



# Software Engineering Course's Code: CSE 305

# Chapter 5. System Models

5.1. System Modelling

5.2. Functional Models

5.3. Structural models

5.4. Behavioral models



- **A model** is an abstract view of a system that ignores system details.
- System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system.
- **System models** can be developed to show the **system's context, interactions, structure and behavior**.
- System models are represented using some graphical notations
  - ❑ Now almost always based on notations in the **Unified Modeling Language (UML)**.

- **System modelling** helps the analyst to understand the **functionality of the system** and **models are used to communicate with customers**.
- Models of the system are used during Requirements Engineering
  - ❑ to help explain the proposed requirements to other system stakeholders
  - ❑ Software Engineers use these models **to discuss design proposals and to document the system for implementation**

.



- **An external perspective**, where you model the **context or environment** of the system.
- **An interaction perspective**, where you model the **interactions** between **a system and its environment**, or between the **components of a system**.
- **A structural perspective**, where you model the **organization of a system** or the structure of the data that is processed by the system.
- **A behavioral perspective**, where you model the **dynamic behavior** of the system and how it responds to events.

# Chapter 5. System Models

5.1. System Modelling

5.2. Functional Models

5.3. Structural models

5.4. Behavioral models



## **Use Case Diagram**

**Captures the system behavior from the users' point of view**



**Captures the system behavior from the users' point of view**

## Purpose of use case diagrams

### Use case

- Focuses on a single behavior of the system from an external point of view
- Describes a function provided by the system that yields a visible result for an actor
- Is typically initiated by an actor
- An actor is any external entity that interacts with the system

### Use case diagram

- Shows the main use cases, actors and their associations





## Textual description of a use case

Similar to a scenario, use cases can be specified textually and consists of:

**Name:** Description of the use case

**Actors:** Users of the system from outside the system boundaries

**Entry conditions:** Conditions which have to be met at start of the use case

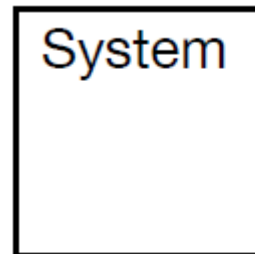
**Event flow:** The actual event flow of a single use case

**Exit conditions:** The state of the system after the use case has finished



## Use case diagram elements

The scope of a system can be represented by a system (shape), or sometimes known as a system boundary. The use cases of the system are placed inside the system shape, while the actor who interact with the system are put outside the system. The use cases in the system make up the total requirements of the system.



System  
boundary



## Use case diagram elements

Actors are the entities that interact with a system. Although in most cases, actors are used to represent the users of system, actors can actually be anything that needs to exchange information with the system. So, an actor may be people, computer hardware, other systems, etc.

Note that actor represents a role that a user can play but not a specific user. So, in a hospital information system, you may have doctor and patient as actors but not Dr. John, Mrs. Brown as actors.

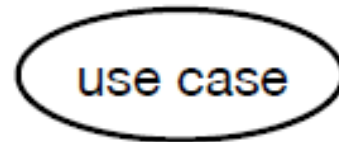


Actor



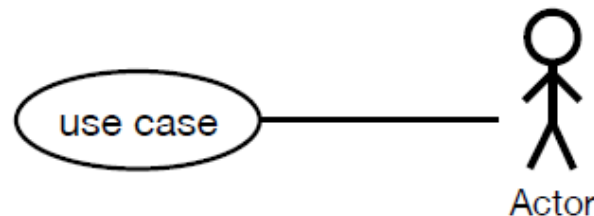
## Use case diagram elements

A use case represents a user goal that can be achieved by accessing the system or software application.



Use case

Actor and use case can be associated to indicate that the actor participates in that use case. Therefore, an association correspond to a sequence of actions between the actor and use case in achieving the use case.

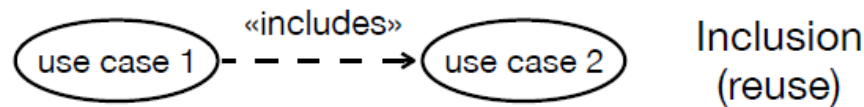


Association

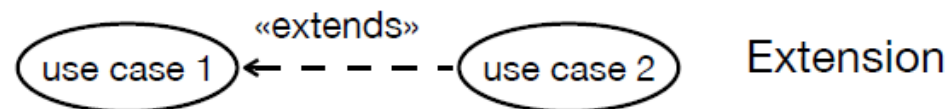


## Use case diagram elements

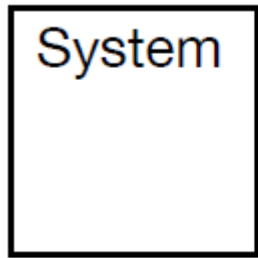
An include relationship specifies how the behavior for the inclusion use case is inserted into the behavior defined for the base use case.



