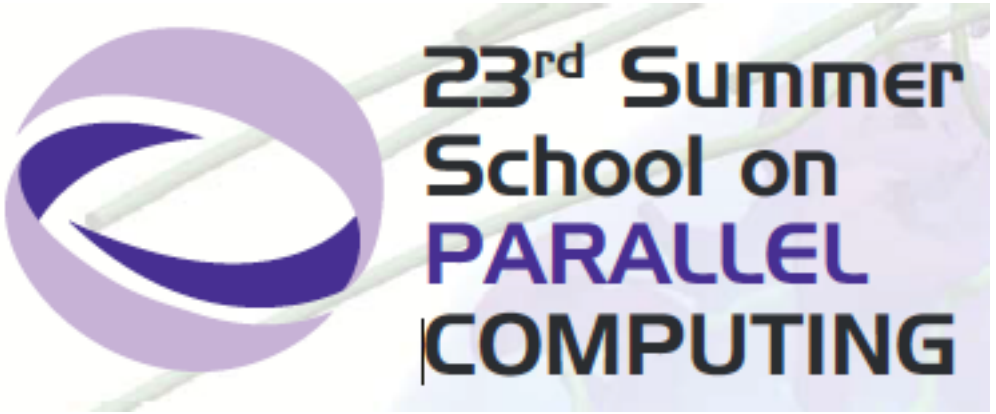


Introduction to UML

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

Software models

- A model is a *description* of the structure and meaning 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

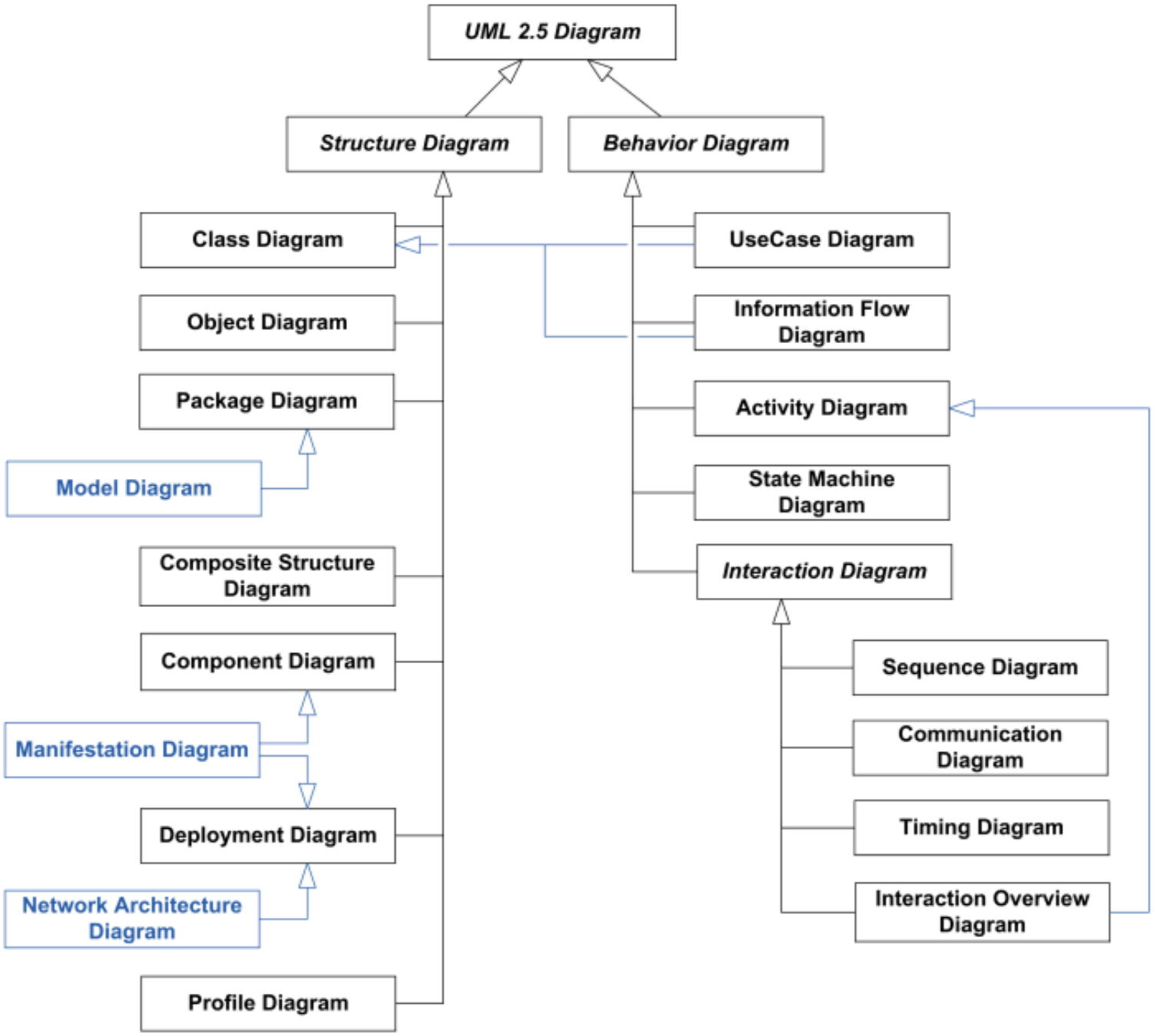
  - 2005: UML 2.0
  - 2007: UML 2.1.2
  - 2009: UML 2.2
  - 2010: UML 2.3
  - 2011: UML 2.4
  - 2013: UML 2.5

pre-UML

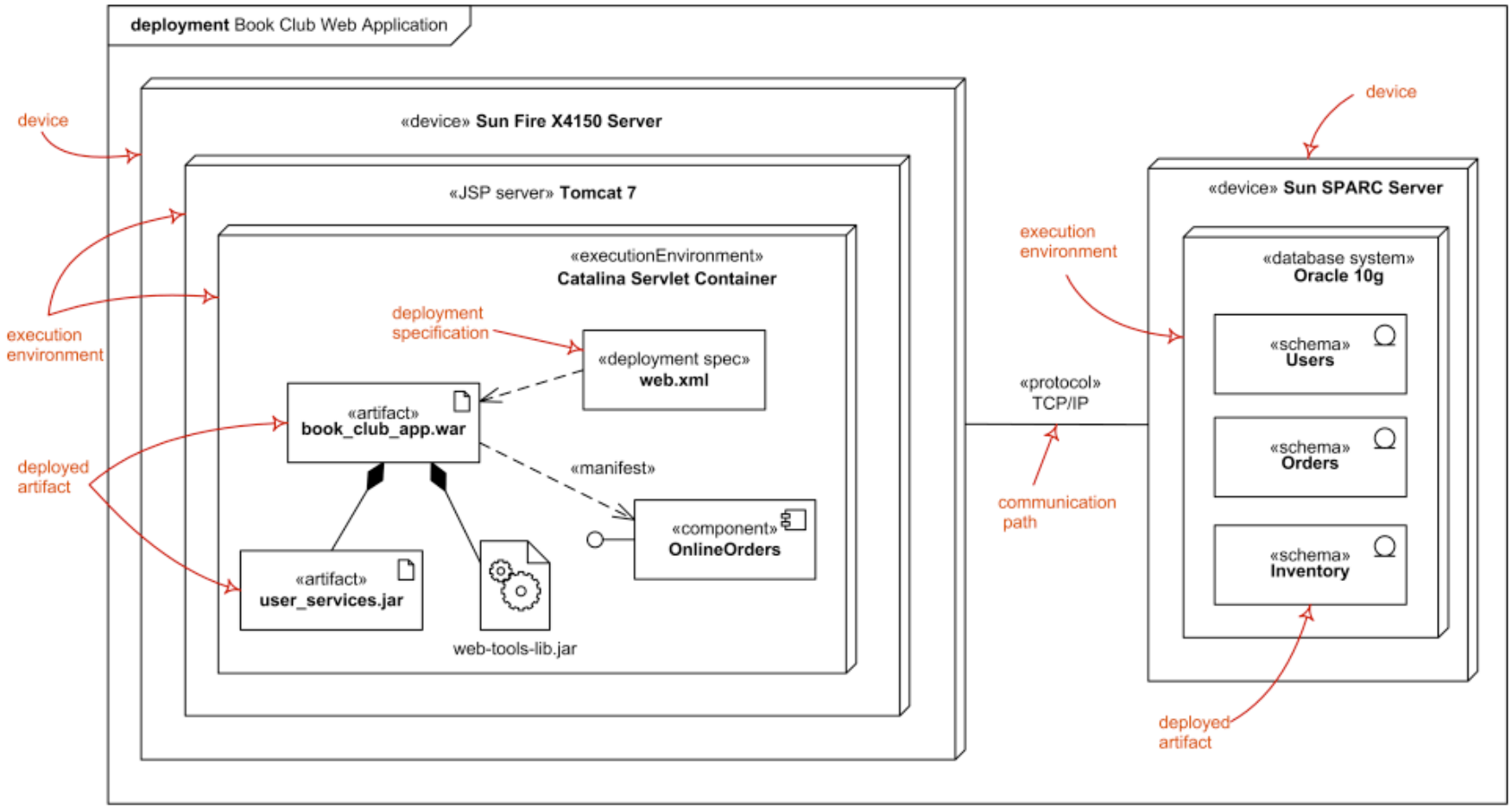
UML 1.x

UML 2.0

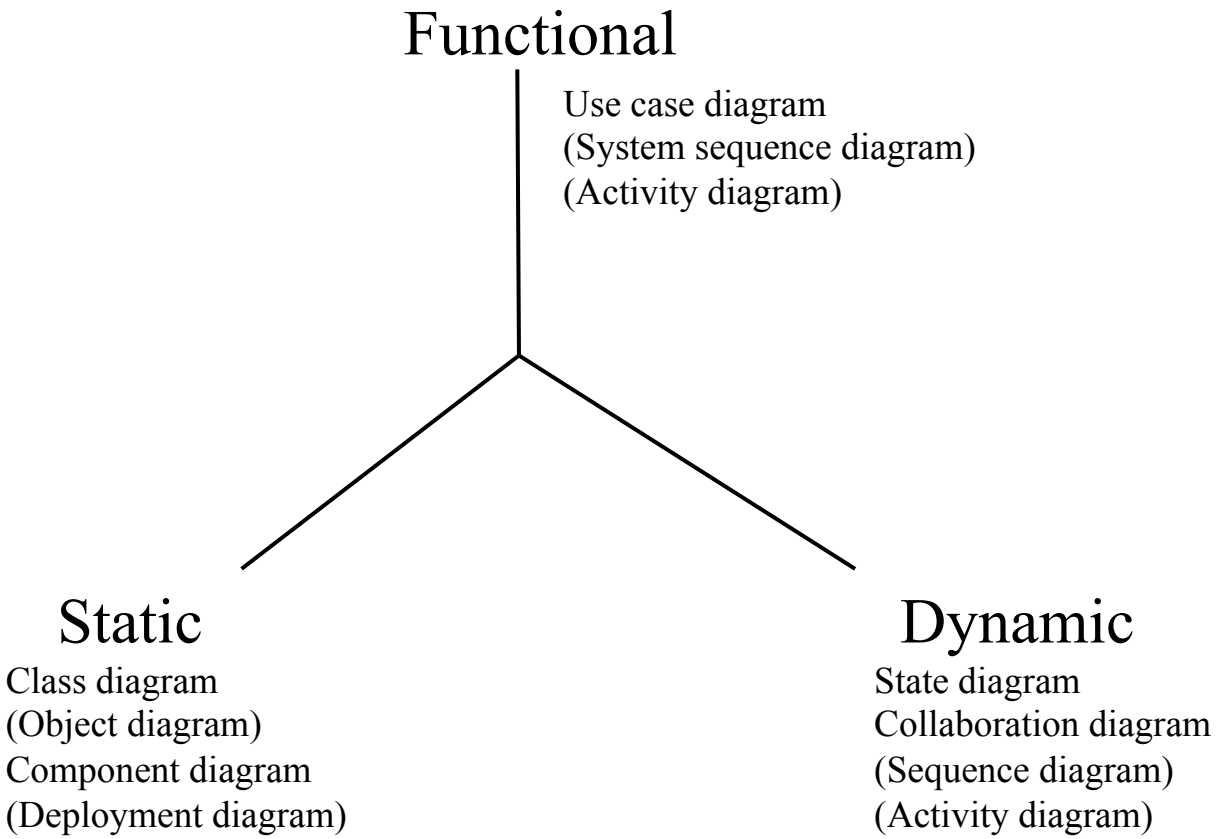
UML 2.5 (2013)


UML 2.5 Diagram
Structure Diagram
Behavior Diagram
Class Diagram
Object Diagram
Package Diagram
Model Diagram
Composite Structure Diagram
Component Diagram
Manifestation Diagram
Deployment Diagram
Network Architecture Diagram
Profile Diagram
UseCase Diagram
Information Flow Diagram
Activity Diagram
State Machine Diagram
Interaction Diagram
Sequence Diagram
Communication Diagram
Timing Diagram
Interaction Overview Diagram

