Introduction to UML

Paolo Ciancarini – paolo.ciancarini@unibo.it
Department of Informatics – University of Bologna

Software models

- A model is a *description* of a system
- A model is always an *abstraction* at "some level": it captures the essential aspects of a system and ignores some details

UML is a modeling language

- A modeling language allows the specification, the visualization, and the documentation of the development of a software system
- The models are *descriptions* which users and developers can use to communicate ideas about the software
- UML 1.* is a modeling language
- UML 2.* is still a modeling language, but it is so “detailed” that can be used also as a programming language (see OMG’s Model Driven Architecture)

Evolution of UML

- OO languages appear, since mid 70's to late 80's
- Between '89 and '94, OO methods increased from 10 to 50
- Unification of ideas began in mid 90's
    - 1994 Rumbaugh joins Booch at Rational
  - 1995 v0.8 draft Unified Method
    - 1995 Jacobson joins Rational (*Three Amigos*)
  - 1996 June: UML v0.9 published

  - 1997 Jan: UML 1.0 offered to OMG
  - 1997 Jul: UML 1.1 OMG standard
  - 1998: UML 1.2
  - 1999: UML 1.3
  - 2001: UML 1.4
    - 2003 Feb: IBM buys Rational
  - 2003: UML 1.5
  - 2004: UML 1.4.2 becomes the standard ISO/IEC 19501

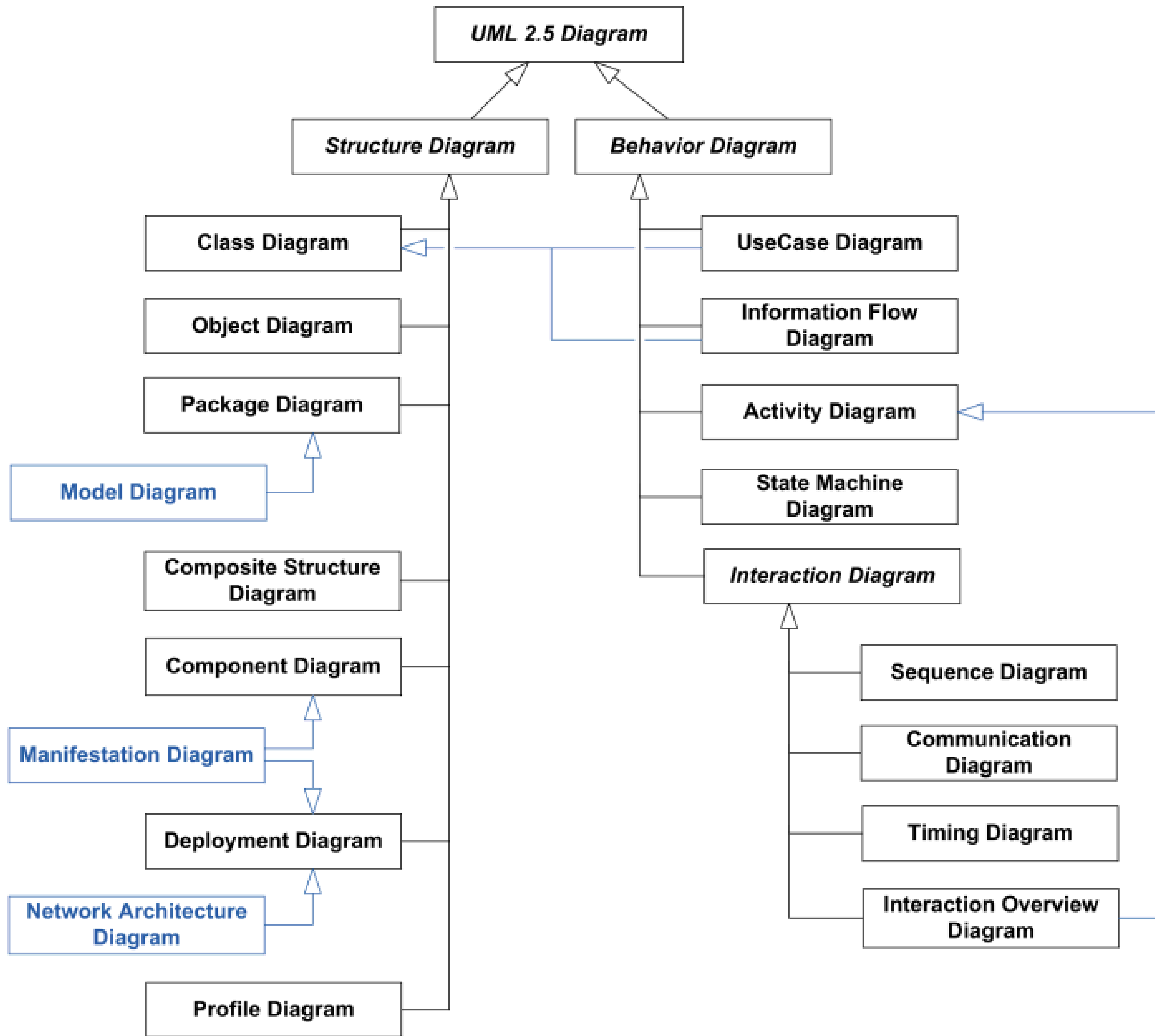
  - 2005: UML 2.0
  - 2007: UML 2.1.2
  - 2009: UML 2.2
  - 2010: UML 2.3
  - 2011: UML 2.4
  - 2013: UML 2.5