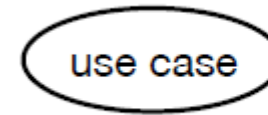
An extend relationship specifies how the behavior of the extension use case can be inserted into the behavior defined for the base use case.



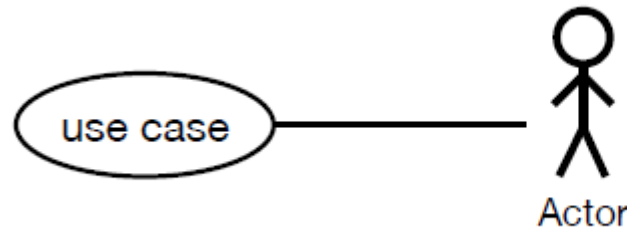
## Use case diagram elements



System  
boundary



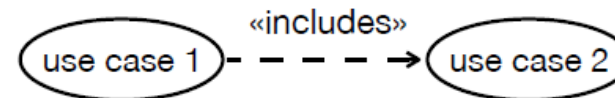
Use case



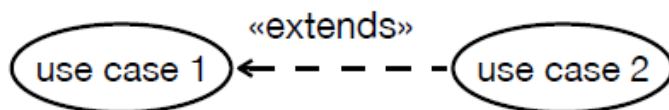
Association



Actor



Inclusion  
(reuse)



Extension



## Example of a use case diagram

- Imagine **an University App** shows **a list of courses that a student can join**
- The app also helps student **to post message** and **start discussion**
- **End Users : Student, Lecturer,**
- User Stories:
  - US1:** As a student, I want to show a list of courses, so that I can choose a Course
  - US2:** As a student, I want to join a course, so that I can enroll in that course
  - US3:** As a student, I want to manage all the enrolled courses, so that I can track my course contents



## Example of a use case diagram

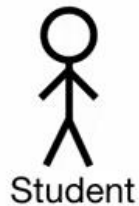
- US4: As a student, I want my account be secure, so that nobody can access my account except me
- US5: As a student, I want to post messages, so that I can express what I am thinking
- US6: As a student, I want to start discussion on course topics, so that I can know about other students' opinion.
- US7: As a lecturer, I want to view the list of courses, so that I can know which courses are offered



