UML Programming

Course

Introduction to UML

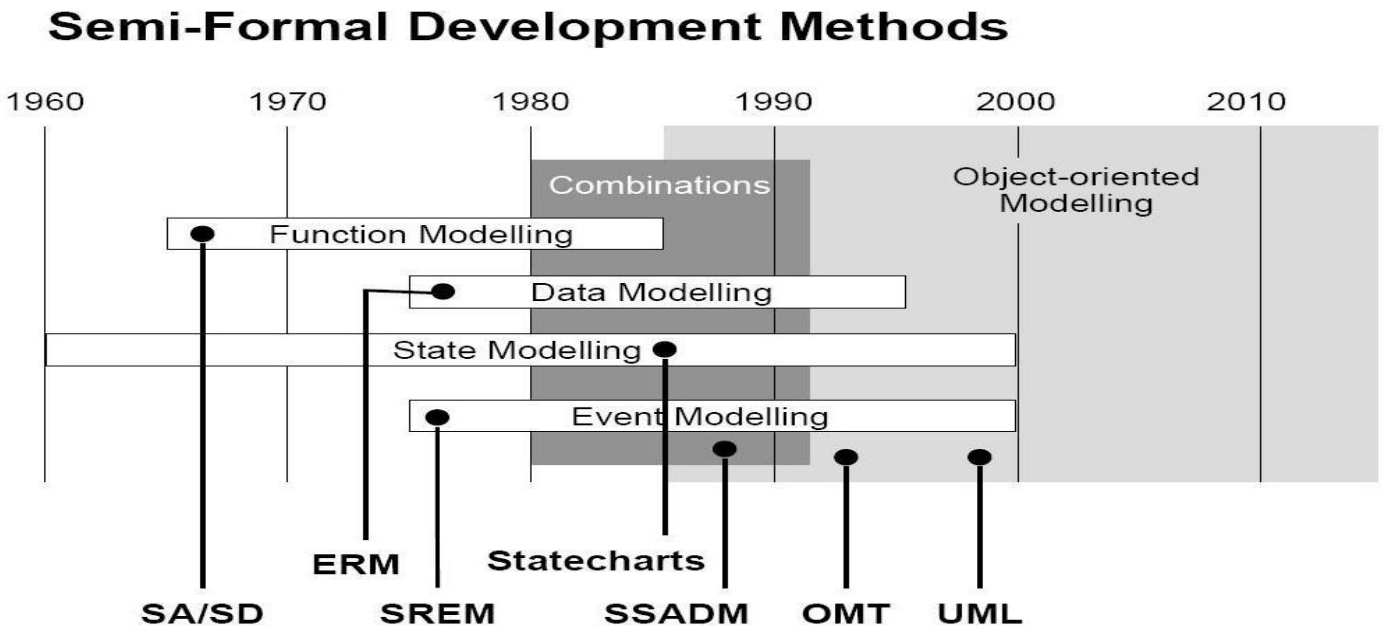
- UML is a large and complex language. Still, there are many requests to represent additional features explicitly that cannot be described conveniently with UML in its current version.
- Therefore, UML provides mechanisms, in particular *stereotypes* and *tagged values,* that allow extensions.
- These extensions may be defined and grouped in so-called *profiles.*

- UML is intended to cover a wide variety of application domains. It is not even restricted for describing software, but offers concepts to describe hardware as well as real-word requirements. Apparently these different domains require different modeling elements.
- Therefore, it became clear quite early on that UML was not going to become a single language fitting all purposes, despite all the efforts to unify its syntax, semantics, and usage.
- Instead, UML is a *family of languages* with one common core. This core is defined in the UML standard documents produced by the Object Management Group (OMG, 2001).

UML and Software Applications

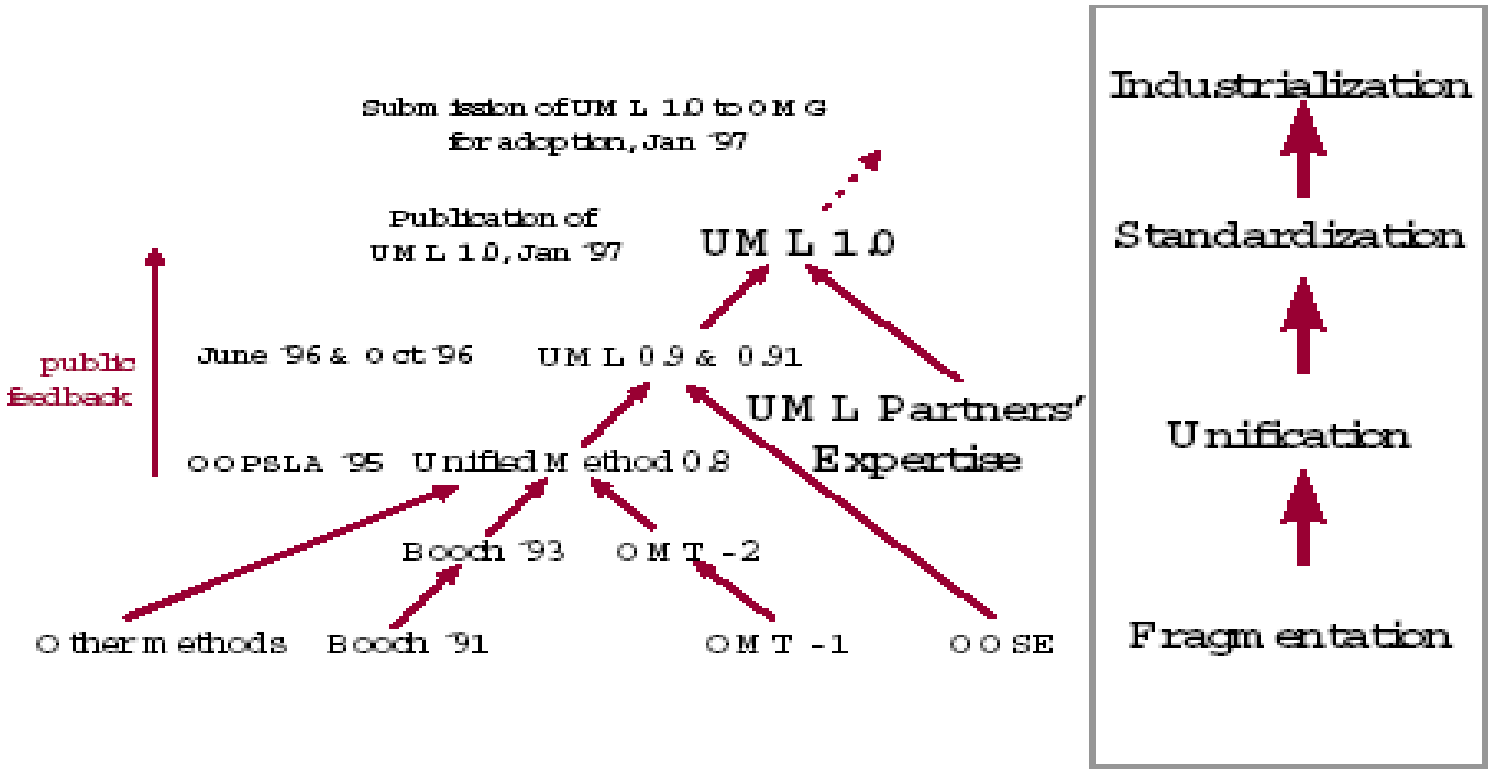
- Nowadays, software systems, like organisms, are far too complex to be described by a single blueprint. Instead, a number of descriptions are necessary so that each can focus on a different aspect of a system.
- Similar to the architecture of a building, ***the core structure of a software system is the foundation*** around which all other aspects are built. Whereas a building is a static artifact, software systems mainly derive their value from their ***behavior***.
- To further complicate the situation, the structure of a software system is not a concrete, touchable thing, which makes its behavior even more difficult to describe and grasp.

UML Evolution



Basics of UML

- Grady Booch diagrams based on **OOD** (Object Oriented Design)
- **OMT** (Object Modeling Technique) oriented for ***analysis*** – James Rumbaugh
- **OOSE** (Object Oriented Software Engineering), developed by Ivar Jacobson used to understand the behavior of the system using ***use cases diagrams***
- UML is a ***standard notation*** and does not involve a using process of these diagrams developing an application


Submission of UML 1.0 to OMG
for adoption, Jan '97
Publication of
UML 1.0, Jan '97
UML 1.0
public
feedback
June '96 & Oct '96
UML 0.9 & 0.91
UML Partners'
Expertise
OOPSLA '95 Unified Method 0.8
Booch '93
OMT-2
Other methods
Booch '91
OMT-1
OOSE
Industrialization
Standardization
Unification
Fragmentation

Canonical Diagram Names

The canonical diagram names used in UML are:

-use case diagram

-class diagram, object diagram

-behavior diagrams:

-state chart diagram

-activity diagram

-interaction diagrams:

-sequence diagram

-collaboration diagram

-implementation diagrams:

-component diagram

-deployment diagram

- Although other names are sometimes given to these diagrams.