pre-UML

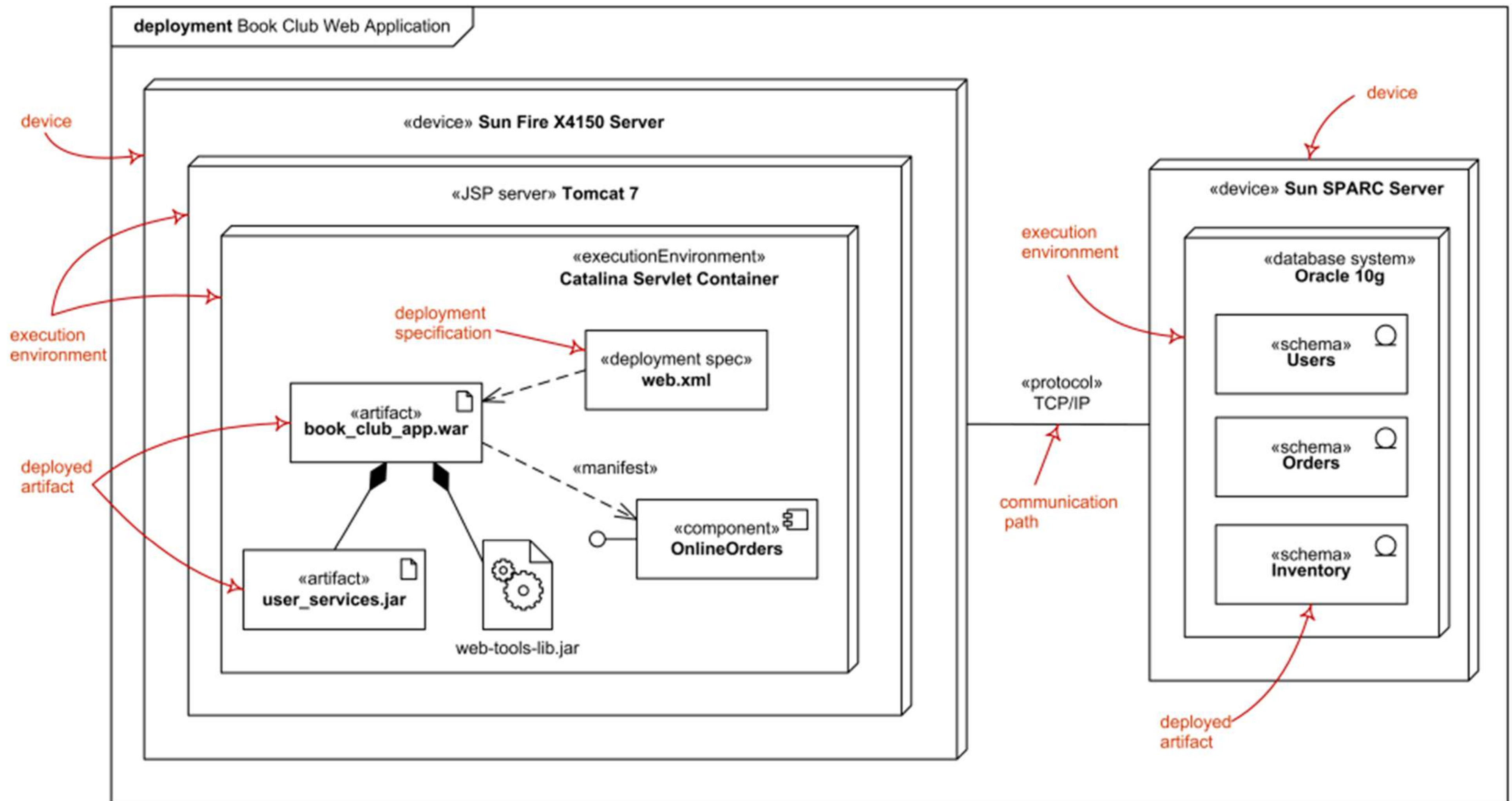
UML 1.x

UML 2.0

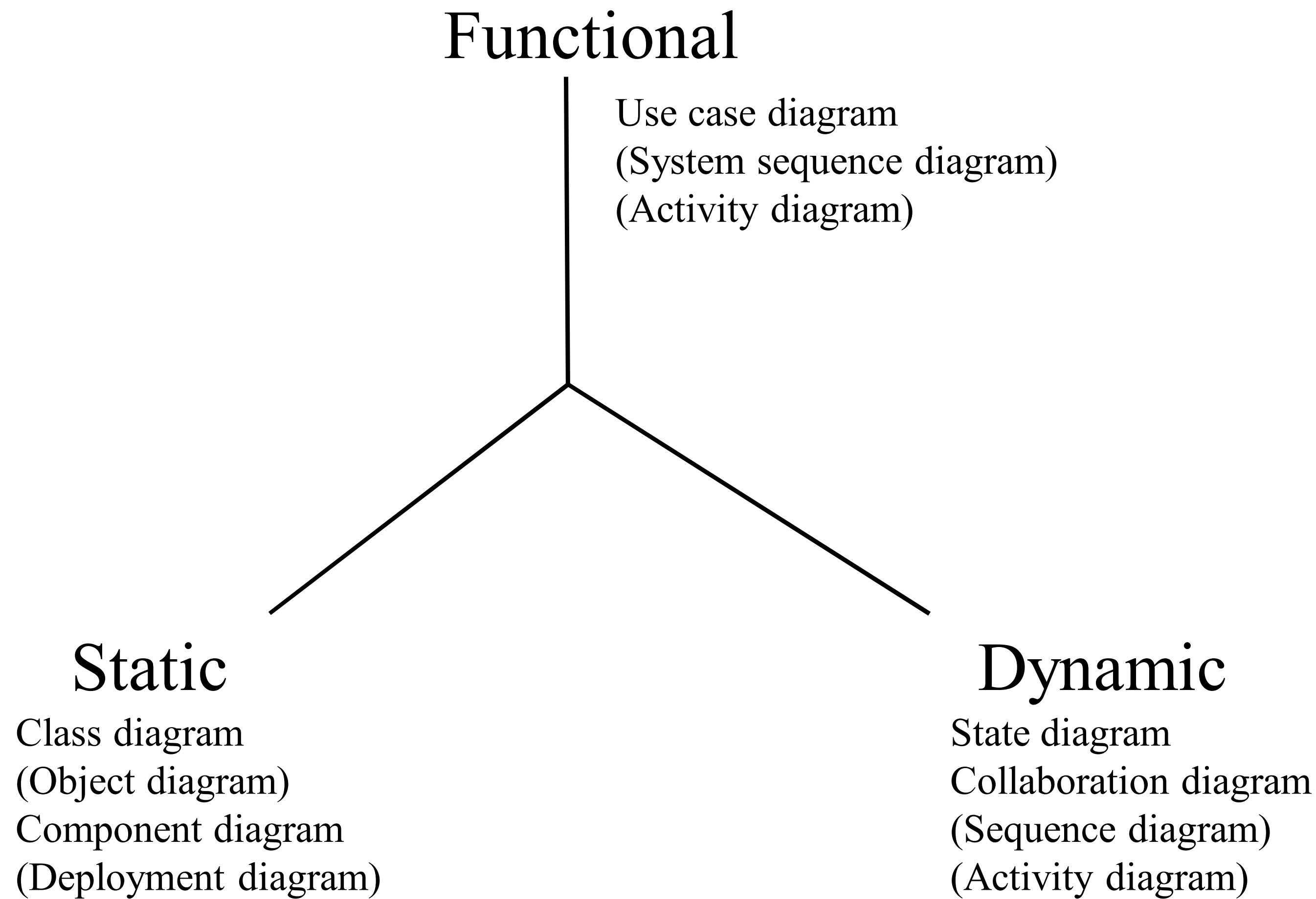
UML includes several visual notations



Deployment diagram



Three modeling axes



Example

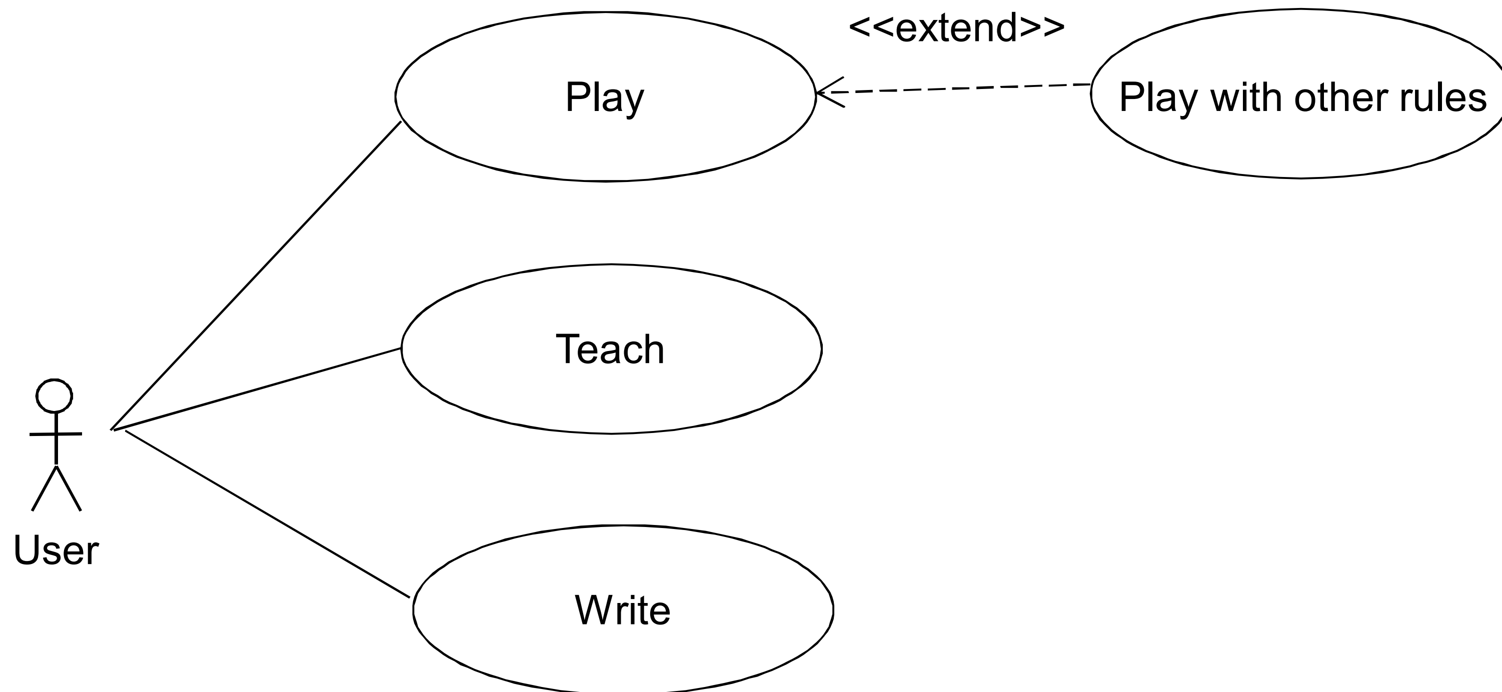
- A chess program could be “stand-alone”, “client-server”, “agent based”, etc.
- Its behavior should always be coherent with the rules of chess
- What is its **goal**? To play and win a chess game against an opponent

Goals and responsibilities

- The very same chess program, with identical structure and behavior, could be used with a different goal?
- For instance, could it be used to learn to play chess? Responsibility of the program: teach chess
- Or to write a chess book, like a chess game editor? Responsibility of the program: write chess texts
- Or to play a game of loser's chess (where who is checkmated wins)? Responsibility: play games with rules slightly different from chess

Each responsibility corresponds to (at least) a use case

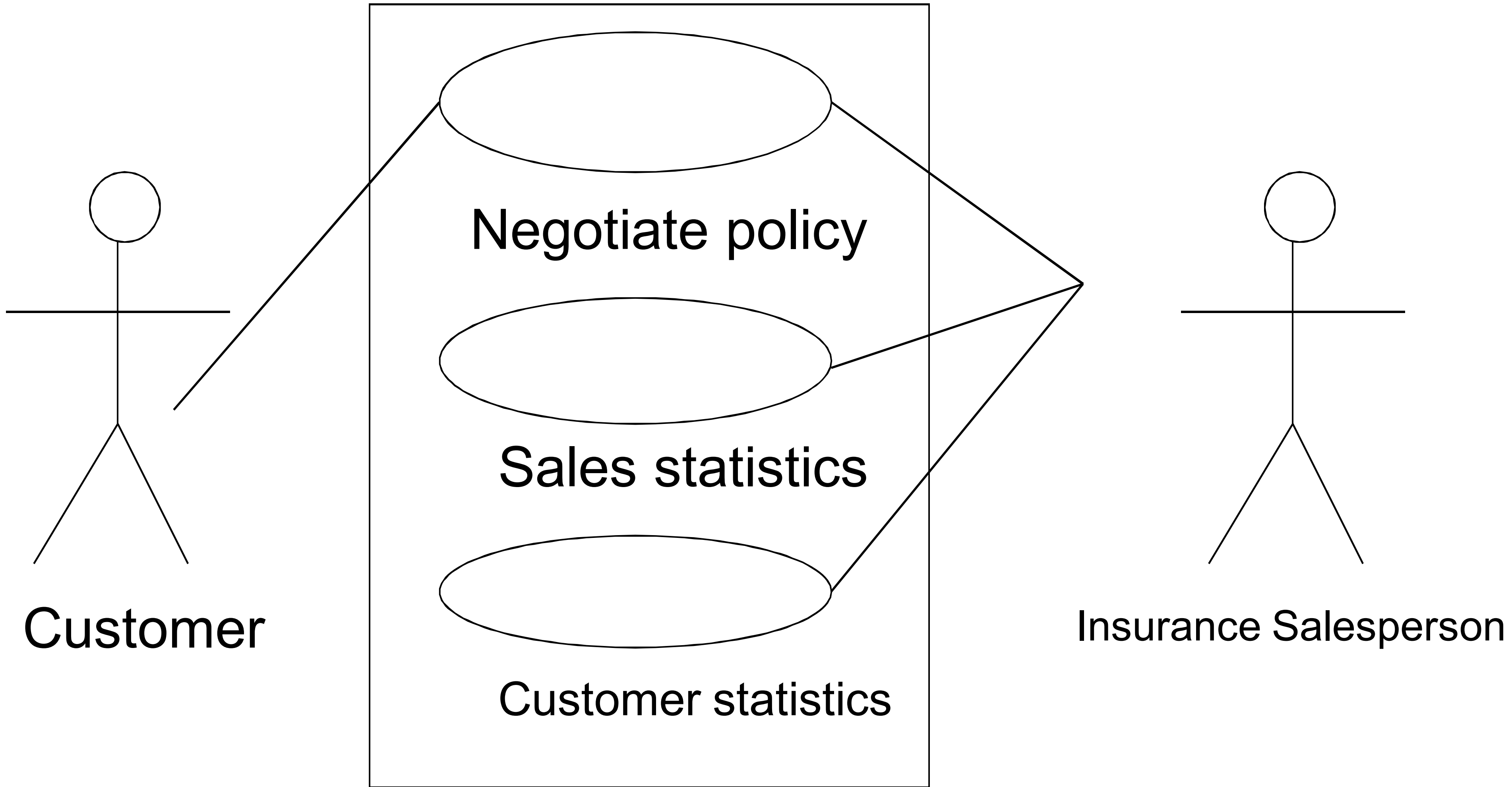
From responsibilities to use cases



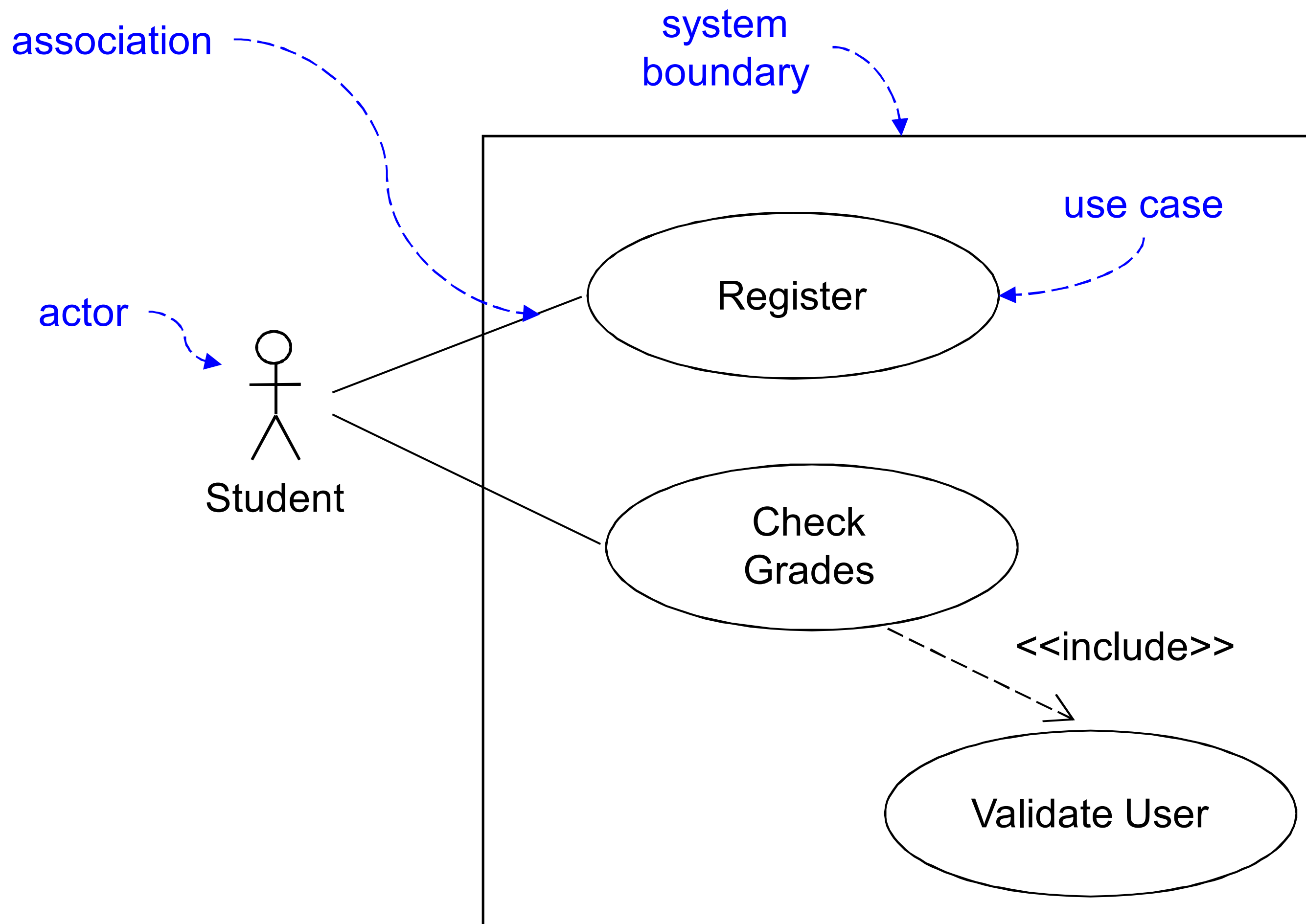
Use Case diagram

- A UC diagram describes the observable behavior of a system
- It describes the main interactions between the system and external entities, including users and other systems
- It is a summary of the main scenarios where the system will be used
- It describes the main user roles
- It is a visual summary of the requirements of the system