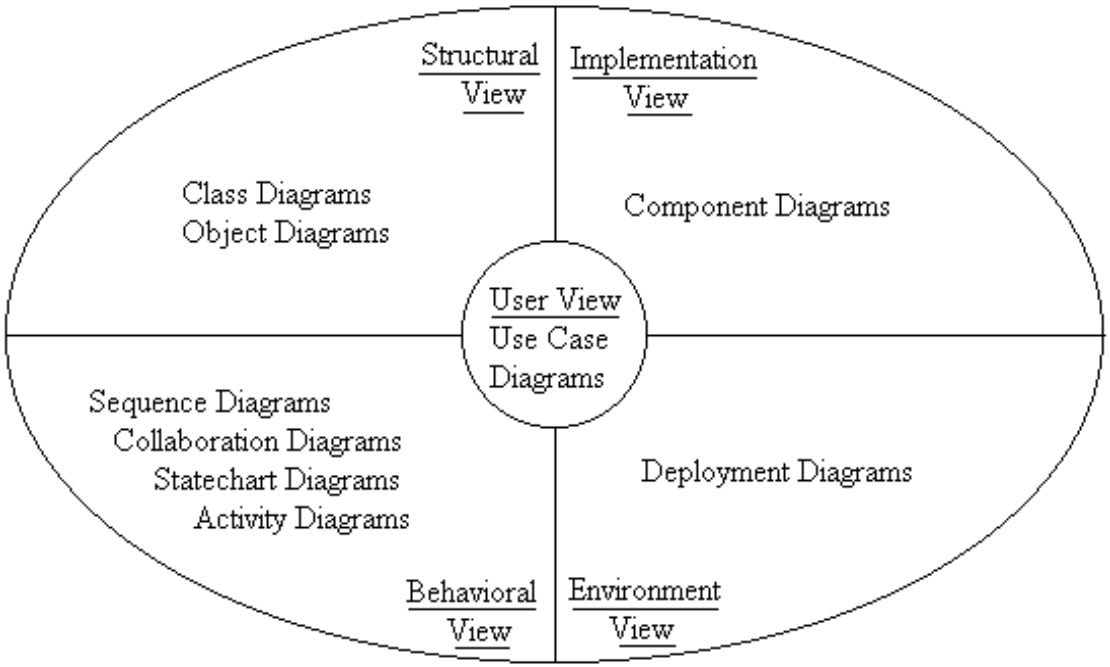
Deployment diagram



Three modeling axes



Another organization



Example

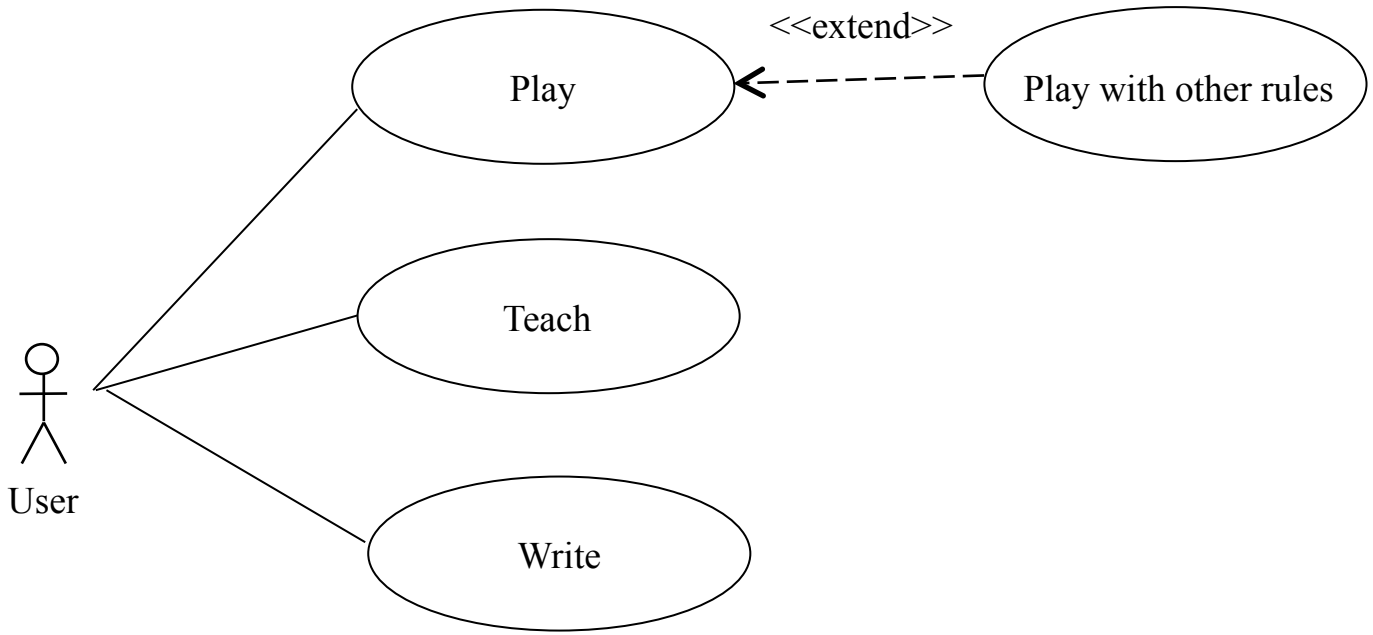
- A chess program could be “stand-alone”, “client-server”, “agent based”, etc.
- Its behavior should always be coherent with the rules of chess
- What is its **goal**? To play and win a chess game against an opponent

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

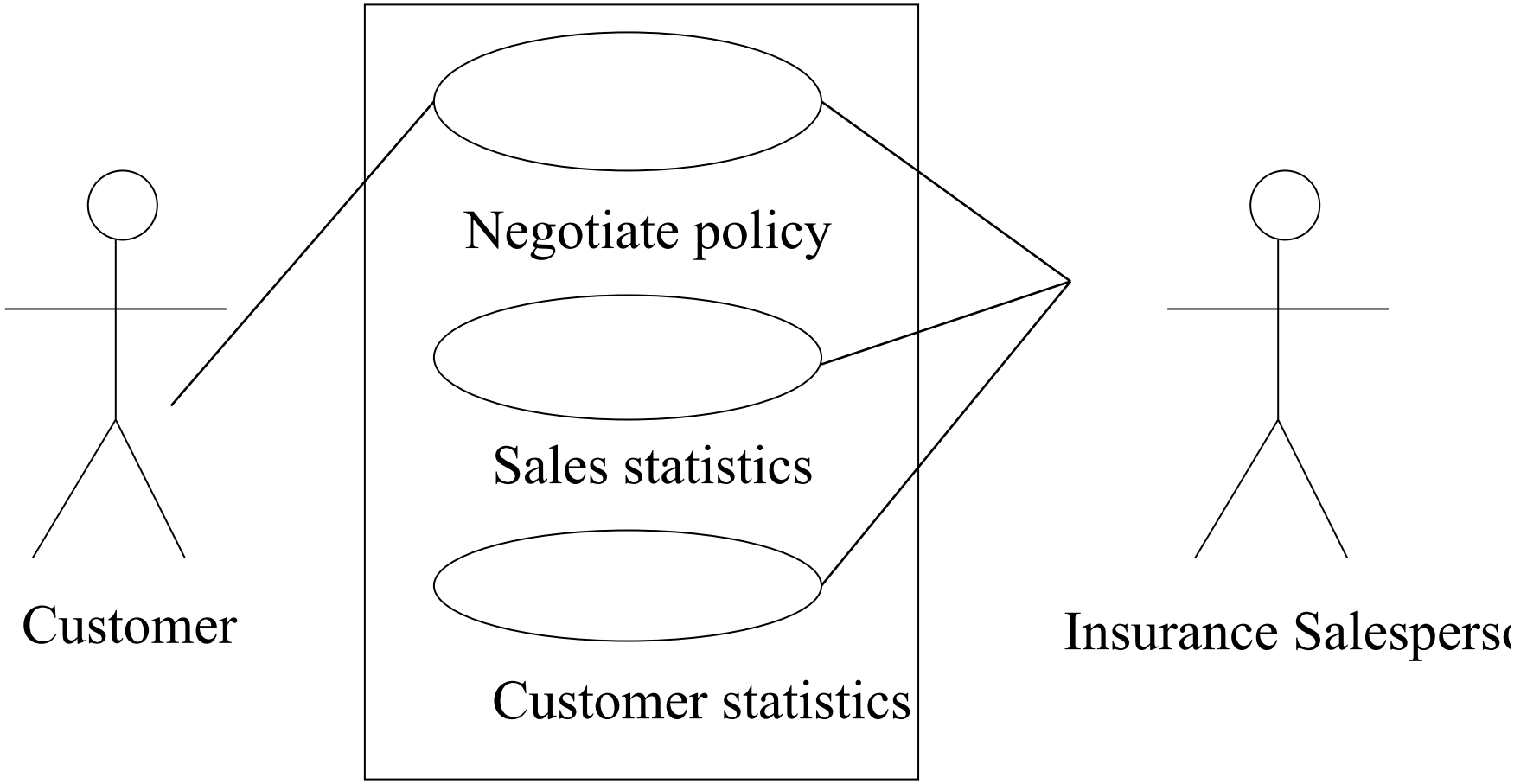
From responsibilities to use cases



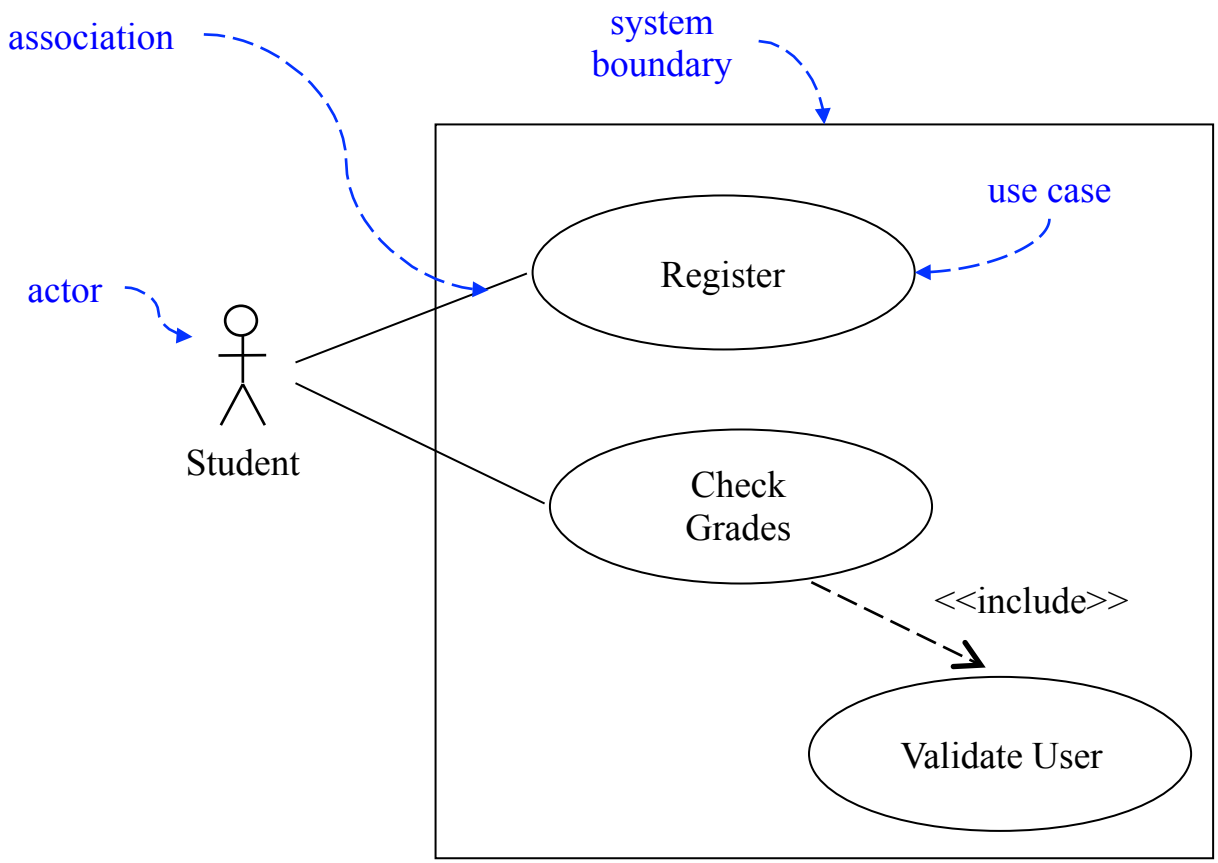
Use Case diagram

- It describes the externally observable behavior of a system, as related to **requirements**
- 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

Example


Negotiate policy
Sales statistics
Customer statistics
Customer
Insurance Salesperso

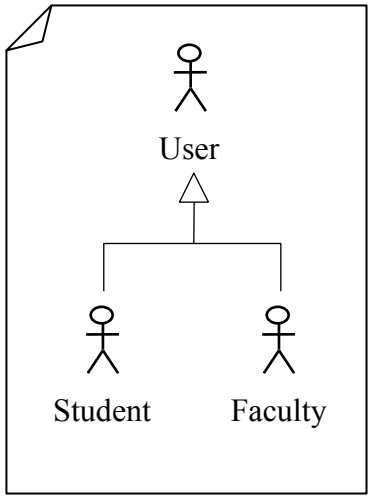
Use Case: elements

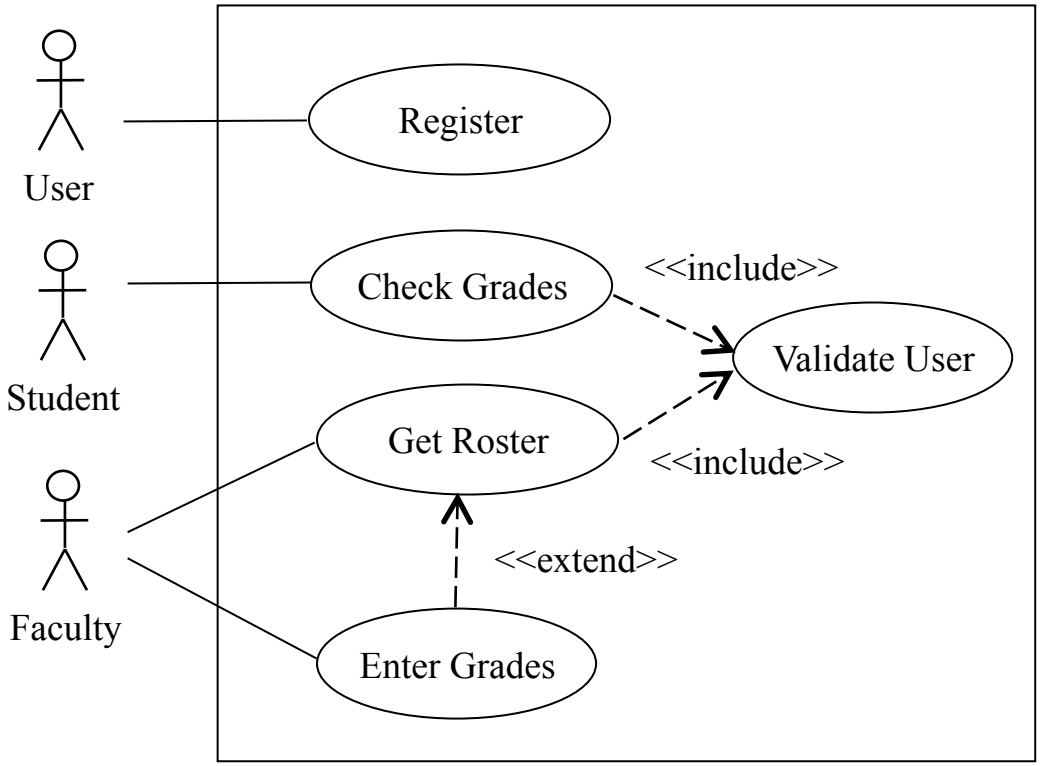


Elements of a Use Case Diagram

- **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


User
Student
Faculty


User
Student
Faculty
Register
Check Grades
Get Roster
Enter Grades
Validate User
<<include>>
<<include>>
<<extend>>

