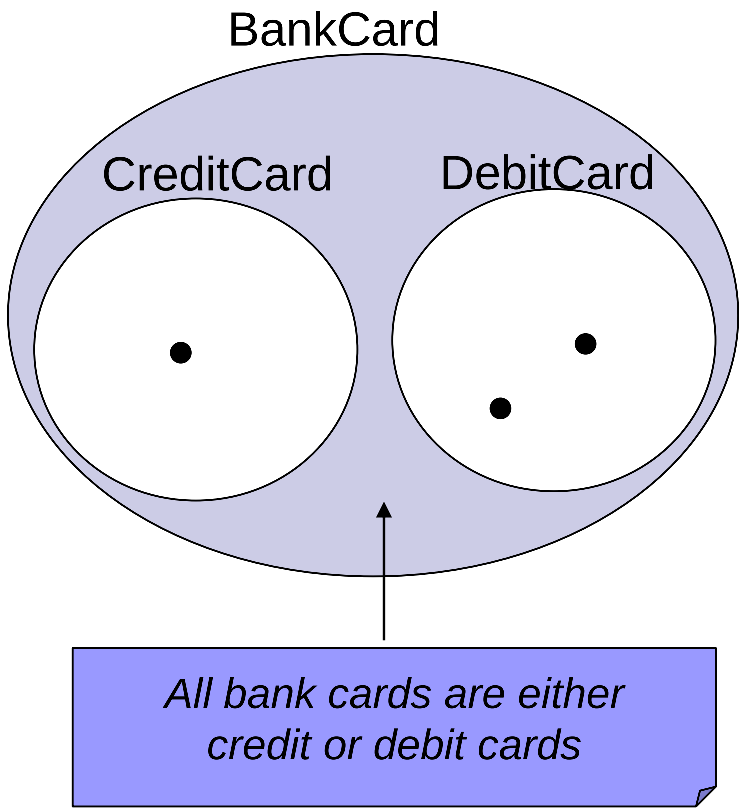


Object Orientation

Abstract Class

- An abstract class cannot be instantiated

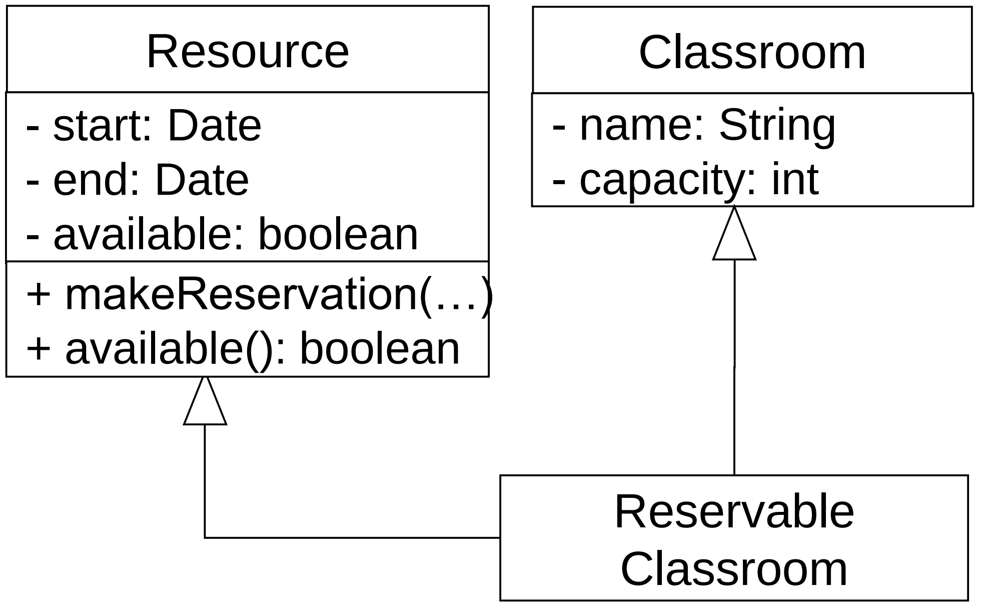


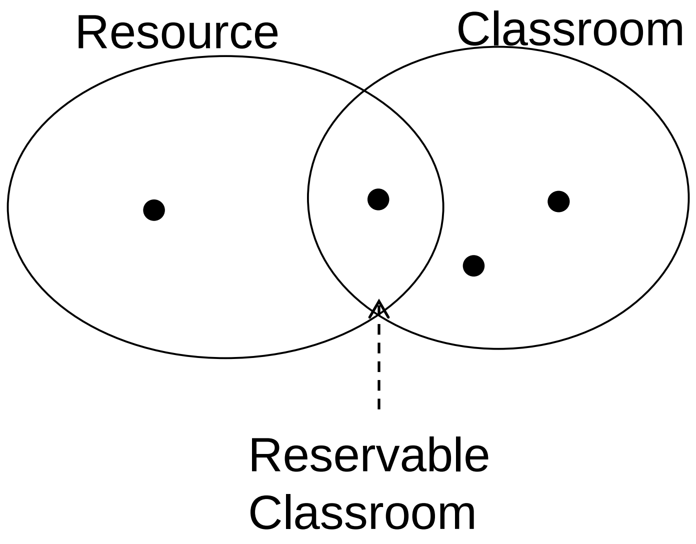


Object Orientation

Multiple Inheritance

- In general, classes can inherit from several superclasses.
- Not allowed in Java. It is allowed in some other programming languages like C++.





Object Orientation
Associations

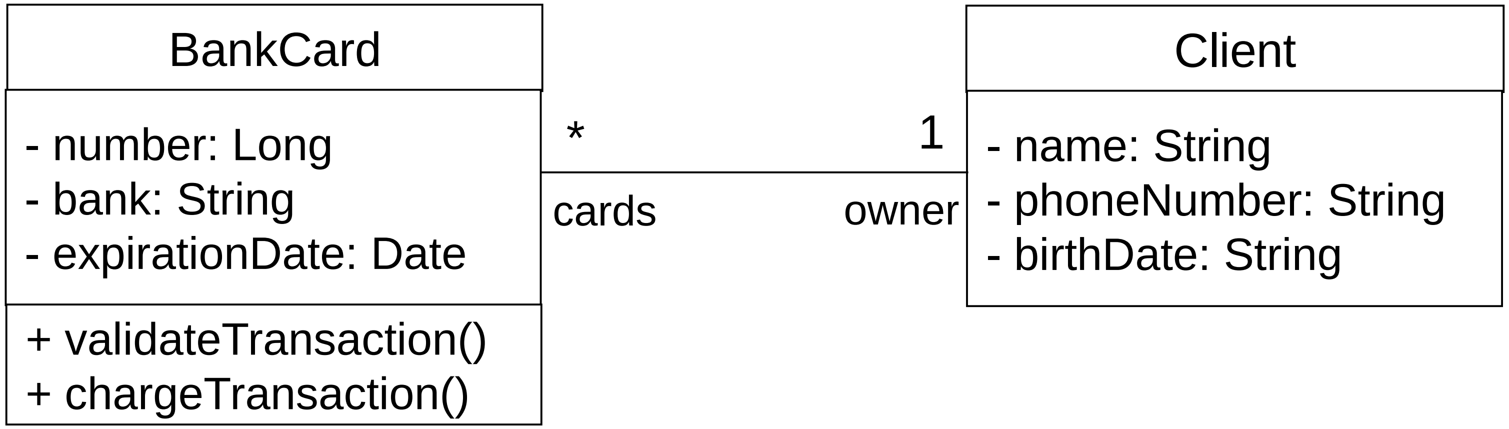
- Objets are not isolated. They must “know each other” in order to be able to invoke methods from other objects.
  - The application functionality is implemented by means of collaborations between different objects.

- Conceptually object association is represented by means of a link (a line) between two classes.
  - In code, this amounts to an object having a reference to another object (an attribute of the first one).

Object Orientation

Associations

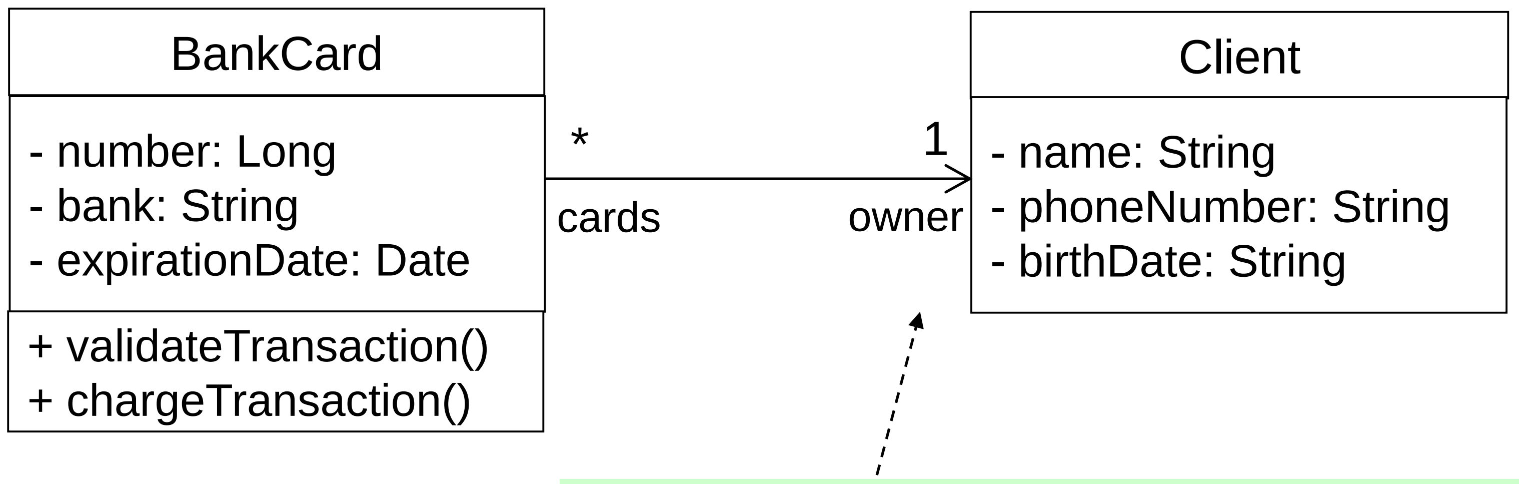
- Multiplicity:
  - Allowed interval of objects that can be related to a source object (and vice-versa).
- Roles:
  - Names of the association ends.



Object Orientation

Associations

- Navigation: Indicates whether an object can access the objects at the other end.