Example



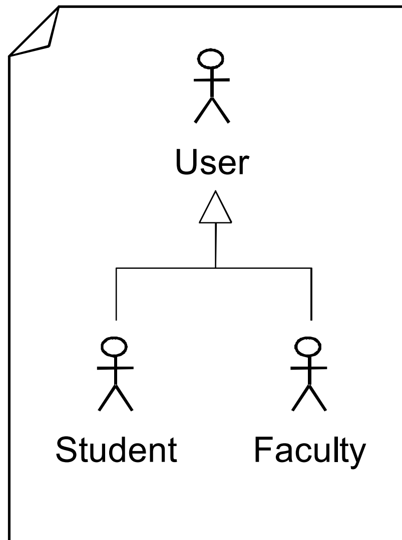
Use Case: elements

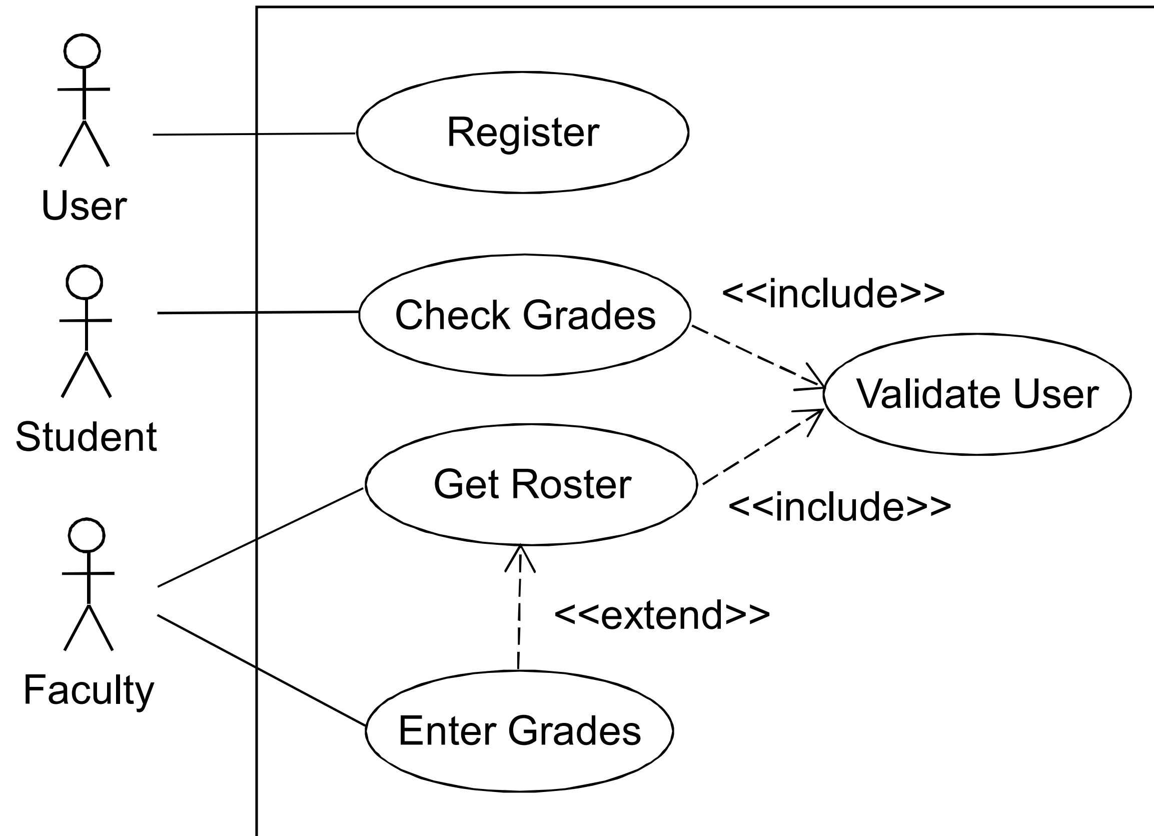


Elements of a Use Case Diagram

- **Actor**:
  - Represents a role played by external entities (humans, systems) that interact with the system
- **Use case**:
  - Describes what the system does (i.e., functionality)
  - Scenario: sequence of interactions between the actors and the system
- **Relationships**:
  - Association between actors and use cases
  - Extension (or generalization) among actors
  - Dependency among use cases: *include* and *extend*

Example





Use Case Scenario

<table>
<tr><td colspan="2">Use Case: Check Grades</td></tr>
<tr><td colspan="2">Description: View the grades of a specific year and semester
Actors: Student
Precondition: The student is already registered
Main scenario:</td></tr>
<tr><td>User</td><td>System</td></tr>
<tr><td>

3. The user enters the year and semester, e.g., Fall 2013.</td><td>1. The system carries out “Validate User”, e.g., for user “miner” with password “allAs”.
2. The system prompts for the year and semester.

4. The system displays the grades of the courses taken in the given semester, i.e., Fall 2013.</td></tr>
<tr><td colspan="2">Alternative:
The student enters “All” for the year and semester, and the system displays grades of all courses taken so far.
Exceptional:
The “Validate User” use case fails; the system repeats the validation use case.</td></tr>
</table>

Exercise

Draw a use case diagram and a related scenario for the following situation:

- A user can borrow a book from a library;
  - extend it with borrowing a journal
- a user can give back a book to the library
  - including the use case when the user is identified

Modeling (parallel, distributed) processes

- Processes and process descriptions
- Activity diagram notation
- Activity diagram execution model
- Making activity diagrams

Processes and their description

A **process** is a collection of related tasks that transforms a set of inputs into a set of outputs.

Process description notations describe workflow processes as well as the computational processes we design.

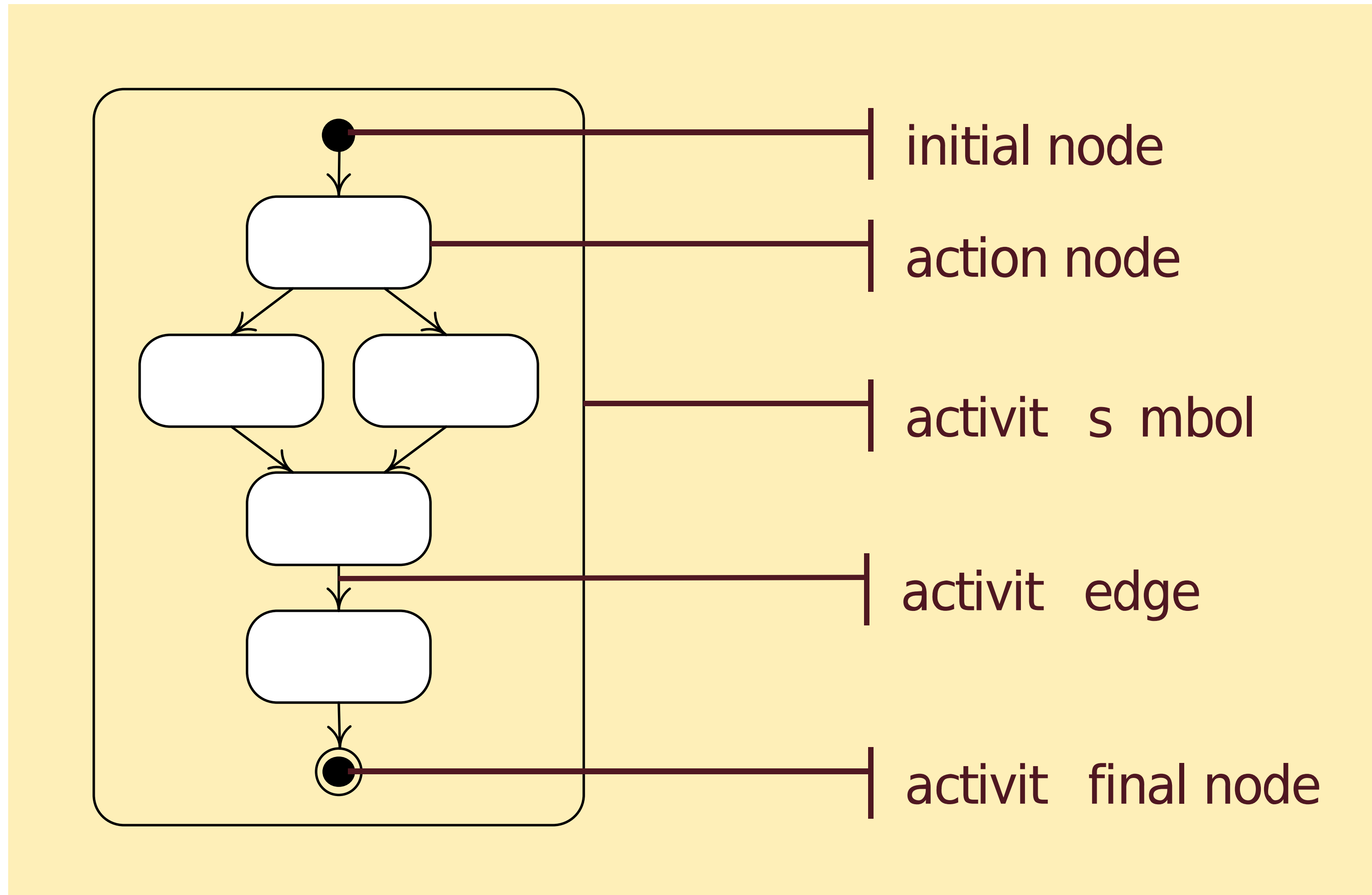
An **activity diagram** shows actions and the flow of control and data between them.

Activities and actions

An activity is a non-atomic task or procedure decomposable into actions.

An action is a task or procedure that cannot be broken into parts.