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>

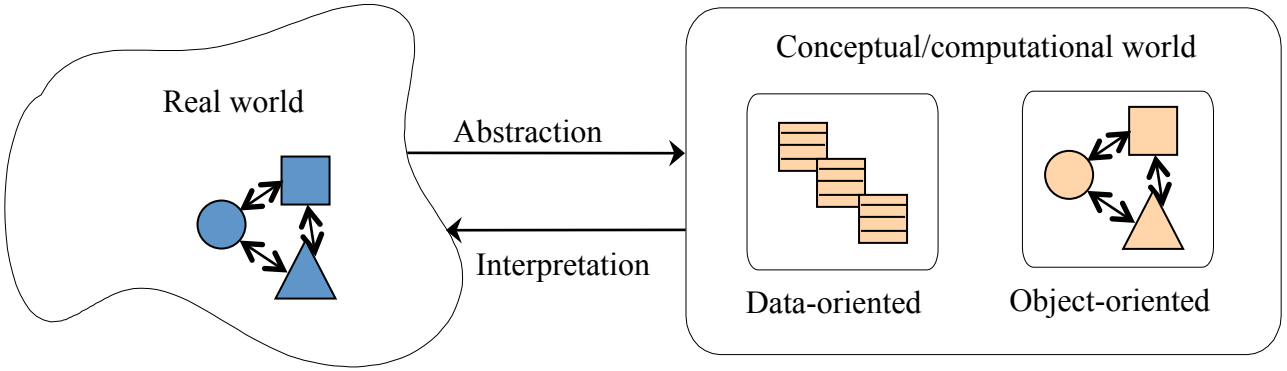
Exercise

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

Object-Oriented Modeling

- Models describe structures of objects and their behavior
- A system is modeled as **a set of objects** that interact by exchanging messages
- 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

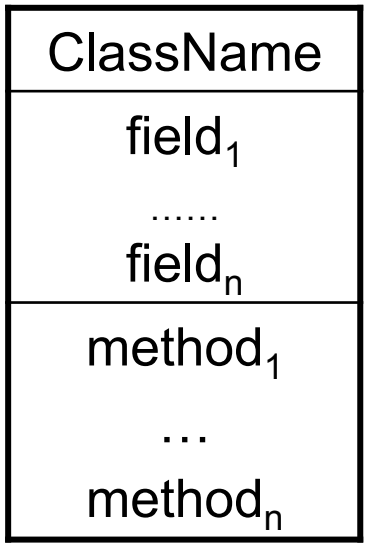
- With UML we model systems made of objects which have relationships among them
- Objects are instances of classes
- Classes define the structure of objects and their relationships

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



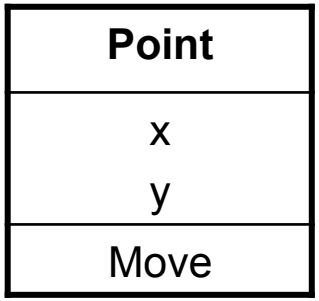
The top compartment shows the class name

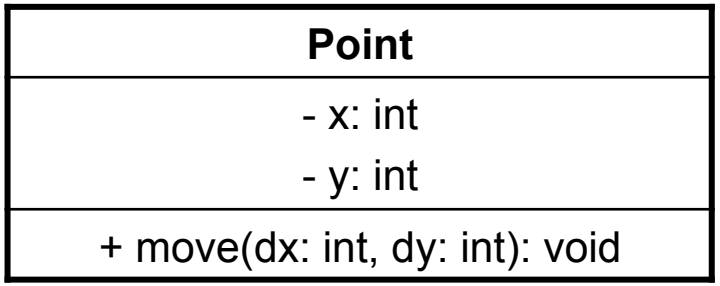
The middle compartment contains the declarations of the fields, or *attributes,* of the class

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

Example

A point class at three different abstraction levels



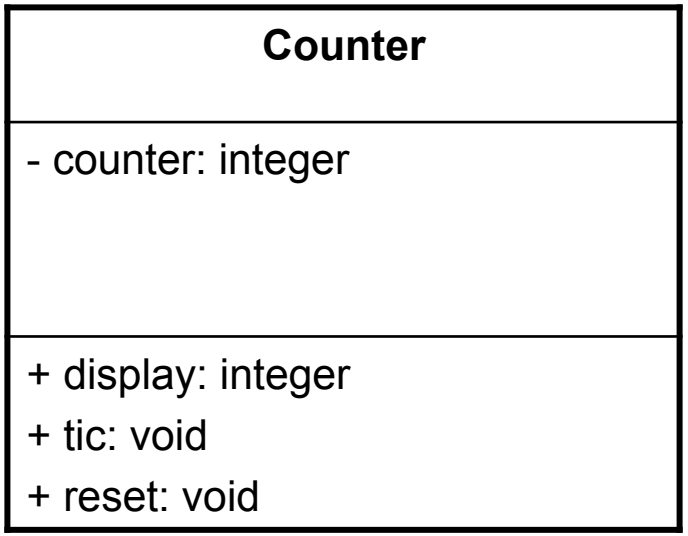


Exercise

Draw a class diagram for the following Java code

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

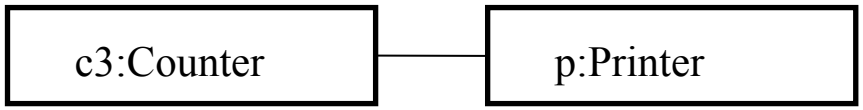
A counter class



A class in UML

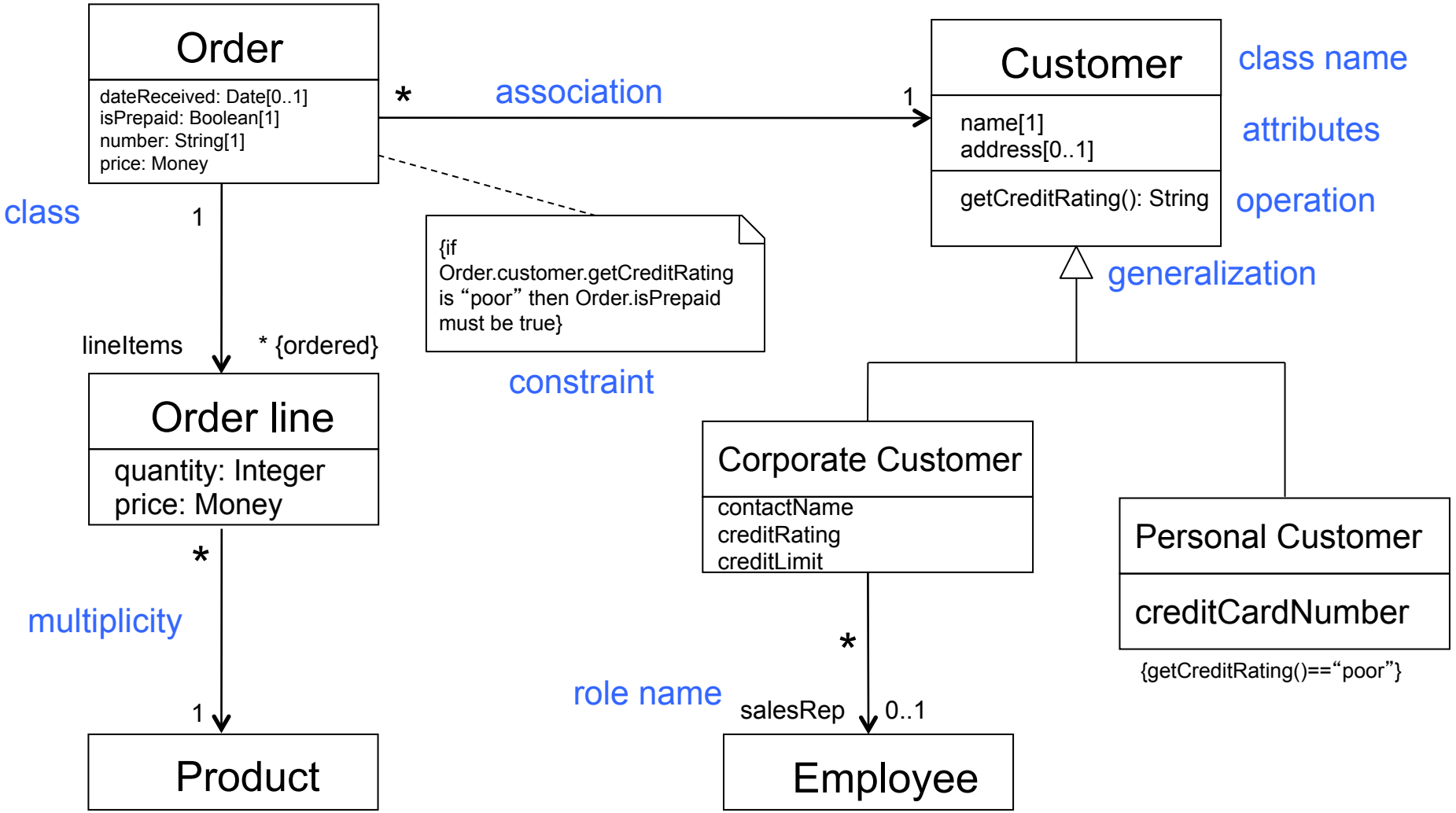
```
class Counter{
  private counter: integer;
  public integer display()
   {return counter};
  public void tic()
   {counter = counter + 1};
  public void reset()
   {counter = 0};
}
```