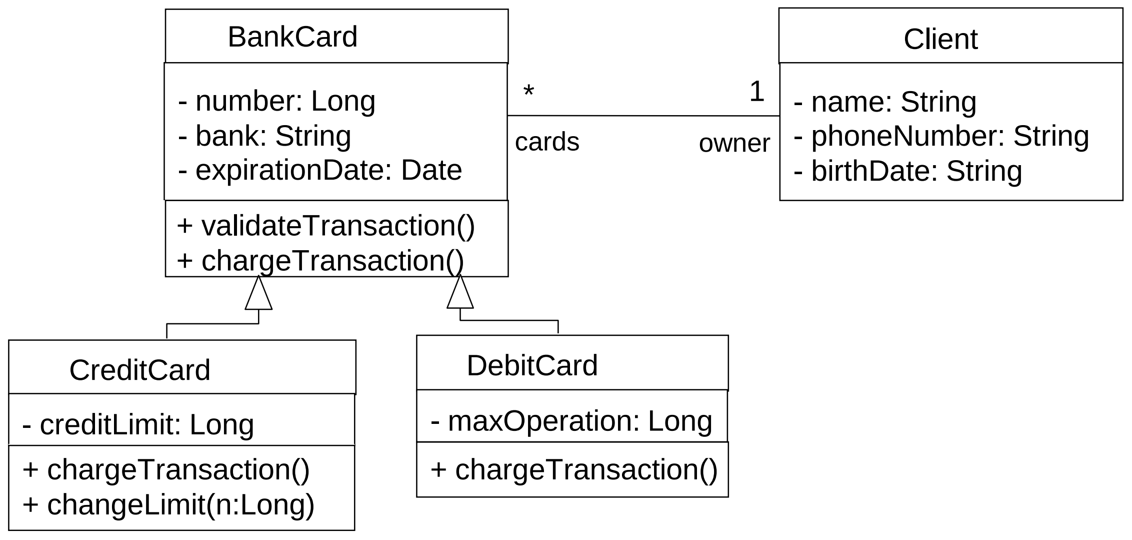


From an object of type BankCard we can access its owner, but not the other way around

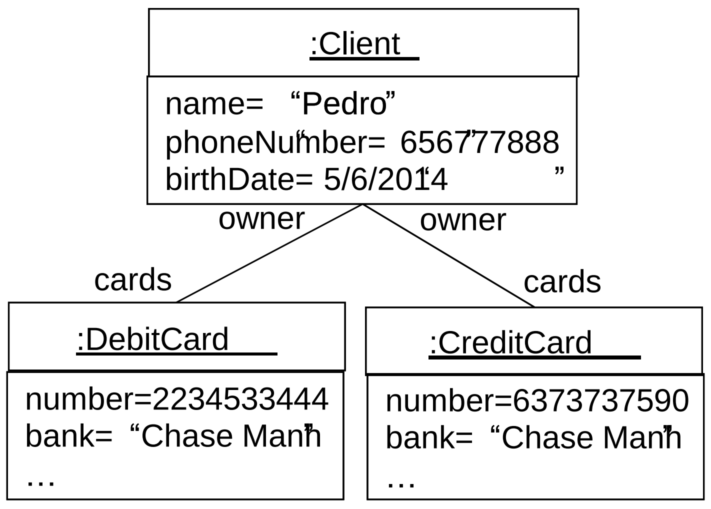
Object Orientation

Inheritance of Associations

- An association that is declared in a base class is inherited by each subclass.



Class diagram

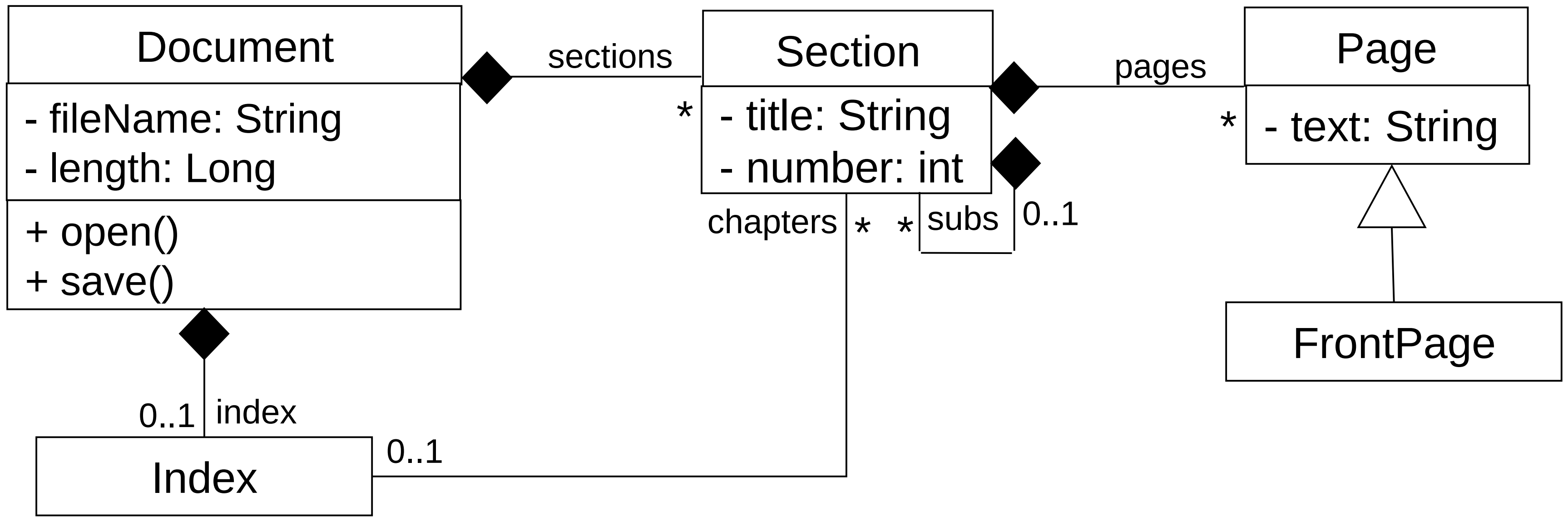


Object diagram

Object Orientation

Composition and Aggregation

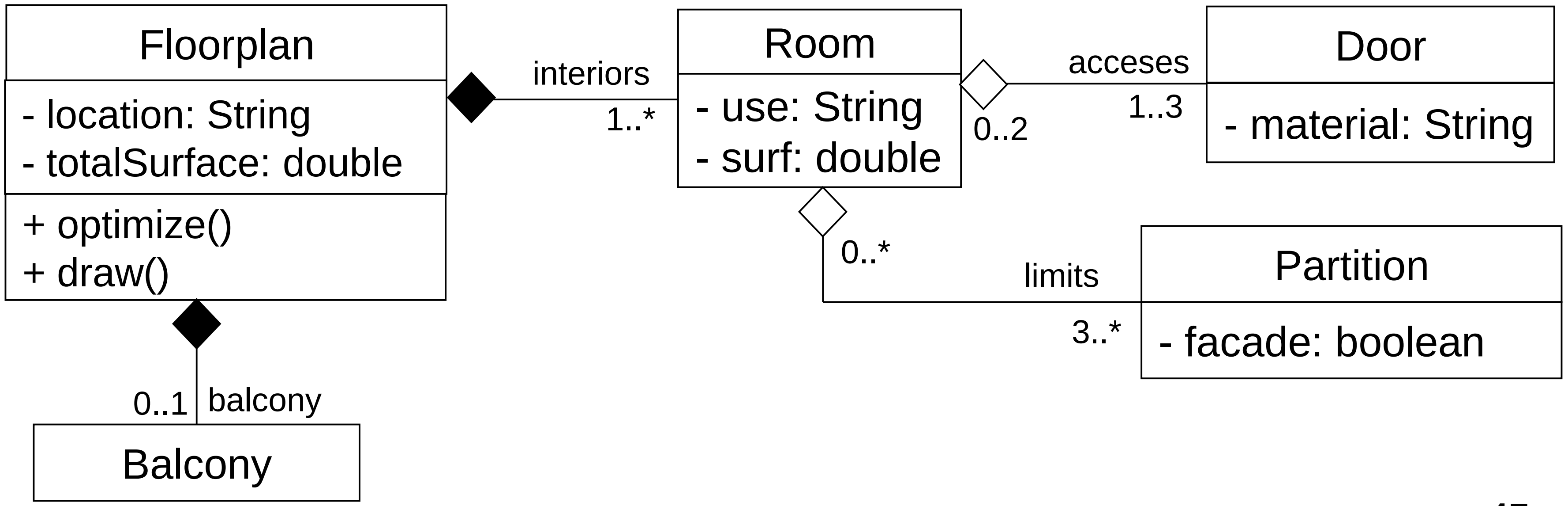
- Some relations have special semantics:
  - ◆ Composition (a class made up of different parts)
  - ◇ Aggregation (weaker composition)



Object Orientation

Composition and Aggregation

- Some relations have special semantics:
  - ◆ Composition (a class made up of different parts)
  - ◇ Aggregation (weaker composition)



Project

- Build the class diagram for the project.

Index

Behaviour modelling

- A class diagram describes the structure of the application, but it does not describe its behaviour:
  - Which actions are performed by each method?
  - How is the state of an object changed when methods are invoked?
  - What are the allowed order for invoking methods?
  - How do several objects collaborate among themselves in order to perform a task?