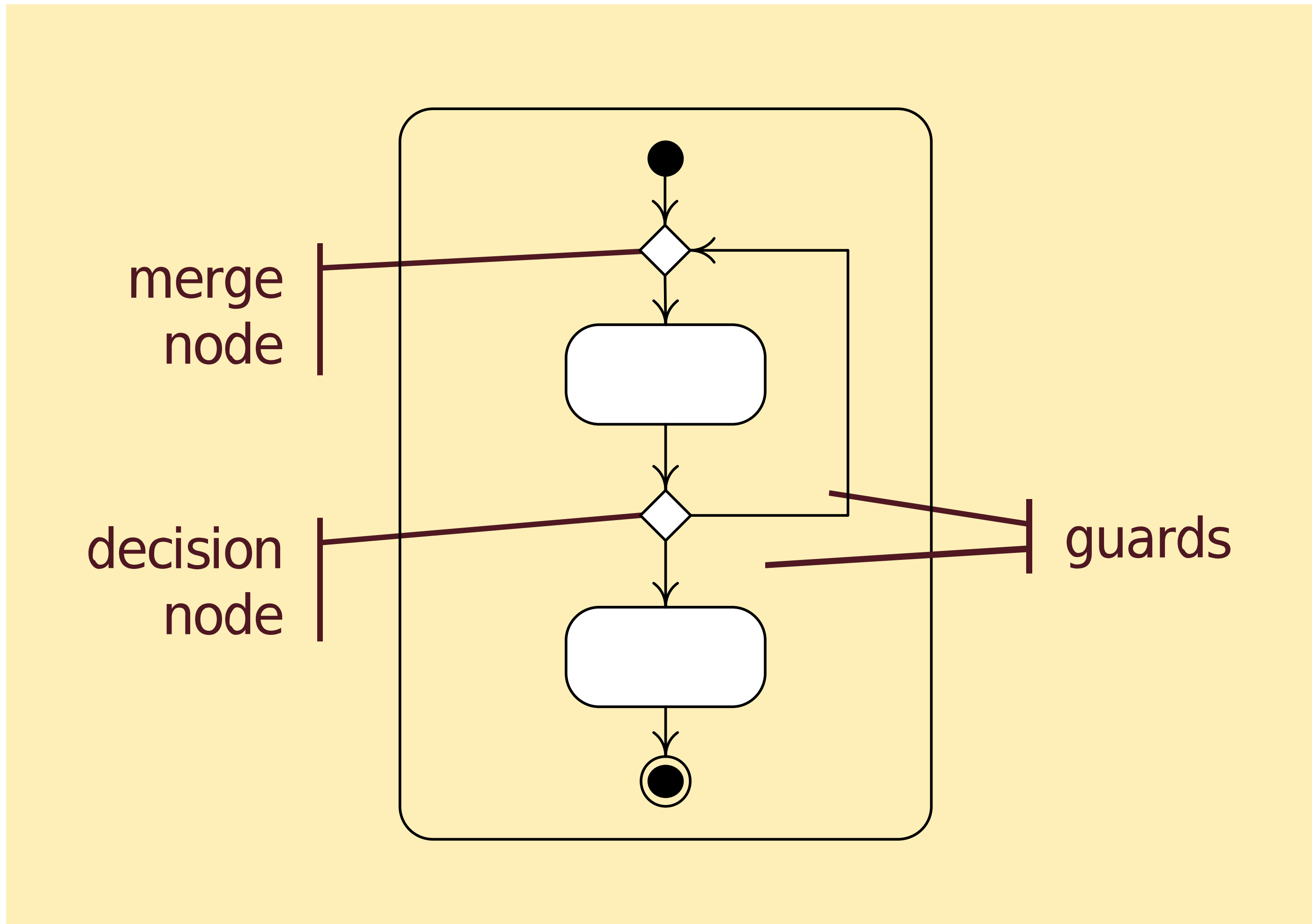
Elements of an activity diagram



Execution model

- The “execution” of an activity diagram is modeled by **tokens** that are produced by action nodes, travel over action edges, and are consumed by action nodes.
- When there is a token on every incoming edge of an action node, it consumes them and begins execution.
- When an action node completes execution, it produces tokens on each of its outgoing edges.
- An initial node produces a token on each outgoing edge when an activity begins.
- An activity final node consumes a token available on any incoming edge and terminates the activity.

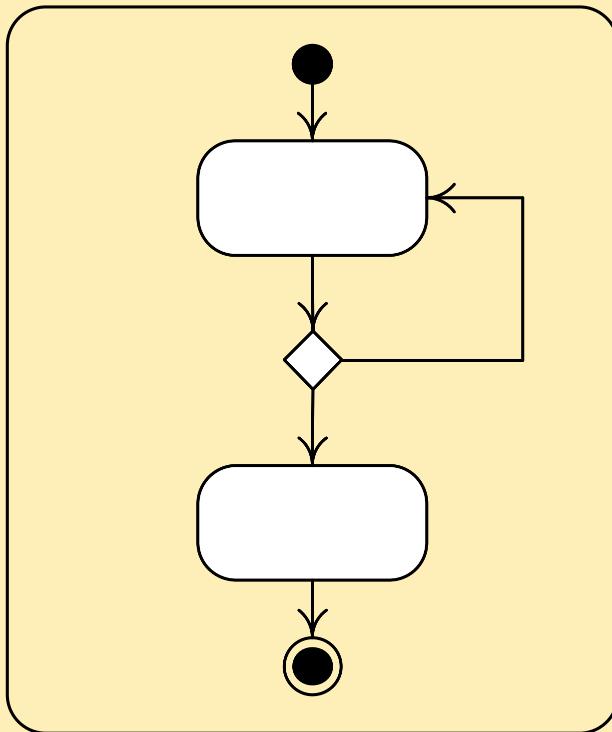
Branching nodes



Branching execution

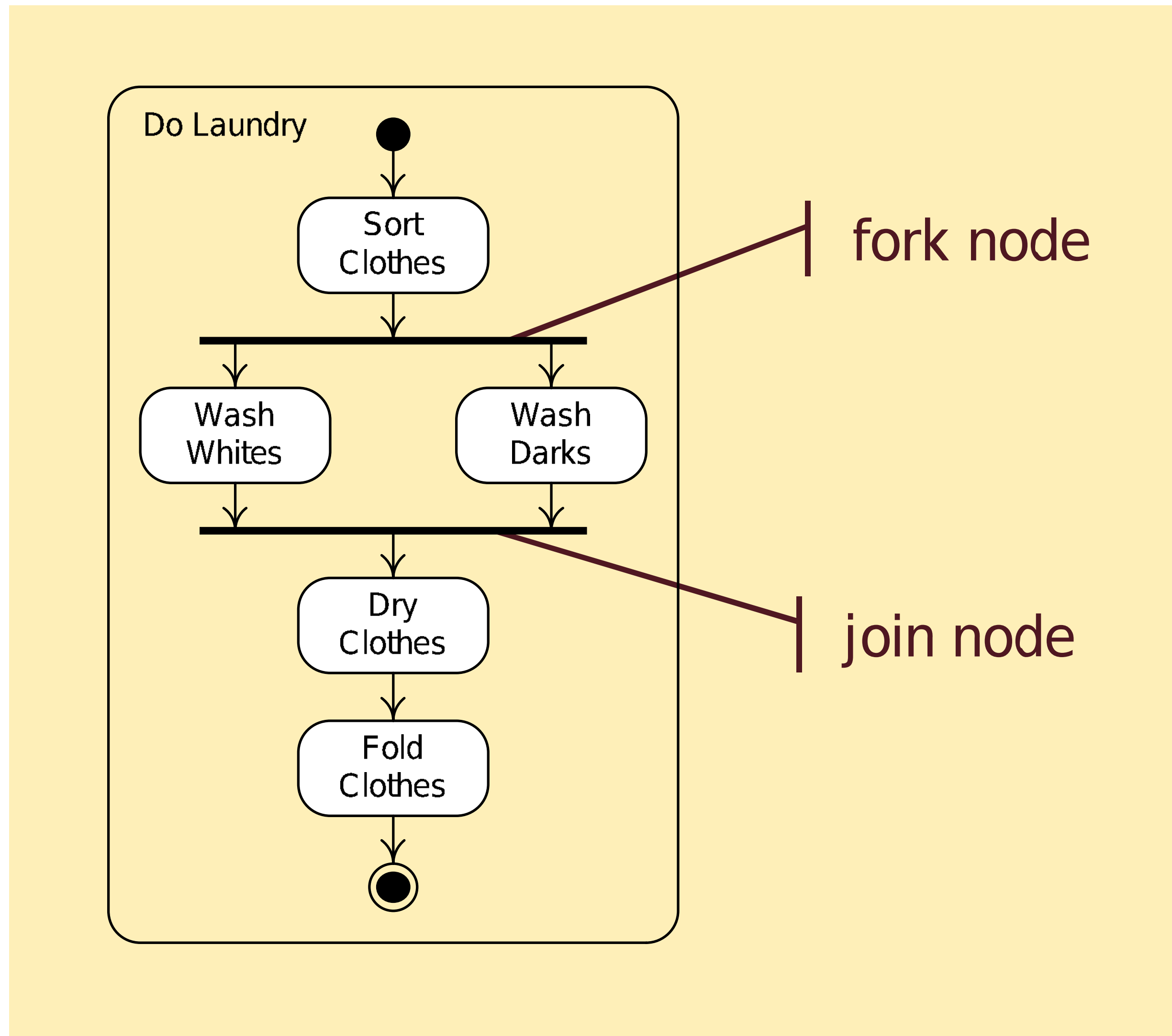
- If a token is made available on the incoming edge of a decision node, the token is made available on the outgoing edge whose guard is true.
- If a token is available on any incoming edge of a merge node, it is made available on its outgoing edge.
- Guards must be mutually exclusive

Deadlock



RunDrier cannot execute: when the activity begins, there is a token on the edge from the initial node but not on the other incoming edge.

Forking and Joining Nodes

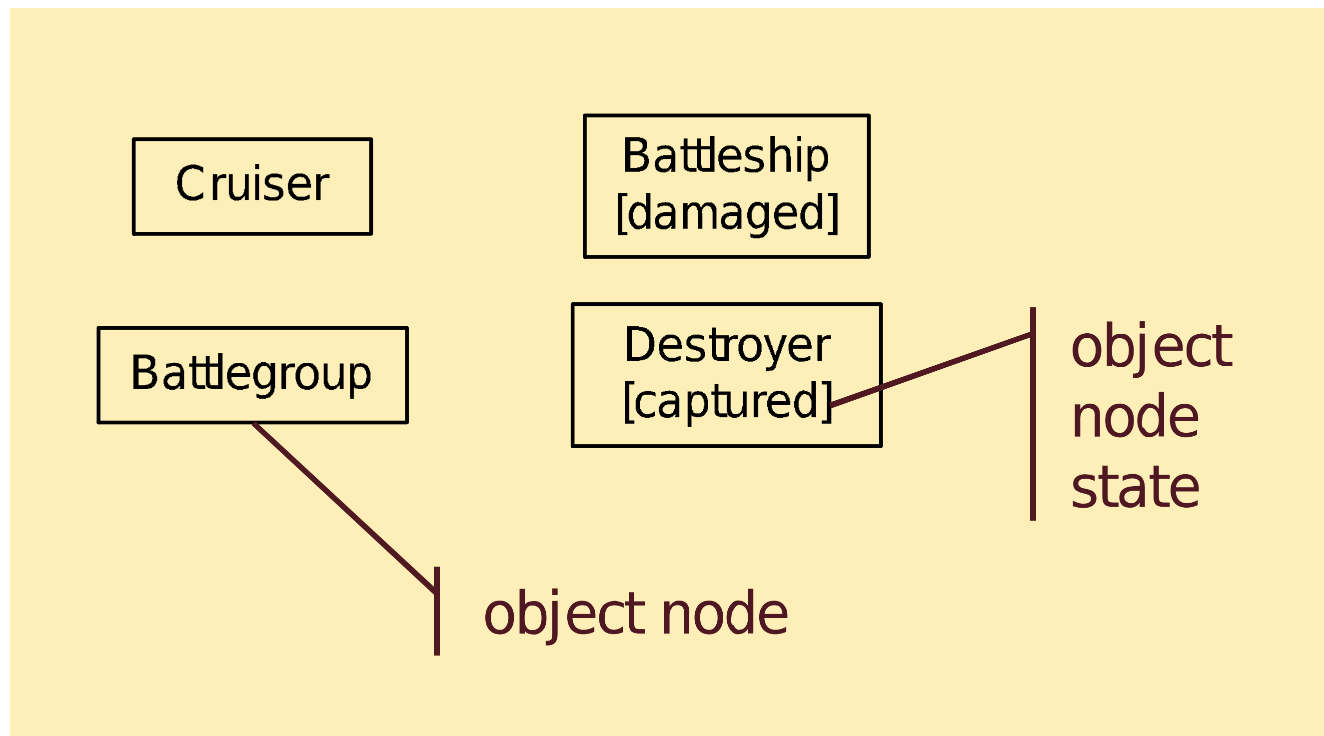


Forking and Joining Execution

- A token available on the incoming edge of a fork node is reproduced and made available on all its outgoing edges.
- When tokens are available on every incoming edge of a join node, a token is made available on its outgoing edge.
- Concurrency can be modeled without these nodes.