A corresponding class
in a programming language



Using an object of type class
in an object oriented system

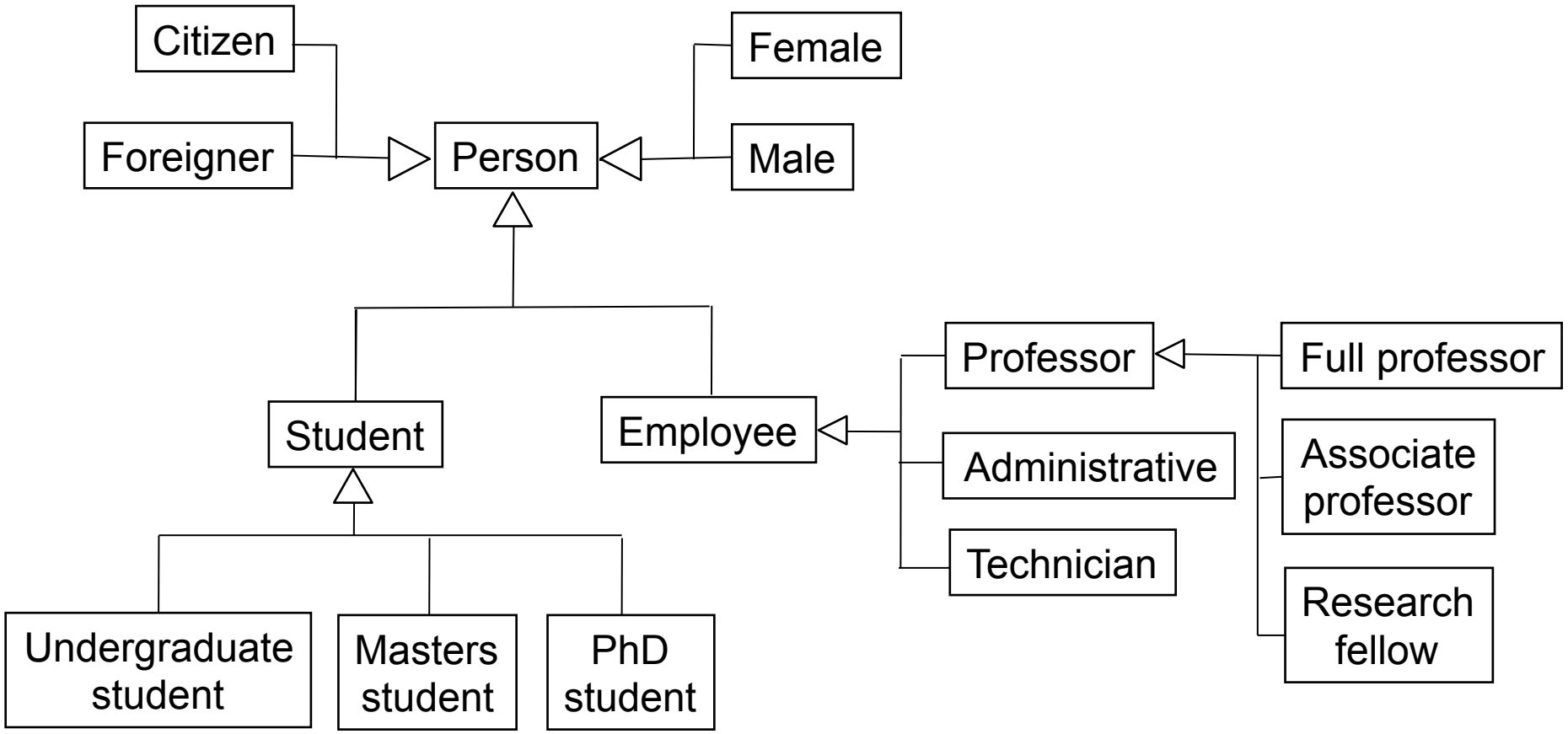
Class diagram: example



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

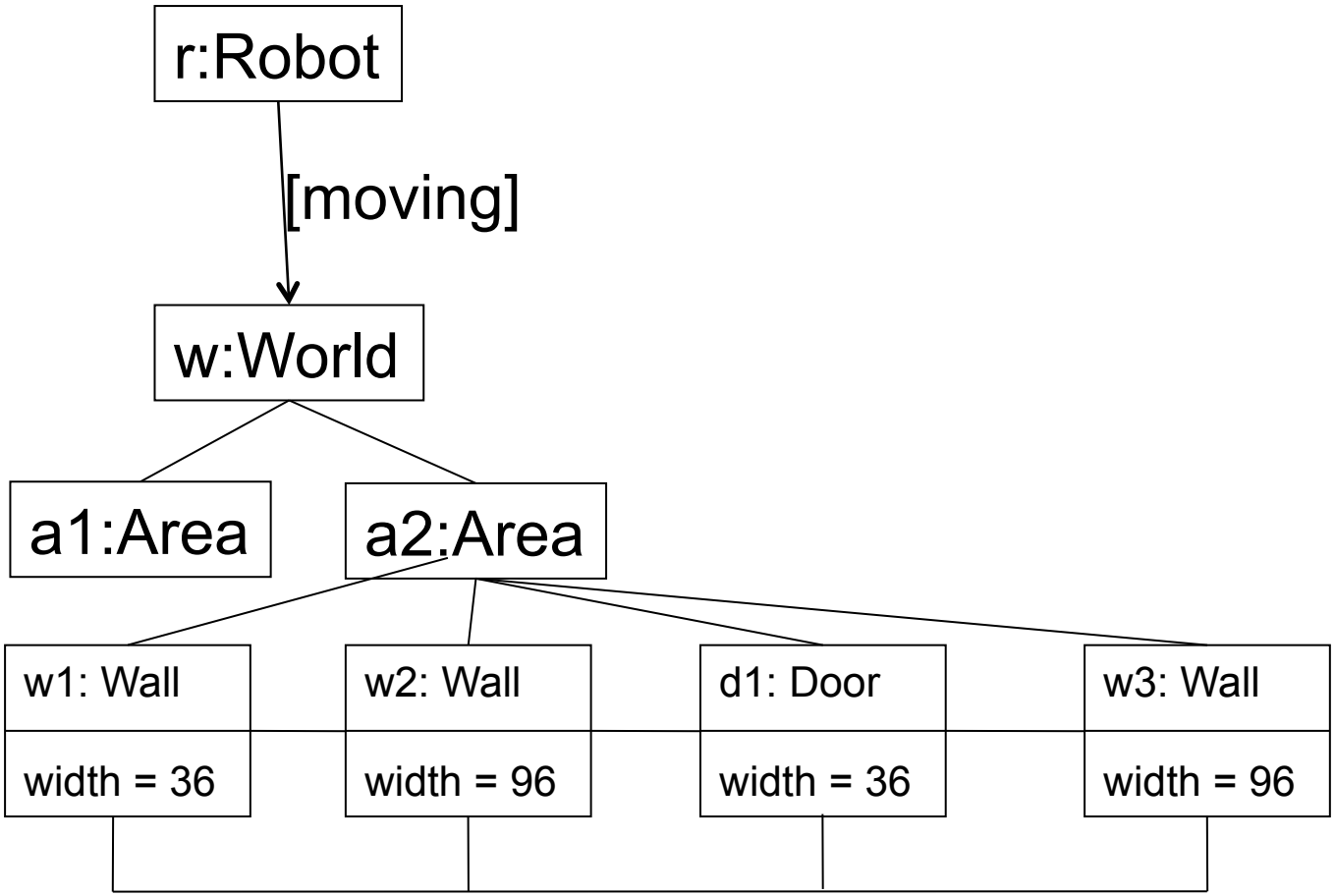
A taxonomy


Citizen
Foreigner
Person
Female
Male
Student
Employee
Professor
Full professor
Administrative
Associate professor
Technician
Research fellow
Undergraduate student
Masters student
PhD student

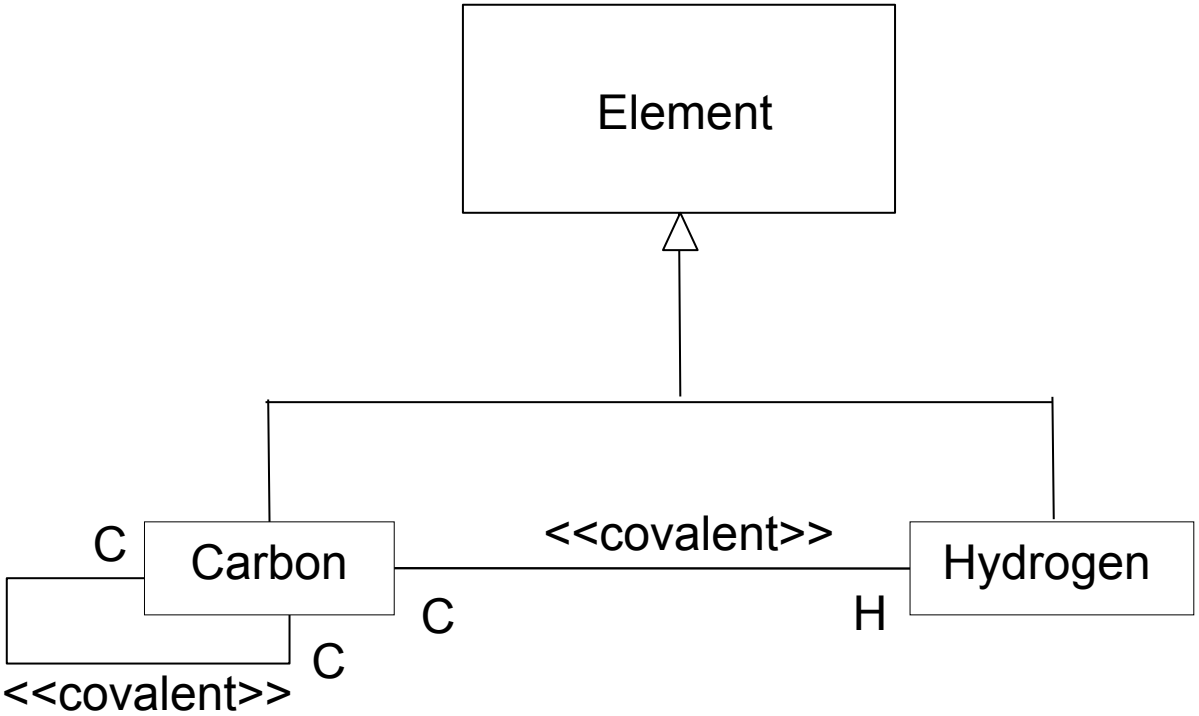
Object diagram

- 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

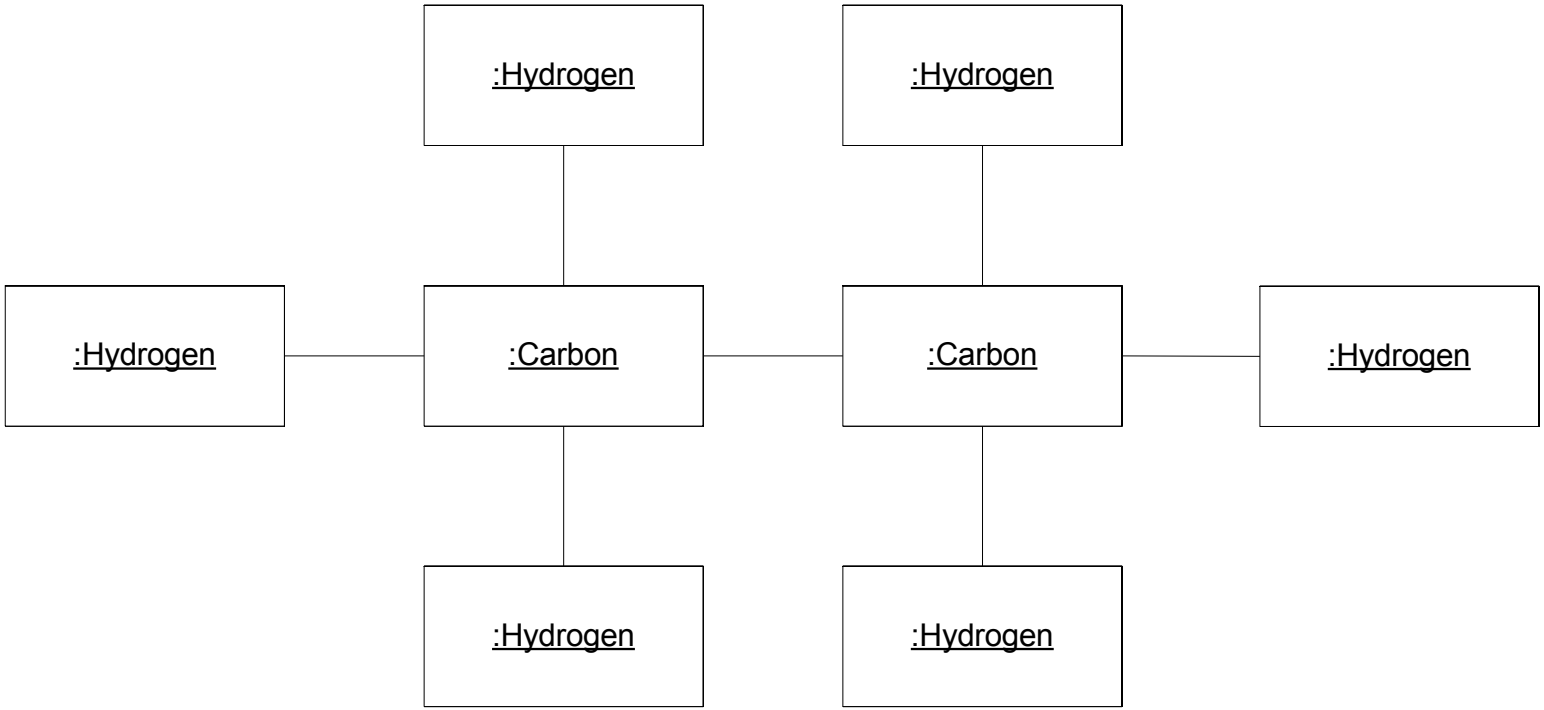
Example (object diagram)



Example: chemical elements (class diagram)



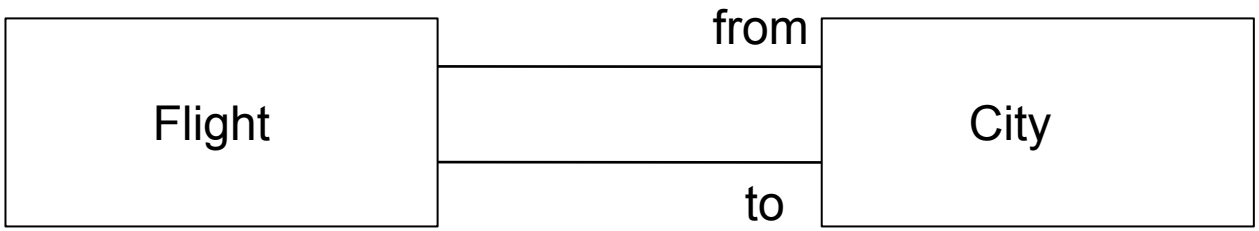
Example: molecule (object diagram)



Objects vs. Classes

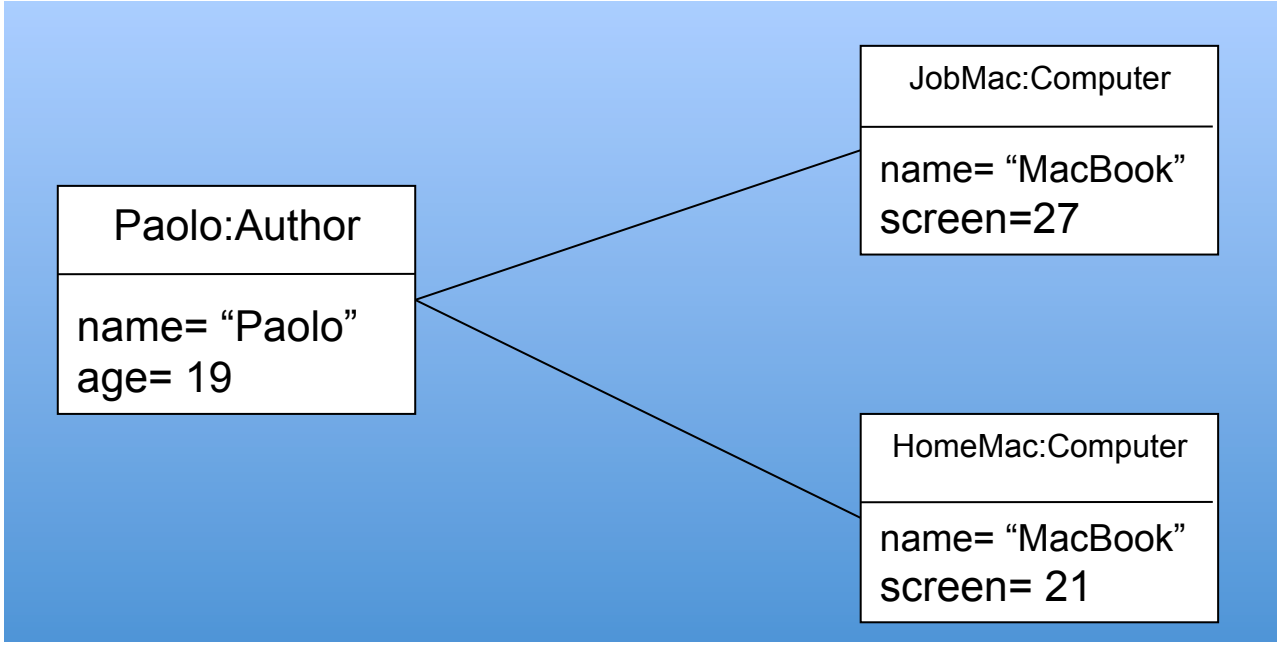
	Interpretation in the real world	Representation in the model
Object	An *object* is anything in the real world that can be distinctly identified	An *object* has an identity, a state, and a behavior
Class	A *class* is a set of objects with similar structure and behavior. These objects are called *instances*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