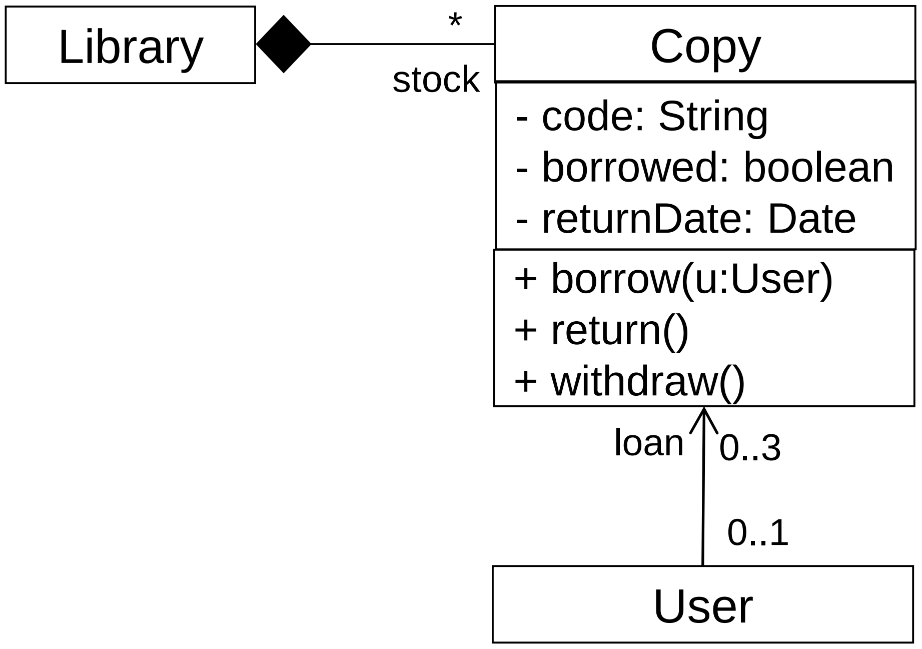
Behaviour modelling

Behaviour diagrams

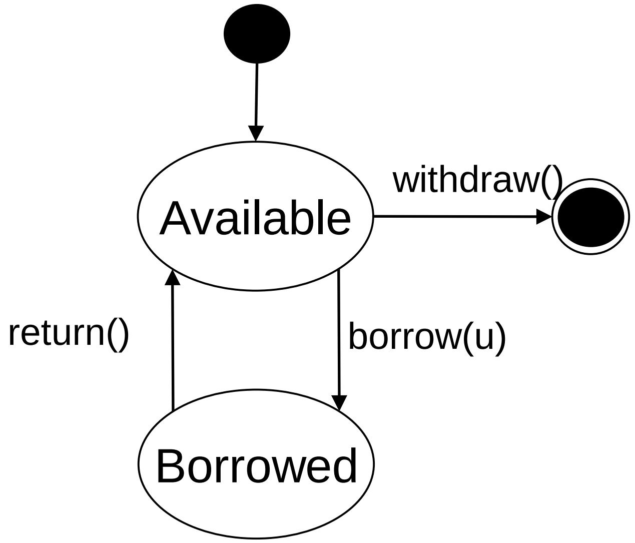
- A class diagram describes the structure of the application, but it does not describe its behaviour:
  - Which actions are performed by each method?
    - Pseudocode (“action semantics” language).
    - Activity diagram
  - How is the **state of an object** changed when methods are invoked?
    - **State transition diagram** (“statecharts”).
  - How do **several objects** collaborate among themselves in order to perform a task?
    - **Sequence diagram**.
    - Collaboration/communication diagram.

State transition diagram

- Associated to a class.
- Describes its evolution when their methods are invoked.
- Similar to a finite automaton.



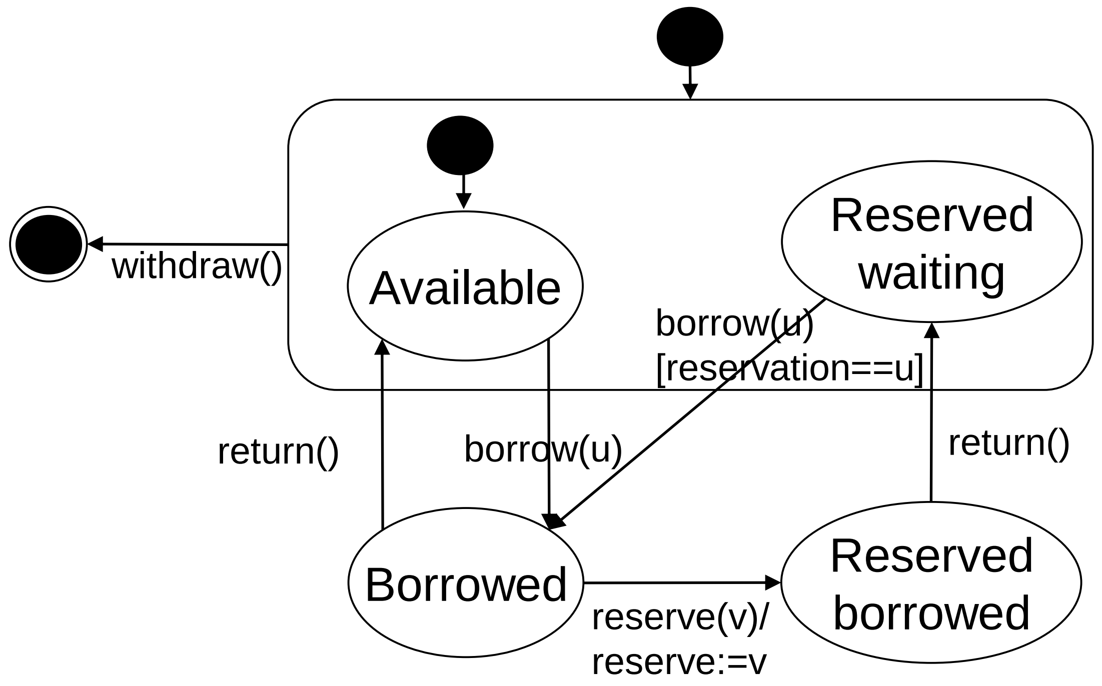
Class diagram



State transition diagram (class: Copy)

State transition diagram

- Hierarchy of states.
- Transitions with guards and actions.



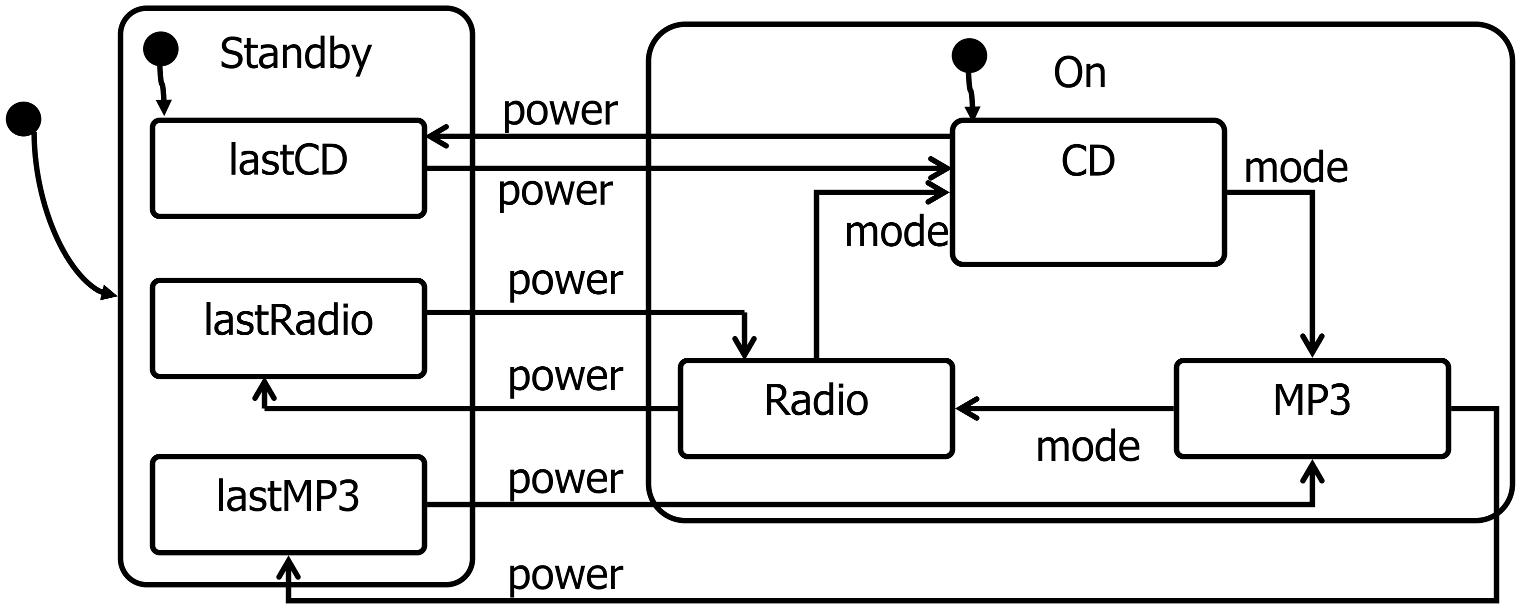
State transition diagram

Exercise

- Model the behaviour of a sound system.
- The system can be ON or in Standby.
- The system has a radio, MP3 and CD player. It is posible to activate them by means of the "*mode*" button.
- When the system is turned on, the last state from its previous use is activated.