## EIU Step 1: Identify System Boundary and Actors

Example of a use case diagram

System boundary



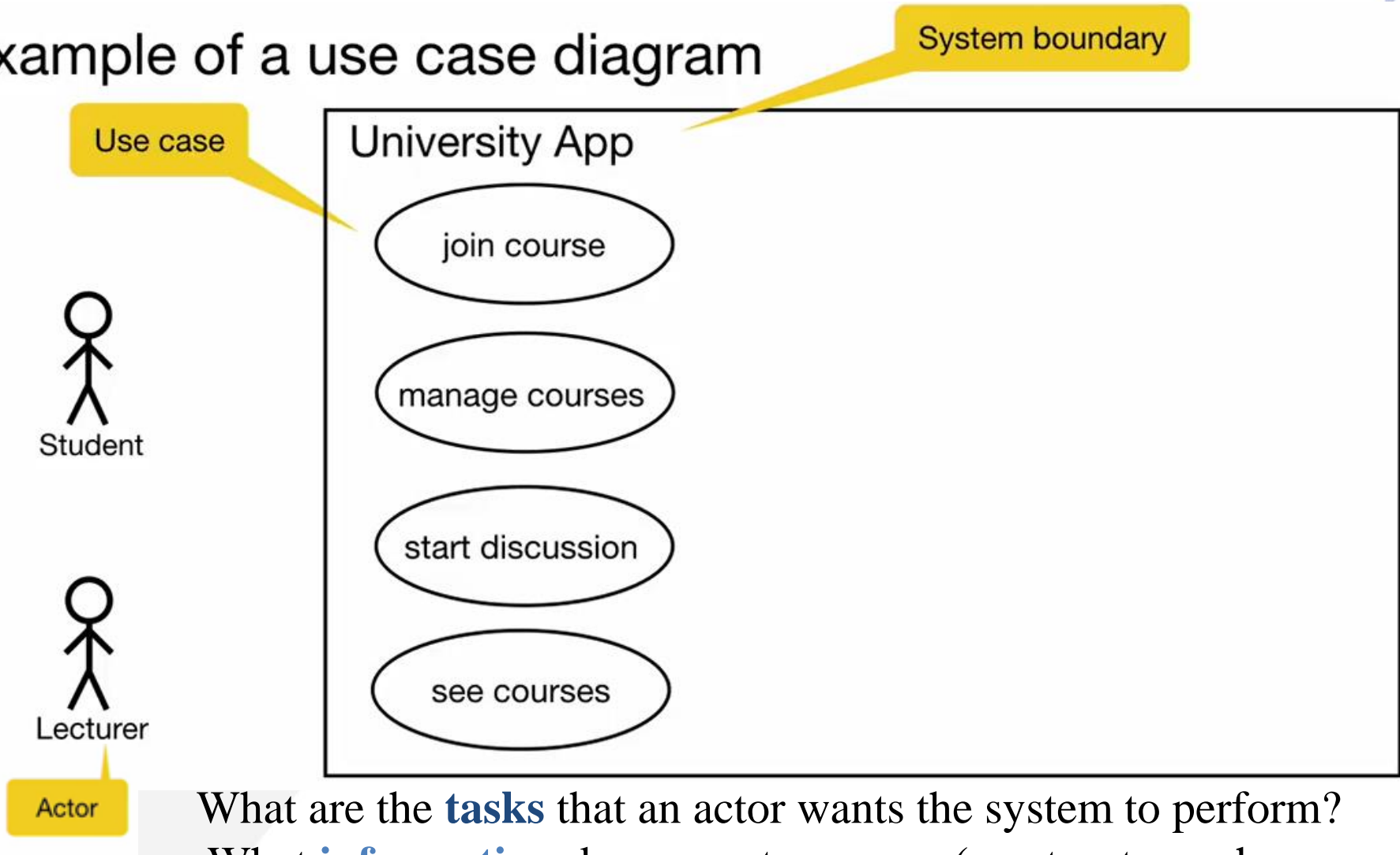
Actor

University App

# Use Case Diagram :

## Step 2: Identify Use Cases

### Example of a use case diagram



What are the **tasks** that an actor wants the system to perform?  
 What **information** does an actor access (create, store, change, remove or read) in the system?