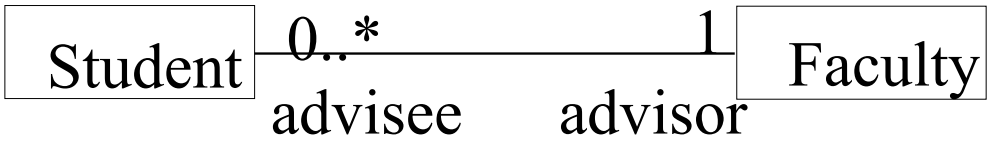
Class diagram



Object diagram that is an instance of the class diagram above

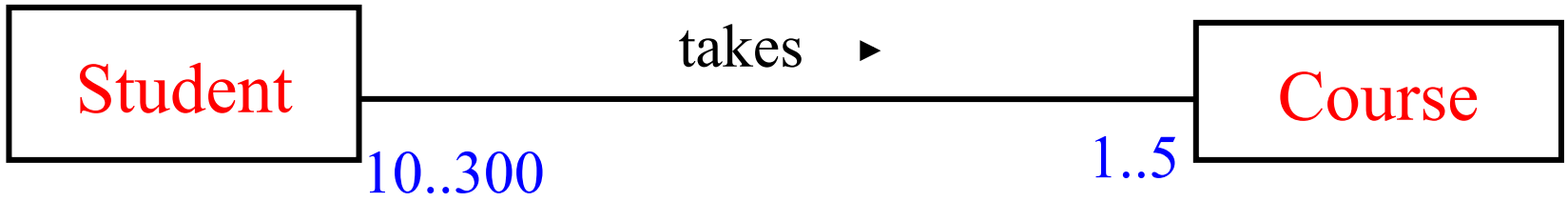
Roles and multiplicity

- 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, …

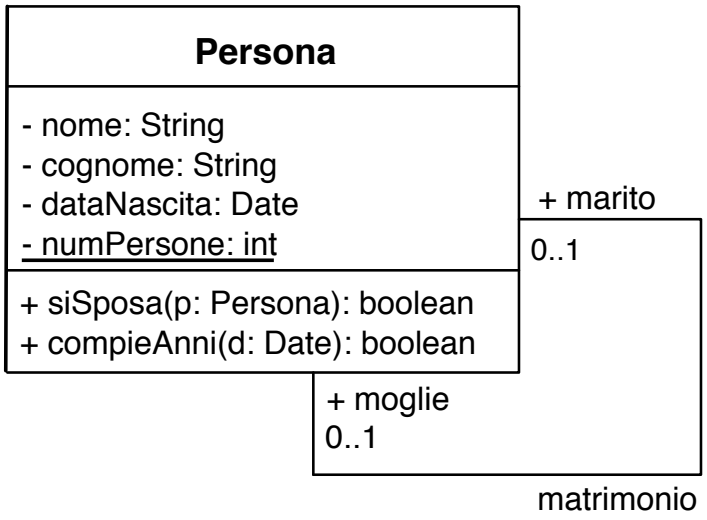


Association example

- 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



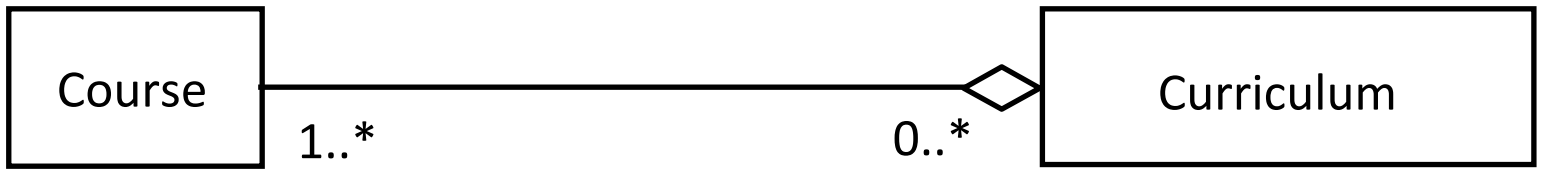

```
class Persona {
   private String nome;
   private String cognome;
   private Date dataNascita;
   private static int numPersone;
   public Persona marito;
   public Persona moglie;

   public boolean siSposa(Persona p) {
      …
   }

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

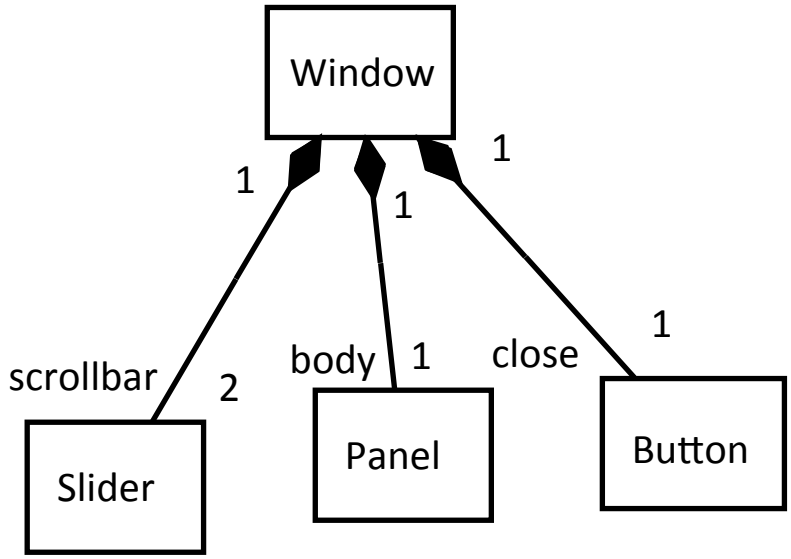
Aggregations

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



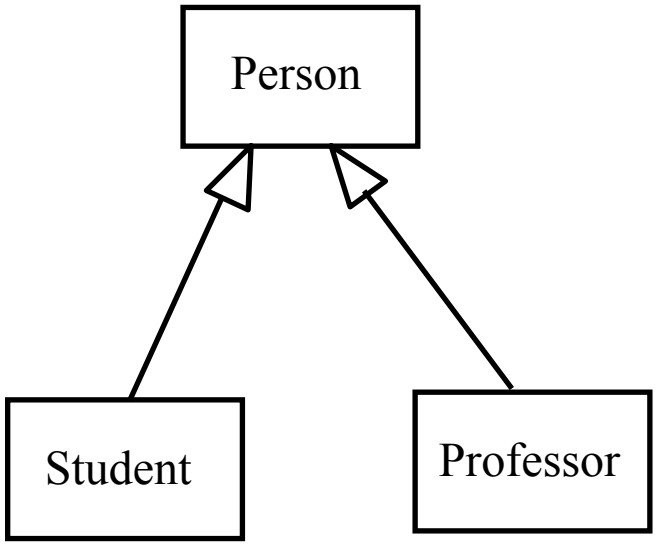
Composites

- 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)



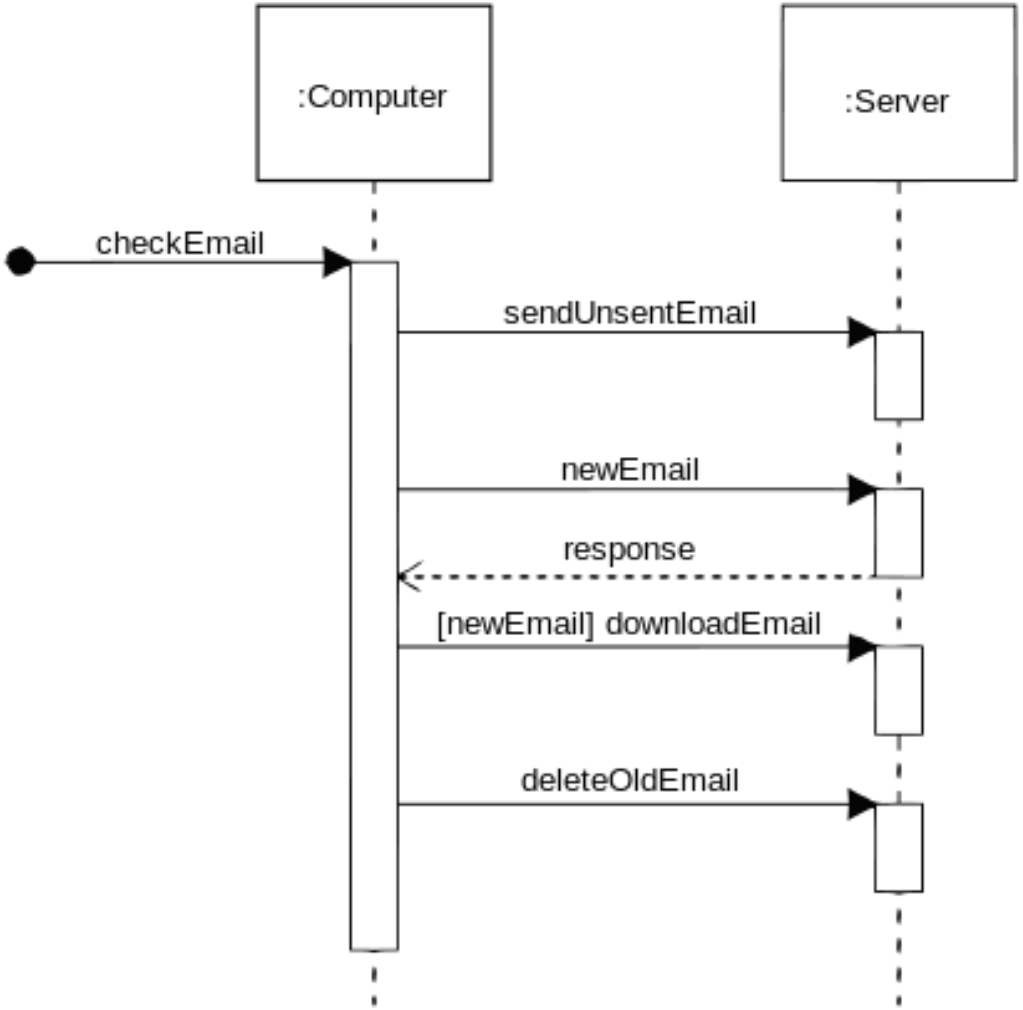
Interaction diagrams

- Models that describe how groups of objects interact (collaborate)
- Class and use case diagrams are useful at capturing the structural nature of a system design in a generalized way
- Interaction diagrams capture dynamic behavior applicable to timing or sequencing requirements

Sequence

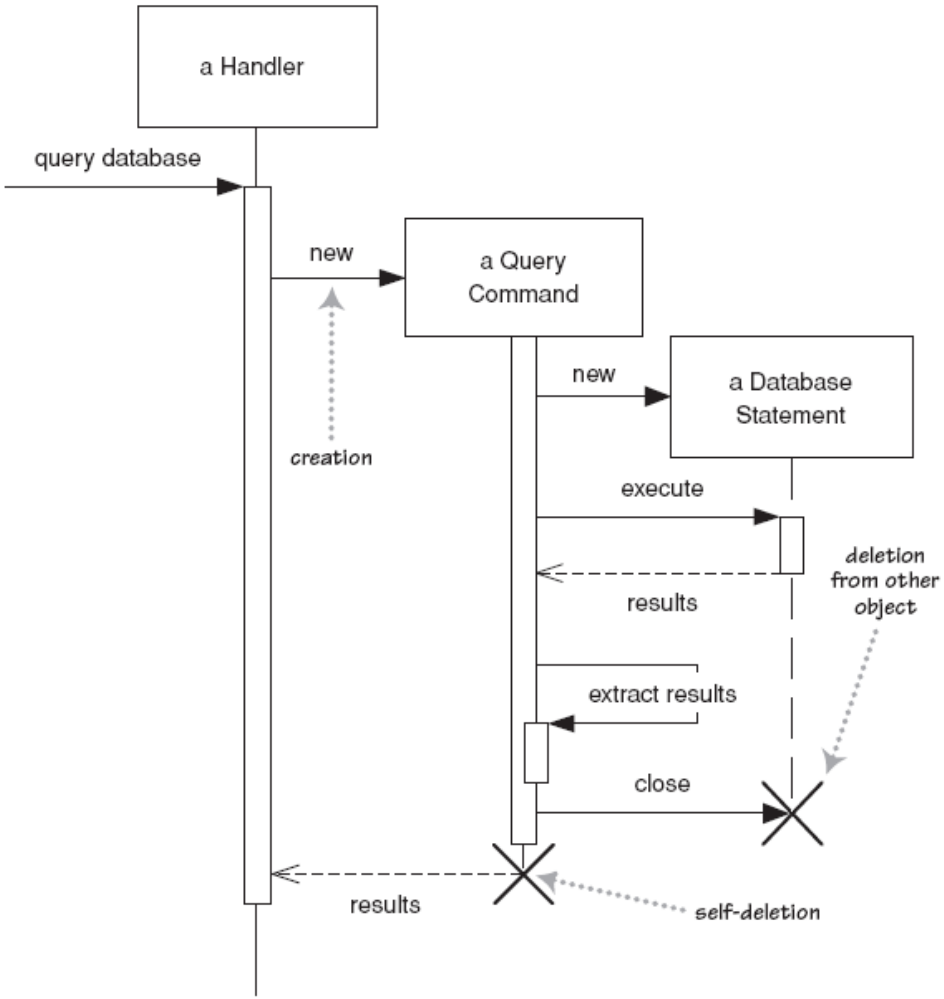
- Boxes at the top are participants (objects)
- Vertical lines are time lines
- Horizontal directed lines are messages
- Unless specifically noted, only sequence (location on the time line) is important, not exact times
- An activation rectangle in the lifeline indicates a focus of control (activation) during which an object is performing an action, either directly or through another object

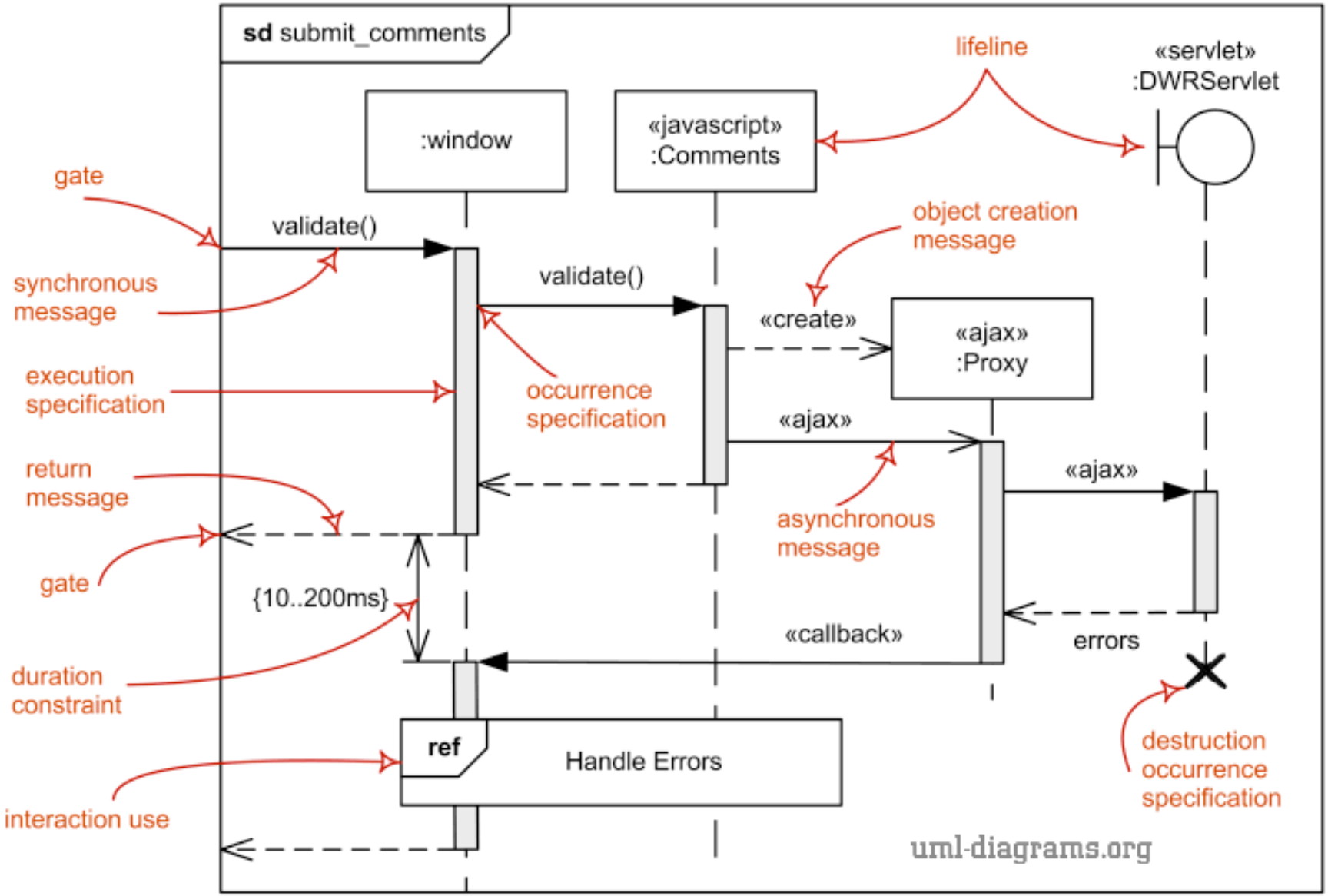
First example



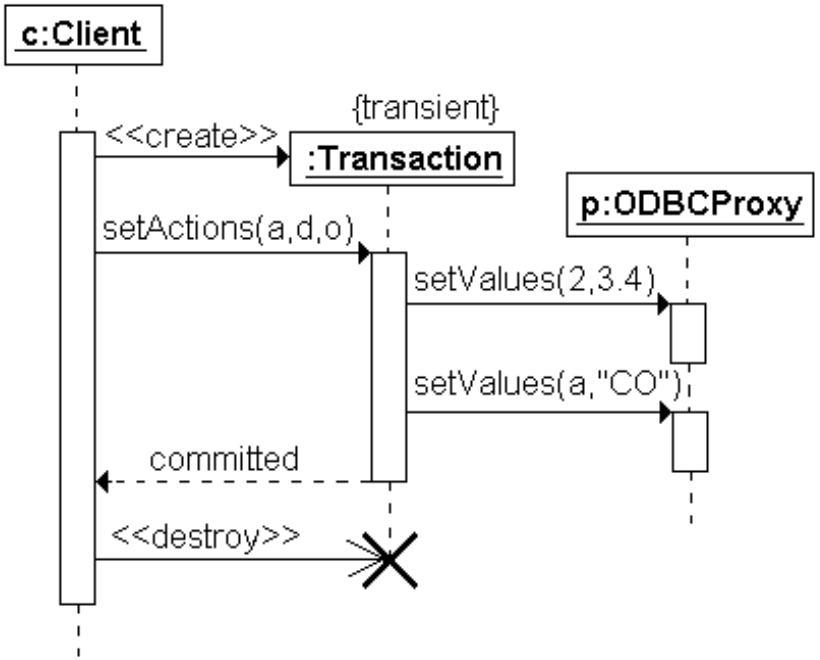
Lifetime of objects

- Creation: arrow with 'new' written above it
  - Notice that an object created after the start of the scenario appears lowe than the others
- Deletion: an X at bottom of object's lifeline
  - Java doesn't explicitly delete objects; they fall out of scope and are garbage-collected

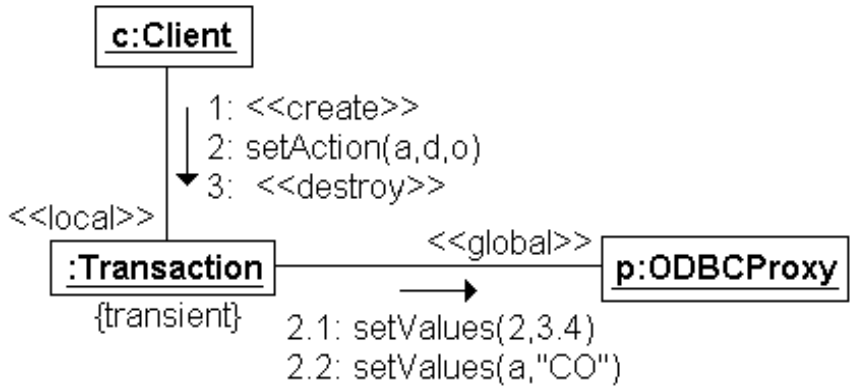



sd submit_comments
lifeline
«servlet»
:DWRServlet
:window
«javascript»
:Comments
gate
validate()
object creation
message
synchronous
message
validate()
«create»
«ajax»
:Proxy
execution
specification
occurrence
specification