State transition diagram

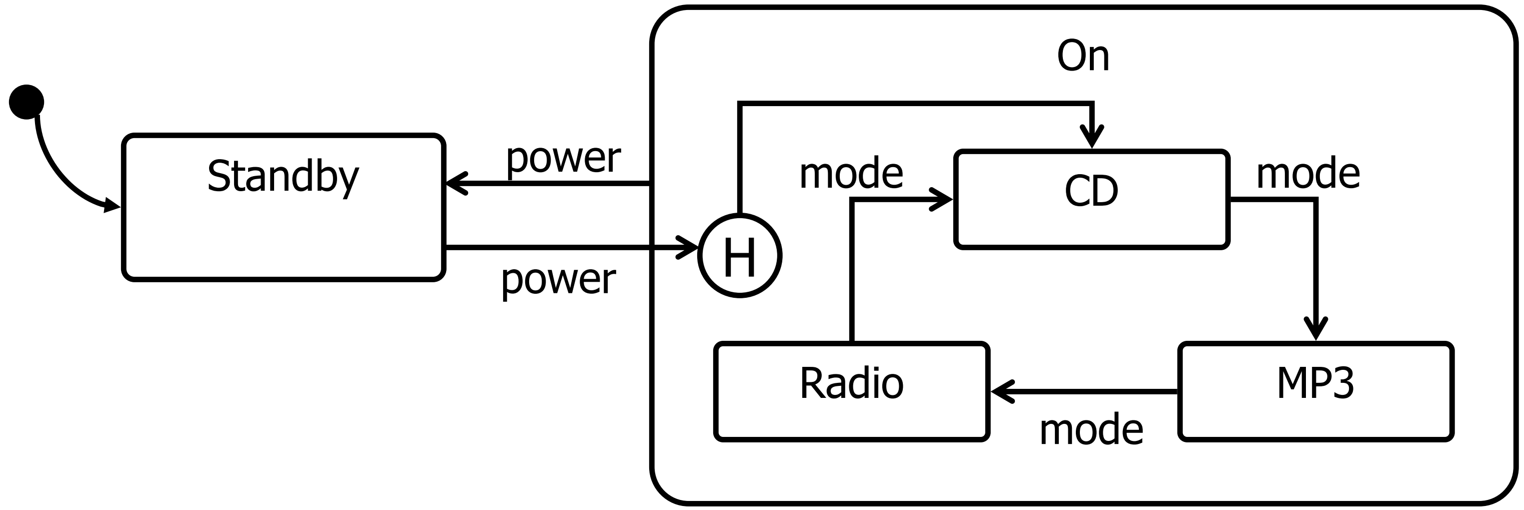
Solution



State transition diagram

History state

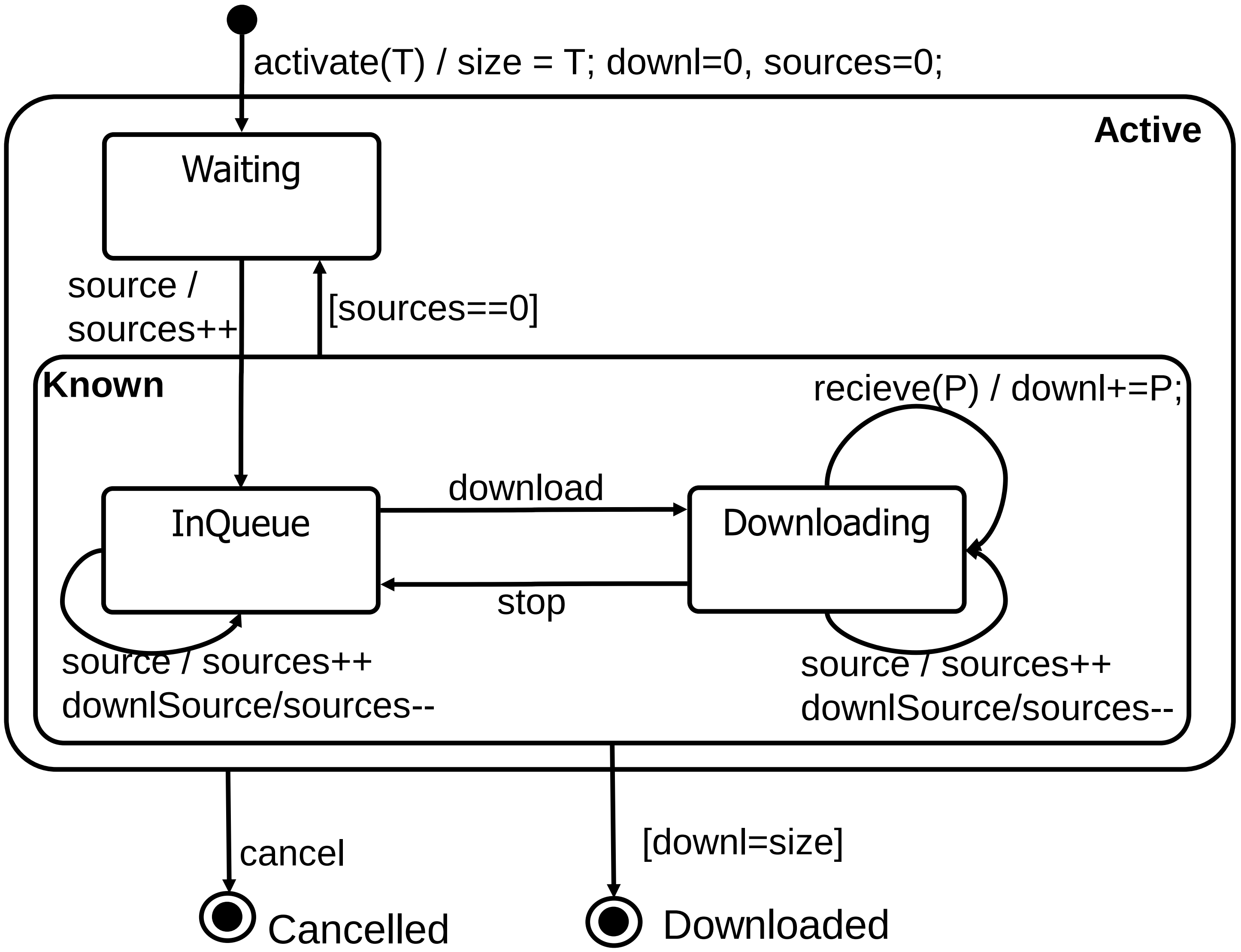
- A ***history*** state remembers the current state the system was in when a hierarchical state was left.
- The first time the hierarchical state is entered, the outgoing transition from the history state is followed.



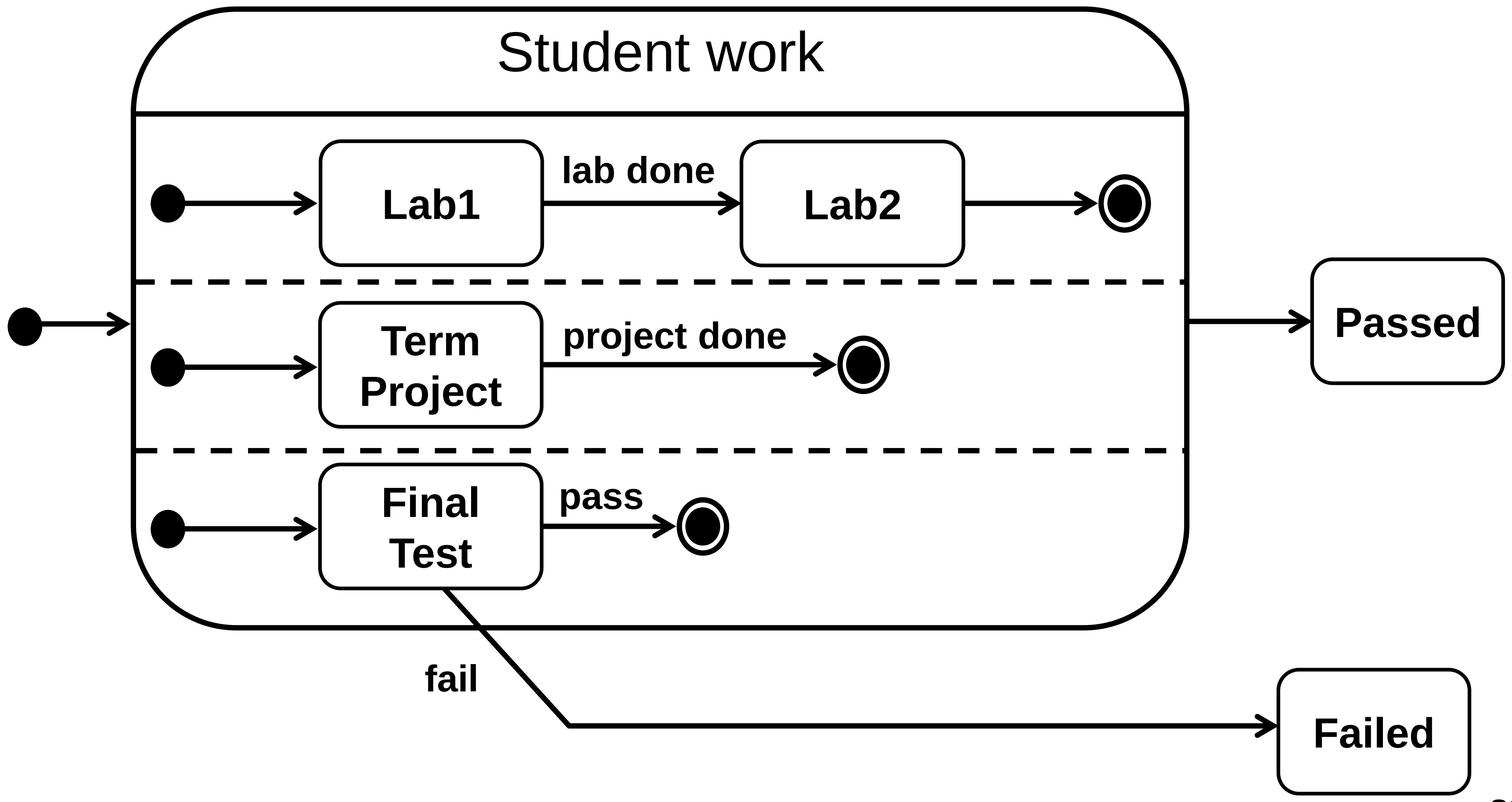
Exercise

- Model the behaviour of a *“shared resource”* in a P2P system. Its life cycle is as follows:
  - A user activates the download of a shared resource with size T.
  - When a source is available for the resource, the resource is enqueued. More sources can be available at any moment.
  - Once the resource is in the queue, at any moment it can start being downloaded.
  - While the resource is being downloaded, data packets of size P arrive to the user device.
  - Downloading can end before all data are transferred or when all data are downloaded.
  - At any moment before the resource is completely downloaded the user can cancel the process.