Object Nodes

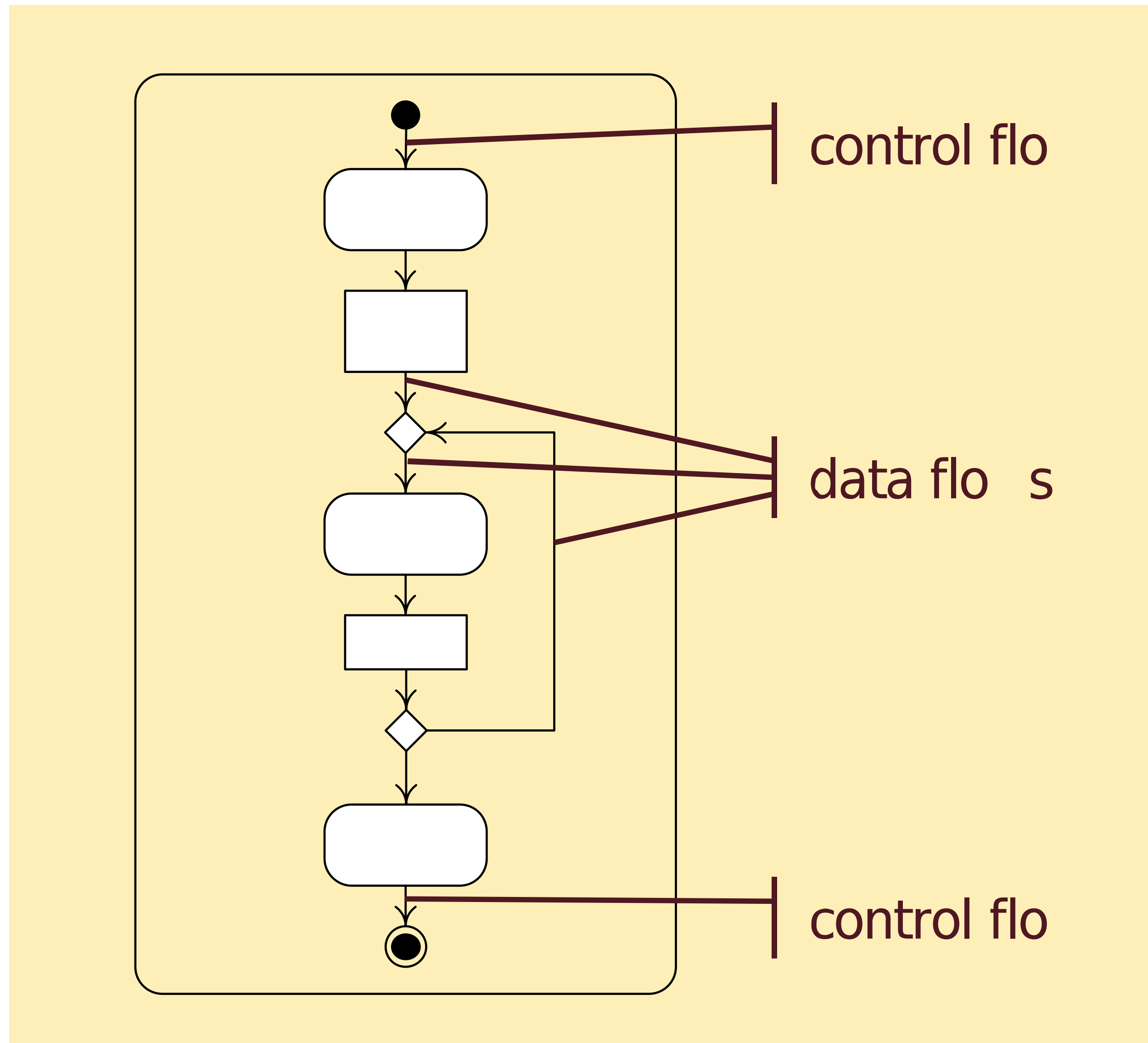
Data and objects are shown as object nodes.



Control and Data Flows

- **Control tokens** do not contain data, **data tokens** do.
- A **control flow** is an activity edge that is a conduit for control tokens.
- A **data flow** is an activity edge that is a conduit for data tokens.
- Rules for token flow through nodes apply to both control and data tokens, except that data is extracted from consumed tokens and added to produced tokens.

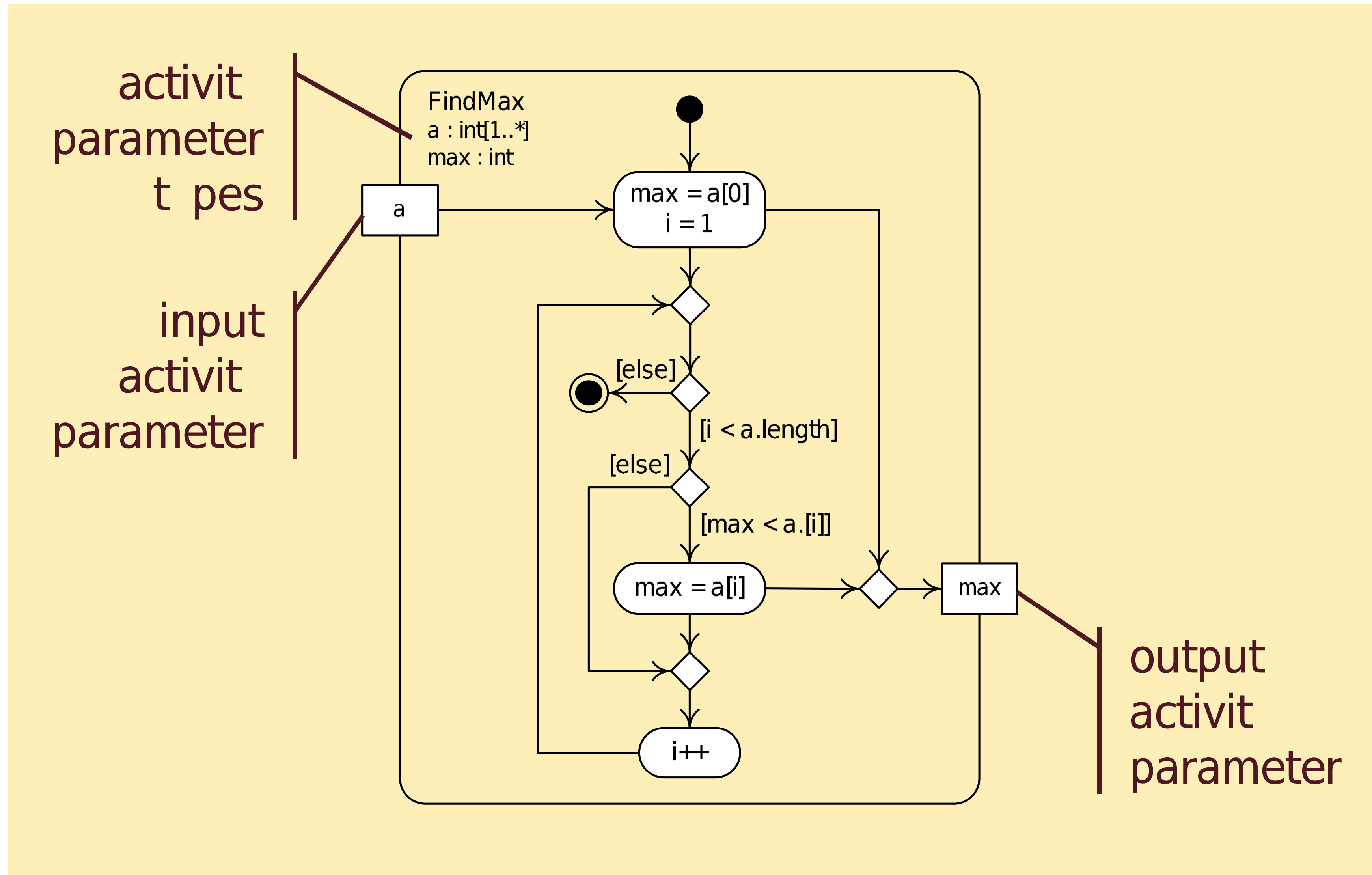
Control and Data Flow Example



Activity parameters

- Activity parameters are object nodes placed on activity symbol boundaries to indicate data or object inputs or outputs.
- Activity parameters contain the data or object name.
- Activity parameter types are specified in the activity symbol beneath the activity name.

Activity Parameter Example



Pragmatics of activity diagrams

- Flow control and objects down the page and left to right
- Name activities and actions with verb phrases.
- Name object nodes with noun phrases.
- Don’t use both control and data flows when a data flow alone can do the job.
- Make sure that all nodes entering an action node can provide tokens concurrently.
- Use the [else] guard at every branch.

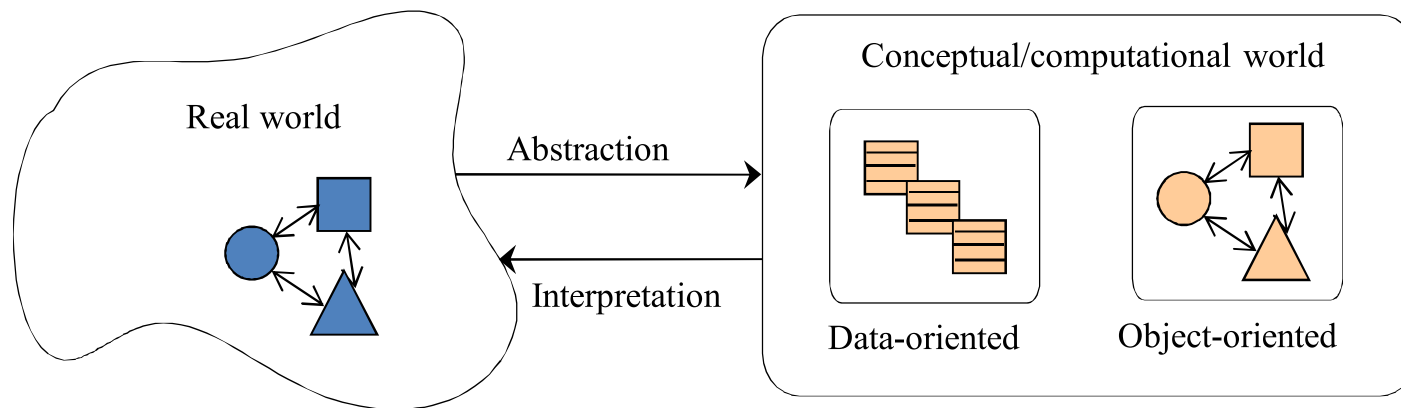
Exercises

Draw an activity diagram for the following situations:

- A user borrows a book from a library and after reading it she gives back the book to the library
- A user asks to buy a book, it is bought, added to the catalog and finally borrowed to the user

Object-Oriented Modeling

- A software system can be modeled as **a set of objects** that interact by exchanging messages
- UML models describe structures of objects and their dynamic behavior
- No semantic gap, seamless development process



Key Ideas of OO Modeling

- Abstraction
  - hide minor details so to focus on major details
- Encapsulation
  - Modularity: principle of separation of functional concerns
  - Information-hiding: principle of separation of design decisions
- Relationships
  - Association: relationship between objects or classes
  - Inheritance: relationship between classes, useful to represent generalizations or specializations of objects
- Object-oriented language model

  = object (class) + inheritance + message send

Main idea

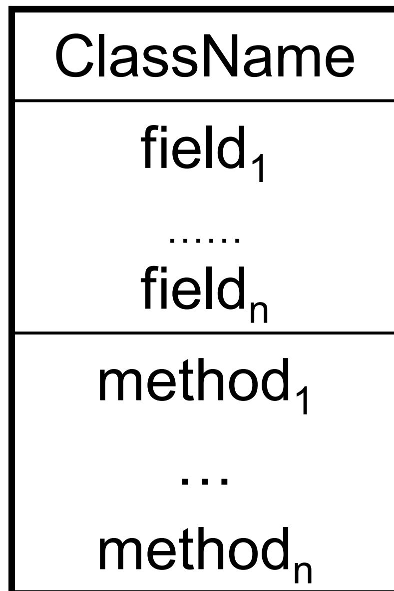
- With UML we model systems made of objects which have some relationships among them
- Objects are instances of classes
- Classes define the structure of objects and their relationships

Class

- A class is the description of a set of objects
- Defines the structure of the states (*attributes*) and the behaviors (*methods*) shared by all the objects of the class (also called *instances*)
- Defines a template for creating instances
  - Names and types of all fields
  - Names, signatures, and implementations of all methods

Notation for classes

- The notation for classes is a rectangular box with three compartments



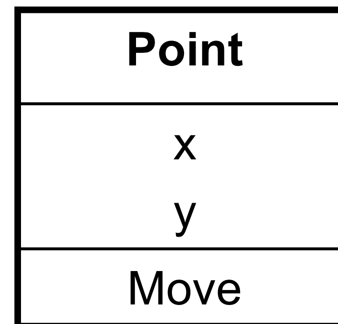
The top compartment shows the class name

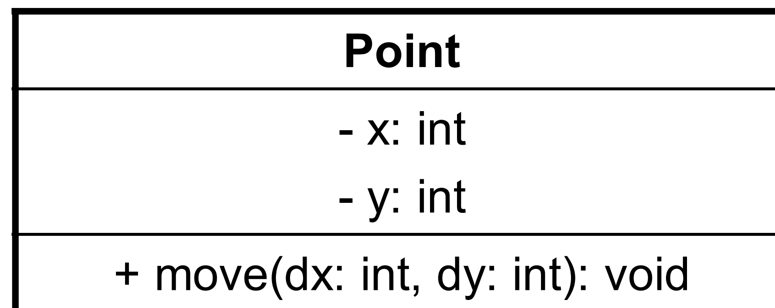
The middle compartment contains the declarations of the fields, or *attributes,* of the class