«ajax»
return
message
«ajax»
asynchronous
message
gate
{10..200ms}
«callback»
errors
duration
constraint
ref
Handle Errors
destruction
occurrence
specification
interaction use
uml-diagrams.org

Sequence diagram



Corresponding collaboration diagram

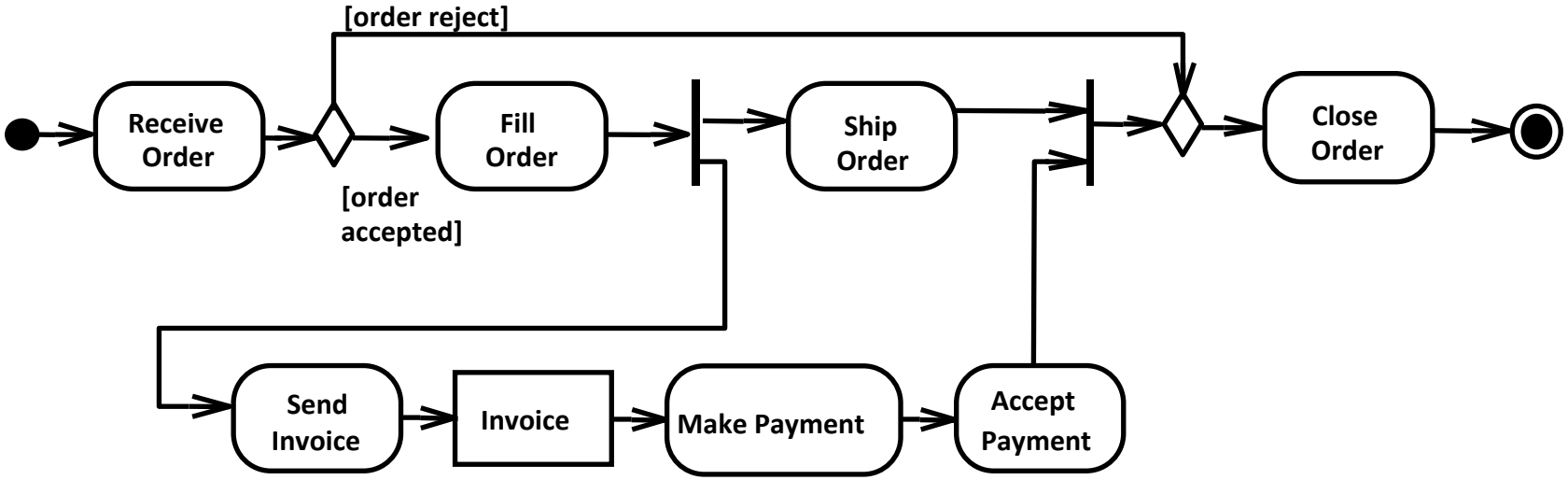


Which to use?

- Choose sequence diagram when only the sequence of events needs to be shown and collaboration among the objects are priority
- Choose a collaboration/communication diagram when the objects and their links facilitate understanding the interactions
- Collaboration diagrams have relatively fixed notation and numbering scheme

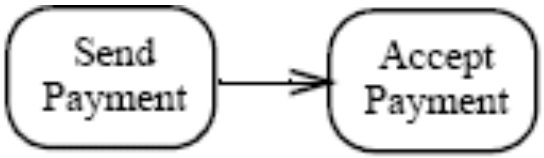
Activity diagrams

- Useful to specify software or hardware system behavior
- Based on data flow models – a graphical representation (with a Directed Graph) of how data move around an information system



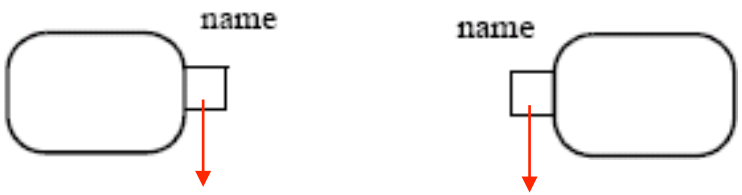
Actions

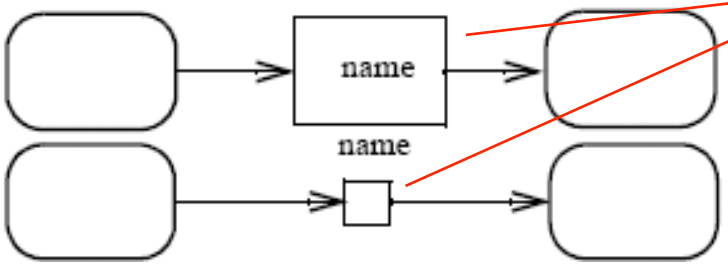
- The fundamental unit of executable functionality in an activity
- The execution of an action represents some transformations or processes in the modeled system



Pins

- Actions can have inputs and outputs, through the pins
- Hold inputs to actions until the action starts, and hold the outputs of actions before the values move downstream
- The name of a pin is not restricted: generally recalls the type of objects or data that flow through the pin

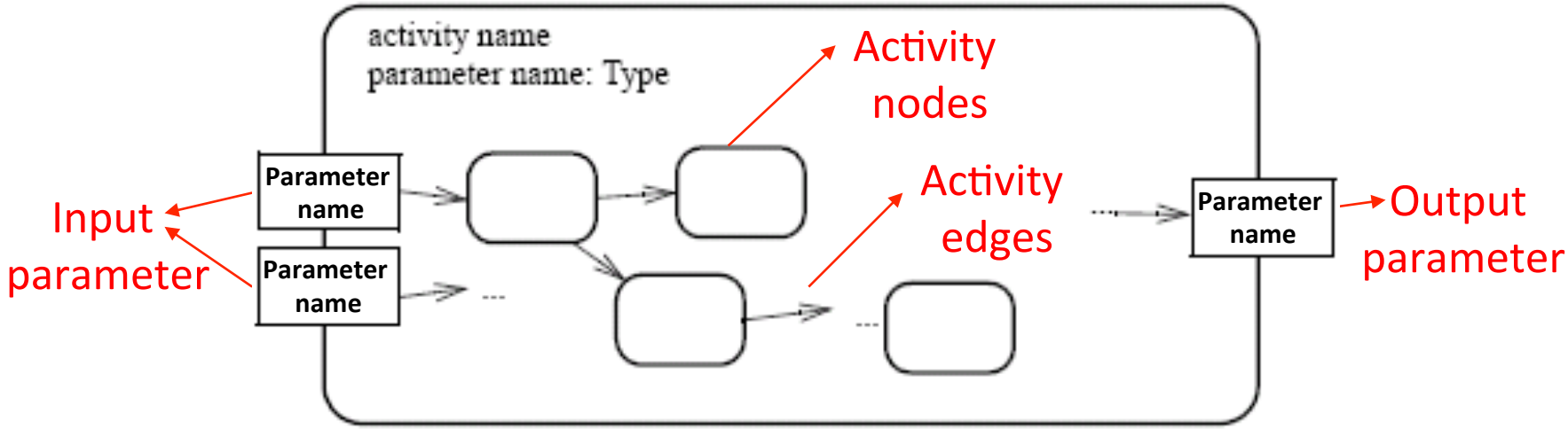




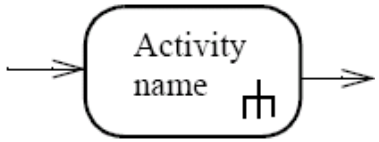
Standalone pin notations: the output pin and the input pin have the same name and same type

Activities

- An activity is the specification of parameterized behaviour as the coordinated sequencing of subordinate units whose individual elements are actions
- Uses parameters to receive and provide data to the invoker

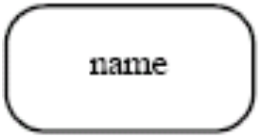


- An action can invoke an activity to describe its action more finely

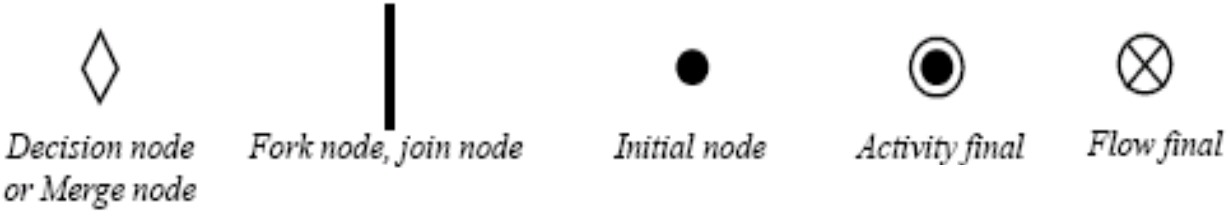


Activity nodes

- Action nodes: executable activity nodes; the execution of an action represents some transformations or processes in the modeled system (already seen)



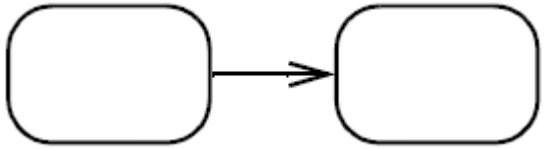
- Control nodes: coordinate flows in an activity diagram between other nodes



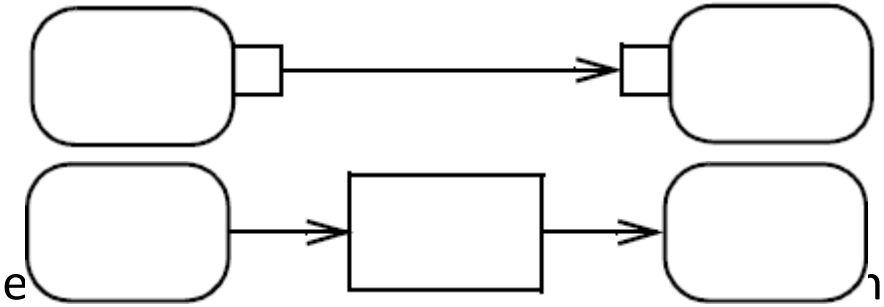