UML 2.x introduced an other **classification based on**:

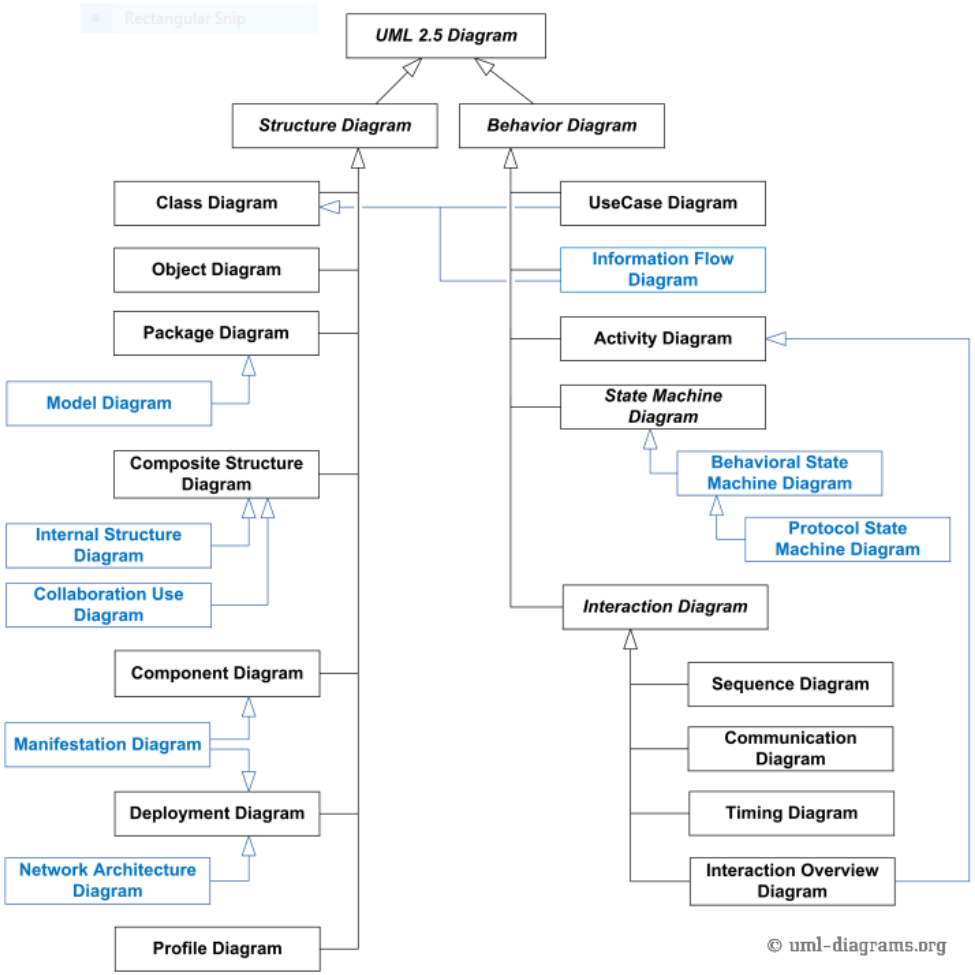
-Structural Diagrams

-Behavioral Diagrams, and a lot of sub-diagrams.

Last version used in 2020-21 is UML 2.5.x:

https://www.uml-diagrams.org/uml-25-diagrams.html

UML 2.5 Overview



UML 2.5 Diagrams Overview.
Note, items in blue are not part of official taxonomy of UML 2.5 diagrams.

Diagrams Classification

- **Static diagrams** – describes the structure and the responsibilities
  - Class, object and use-case diagrams
- **Dynamic diagrams** – describes the behavior and the interactions
  - Sequence, collaboration, state and activity diagrams
- **Architectural diagrams** – describes the executable components and the physical location
  - Implementation (components and deployment diagrams)

UML notations

- The various notations provided by UML are each designed to focus on individual aspects of a software system.
- The ***static structure*** of a piece of software is described by the most important part of UML, the ***class diagram****.* A class diagram gives an *overview of the whole system* – it captures the structure and relationships between the components.
- ***Object diagrams*** are closely related to class diagrams. They are a useful and an important technique to describe *snapshots* of a system.

- UML provides several notations to describe different aspects of ***behavior***. ***Sequence diagrams*** are among the most prominent, for they capture interactions between objects. They describe how several *objects interact in time* with each other through sending and receiving messages. They normally *do not consider the state* of participating objects.

- ***Collaboration diagrams*** provide almost the same information, but in a different way, using *multilevel numbers*.
- Collaboration diagrams often turn out to be *harder to read* than sequence diagrams – it is more difficult to understand the overall message flow.
- Additionally, collaboration diagrams are more difficult to manipulate, e.g. through adding new participating objects, filtering irrelevant messages, or combining two diagrams.

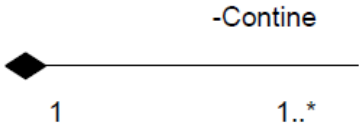
- ***Statechart diagrams*** describe the *life cycle of single objects* using *state machines*.
- State charts combine information about an object's state, including information about its behavior.
- ***Deployment diagrams*** show how the software is deployed considering computing elements.

Common UML Mechanism

- **Stereotypes** – allows specialization
- **Labels** – allows extension of attributes
- **Notes** - comments
- **Constraints** – extends the semantic of the metamodel
- **Dependence relation** – a unidirectional relation
- **Dualities** of (type-instance) and (type-class)

Notations

Simple aggregation

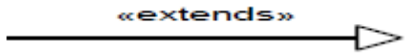


Composition

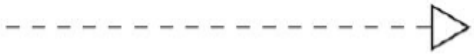
Transition

Dependencies

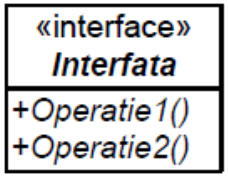
Derivation



Extends



Implements



Interface representation

«Stereotip» Nume_MetaClasa::**Nume_Clasa**
+Atribut public : Tip_Data = Valoare_Initiala #Atribut protejat : Tip_Data = Valoare_Initiala -Atribut privat : Tip_Data = Valoare_Initiala
+Operatie publica(in Parametru_1 : Tip_Data = Valoare_implicita, out Parametru_N : Tip_Data) : Tip_Data #Operatie protejata(in Parametru_1 : Tip_Data, inout Parametru_i : Tip_Data, out Parametru_N : Tip_Data) : Tip_Data -Operatie privata() : Tip_Data

Class representation

Class diagrams

- Class diagrams are the most important notation for object modeling. It express the static structure of the system.
- A class diagram serves several important *purposes*:

-a class diagram gives an *overview* of the *structure type* of a system;

-*associations* and *generalizations* shown in a class diagram describe *structural relationships* among classes;

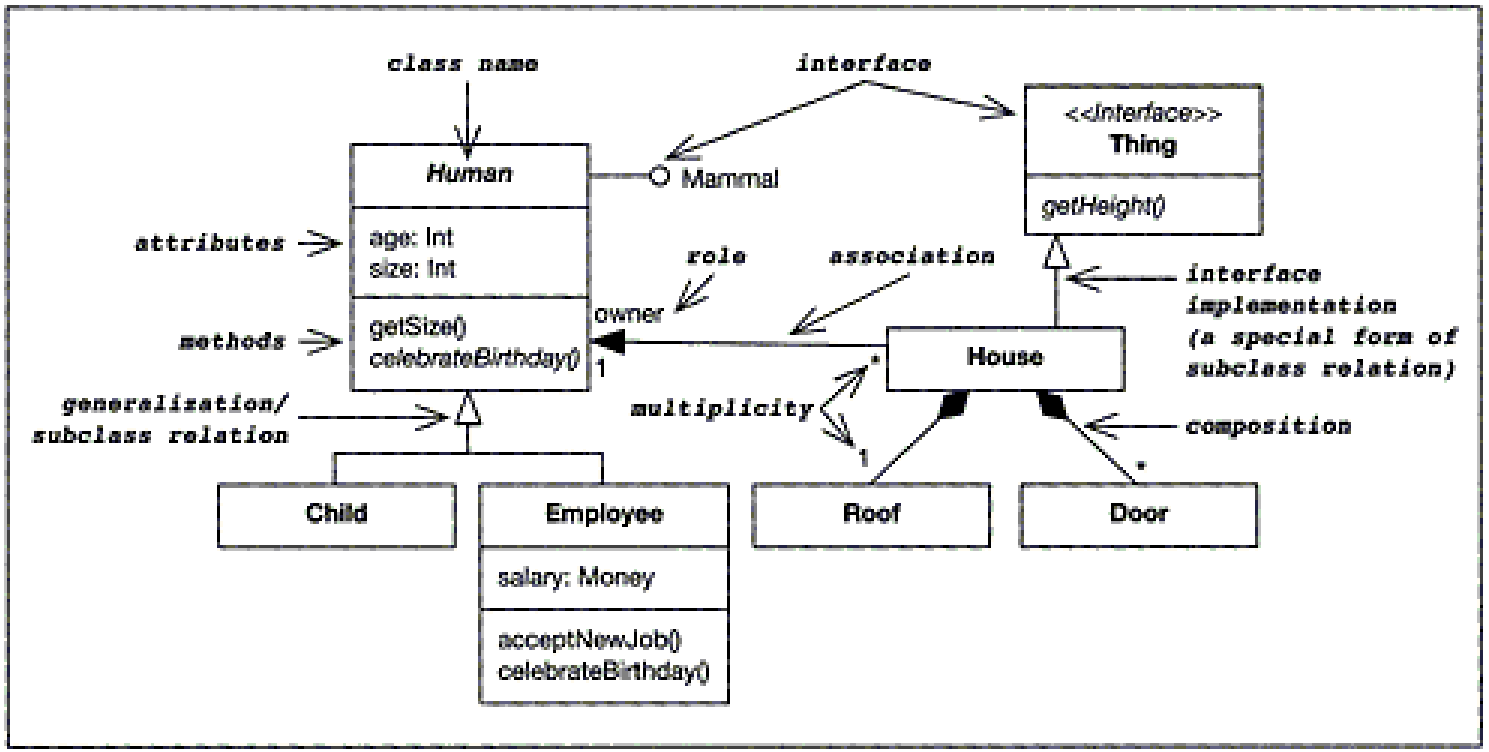
-class diagrams contribute to the *description* of the system architecture, since components are implemented by (sets of) classes;

-several other kinds of UML diagrams, such as sequence and object diagrams, are based on class diagrams;

-a class diagram describes the set of *possible states* of the overall system and therefore acts as a *constraint* on the possible states of the system;

-attaching *notes* that contain constraints, or implementation bodies to elements of class diagrams allows us to *describe behavior* to some extent – *notes* can also be used to describe additional information, such as the last date of change, the review status, etc.

Class diagram- Figure 1



- The Figure 1 shows a sample UML class diagram that contains all the elements provided by the UML standard.
- The diagram consists of six *classes* (namely Human, Child, Employee, House, Roof, and Door) and two *interfaces* (Mammal and Thing).
- The *interfaces* are presented in different shapes. Thing is drawn as a class with the so-called stereotype «interface» on it, whereas Mammal is drawn as a small circle without any indication of an interface method.

- Class House *implements* the Thing interface. Classes Child and Employee are *subclasses* of Human – a *generalization relation* is shown in the form of an unfilled triangular arrowhead. Generalization is a relationship between a subclass and a superclass.
- Compared to Java, where a subclass may have only one direct superclass, the UML standard allows classes to inherit from as many superclasses as desired. In Java, however, each class may implement an arbitrary number of interfaces. This is sufficient to allow general type hierarchies, but avoids technical involvement when inheriting the same attribute or method several times along different paths.