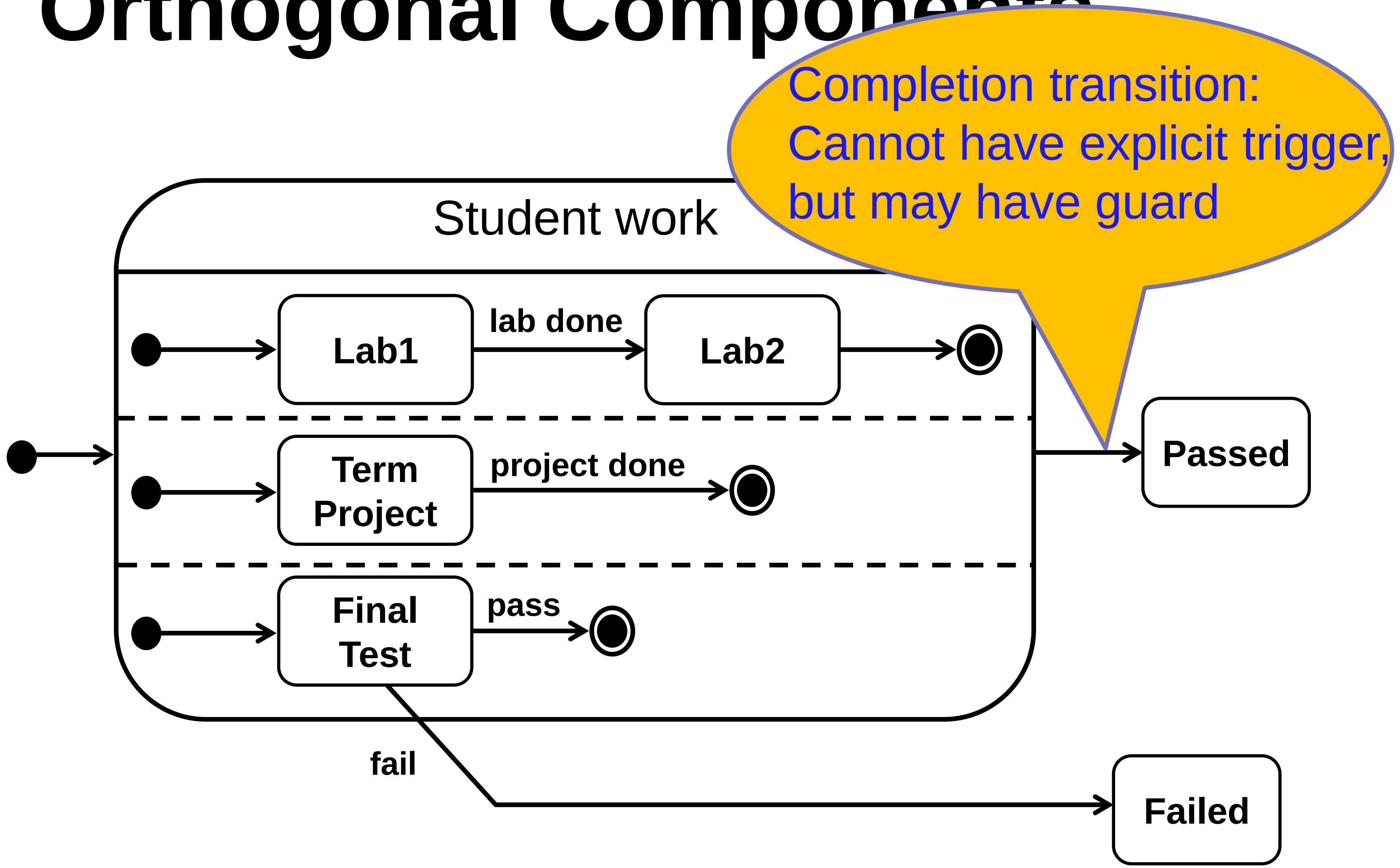
Exercise solution



Orthogonal Components



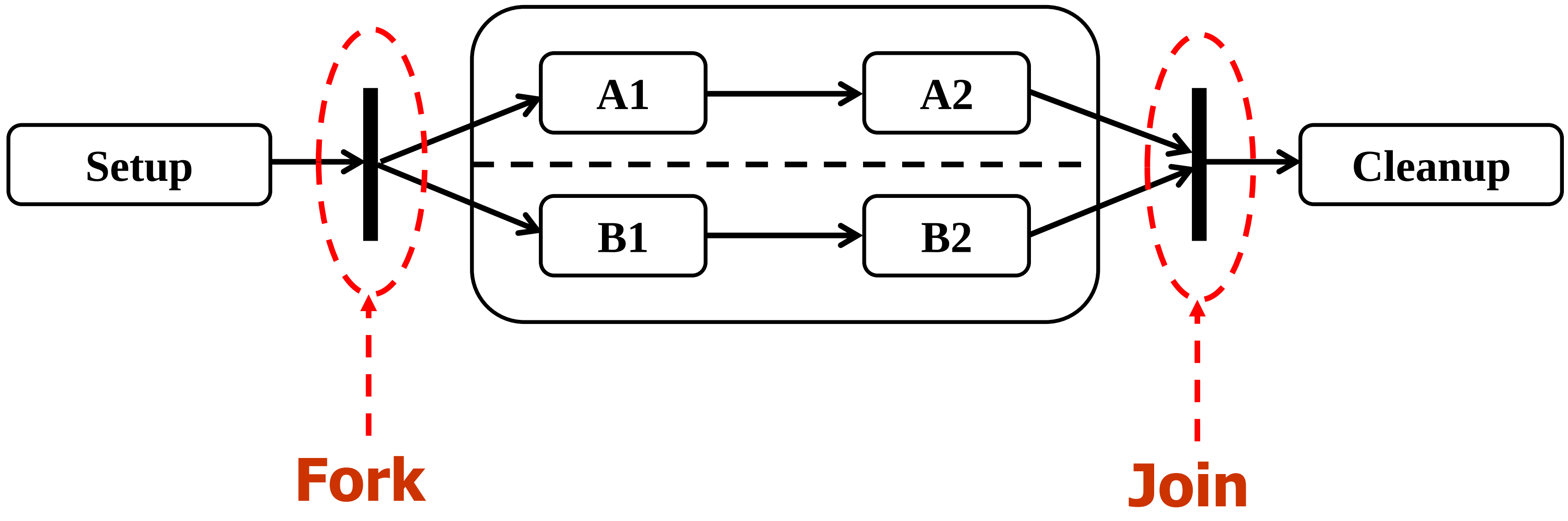
Orthogonal Components



Orthogonal Components

Synchronization pseudostates

- Fork and Join

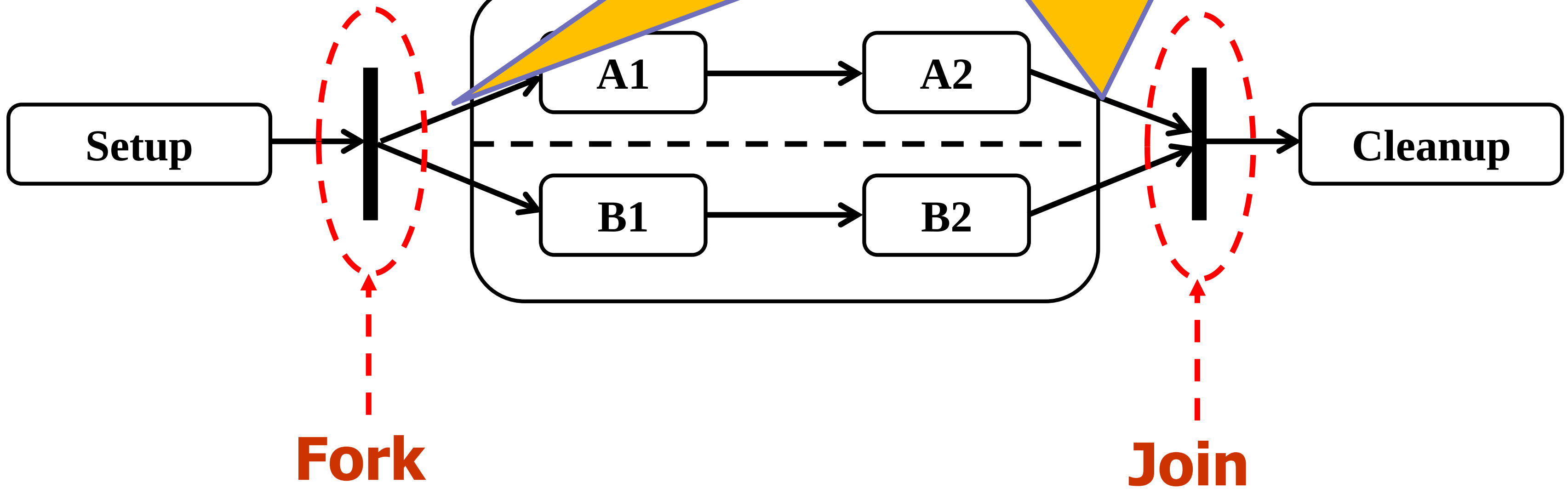


Orthogonal Components

Synchronization pseudostat

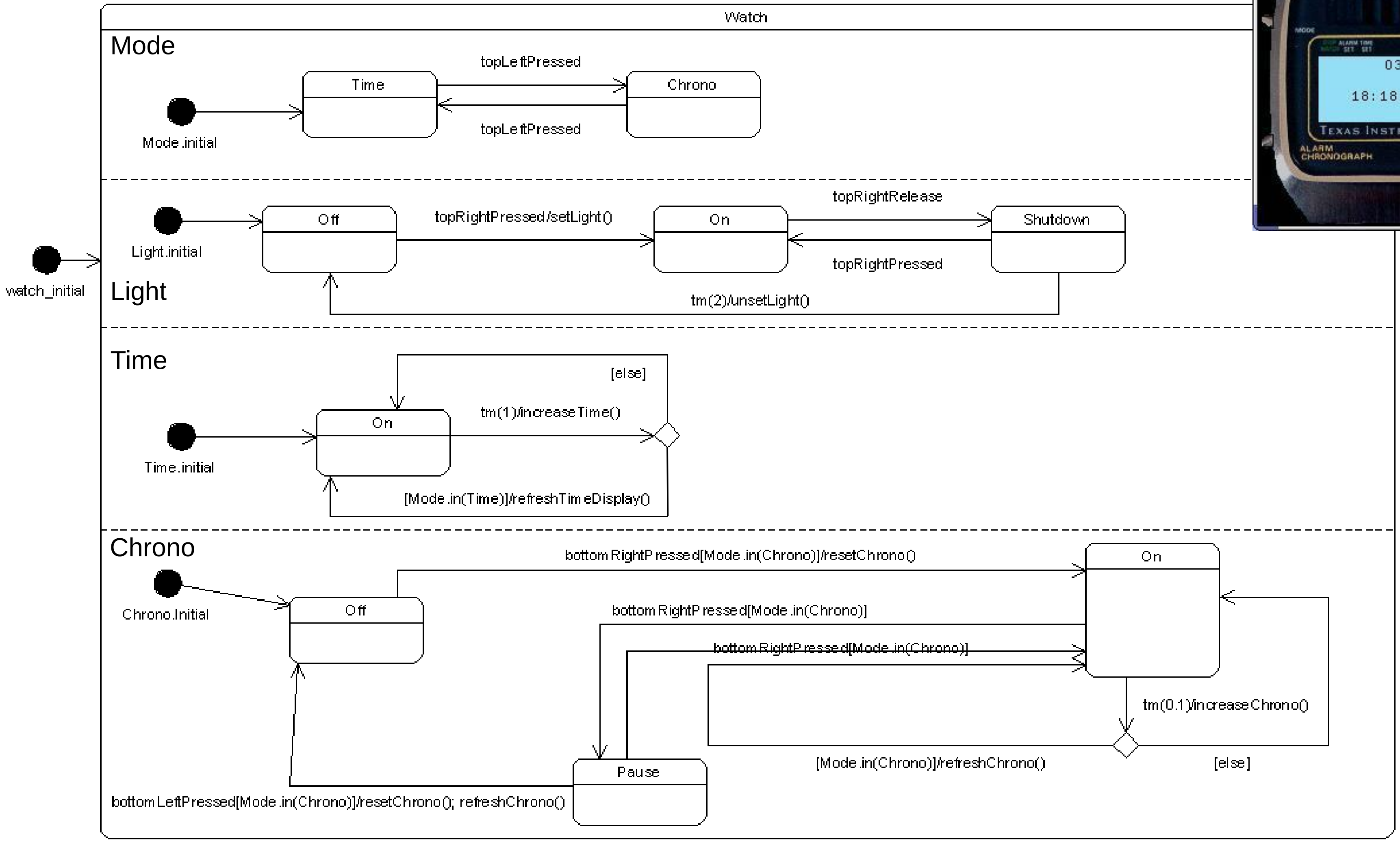
- Fork and Join

Transitions *from* a fork or *into* a join:
Cannot have explicit trigger nor guard