- Object nodes: indicate an instance of a particular object, may be available at a particular point in the activity (i.e Pins are object nodes)

Activity edges

- Control flow edge - is an edge which starts an activity node after the completion of the previous one by passing a control token



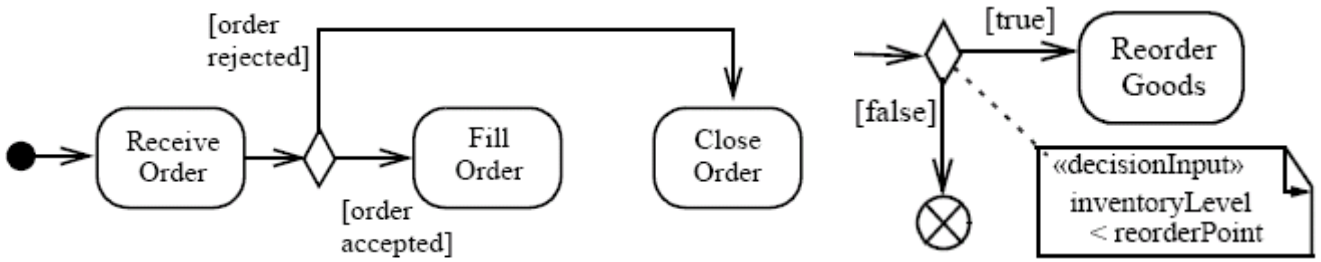
- Object flow edge - models the flow of values to or from object nodes by passing object or data tokens



- Weight can de ns that must traverse the edge at the same time

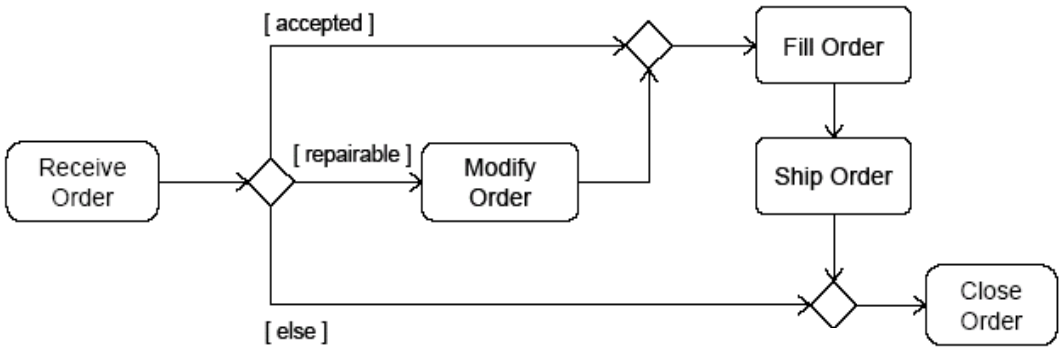
Decision nodes

- Route the flow to one of the outgoing edges (tokens are not duplicated)
- Guards are specified on the outgoing edges or with the stereotype «decisionInput»
- There is also the predefined guard [else], chosen only if the token is not accepted by all the other edges
- If all the guards fail, the token remains at the source object node until one of the guards accept it



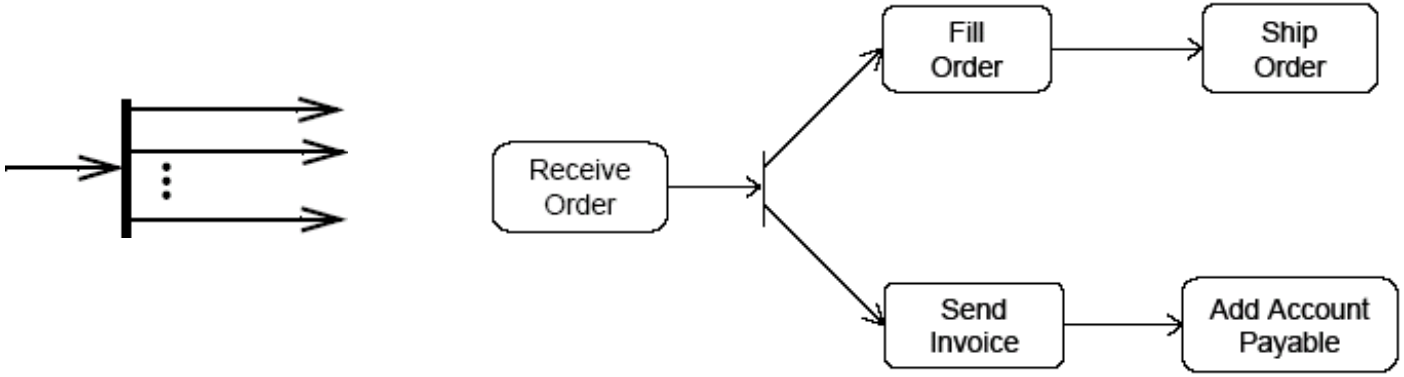
Merge nodes

- Bring together multiple alternate flows
- All controls and data arriving at a merge node are immediately passed to the outgoing edge
- There is no synchronization of flows or joining of tokens



Fork nodes

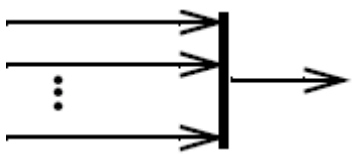
- Fork nodes split flows into multiple concurrent flows (tokens are duplicated)

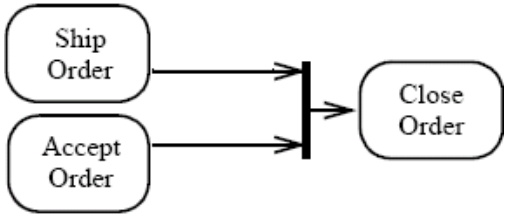


- UML 2.0 activity forks model unrestricted parallelism

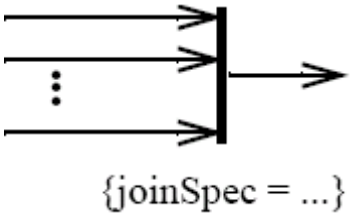
Join nodes

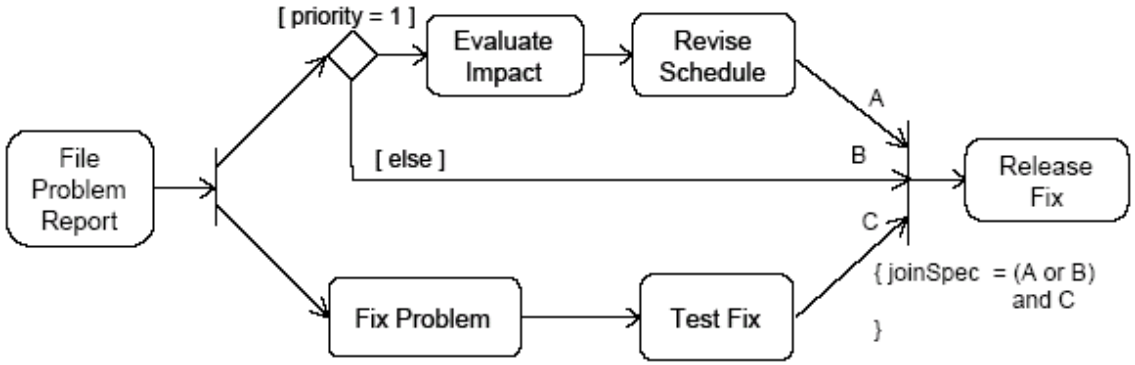
- Join nodes synchronize multiple flows





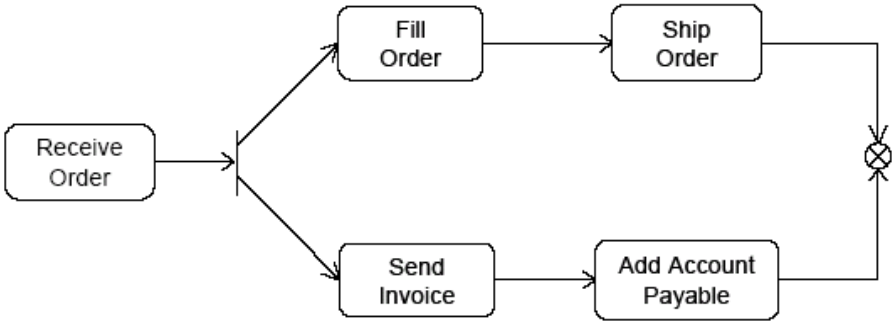
- Generally, controls or data must be available on every incoming edge in order to be passed to the outgoing edge, but user can specify different conditions under which a join accepts incoming controls and data using a join specification



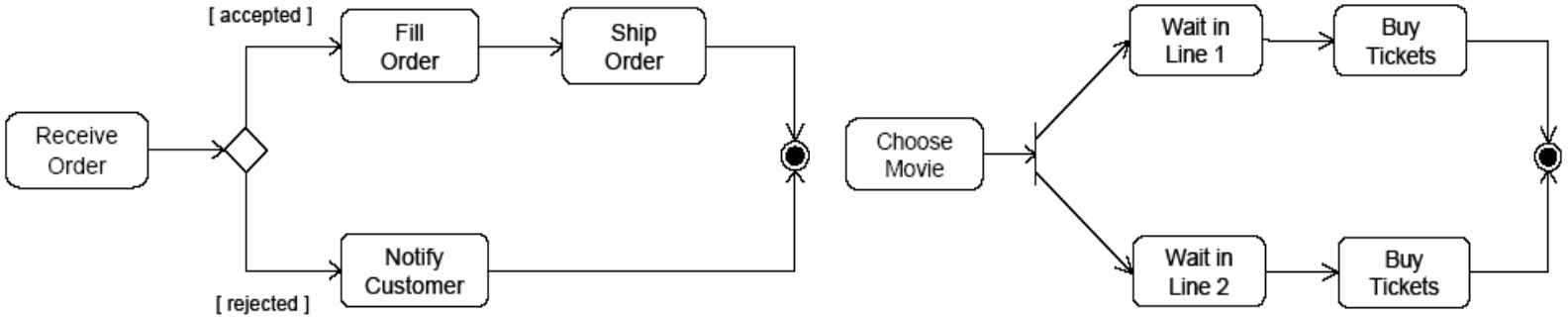


Final nodes

- Flow final:
  - destroys the tokens that arrive into it
  - the activity is terminated when all tokens in the graph are destroyed

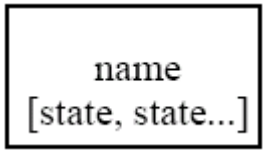


- Final node:
  - the activity is terminated when the first token arrives

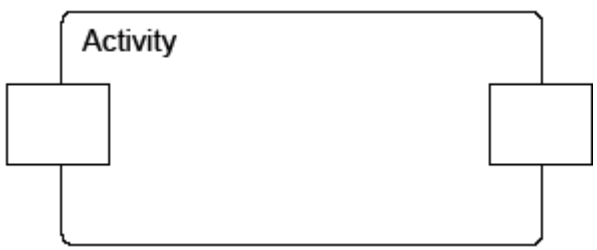


Object nodes

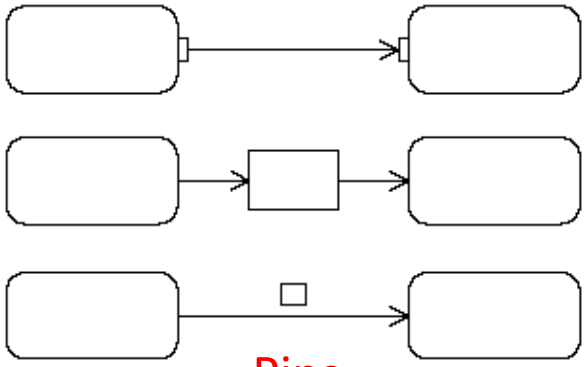
- Hold data temporarily while they wait to move through the graph
- Specify the type of values they can hold (if no type is specified, they can hold values of any type)
- Can also specify the state of the held objects