- Furthermore, the *implementation relationship* between classes and interfaces and the *generalization relation* between classes are quite similar. Both are therefore visualized in the same way through triangles with a solid line.(*)
- An ***abstract class*** is drawn with a class name in italics, such as class *Human.*
- *(*) The UML standard 1.4 proposes the use of a dashed line for interface implementation, but that distinction from inheritance is not necessary because it is clear already from the context of the line.*

- The unnamed *association* between the classes Human and House represents ownership. It has one *role* and two *multiplicity identifiers* attached to it. The role name 'owner' can be used to navigate from an object of class House to the corresponding Human object (its owner).
- A *solid arrow* shows the navigability of the association in this direction.
- *Multiplicity 1* describes that there is exactly one such object. Note that associations may have quite a number of different realizations, but in many cases it is a good choice to use an attribute as the implementation choice.
- *Multiplicity* * in the other direction indicates that many objects of class House may belong to one Human object. However, as there is no arrow associated with it, it is unclear whether navigation in the direction described is possible.

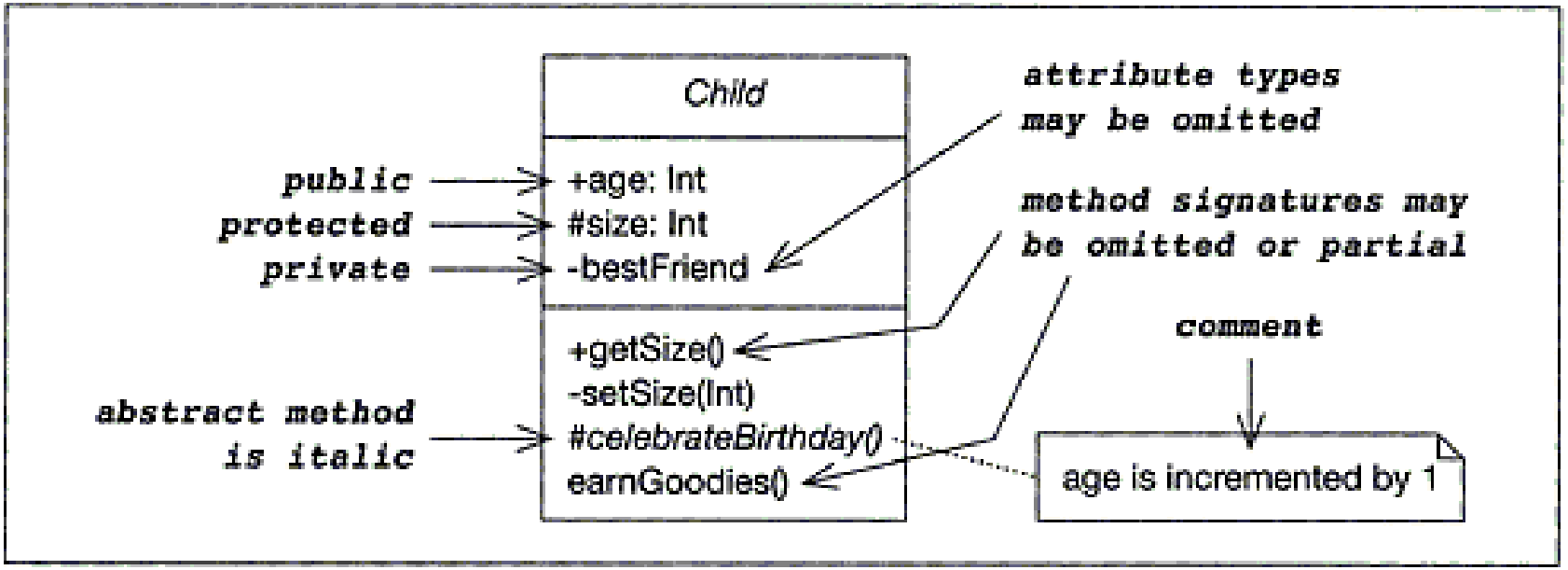
- Classes and interfaces do have *signatures*.
- An interface signature consists of methods, while a class signature consists of methods and attributes.
- Each *method* is described through its name, argument types, and return type.
- An *attribute* signature consists of name and type.
- The class diagram from the Figure 1 omits method and attribute signature lists for most classes; only the classes Human and Employee, and the interface Thing carry a signature.
- *Abstract methods* are denoted in italics, as exemplified by *celebrateBirthday()* in class Human.

- House objects are composed of objects of classes Roof and Door. Each House object consists of exactly one Roof object (multiplicity 1) and an unspecified number of Door objects (multiplicity *).
- There are a number of different flavors of **object composition.**
- **A *weak form*, namely *aggregation***, is represented through an *unfilled diamond*. Aggregation has a number of different interpretations, with some of them almost identical to normal associations. Thus, one recommendation is not to use this weak form at all.
- **The *strong form*, namely *composition*** (represented by a *black diamond*), is characterized by coincident lifetimes and unshared composed objects.

- This means that the *composed elements* do not live longer than the composition itself, and that each composed element belongs to, at most, one composition.
- In this example, it means that whenever a house is destroyed, the corresponding roof and doors must also be destroyed, and that each door and roof belong only to one house.

- *Attributes and methods* are given in two compartments within a class rectangle.
- Figure 2 shows how to represent these elements together with their visibility.
- Visibility marker '+' denotes *public* attributes or methods. It corresponds to Java's public language construct, allowing access from any other object.
- *Private* access is marked by '-'. It corresponds to Java's 'private' language construct, allowing access only within the scope of that object.
- Finally, visibility marker '#' expresses that a class and its subclasses may access this element. In C++ this is called '*protected*', whereas in Java the same concept was originally called *'private protected*'.

Class elements –Figure 2



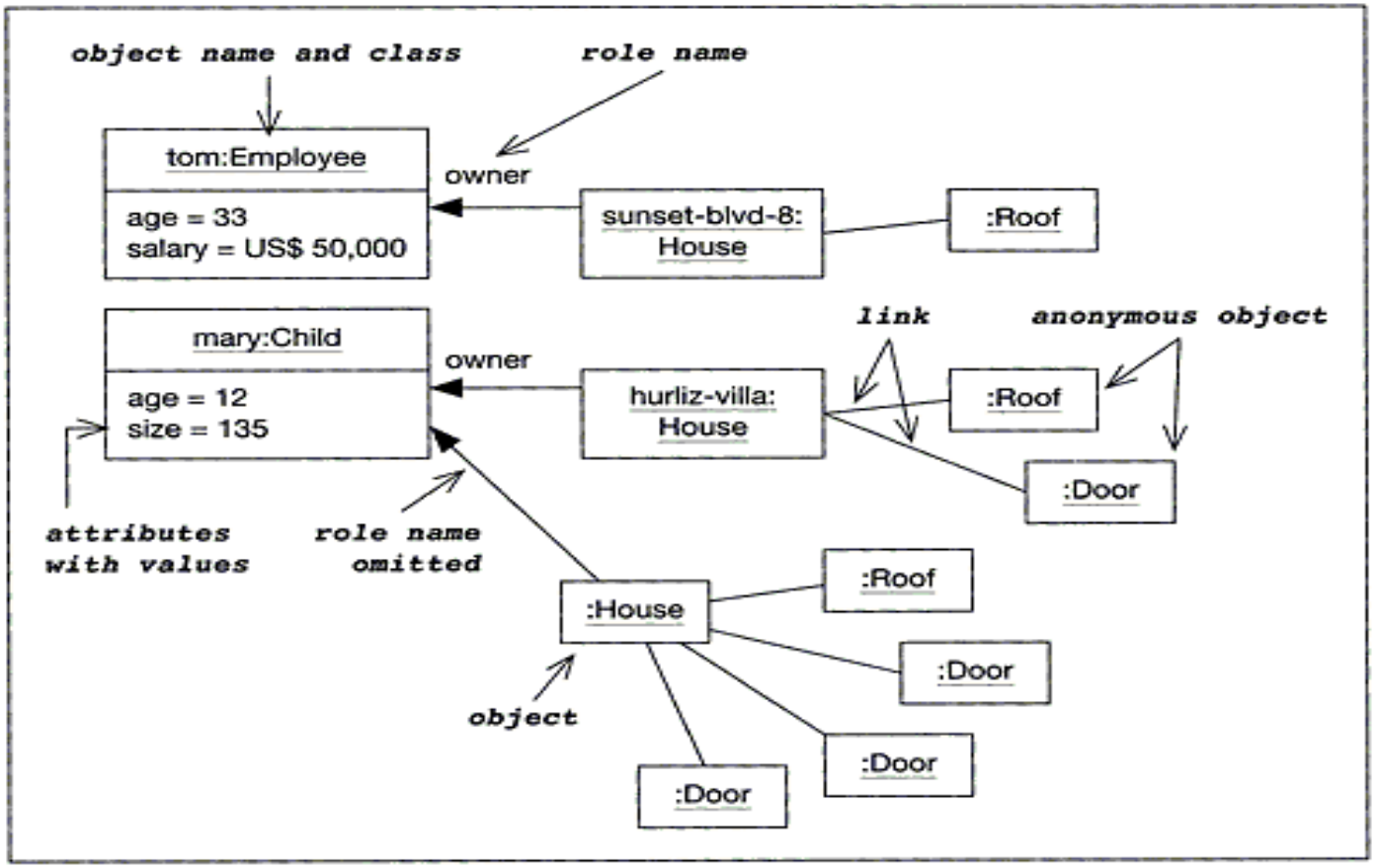
- Figure 2 also contains a *comment* in a *rectangle with a dog's ear*.
- Comments can contain informal explanations (such as the one above), pseudo-code, such as age+=1, or properties specified in the Object Constraint Language (OCL), such as age=age@pre+1.
- OCL is part of the UML standard. It is a textual supplement that allows formal specification of conditions.

Object diagrams

- UML *object diagrams* are closely related to class diagrams. Because both deal with the structure and the data of the system, object diagrams are often not treated as a stand-alone notation but as a variant of class diagrams.
- Object diagrams are a useful and important technique for *describing snapshots of a system* – indeed, important enough to regard and introduce this kind of diagram as a notation on its own. Object diagrams are useful for conveying certain kinds of information that class diagrams cannot.
- Object diagrams contain objects and, thus, *describe a system on an instance level that might occur at runtime*. A set of object diagrams can describe variants of *object structures* to the very same class diagram. A single object diagram thus describes an *actual* object structure, whereas a single class diagram describes a *set of potential* object structures.