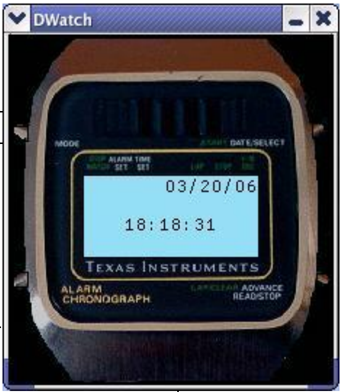


Orthogonal components

Example: a digital watch

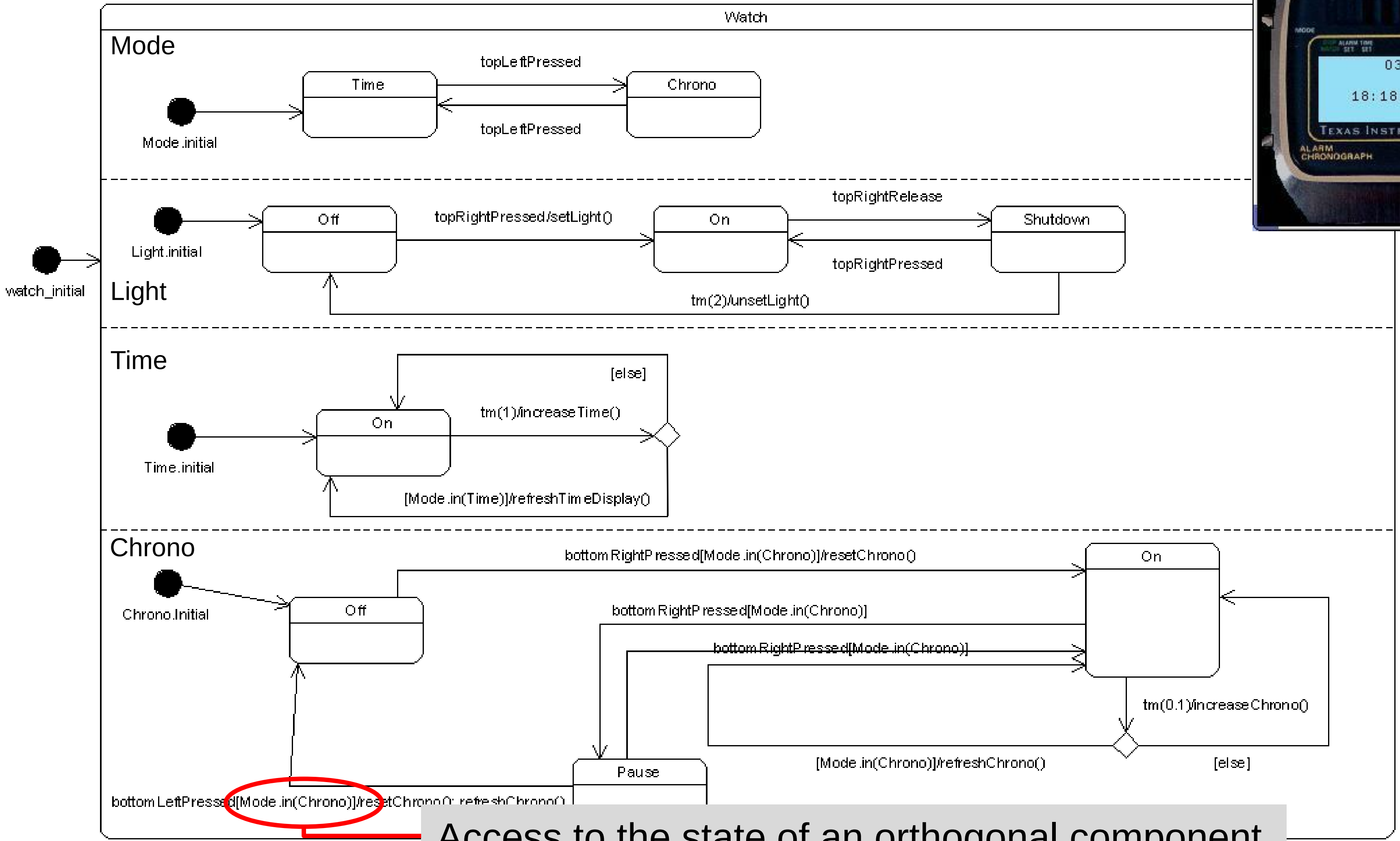


Watch
Mode
topLeftPressed
Time
Chrono
Mode.initial
topLeftPressed
topRightRelease
Off
topRightPressed/setLight()
On
Shutdown
Light.initial
topRightPressed
watch_initial
Light
tm(2)/unsetLight()
Time
[else]
tm(1)/increaseTime()
On
Time.initial
[Mode.in(Time)]/refreshTimeDisplay()
Chrono
bottomRightPressed[Mode.in(Chrono)]/resetChrono()
On
Chrono.Initial
Off
bottomRightPressed[Mode.in(Chrono)]
bottomRightPressed[Mode.in(Chrono)]
tm(0.1)/increaseChrono()
Pause
[Mode.in(Chrono)]/refreshChrono()
[else]
bottomLeftPressed[Mode.in(Chrono)]/resetChrono(); refreshChrono()

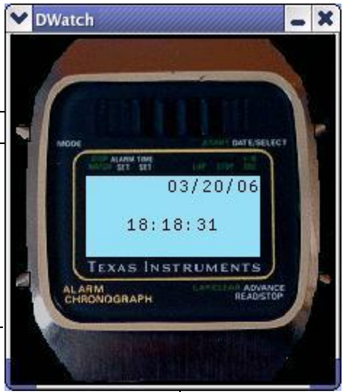


Orthogonal components

Example: a digital watch



Watch
Mode
Mode.initial
Time
Chrono
topLeftPressed
topLeftPressed
watch_initial
Light.initial
Light
Off
On
Shutdown
topRightPressed/setLight()
topRightRelease
topRightPressed
tm(2)/unsetLight()
Time
Time.initial
On
[else]
tm(1)/increaseTime()
[Mode.in(Time)]/refreshTimeDisplay()
Chrono
Chrono.Initial
Off
On
Pause
bottomRightPressed[Mode.in(Chrono)]/resetChrono()
bottomRightPressed[Mode.in(Chrono)]
bottomRightPressed[Mode.in(Chrono)]
tm(0.1)/increaseChrono()
[Mode.in(Chrono)]/refreshChrono()
[else]
bottomLeftPressed[Mode.in(Chrono)]/resetChrono(); refreshChrono()



Access to the state of an orthogonal component

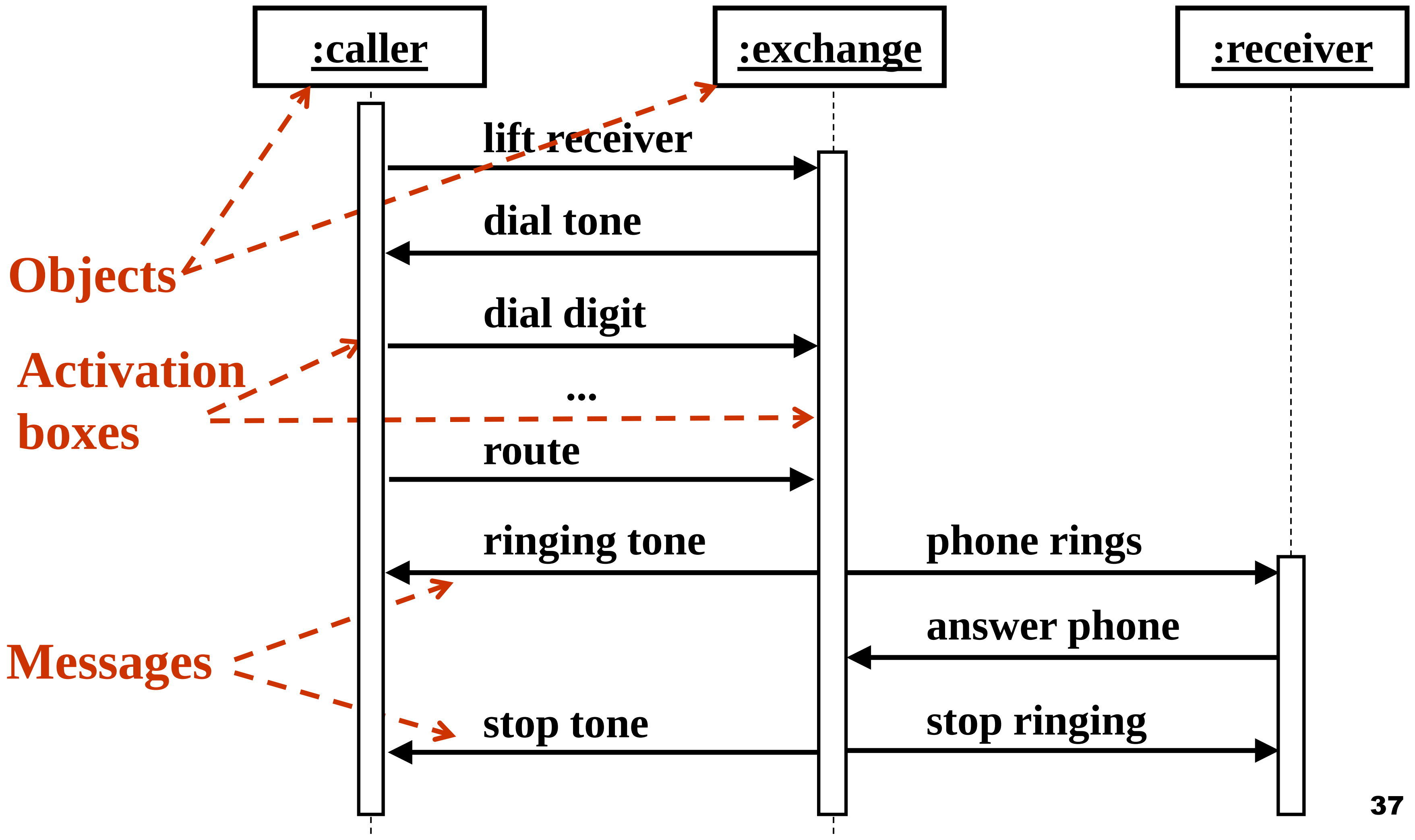
State transition diagram

- Select two classes from your project and build their state transition diagrams.