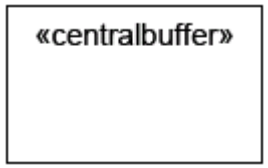
- There are four kinds of object nodes:



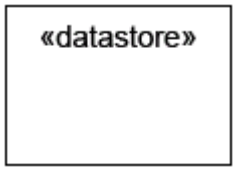
Activity Parameter Nodes



Pins
(three differents notations)



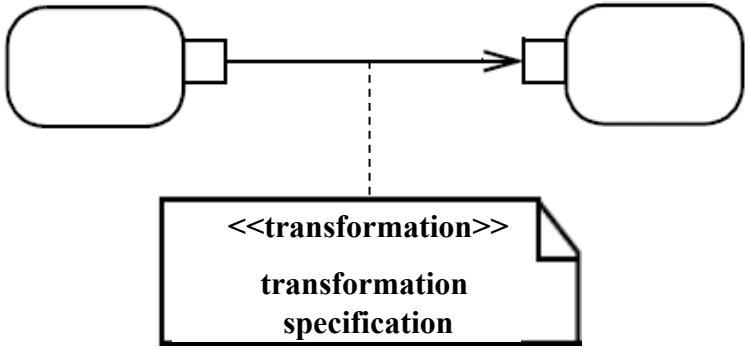
Central Buffer Nodes

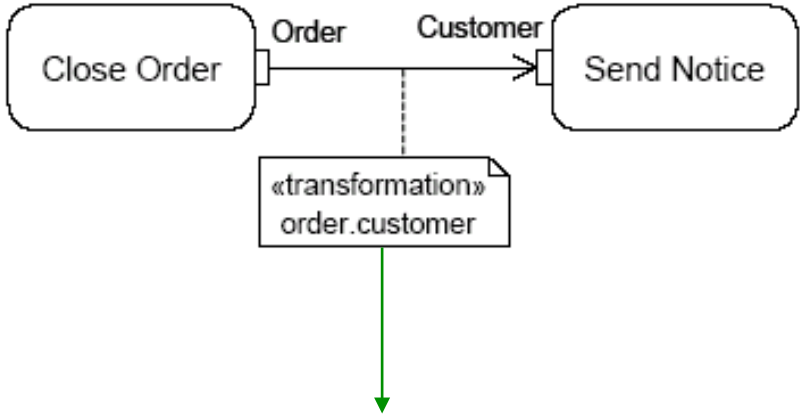


Data Store Nodes

Activity edges - transformation

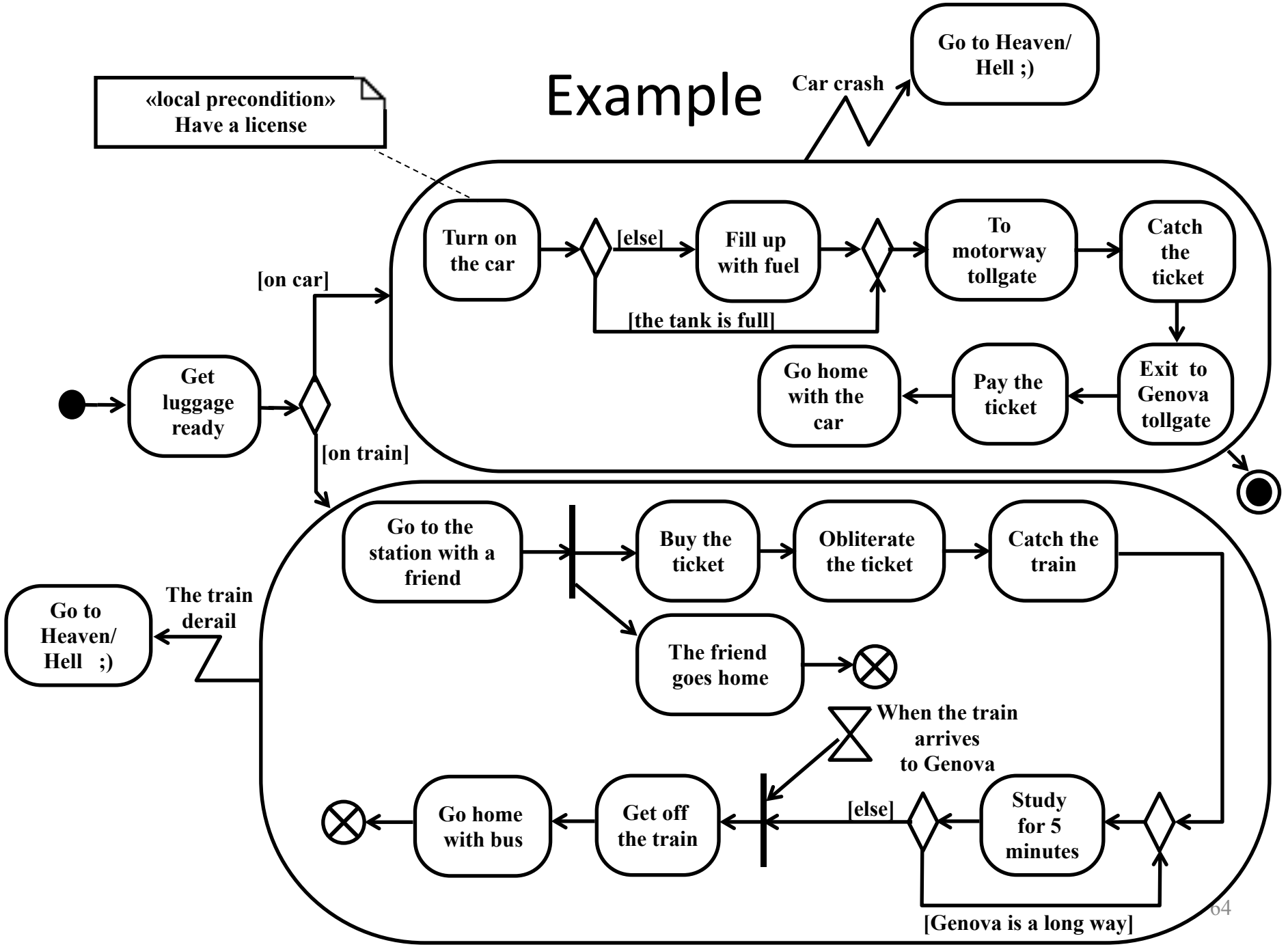
- It is possible to apply a transformation of tokens as they move across an object flow edge (each order is passed to the transformation behaviour and replaced with the result)





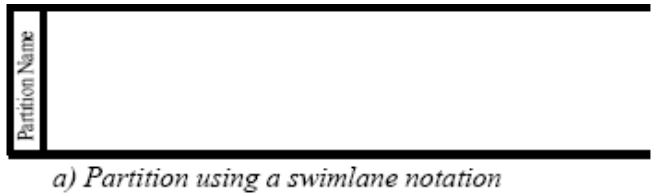
In this example, the transformation gets the value of the attribute Customer of the Order object

Example


«local precondition»
Have a license
Car crash
Go to Heaven/
Hell ;)
Turn on
the car
[else]
Fill up
with fuel
To
motorway
tollgate
Catch
the
ticket
[the tank is full]
[on car]
Get
luggage
ready
Go home
with the
car
Pay the
ticket
Exit to
Genova
tollgate
[on train]
Go to the
station with a
friend
Buy the
ticket
Obliterate
the ticket
Catch the
train
The train
derail
Go to
Heaven/
Hell ;)
The friend
goes home
When the train
arrives
to Genova
Go home
with bus
Get off
the train
[else]
Study
for 5
minutes
[Genova is a long way]

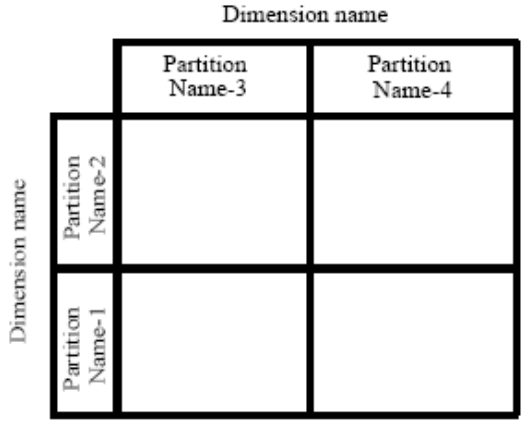
ActivityPartition

- Partitions divide the nodes and edges for identifying actions that have some characteristics in common
- They often correspond to organizational units in a business model
- Partitions can be hierarchical and multidimensional
- Additional notation is provided: placing the partition name in parenthesis above the activity name



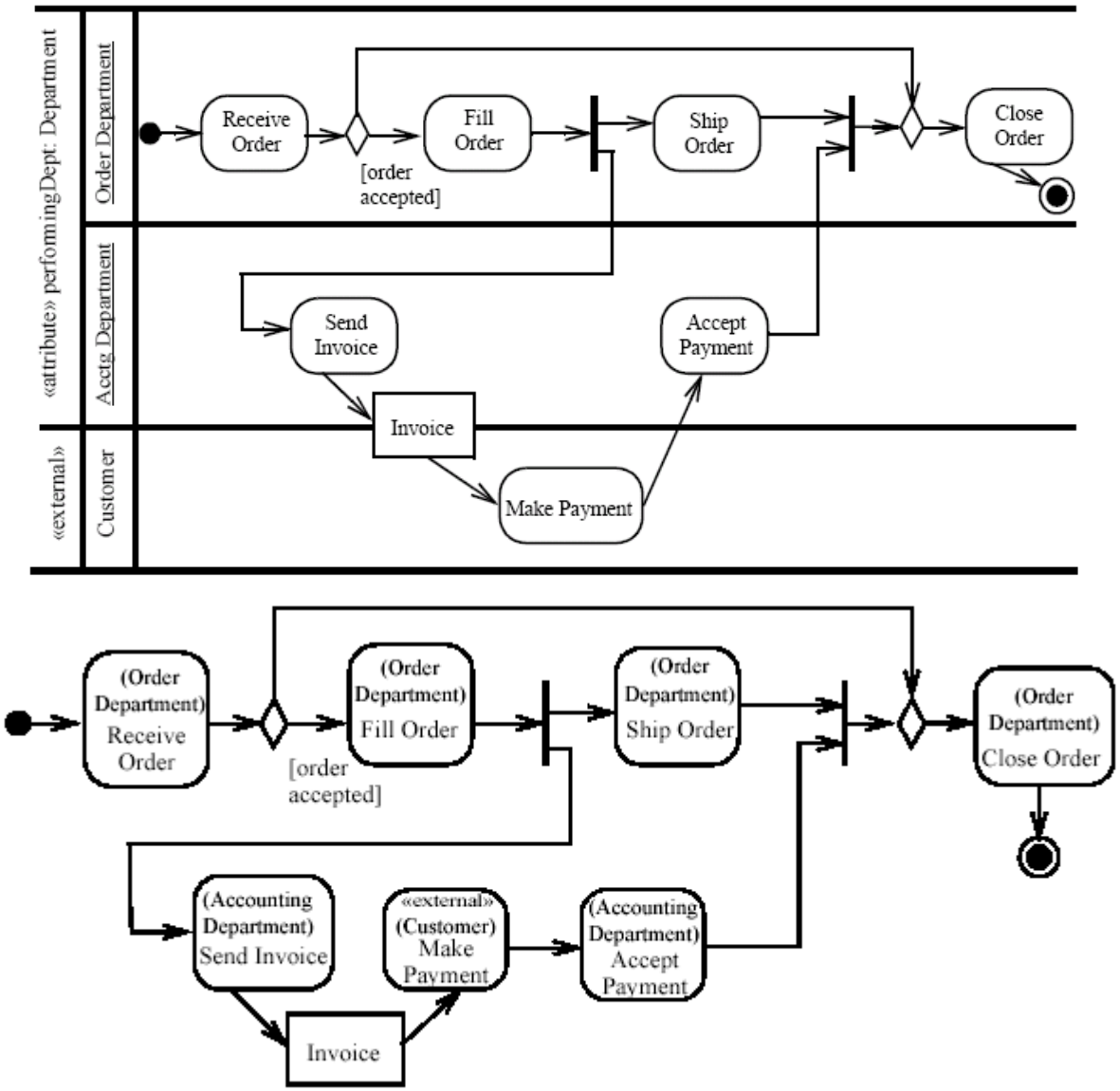
a) Partition using a swimlane notation

b) Partition using a hierarchical swimlane notation



c) Partition using a multidimensional hierarchical swimlane notation

Example


«attribute» performingDept: Department
Order Department
Acctg Department
«external»
Customer
Receive Order
[order accepted]
Fill Order
Ship Order
Close Order
Send Invoice
Accept Payment
Invoice
Make Payment
(Order Department) Receive Order
[order accepted]
(Order Department) Fill Order
(Order Department) Ship Order
(Order Department) Close Order
(Accounting Department) Send Invoice
«external» (Customer) Make Payment
(Accounting Department) Accept Payment
Invoice

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