- The object diagram in Figure 3 shows one sample object structure. It is consistent with the class diagram in Figure 1.
- Objects, like classes, are drawn as rectangles. However, for a clear distinction from classes, the names of objects are *underlined*.
- A name consists of two parts – an object name, and its class name. For example, mary:Child is an object of class Child with the name 'mary'. Note that the object name must neither be confused with a variable referring to an object nor with the object identity; although in the implementation, these object names are often mapped to variable names.
- The object name in an object diagram is used to identify the object within the diagram. In *a running system, the very same diagram structure may occur several times*, thus allowing several objects to take the role of 'mary' – 'mary' can be regarded as a *prototypical* object describing a *role.*

Object diagram – Figure 3



- The *object name is optional* and may be used to refer to that object in the explaining text. Object names in the diagram are primarily used to distinguish instances of the same class, since one diagram may contain an arbitrary number of objects of the same class. The class name can also be left out, provided that the class of the object is clear from the context in which it is presented.
- Each rectangle in a diagram stands for a unique object, even if the objects are not distinguished by explicit names. Objects that are not given explicit names are called *anonymous* objects.
- In order to clarify a certain *snapshot* the object attributes may show concrete *values.* Although subclass structures are not shown in object diagrams, the inherited attributes can be mentioned explicitly (e.g. attribute 'age' in Figure 3). Methods are usually not shown in an object diagram.

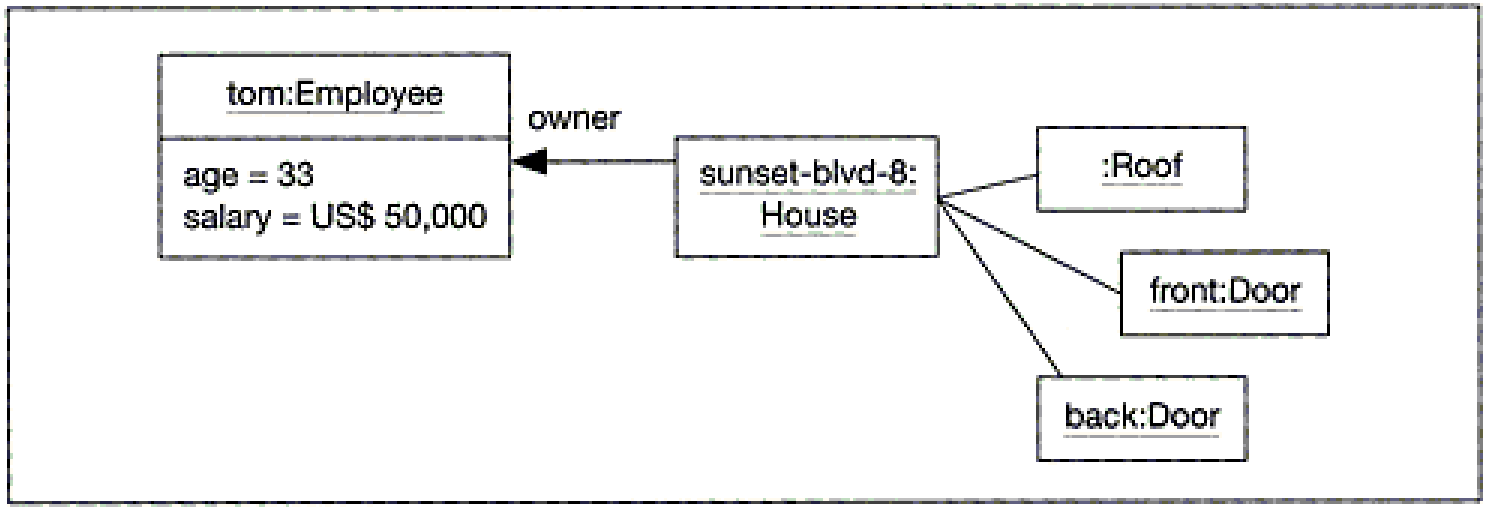
- In the same way as an *object* is an instance of a *class*, a *link* is an instance of an *association*. According to the association in the class diagram in Figure 1, each House object has exactly one link to a Human object, namely its owner. In the opposite direction, Human (and therefore also Employee and Child) objects may have an arbitrary number of links.
- The given association has an arrow from class House to Human, thus allowing navigation starting from houses. This is reflected by the link carrying an arrow in that direction.
- Although links in standard UML are bidirectional in nature, *an implementation might only allow navigation in one direction*.

- In this example, the *role name* 'owner' can be added to a link. A role name can be used in an object diagram to clarify which association a link represents. Alternatively, if an association name is given, it may also be attached to its links.
- Object diagrams reflect the composition of House objects consisting of exactly one Roof object, and an arbitrary number of Door objects – composition is realized by means of links to the aggregated objects.
- As the class diagram in Figure 1 does not contain *navigation arrows* for the composition relationships, the object diagram also does not show navigation arrows. If desired and inferable from the context of a link, the diagram may also drop association and role names.

Exemplar nature of object diagrams

- During runtime, a system may dynamically change its structure. New objects are created, and others become obsolete and are eventually removed from the system by garbage collection. Of course, the linkage between objects changes.
- An *object diagram only shows a snapshot* of the system. A snapshot is a partial view of the system because it does not usually show all objects and links.
- Furthermore, it shows the structure of the system only at a particular point in time. Figure 4, for example, represents another object diagram consistent with the class diagram in Figure 1.

Object diagram- Figure 4



- Object diagrams are therefore of exemplar nature. Several of them may *complement the information conveyed by a class diagram*, because the latter describes all the potential snapshots.
- A class diagram is a constraint on snapshots in the sense that each snapshot of a system actually has to conform to the class diagram.
- An object diagram, instead, is useful to exemplify a *particular situation*, e.g. during the initialization phase, before or after a particular operation, or an erroneous state. Also, object diagrams can be used to describe the structure of a composite object or a desired object structure – and they are especially useful in describing standard situations within a framework, or in describing how objects of newly added classes integrate with existing framework objects.

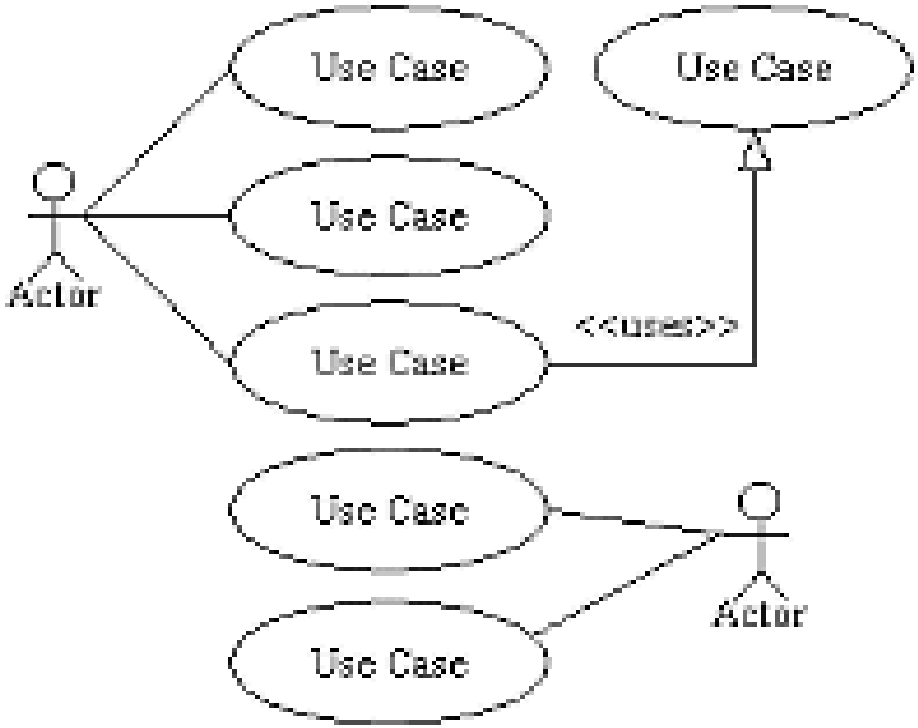
- Several object diagrams may coexist to exemplify different situations. In the case that explicit names are given, it is even possible to specify *overlapping object structures*.
- For example, in Figures 3 and 4 employee Tom owns sunset-blvd-8. It is important to keep in mind that an additional textual explanation may be needed to clarify which situation is described with these diagrams.
- Both, may for example, describe different situations in the life of Tom, e.g. before and after winning the lottery.

- The situation captured by an object diagram *does not necessarily occur while* the system is running – it might happen that the described object composition never occurs.
- On the other hand, an *object structure may also occur more than once* – it may repeatedly occur at different points in time, or may occur several times in the same system snapshot.
- One of the *most prominent uses for object diagrams is to describe an object structure that is created initially* and that statically holds for the whole system's execution, or until its objects are deleted.

- Object diagrams give to a developer a useful clue about the *structure of the framework to be used*. Object diagrams are particularly useful in explaining framework architectures.
- **In complex situations, or when association multiplicity of more than 1 is involved, the diagrammatic depiction of several distinct objects of the same class allows the illustration of situations that cannot be described well using class diagrams alone**.
- Thus, object diagrams become a useful tool for the discussion of framework construction principles.
- Many pattern descriptions also use object diagrams to exemplify the object structures on which they operate.