The bottom compartment contains the declarations of the *methods* of the class

Example

A point class at three different abstraction levels



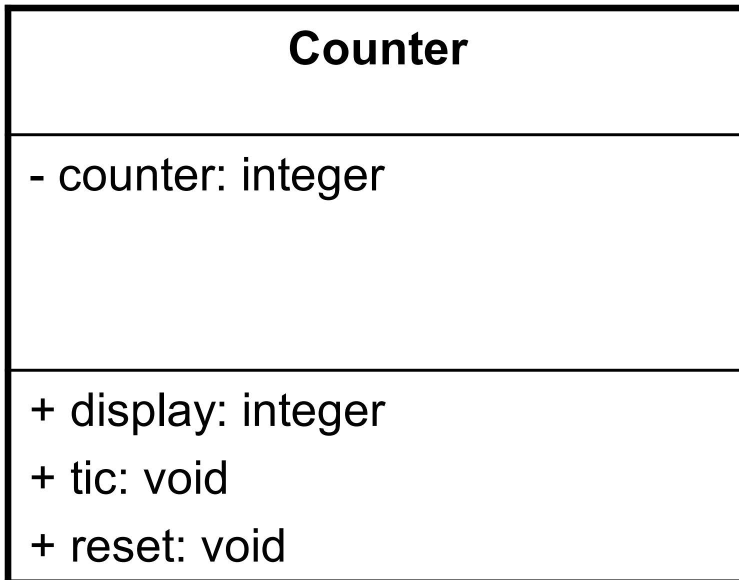


Exercise

Draw a class diagram for the following Java code

```
class Person {
  private String name;
  private Date birthday;
  public String getName() {
    // …
  }
  public Date getBirthday() {
    // …
  }
}
```

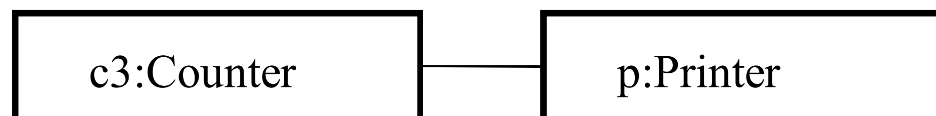
A counter class



A class in UML

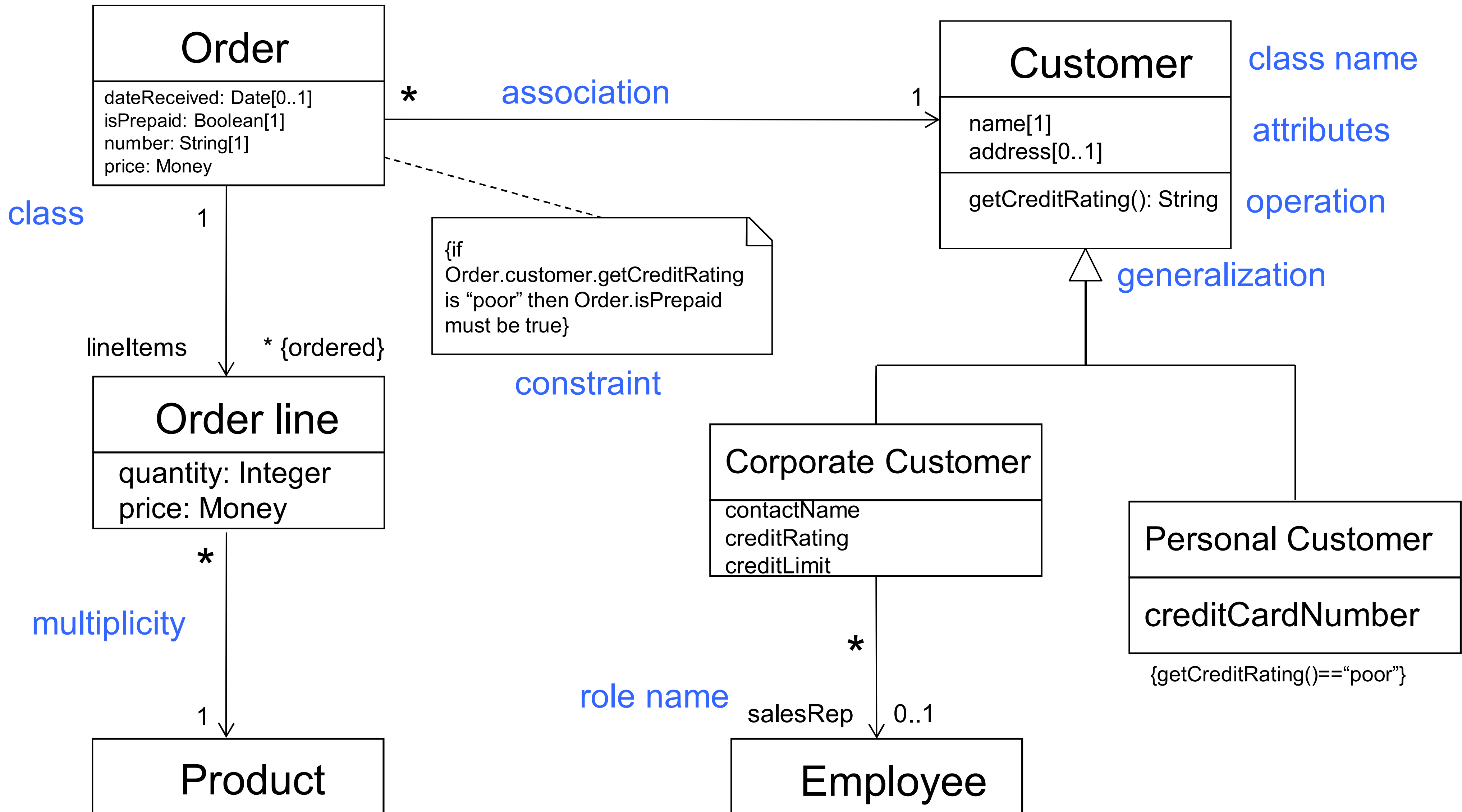
```
class Counter{
  private counter: integer;
  public integer display()
   {return counter};
  public void tic()
   {counter = counter + 1};
  public void reset()
   {counter = 0};
}
```

A corresponding class
in a programming language



Using an object of type class
in an object oriented system

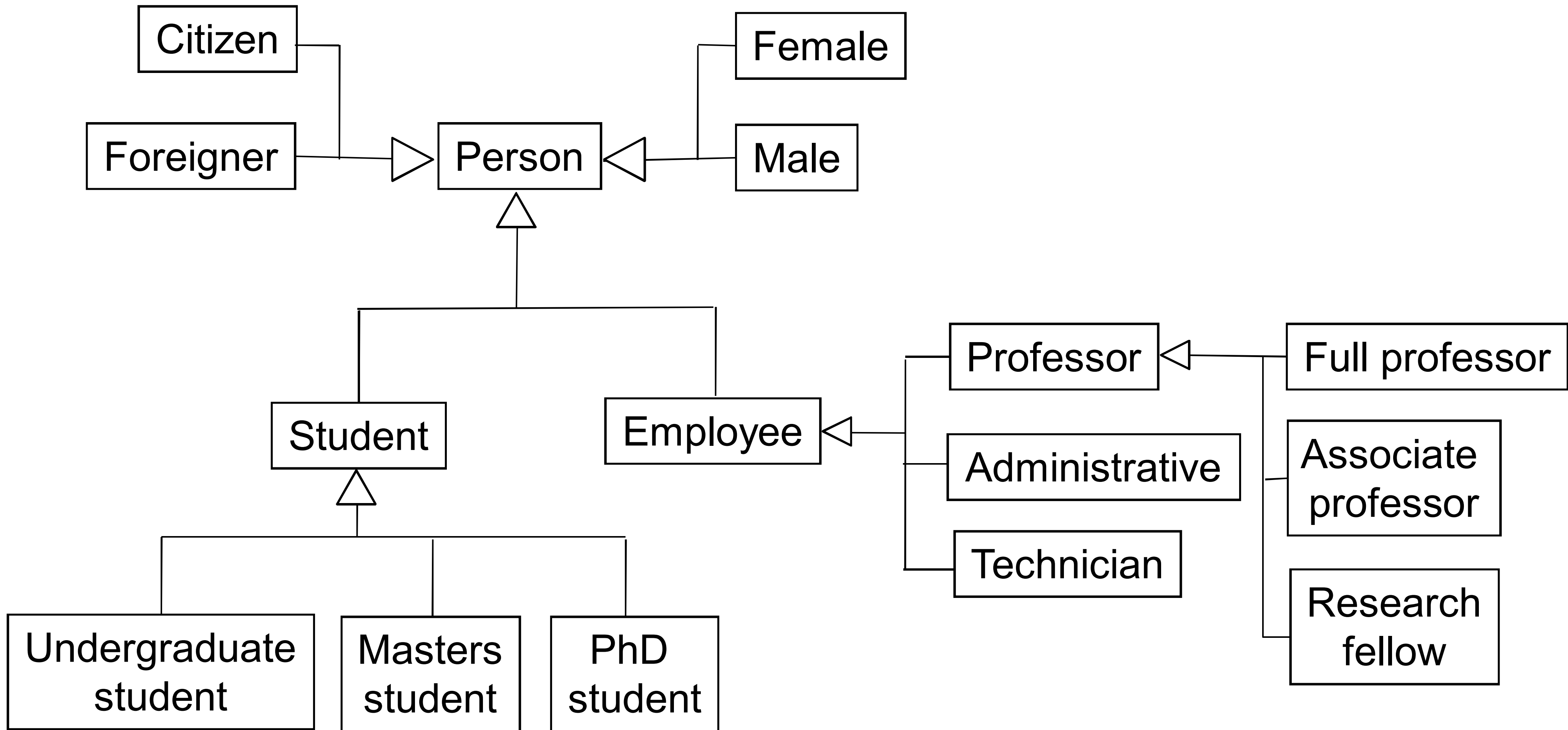
Class diagram: example



Example

- A university is an organization where some persons work, some other study
- There are several types of roles and grouping entities
- We say nothing about behaviors, for the moment