# Use Case Diagram :

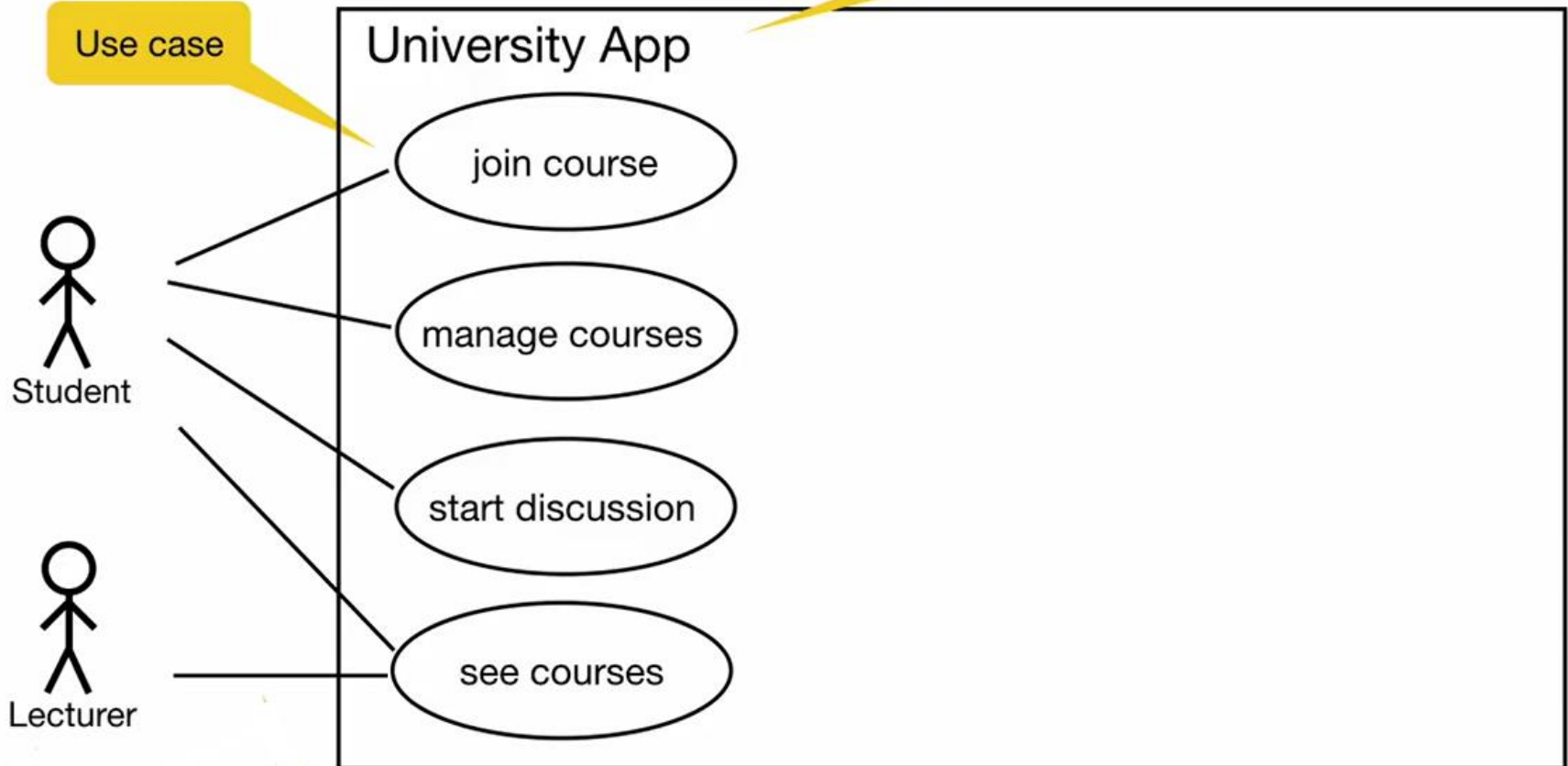
## Step 3: Identify Association



### Example of a use case diagram

System boundary

Use case

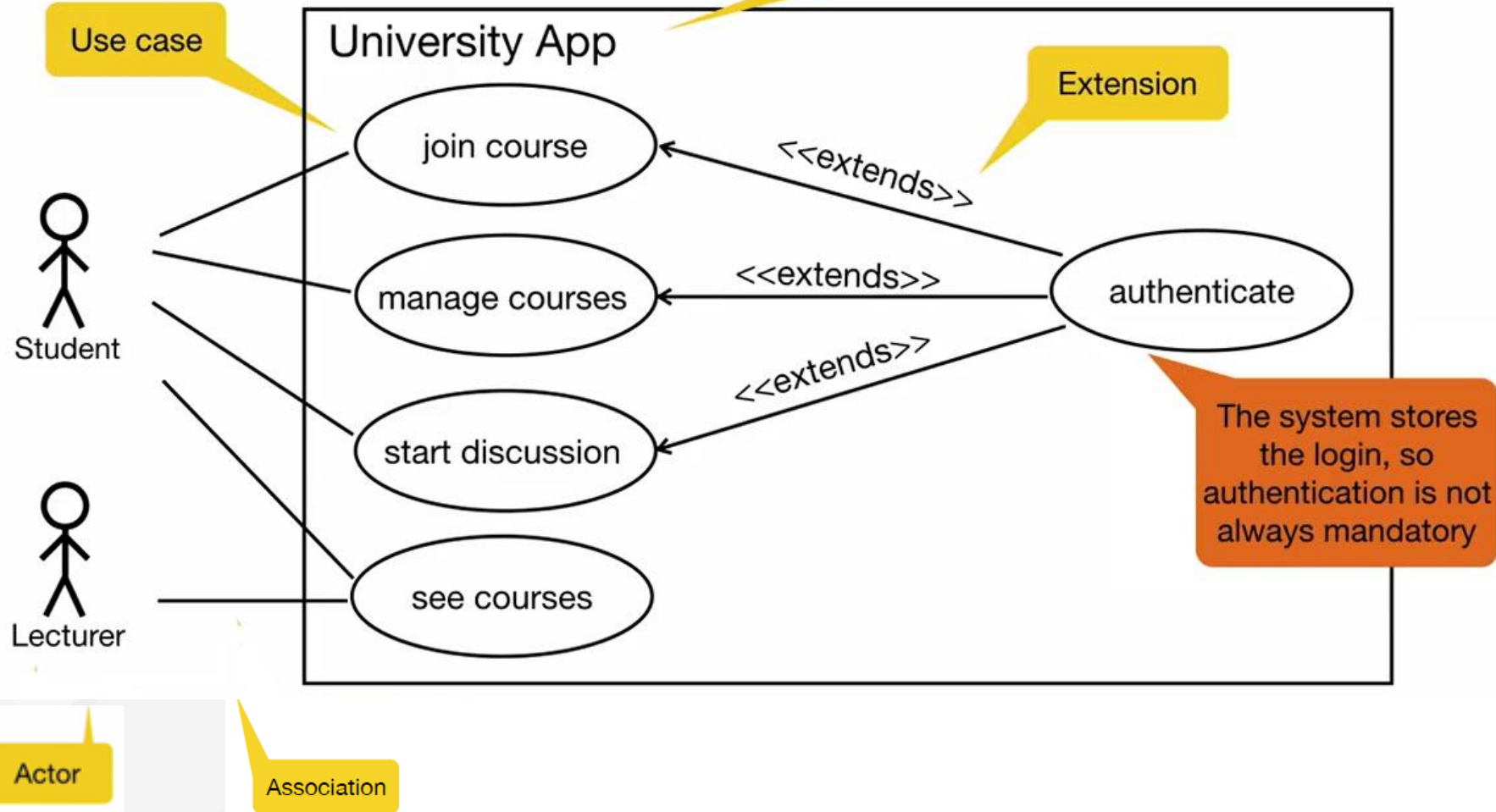


Actor