Use Cases Diagrams

- Use case diagrams ***model the final functionality of system*** using *actors* and *use cases*.
- ***Actors*** are played roles of different persons and systems that interact with the developing system. Are represented by *styled humans*
- ***Use cases*** are oriented on a **scope** and represent *what the system must to do* and *not how to do*. Are represented by an *oval*


Use Case
Use Case
Use Case
Use Case
Use Case
Use Case
Actor
Actor
<<uses>>

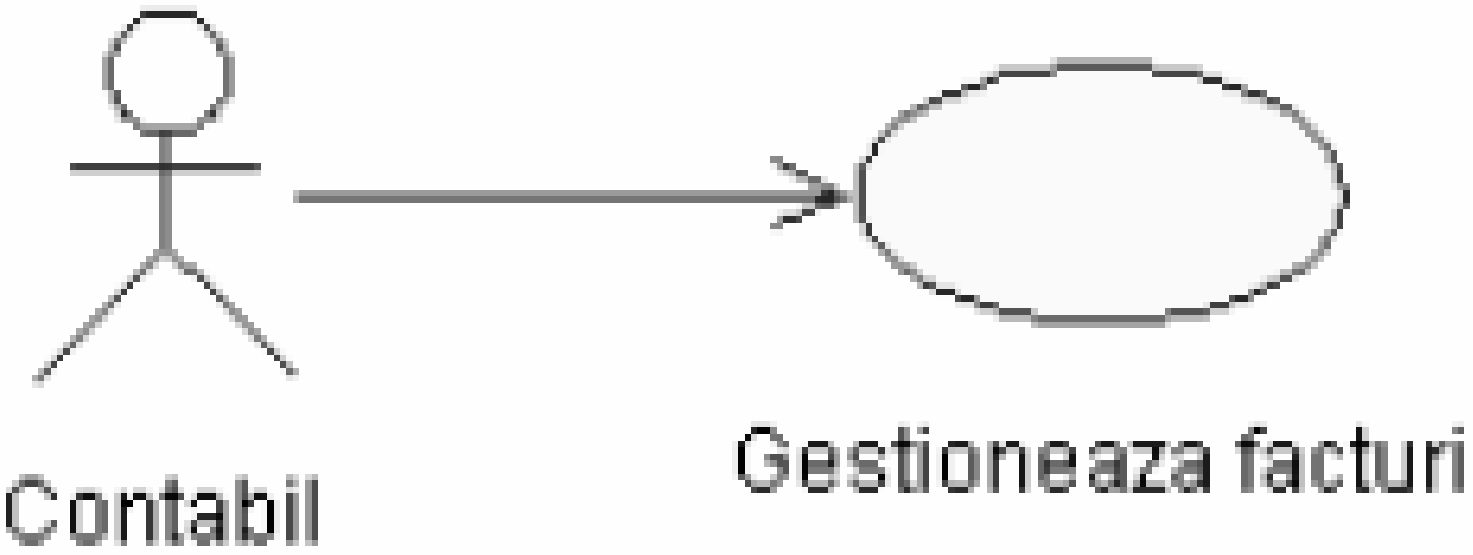
Roles in Use Cases

- 1. **Actors and use-cases** involves association (communication)
- 2. **Roles between use-cases**

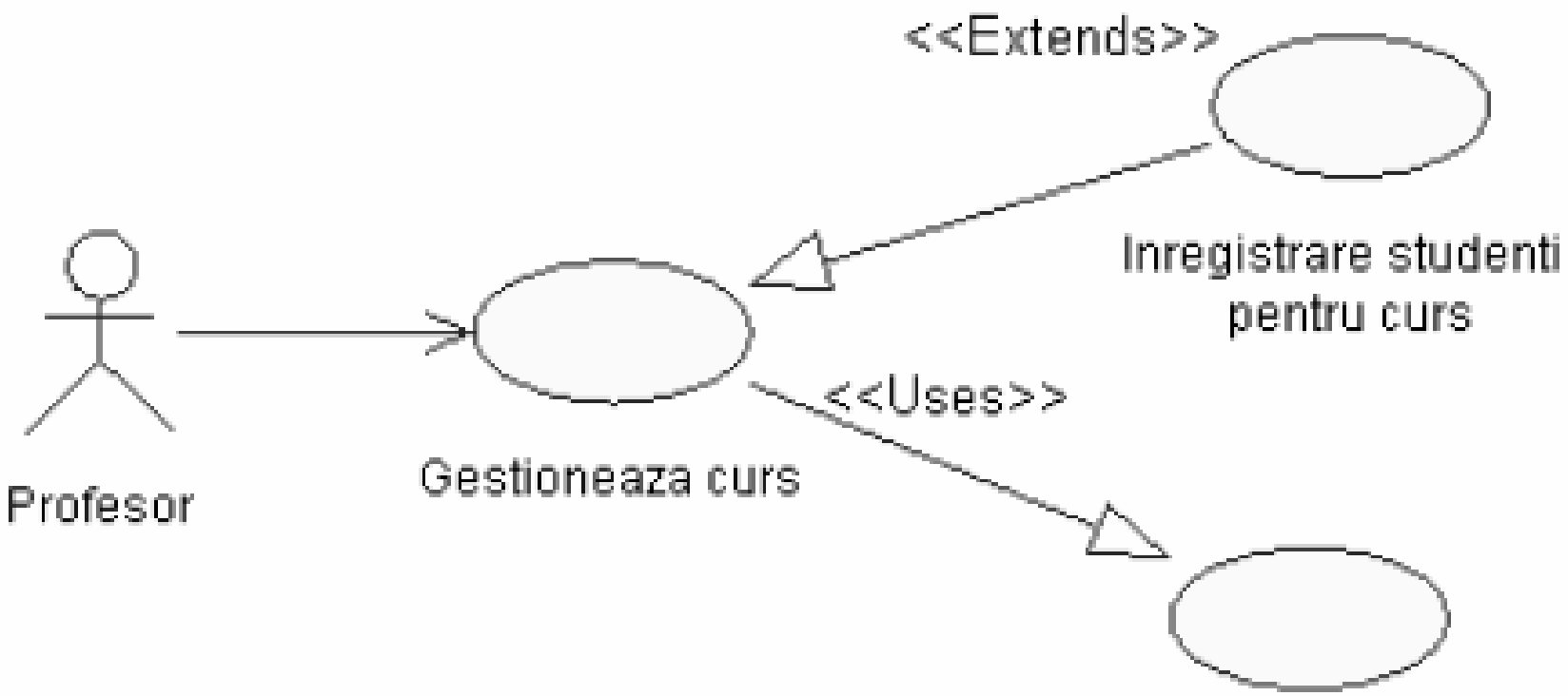
- <<uses>> a use case that *uses* the functionality of an other use case

  - <<extends>> an optional behavior only in some cases
- 3. **Between actors**
  - Generalization, as an *extension* between 2 use cases
  - Dependency, an actor *depends* by an other actor