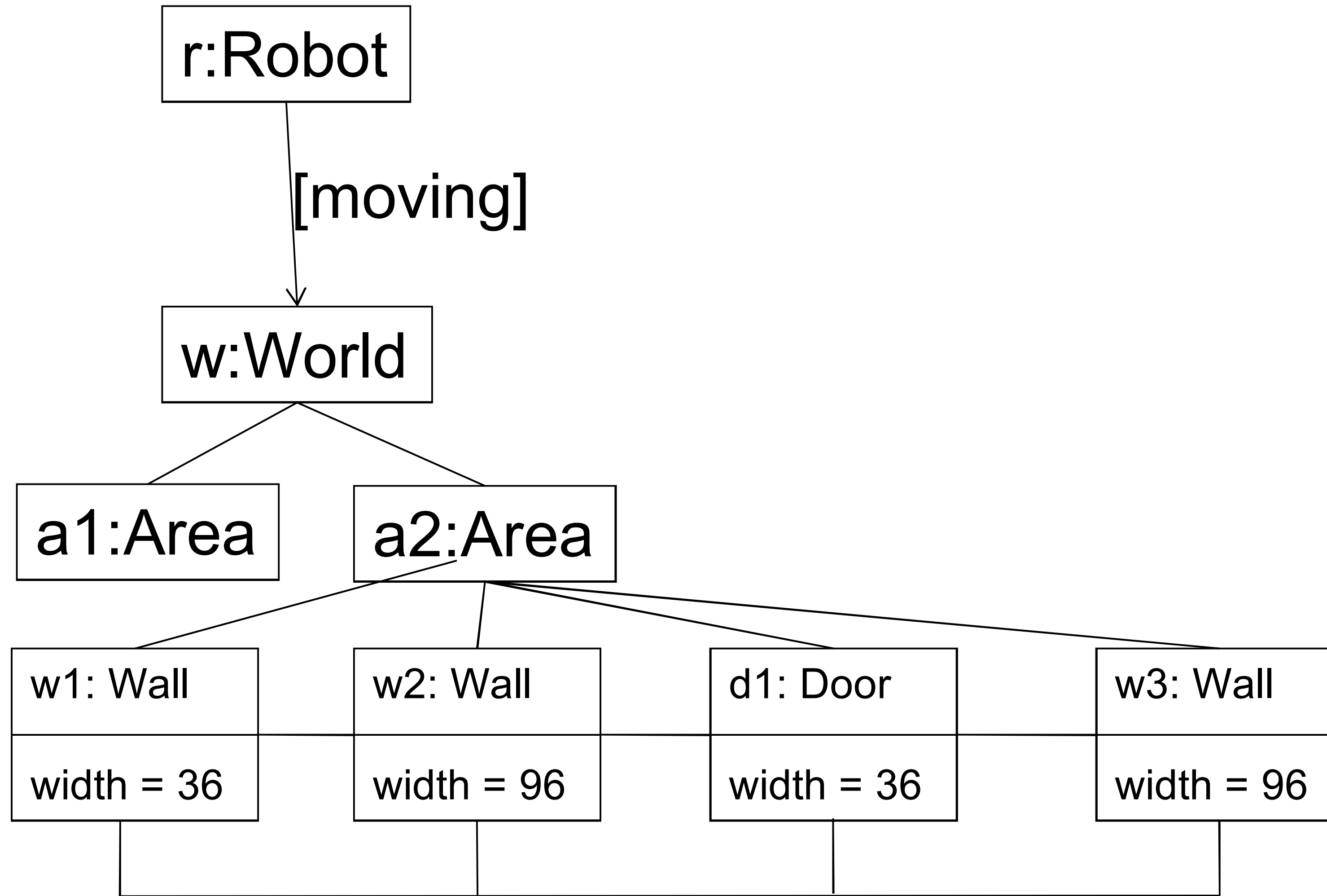
A class diagram



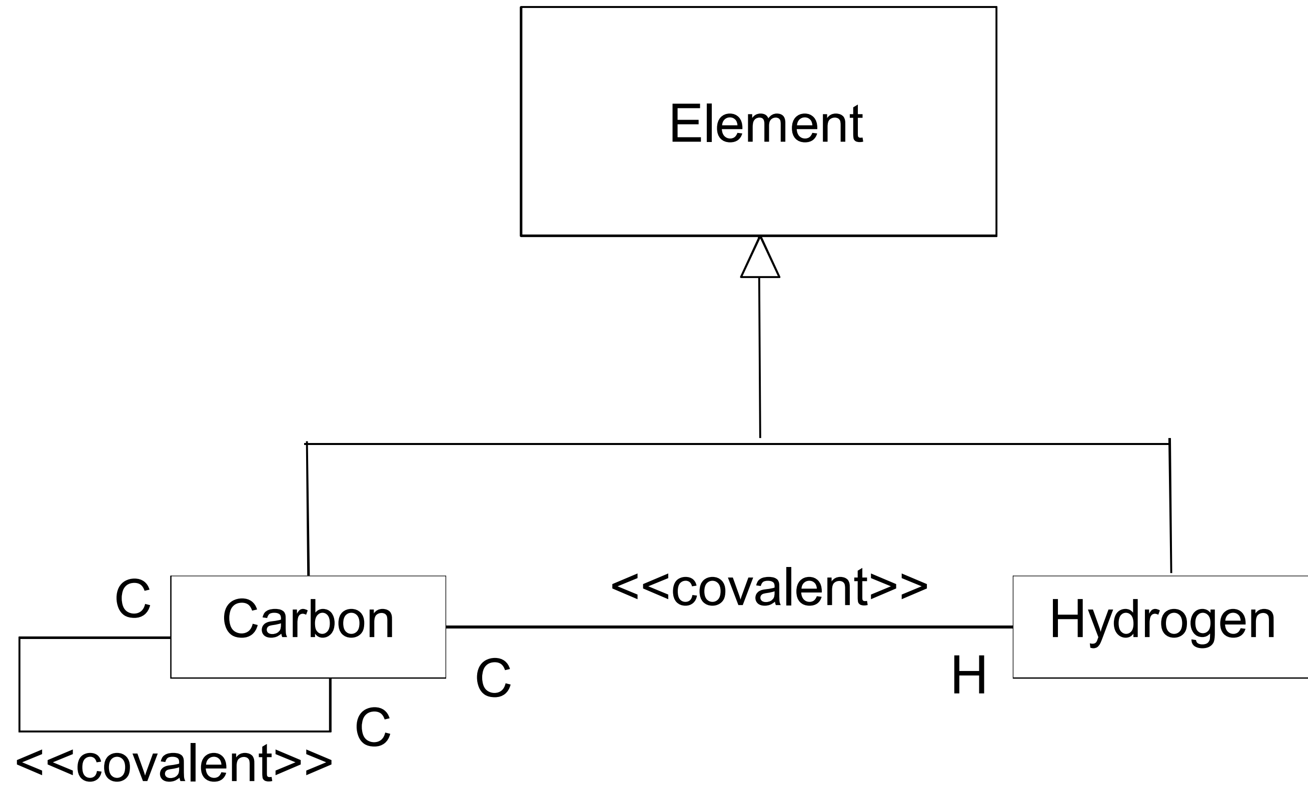
Object diagram

- An object diagram represents a “snapshot” of a system composed by set of objects
- An object diagram looks like a class diagram
- However, there is a difference: values are allocated to attributes and method parameters
- While a class diagram represents an abstraction on source code, an object diagram is an abstraction of running code

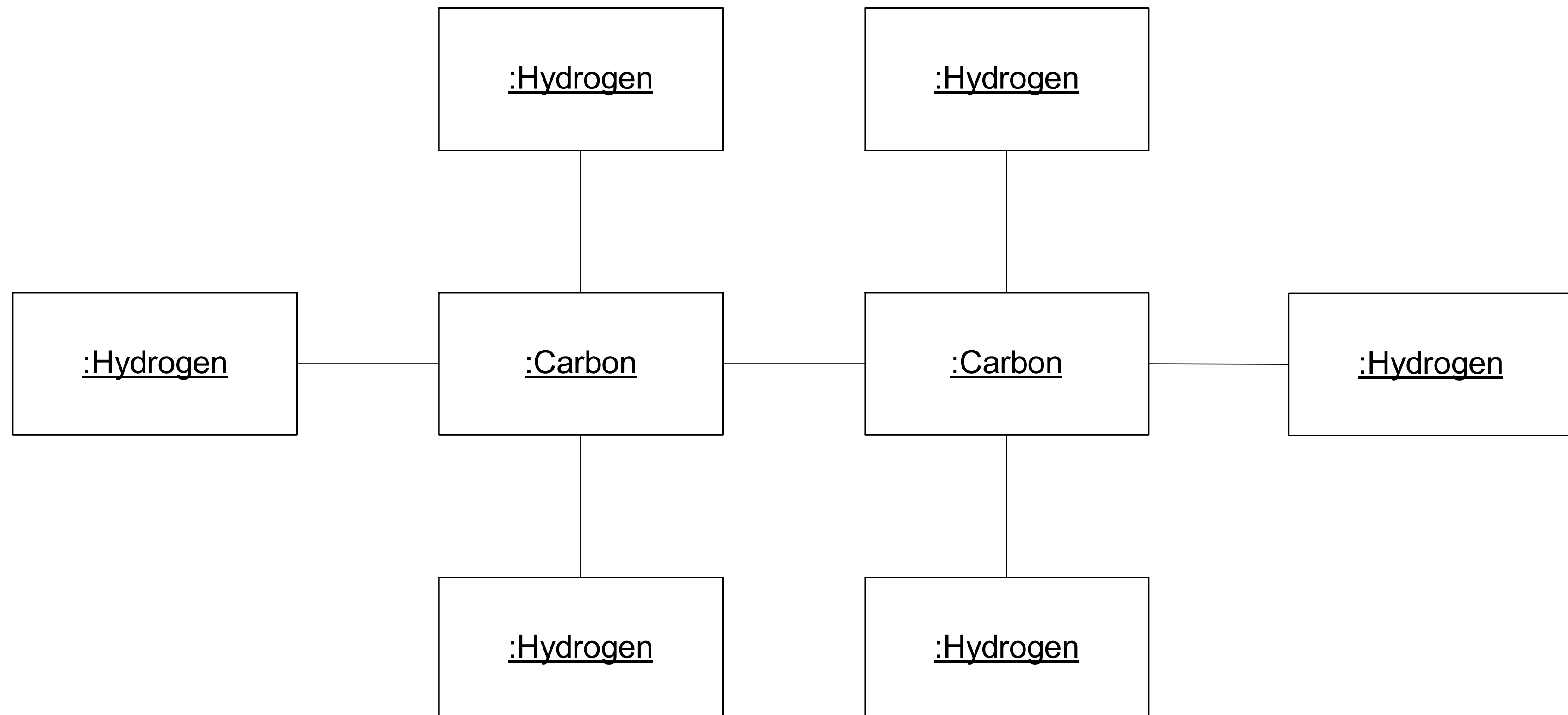
Example (object diagram)



Example: chemical elements (class diagram)



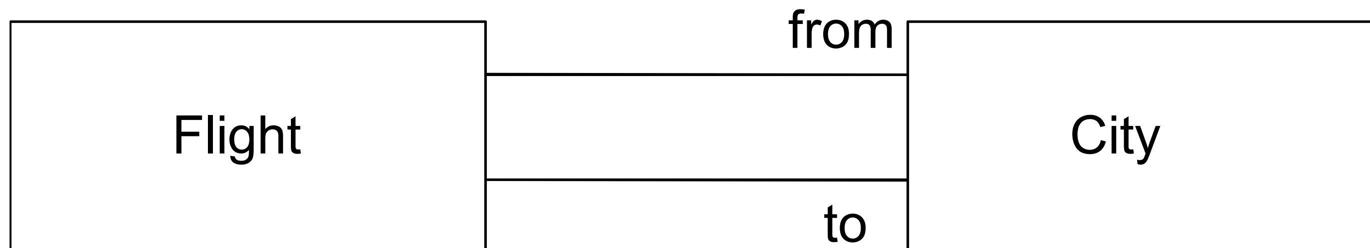
Example: molecule (object diagram)

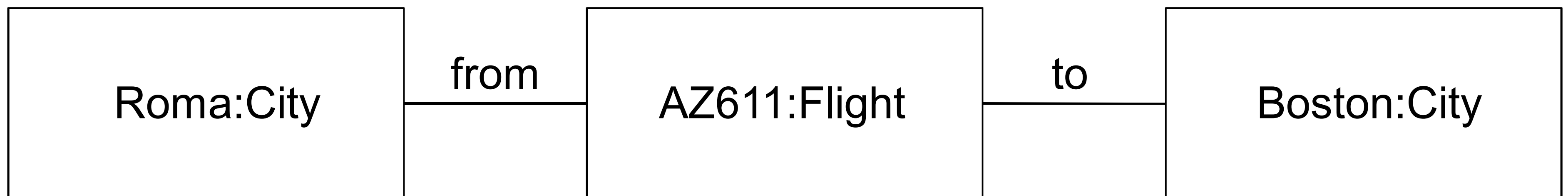


Objects vs. Classes

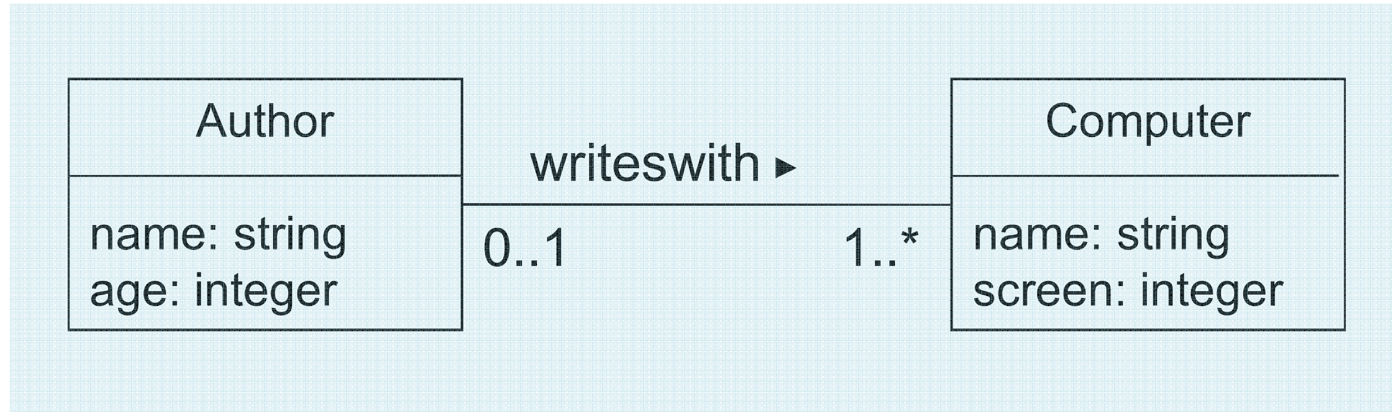
	Interpretation in the real world	Representation in the model
Object	An *object* is anything in the real world that can be distinctly identified	An *object* has an identity, a state, and a behavior
Class	A *class* is a set of objects with similar structure and behavior. These objects are called *instances* of the class	A *class* defines the structure of states and behaviors that are shared by all of its instances