Association

# Step 4: Identify extend and include relationships

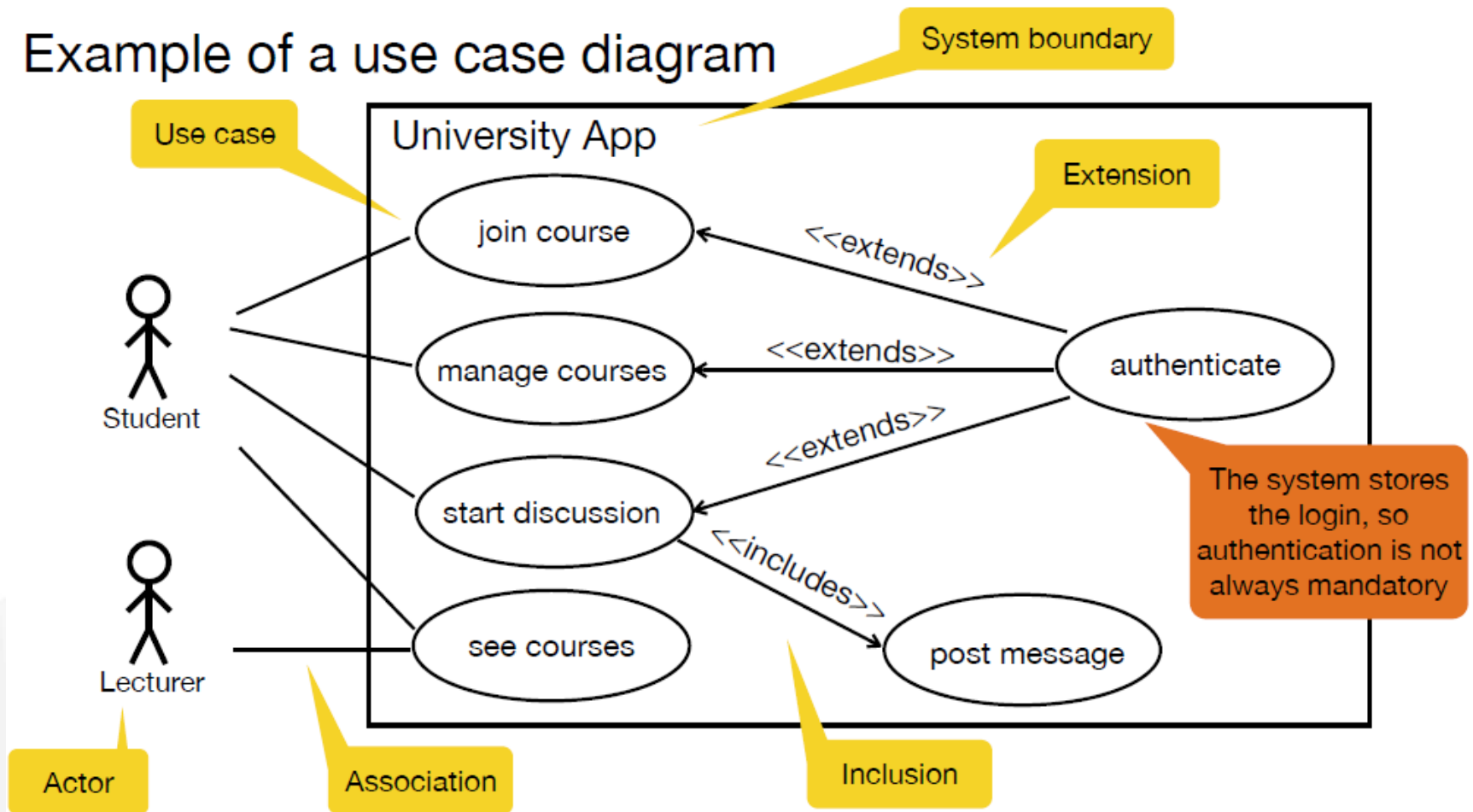
## Example of a use case diagram



# UML Use Case Diagram

## Step 4: Identify extend and include relationships