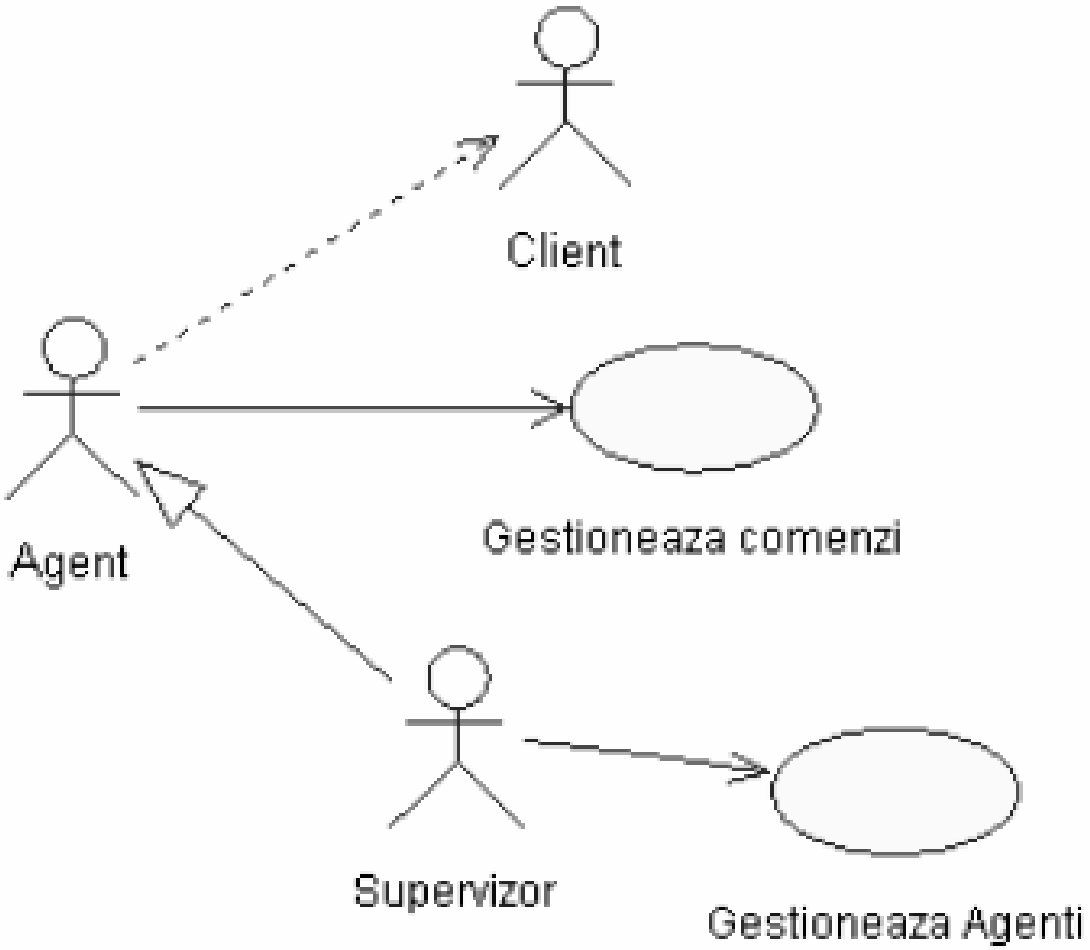
Actors use-cases roles



Roles between use-cases



Roles between actors

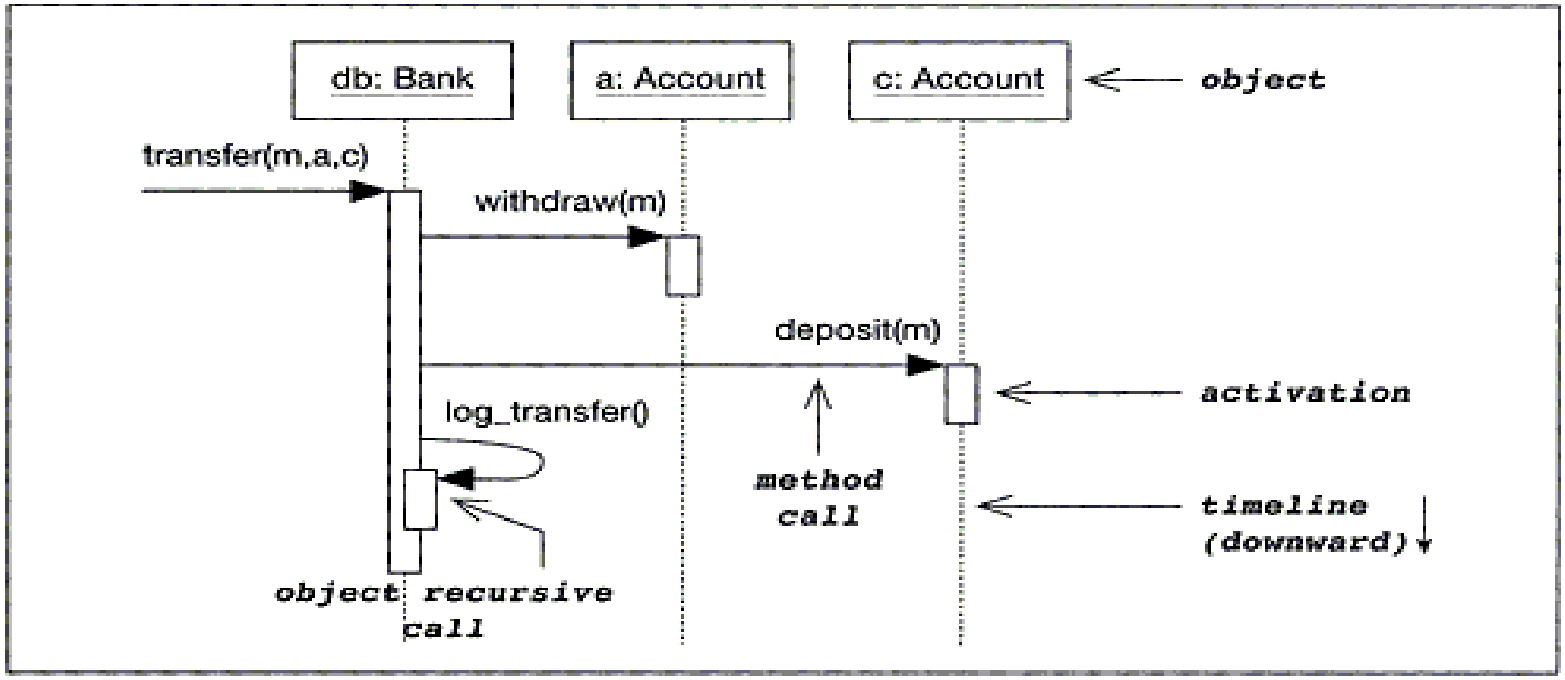


Sequence diagrams

- It is inevitable that *behavioral* notations are needed in addition to *structural* notations. In order to describe the overall behavior of a system, it is useful to look at the ***interactions*** among its components, and its interactions with the environment.
- Sequence diagrams present *interactions in two dimensions*. At the top level, the *involved objects* are shown. From top to bottom, the order of the method calls between the objects is shown according to a *timeline.*
- A sequence diagram *tells a story about a system*. The story describes a certain aspect of the running system – namely a number of collaborating objects and how they interact in a particular situation.

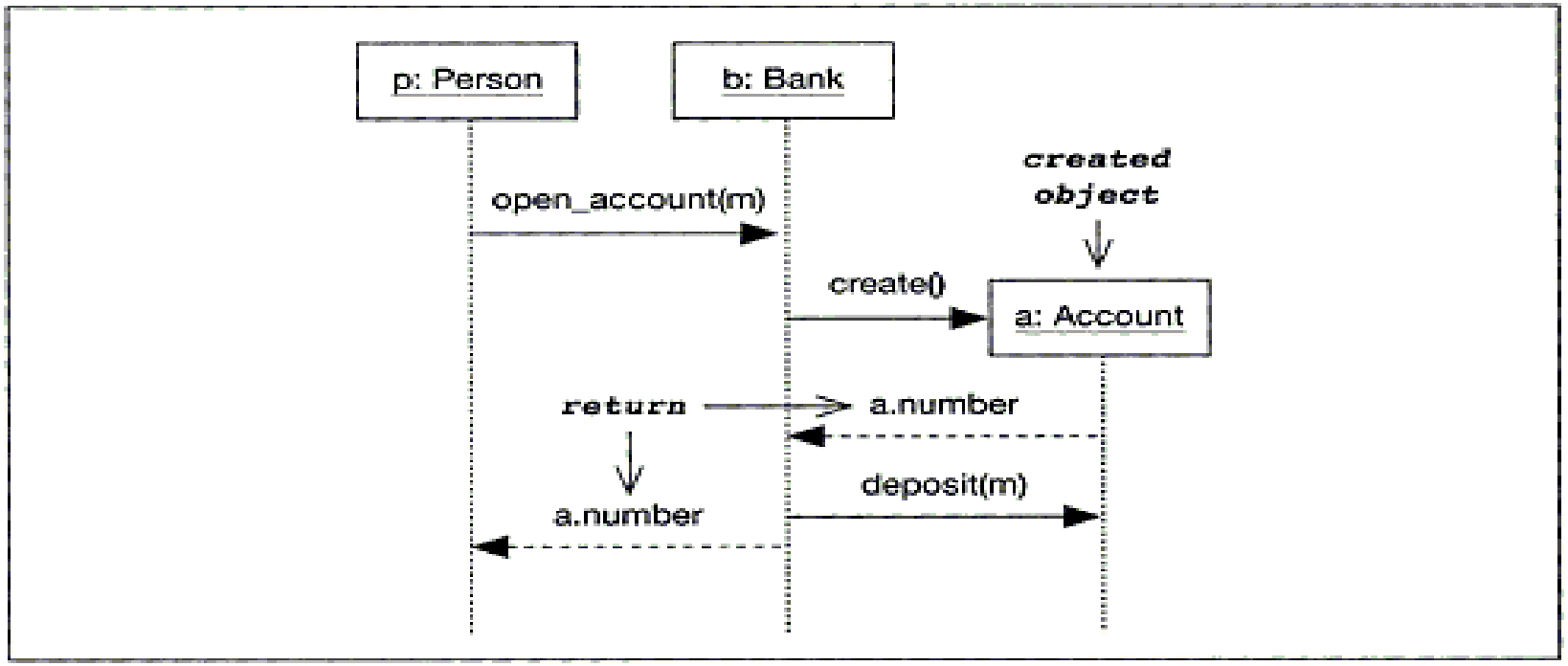
- The sequence diagram in Figure 5 presents an interaction between one *Bank object* with the name *db* and two *Account objects* with the names *a* and *c*.
- The diagram shows how these objects interact in order to transfer an amount, *m,* between the two accounts.
- A *transfer (m, a, c)* message is issued by the environment–possibly by some user interface component.
- The message activates the corresponding method in the Bank object, leading to three more calls.
- One of these calls is *object recursive* – the *log_transfer()* method is invoked on Bank itself
- The duration of a method can be shown by using an *activation box.* However, we use the *timeline* only qualitatively, which means the actual length of an activation box or distance between two method calls does not give any quantitative measure about execution time duration.
- The objects in the sequence diagram use the same naming conventions as in object diagrams.

Sequence diagram – Figure 5



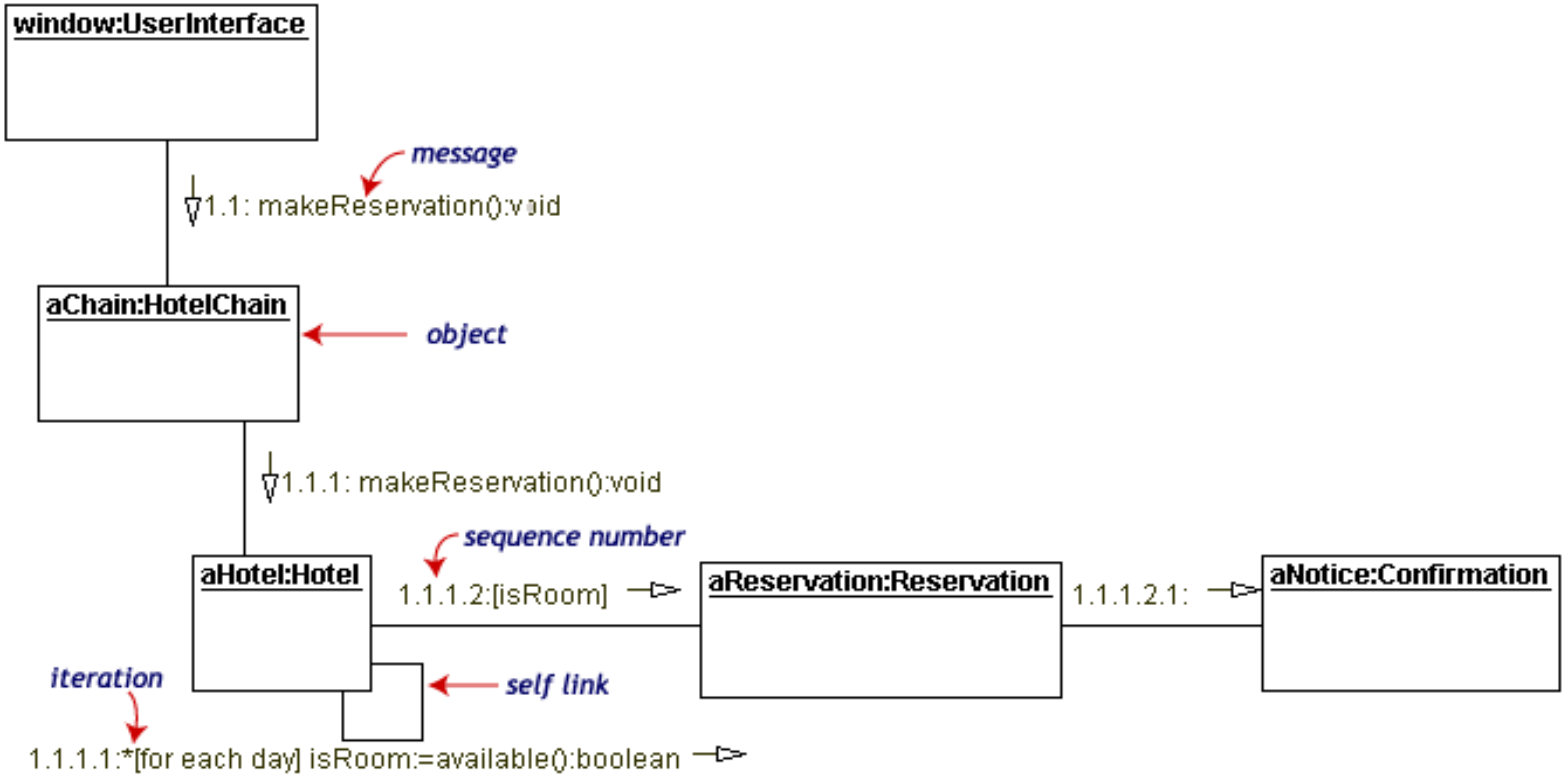
- In a system that applies information hiding, an attribute is normally only changed by the object owning that attribute. Thus, *method calls* are the important *interactions between objects.*
- Method calls often have *return values*. If it is relevant to show a return value in a diagram, *a dashed arrow* can be used. It also happens that an object is created dynamically during an interaction sequence.
- Both *object creation* and *return values* can be described in a sequence diagram, as shown in Figure 6.
- In this figure, activations are omitted. Instead, two return arrows are shown carrying the account number of the newly created *Account* object.
- This object is not drawn at the top level in the diagram in order to indicate that it is created as a reaction to the Bank object's *open_account()* request.