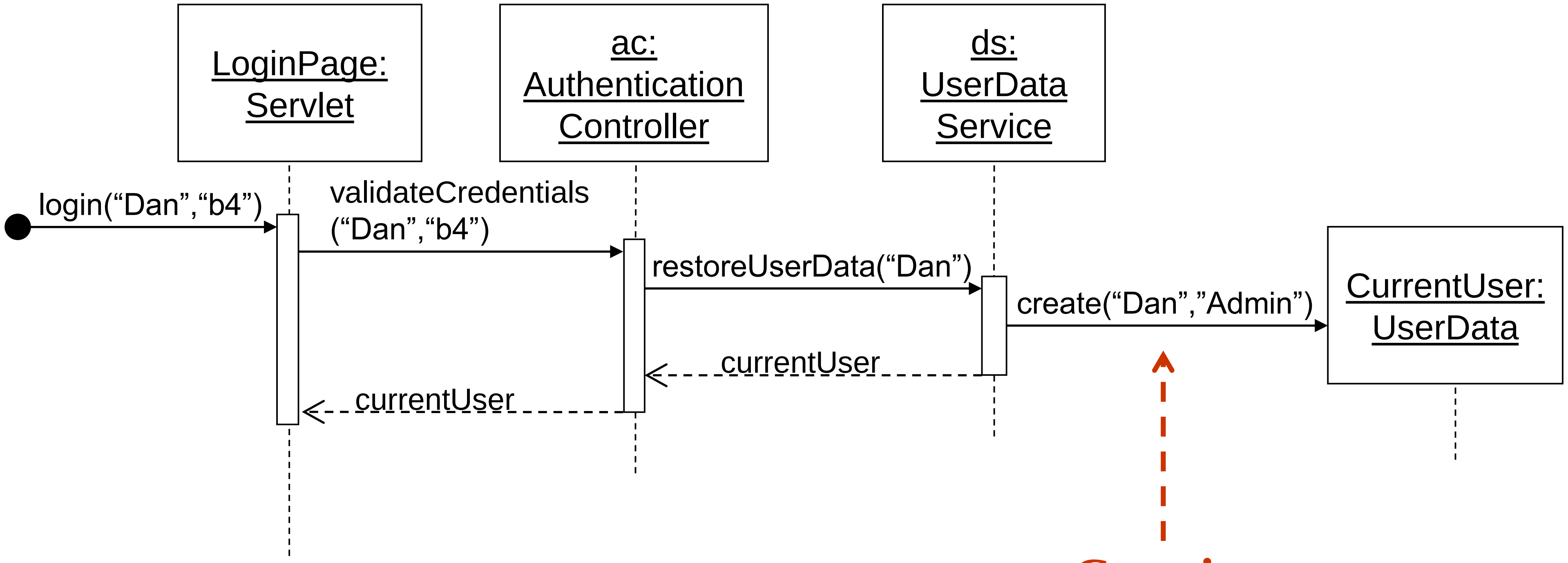
Sequence diagram

- Represents a sequence of messages that are activated by and sent to **a set of objects**.
- Each object has a lifeline represented vertically.
  - Time evolution is represented downwards.
  - Message invocation: arrows joining lifelines.
  - Activation boxes.

Sequence diagram



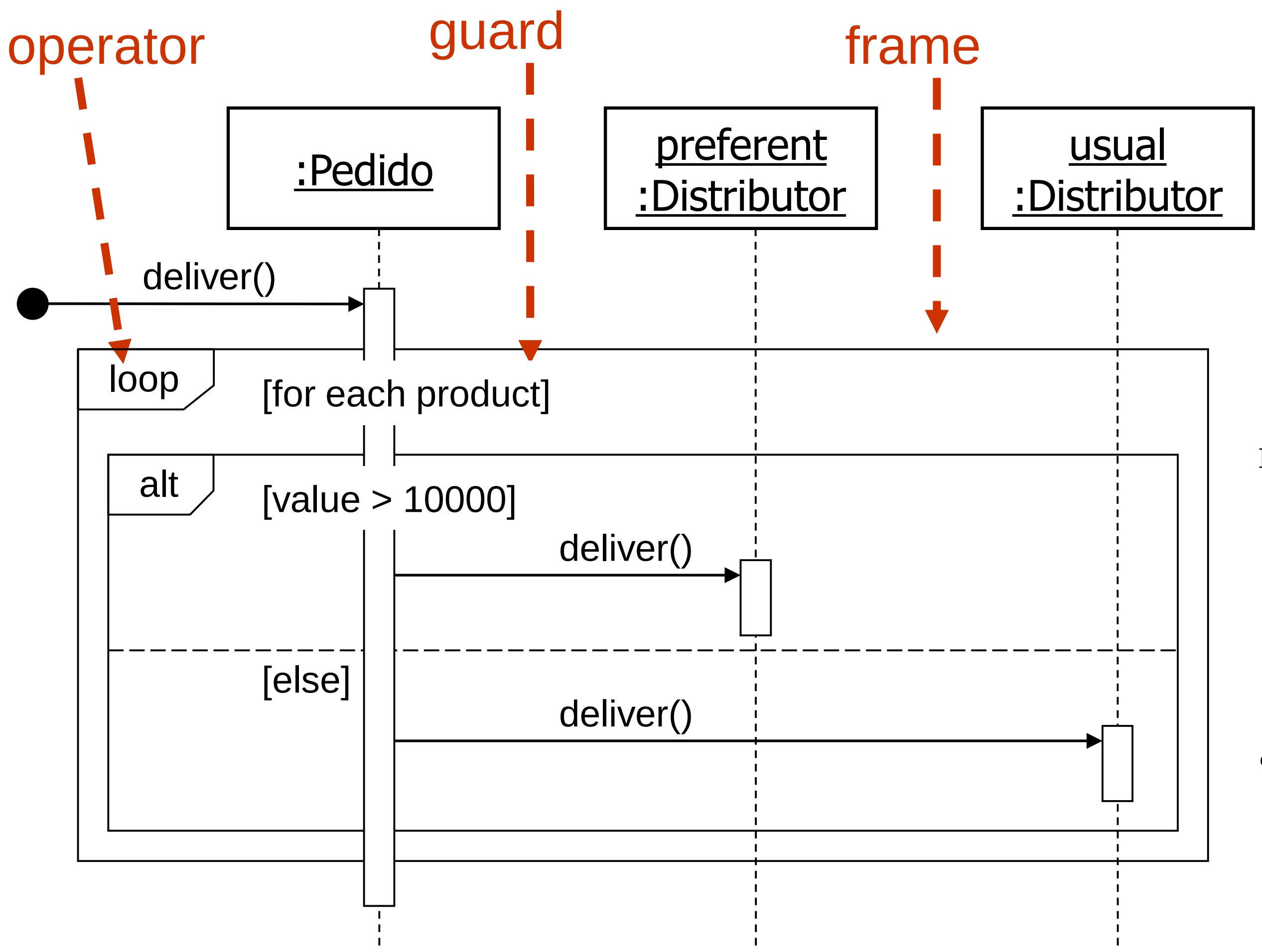
Sequence diagram



Creation message

Sequence diagram

Operators



```
procedure deliver()
   foreach product:
      if product.value>10000
         preferent.deliver()
      else
         usual.deliver()
      end if
   end for
end procedure
```

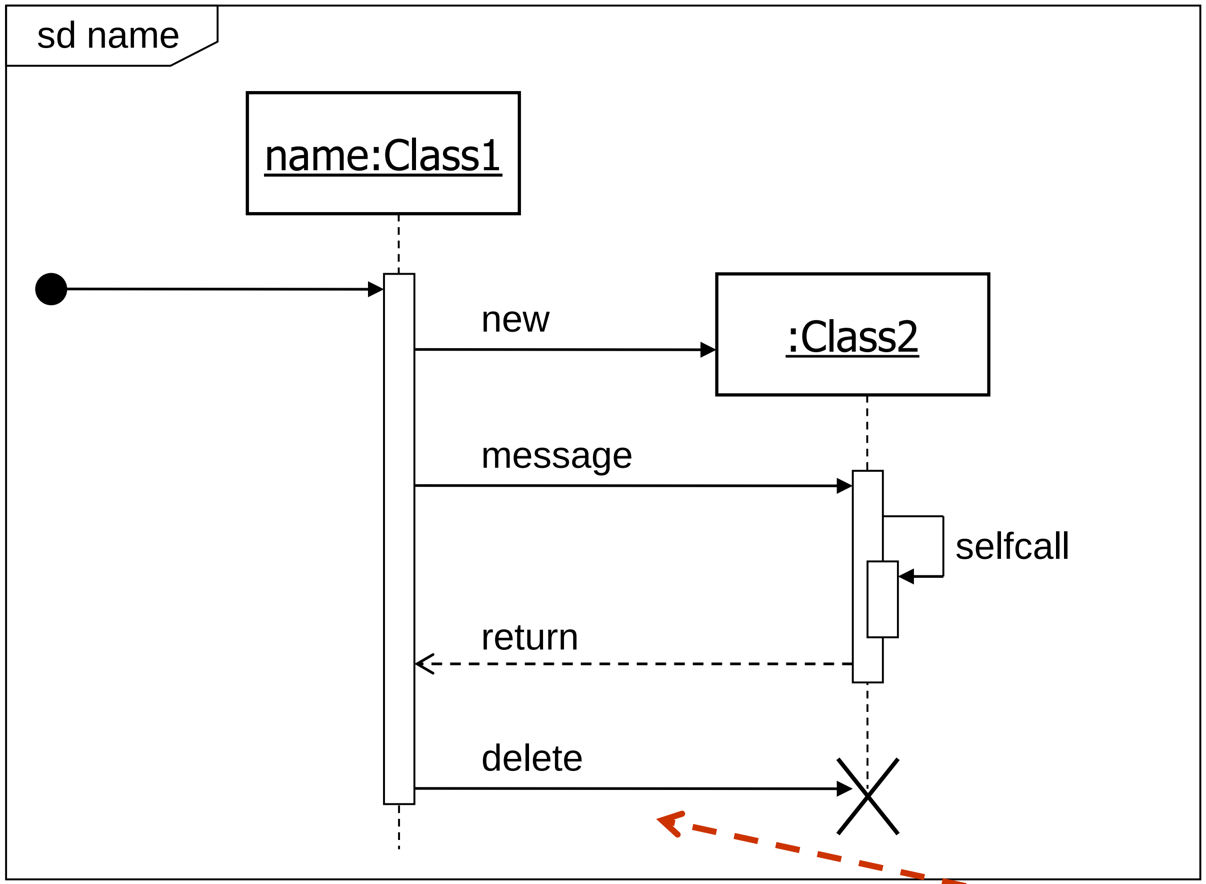
Sequence diagram

Operators

- Combined fragments, operators:
  - **Alternative (*alt*)**: selection (by means of a guard) of an interactión. Múltiple fragments, only the one that satisfies the guard is executed.
  - **Option (*opt*)**: equivalent to an *alt* operator with just one fragment. It is executed if the guard is satisfied.
  - **Loop (*loop*)**: the fragment is executed several times. The guard indicates how to iterate.
  - **Negative (*neg*)**: defines an invalid interaction.
  - **Parallel (*par*)**: each fragment is executed in parallel.
  - **Critical region (*critical*)**: there can be only one process executing the fragment at each instant.
  - **Sequence diagram (*sd*)**: encloses a sequence diagram.
  - **Reference (*ref*)**: the frame refers to an interaction that is defined in another diagram. It covers the lines involved in the interaction. It can include parameters and a return value.