Example of a use case diagram



# Chapter 5. System Models

5.1. System Modelling

5.2. Functional Models

5.3. Structural models

5.4. Behavioral models



## Class Diagram

**Static models, which show the structure of the system**





## Purpose of class diagram

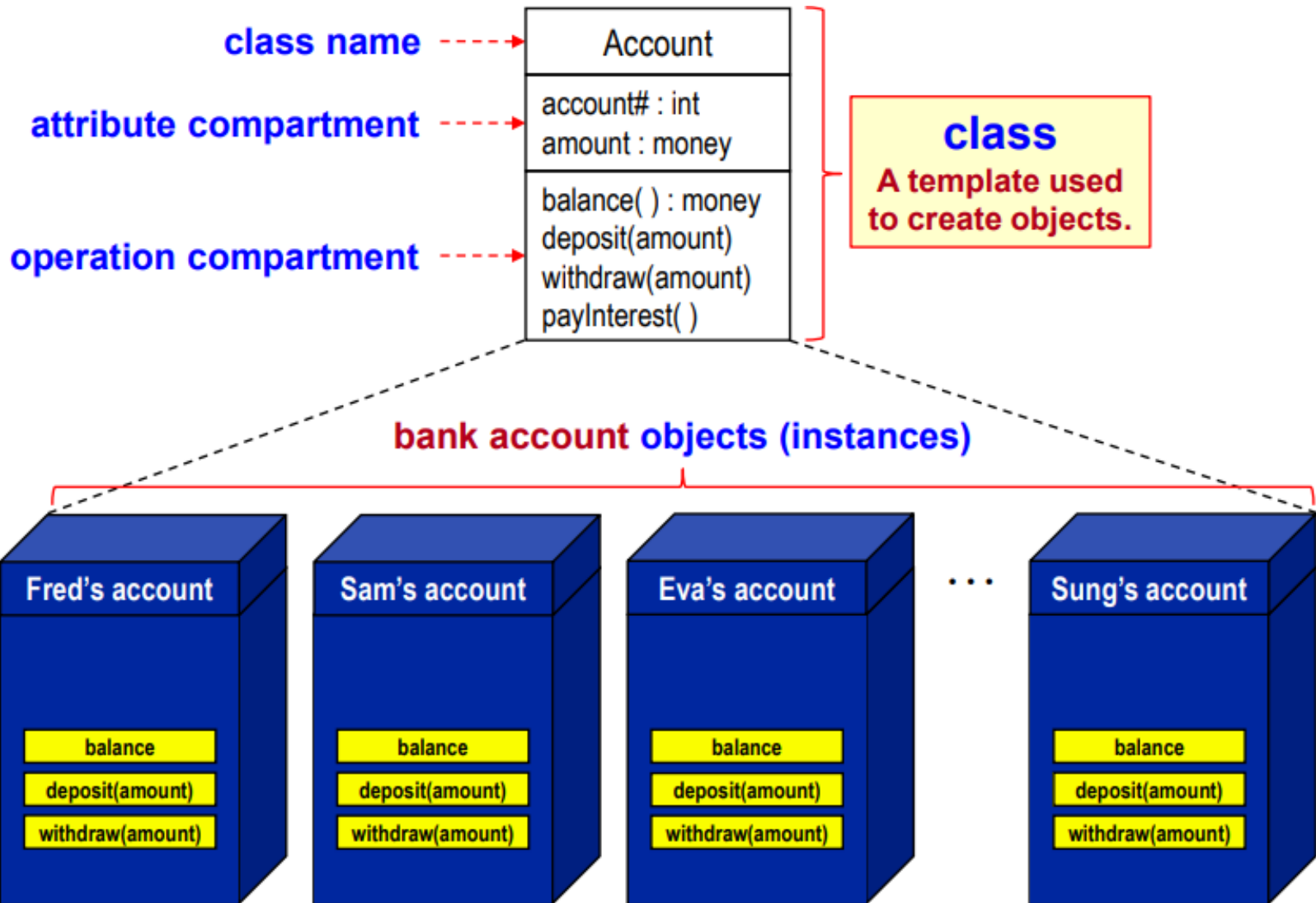
Contains the structure of objects and their dependencies to communicate the main concepts of a software system