Return parameters and object creation in sequence diagrams –Figure 6



Considerations about collaboration diagrams

- UML provides an alternative way to present interaction sequences, namely *collaboration diagrams*.
- These diagrams exhibit information similar to sequence diagrams, but present it in a different form. Instead of showing a timeline, ***a numbering of the interactions*** between the collaborating objects is used.
- Thus, collaboration diagrams can use two dimensions to arrange objects, instead of only the horizontal axis as in sequence diagrams.
- Collaboration diagrams are thus similar to object diagrams in layout and intuition.
- In practice, following the time axis of sequence diagrams is often a more intuitive way to grasp the interactions than following the numbered arrows of collaboration diagrams.

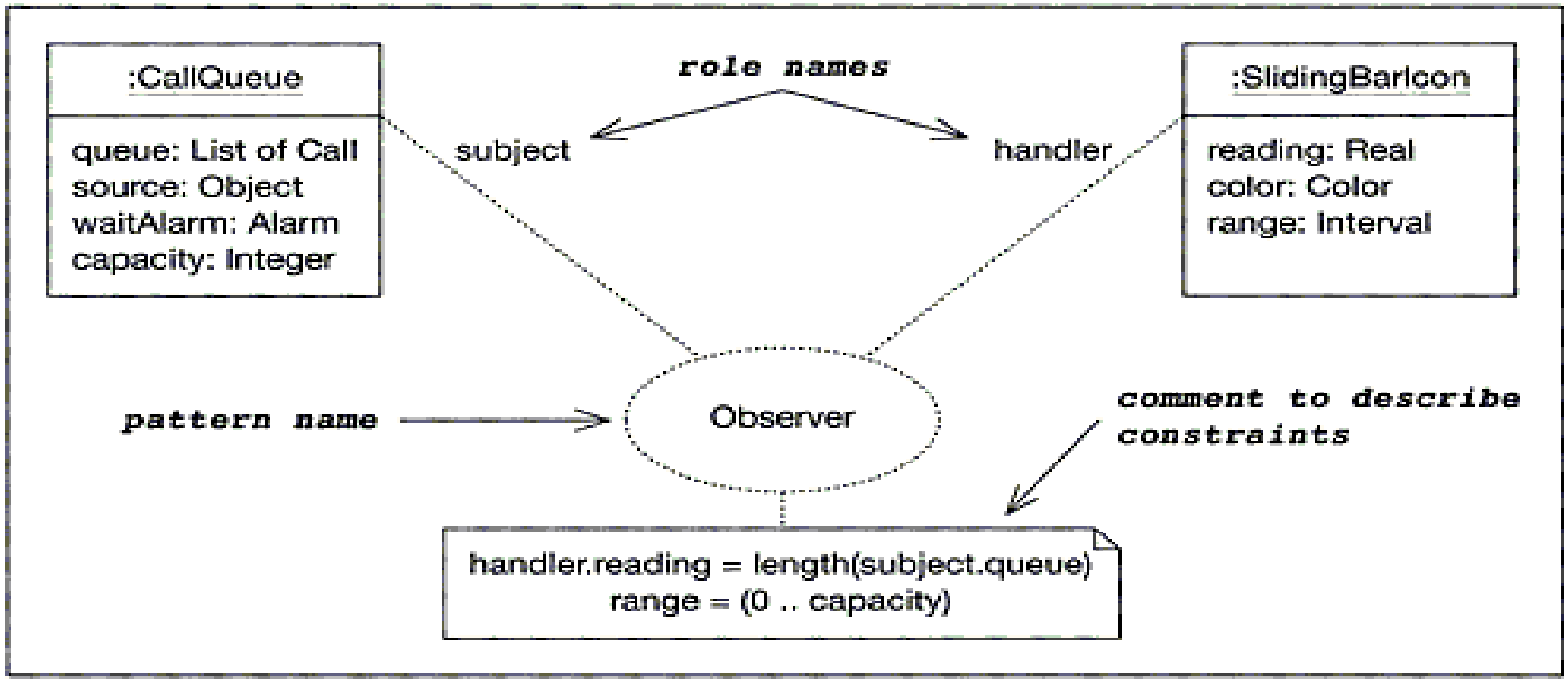

window:UserInterface
message
1.1: makeReservation():void
aChain:HotelChain
object
1.1.1: makeReservation():void
sequence number
aHotel:Hotel
1.1.1.2:[isRoom]
aReservation:Reservation
1.1.1.2.1:
aNotice:Confirmation
iteration
self link
1.1.1.1:*[for each day] isRoom:=available():boolean

Facilities offered by Collaboration diagrams

- -***concurrency*** specified by a letter
- -***self (repeated)*** specified by an asterisk *
- -***multithreading***:
  - ***{concurrency = sequential}***
  - ***{concurrency = concurrent}***
  - ***{concurrency = guarded}***
- ***Asynchronous, balking call***
- ***Refinements*** as synchronizations, preconditions, etc.

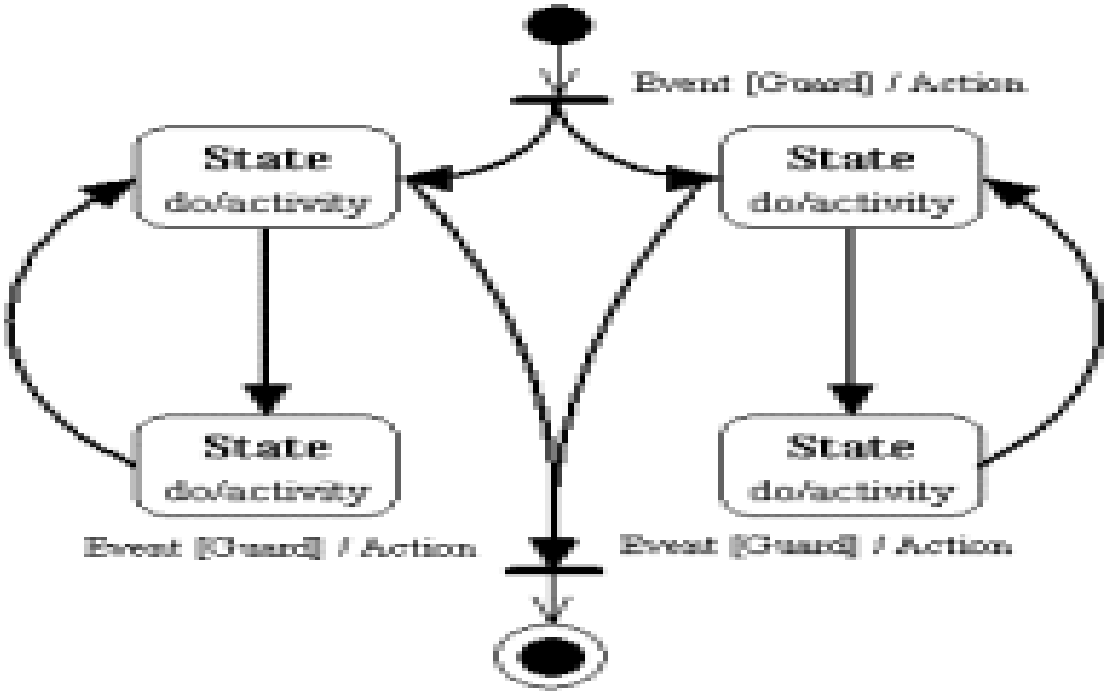
- UML also provides concepts to describe aspects of ***design patterns***, by using *collaboration diagrams*. Figure 7 is taken directly from the UML standard document.
- It demonstrates the application of the ***Observer design pattern*** to two respective classes. It uses constraints in a comment to describe structural invariants between participating classes.
- The ellipse denotes a *pattern*, and lines serve as *connections* to participating classes.
- However, given the generally high density of design patterns in a framework, such a representation could quickly clutter a framework class diagram.
- Another drawback is that it does not show which attributes or methods actually participate in a design pattern.