Sequence diagram

Operators

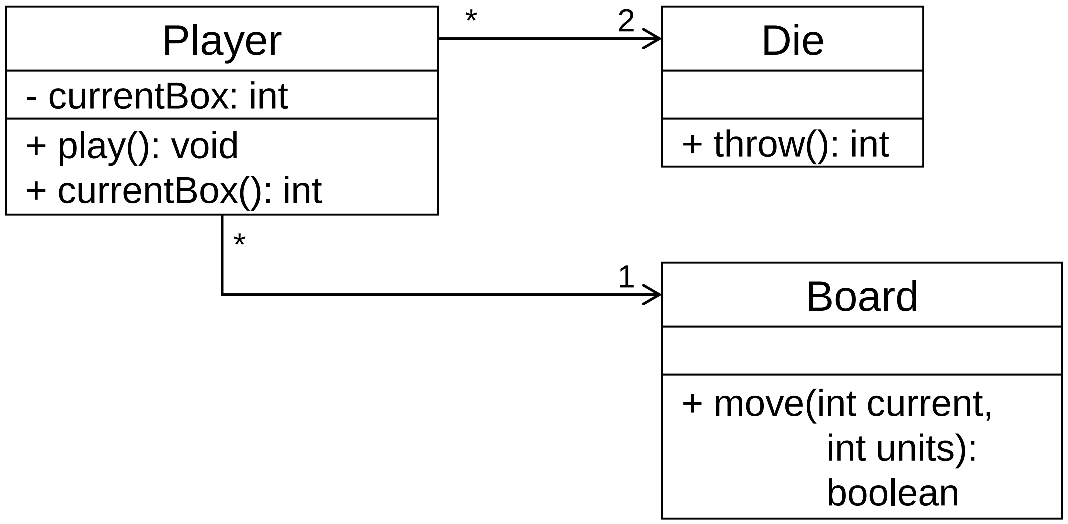


destruction message

Sequence diagram

Exercise

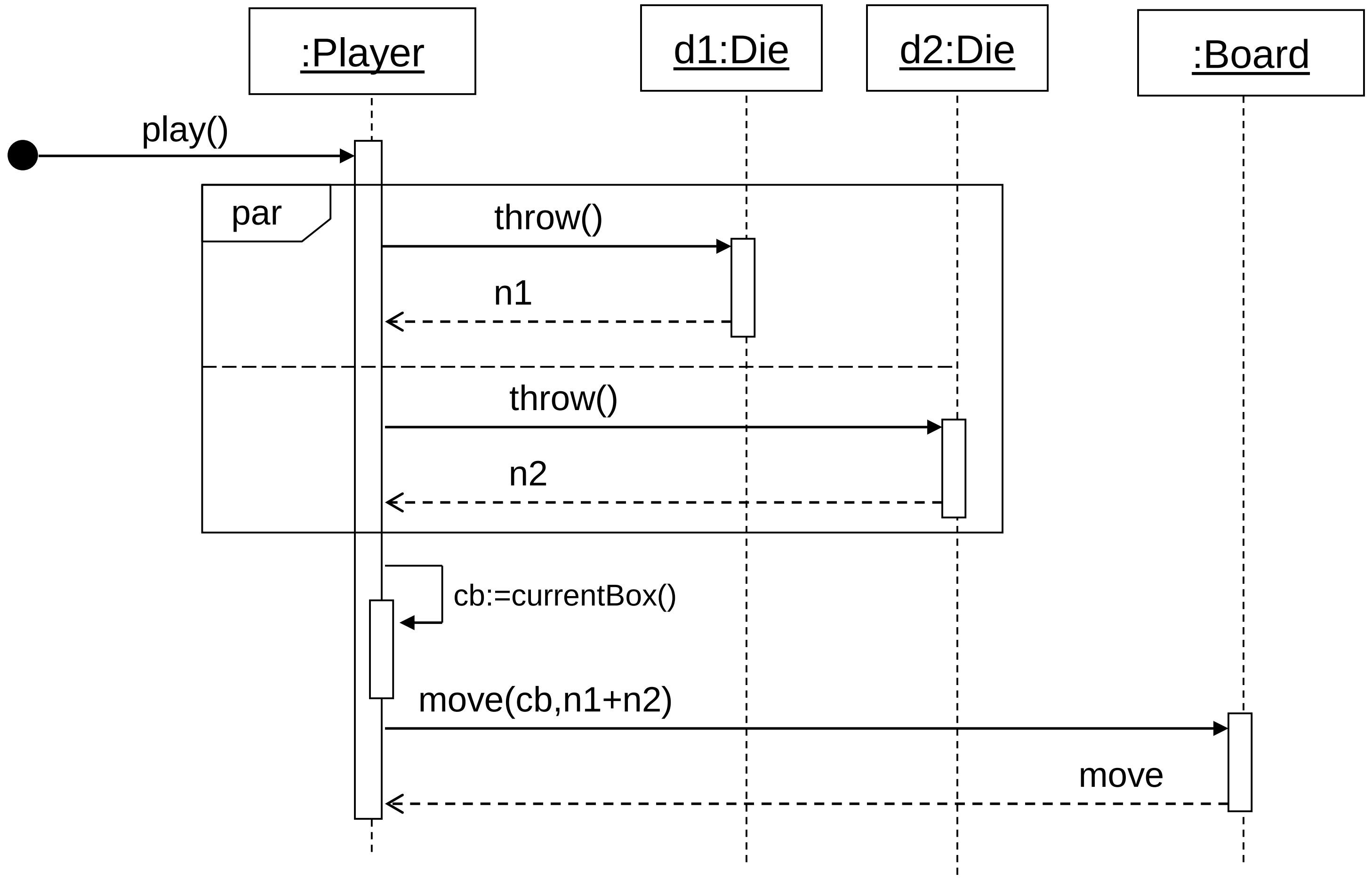
- Specify the sequence diagram of the “play” operation defined in the Player class for the ludo(*) game.



(*) ludo=parchís.

Sequence diagram

Exercise



Sequence diagram

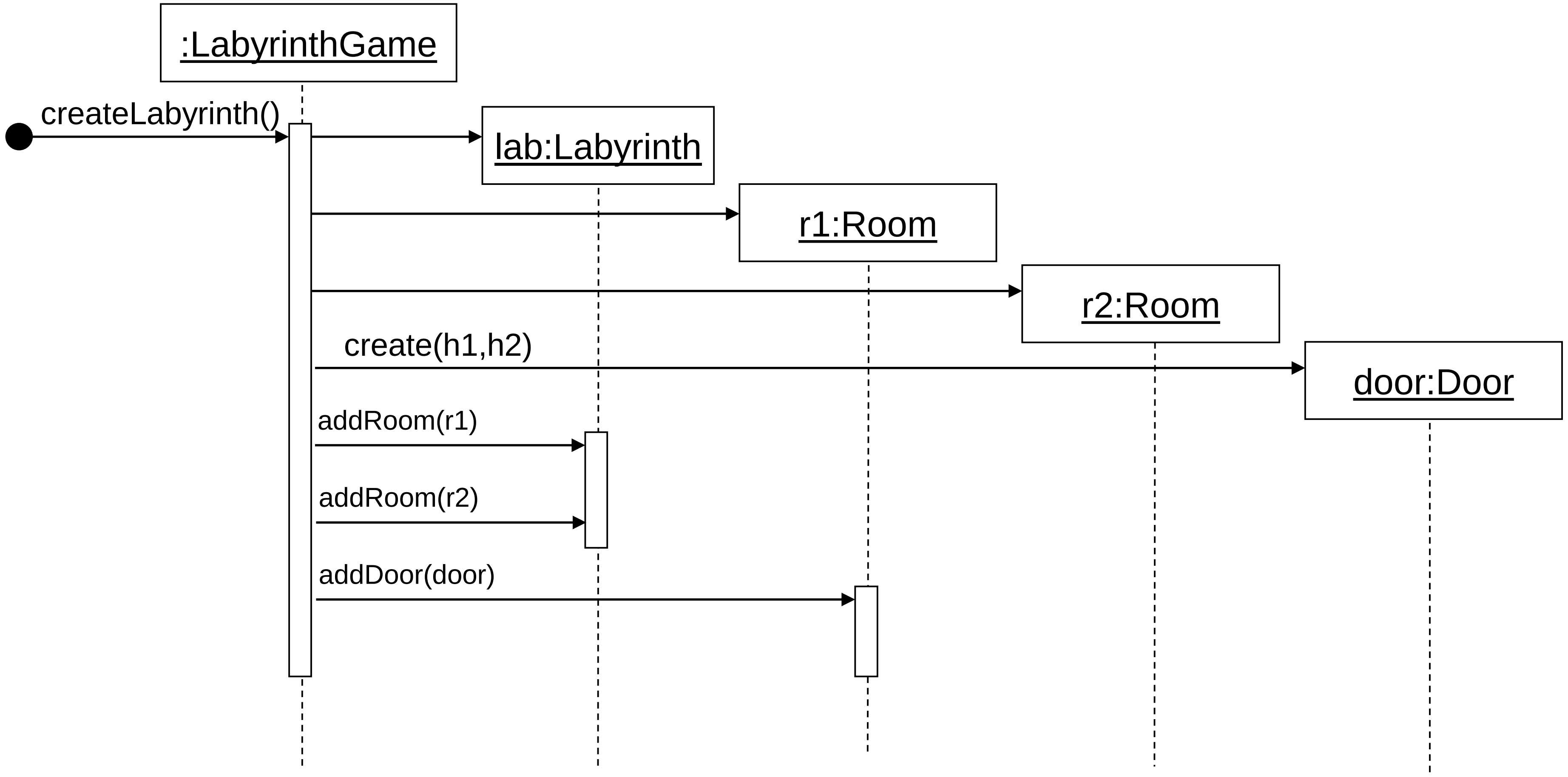
Exercise

Specify the sequence diagram for the method “createLabyrinth”

```
public class LabyrinthGame {
   public Labyrinth creteLabyrinth () {
      Labyrinth lab = new Labyrinth();
      Room r1 = new Room();
      Room r2 = new Room();
      Door door = new Door(r1, r2);
      lab.addRoom(r1);
      lab.addRoom(r2);
      r1.addDoor(door);
      return lab;
   }
}
```

Sequence diagram

Exercise



Sequence diagram

- Select two relevant scenarios from your project and build their sequence diagrams.
  - You can select some of the scenarios from the use cases.

Index

Requirements Traceability

- At the end of the design phase, it is necessary to check whether the current project requirements, elicited in the analysis phase, are being met.
- A Requirements Traceability Matrix helps correlating requirements and functional elements created in the solution domain.

	Functional Elements			
Requirements	Class1.method1	Class1.method2	…	ClassN.methodM
Requirement 1	X	X		
Requirement 1.1		X		
…				
Requirement N	X			X
Requirement N.M				X

Requirements Traceability

- Create a Requirements Traceability Matrix for your project.
  - This matrix will be later on updated and re-submitted after the coding phase

Bibliography

There are many books and manuals about UML2.0. Here you can find some of them:

- Class diagrams:
  - Using UML. Stevens&Poley. Addison Wesley. 1999. Chapters 5 y 6.
  - UML Distilled, 3rd Edition. Fowler. Addison Wesley. Chapter 3.

- Object diagrams:
  - UML Distilled, 3rd Edition. Fowler. Addison Wesley. Chapter 6.

- State transition diagrams:
  - Using UML. Stevens&Poley. Addison Wesley. 1999. Chapter 11.
  - UML Distilled, 3rd Edition. Fowler. Addison Wesley. Chapter 10.

- Sequence diagrams:
  - UML Distilled, 3rd Edition. Fowler. Addison Wesley. Chapter 4.