Class diagrams denote systems of objects

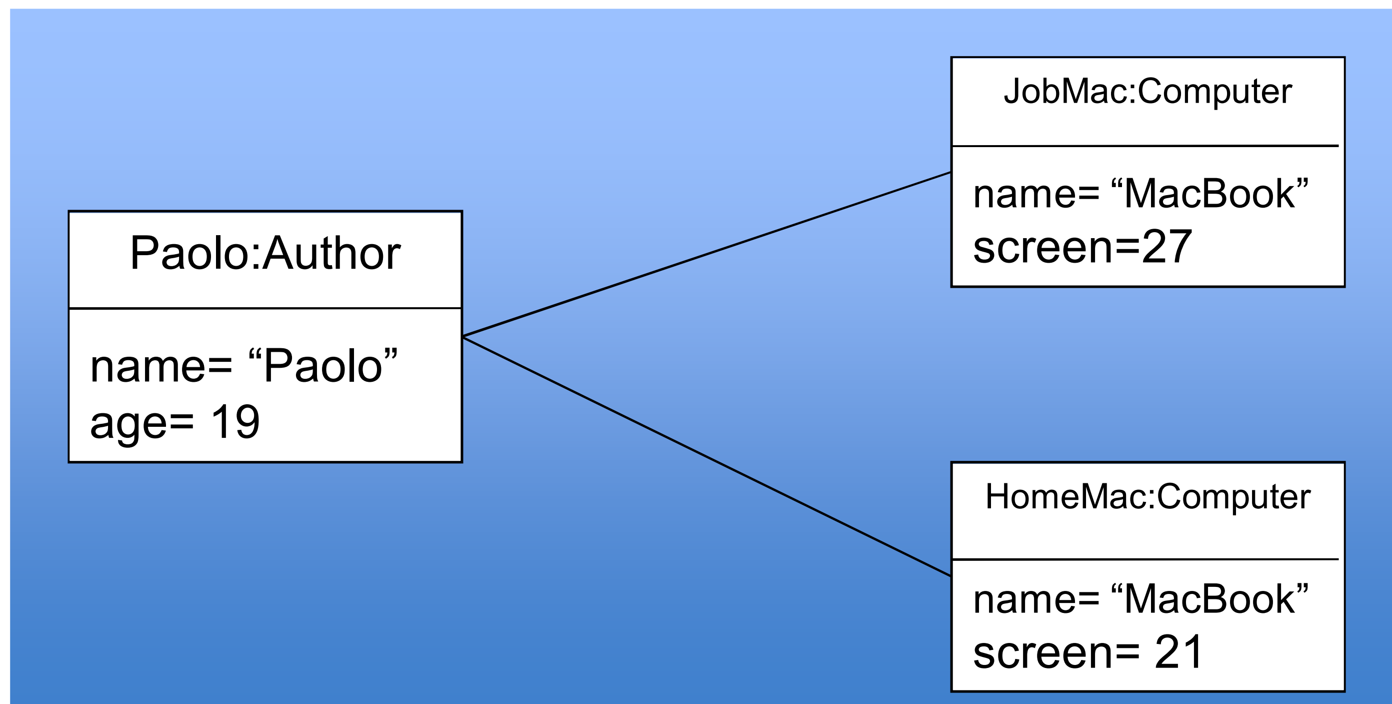




Object diagram instantiating a class diagram



Class diagram



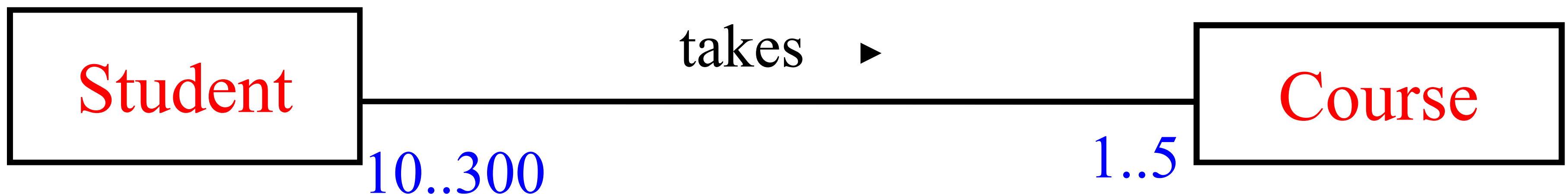
Object diagram that is an instance of the class diagram above

Roles and multiplicity

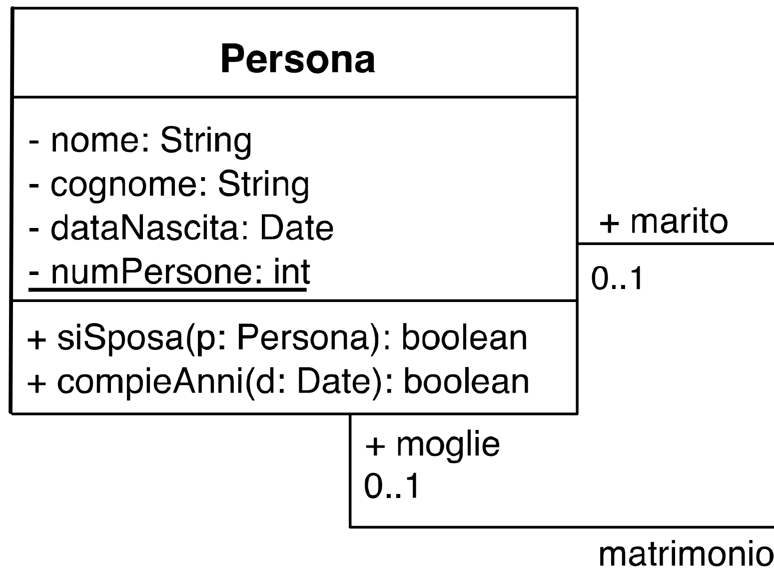
- An association line may have a *role name* and a *multiplicity specification*
- The multiplicity specifies an integer interval, e.g.,
  - *l..u* closed (inclusive) range of integers
  - i singleton range
  - 0..* nonnegative integer, i.e., 0, 1, 2, …

Association example

- A Student can take up to five Courses
- Every Student has to be enrolled in at least one course
- Up to 300 students can enroll in a course
- A class should have at least 10 students



Example



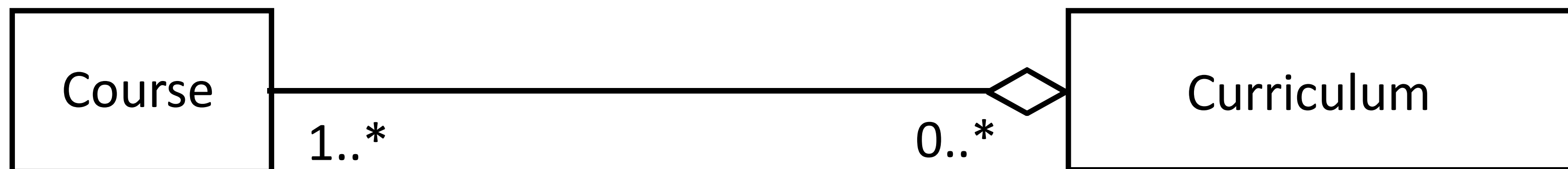

```
class Persona {
   private String nome;
   private String cognome;
   private Date dataNascita;
   private static int numPersone;
   public Persona marito;
   public Persona moglie;

   public boolean siSposa(Persona p) {
      ...
   }

   public boolean compieAnni(Date d) {
      ...
   }
}
```

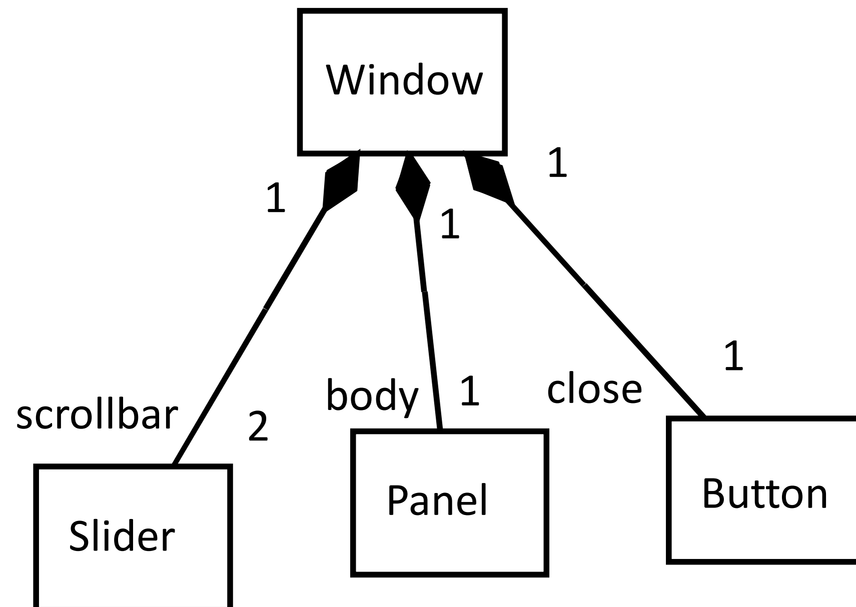
Aggregations

- They are specialized associations that stress the containment between the two classes
- We have a part-of relationship



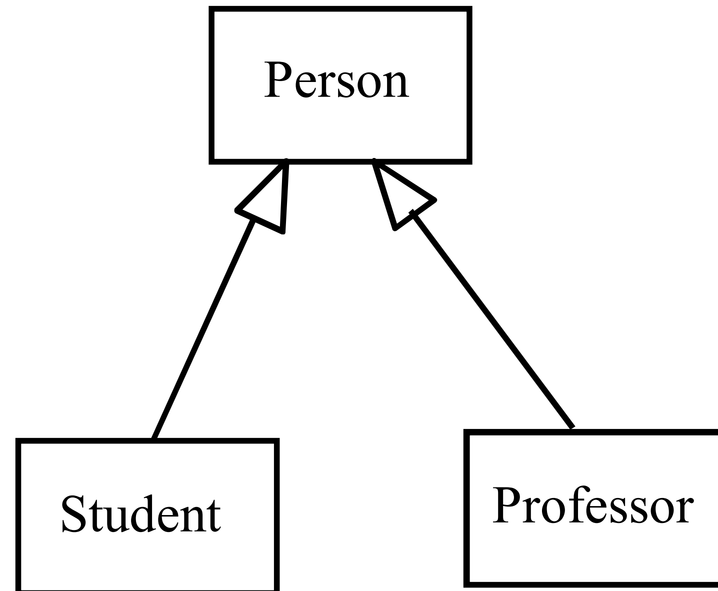
Composites

- Composites are heavy aggregations
  - The contents is subordinated to the container
  - For example, deleting the container means deleting the contents as well



Inheritance (Generalization)

- Makes common properties explicit
- Inheritance is an elegant modeling means, but
  - It is not mandatory
  - Maybe we must add properties
  - Maybe we must refine/modify other properties
- We can work
  - Bottom-up (Generalization)
  - Top-down (Specialization)



Conclusions

- UML is a modeling language born for object oriented (software) systems
- It especially effective for describing complex systems and reusing design ideas
- Next lecture deals with the topic of reusing design ideas expressed in UML