UML suggestion for pattern description –Figure 7



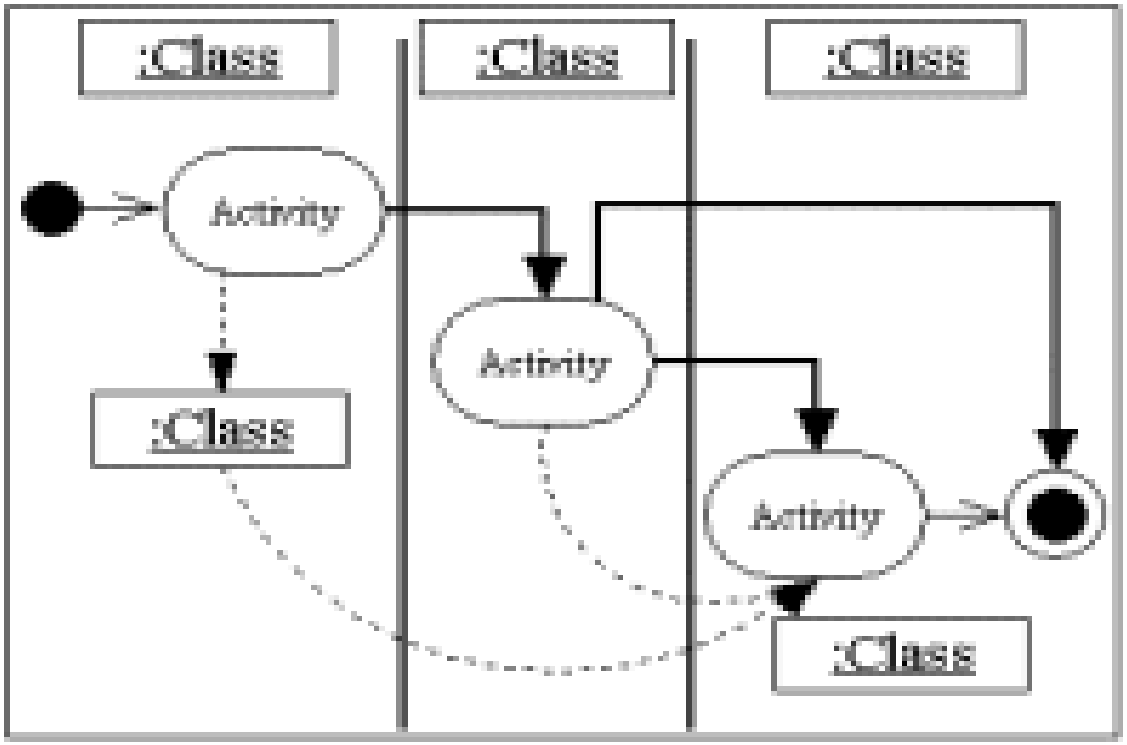
State Machine Diagram

- ***Statechart diagrams*** describe the *dynamic behavior of a system* ***in response to external stimuli*****.**
- Statechart diagrams are especially useful in ***modeling reactive objects*** whose states are triggered by specific events.
- Describes the *states* an object or interaction may be in, as well as the *transitions between states*.
- Formerly referred to as a state diagram, state *chart* diagram, or a state-*transition* diagram


Event [Guard] / Action
State
do/activity
State
do/activity
State
do/activity
State
do/activity
Event [Guard] / Action
Event [Guard] / Action

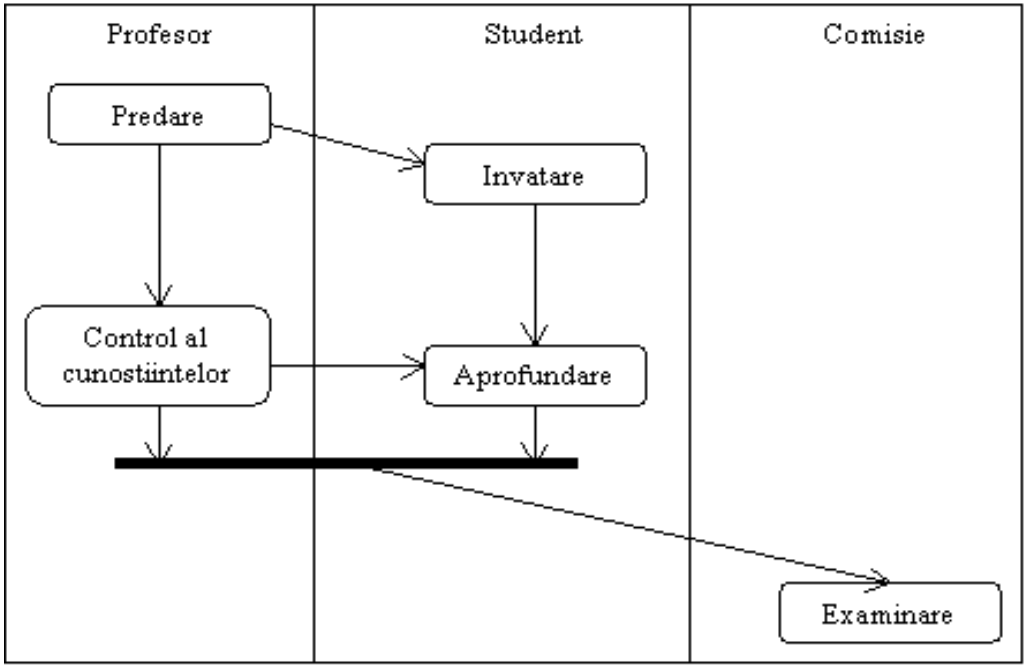
Activity Diagrams

- Activity diagrams illustrate the *dynamic nature of a system* by ***modeling the flow of control*** from activity to activity.
- An ***activity*** represents an operation on some class in the system that results in a change in the state of the system.
- Typically, activity diagrams are used to *model workflow or business processes* and *internal operation*.


:Class
:Class
:Class
Activity
Activity
Activity
:Class
:Class

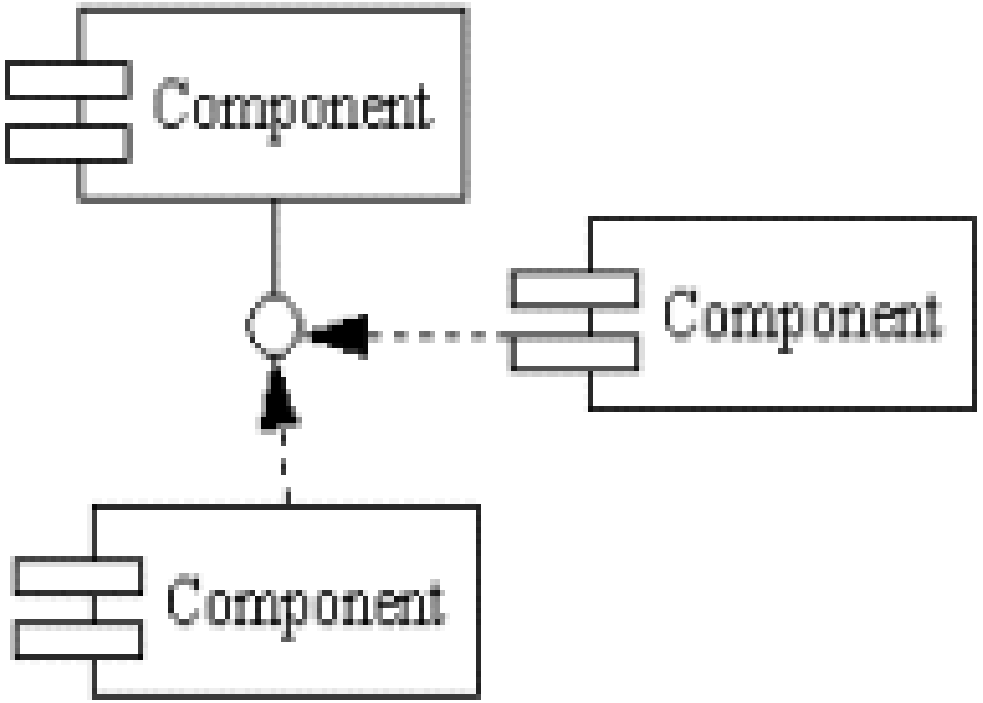
- An *activity* is represented by a *round rectangle* as the states but the roundness is a *semi-circle*
- The activities are linked to automated transactions and represents synchronizations between control streams represented by *synchronization bars* (fork, join)

Partitia unei diagrame de activitati in culoare de activitati.



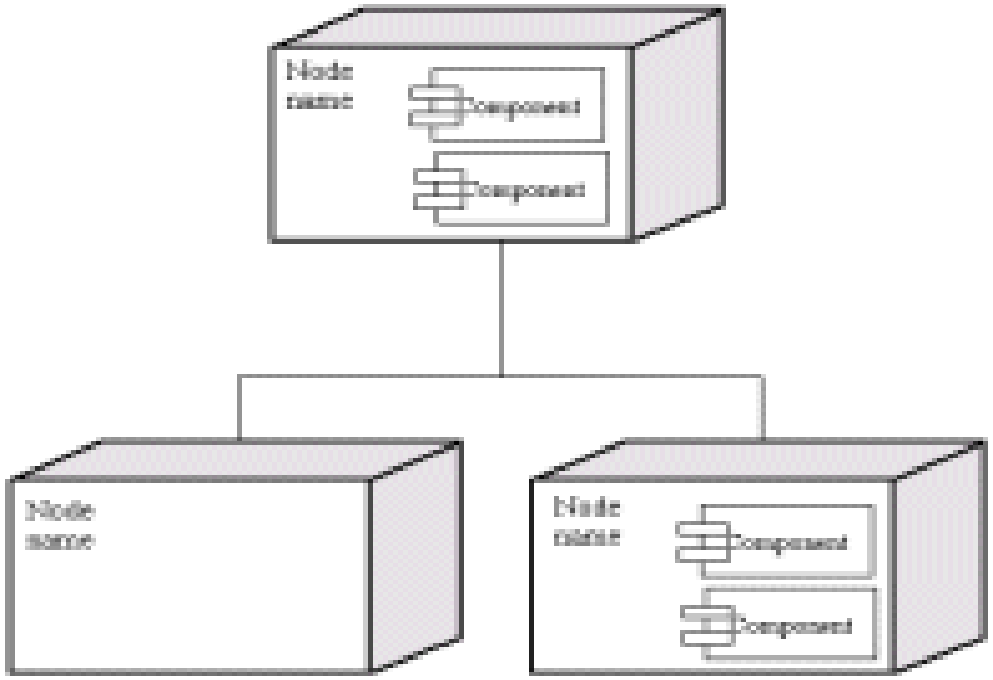
Component Diagrams

- Component diagrams describe the organization of physical software components, including source code, run-time (binary) code, and executables.
- Contains:
  - Subsystems
  - Components
    - Main programs
    - Modules
    - Subprograms
    - Processes
  - Dependences


Component
Component
Component

Deployment Diagram

- Shows the *execution architecture* of systems.
- Deployment diagrams depict the *physical resources* in a system, including nodes, either hardware or software execution environments, components, and connections.


Node
name
Component
Component
Node
name
Node
name
Component
Component

Example - lab

- The aim of this example is to show how to use UML in "real" software development environment.

- The problem that will be solved is the: **Elevator Problem**,
- from basic UML classes to pseudo code program