- **Classes:** Abstract representation of an object which defines its structure and its functionality
- **Associations:** Define the relationships between objects and their corresponding dependencies / hierarchies

Models the individual concepts of the **application domain** that are manipulated by the system, their properties and their relationships

Visual dictionary of the main concepts visible to the user





An **attribute** describes the **data values** held by objects in a class.

- Attribute properties:

- name: unique within a class, but not across classes.
- type: the domain of values – string, integer, money, etc.
- visibility: who can access the attribute's values.  
public (+), private (–), protected (#), package (~)
- initial value [optional]: the attribute's initial value.
- multiplicity [optional]: the number of simultaneous values.
- changeability: whether the value can be changed.  
unspecified (default)      readOnly

For modeling, name and type should always be specified.

Account
account# : int amount : money
balance( ) : money deposit(amount) withdraw(amount) payInterest( )



**An *operation* describes a function or transformation that may be applied to or by objects in a class.**

- Operation properties:

- operation signature

operation name  
parameter names  
result type

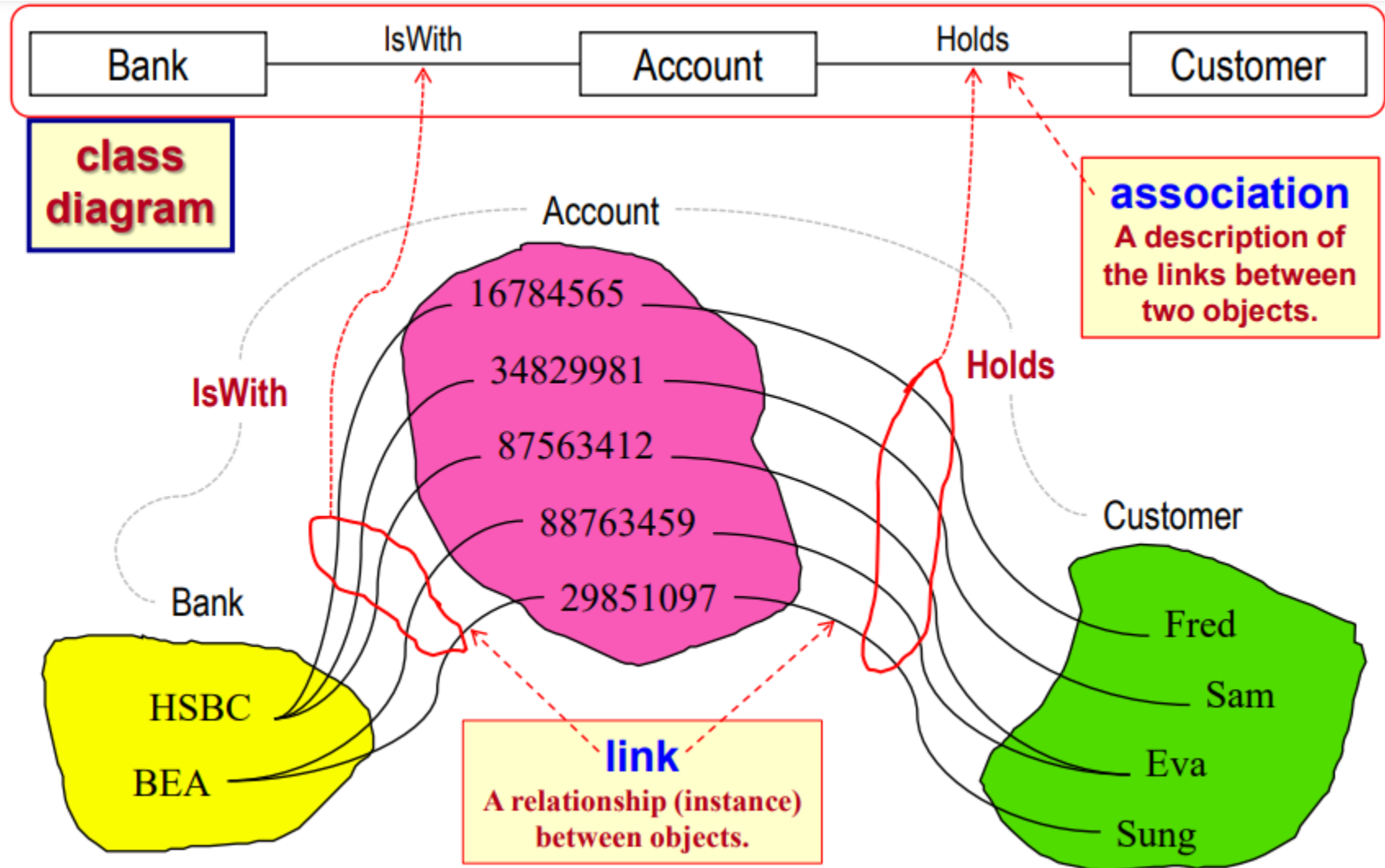
**For modeling, all should always be specified.**

- visibility

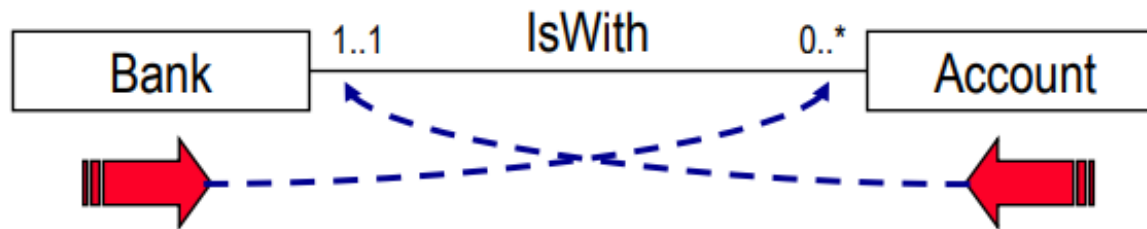
public (+), private (–), protected (#), package (~)

Account
account# : int amount : money
balance( ) : money deposit(amount) withdraw(amount) payInterest( )

- An operation instance (its implementation) is called a **method**.



**Multiplicity** specifies a restriction on the number of objects in a class that may be related to an object in another class.



**For a given bank**, how many accounts can it have?

☞ A bank may have no accounts or it may have many accounts.

**For a given account**, how many banks can it be with?

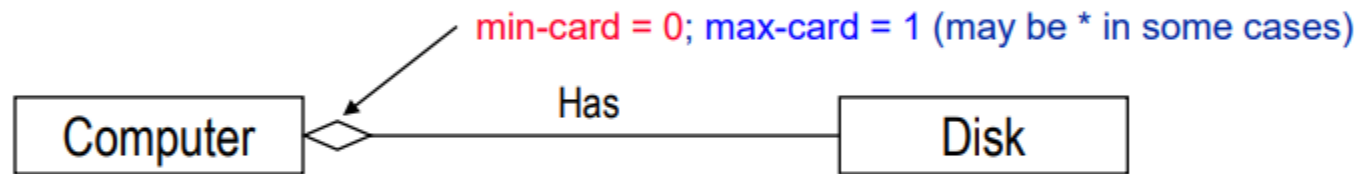
☞ An account must be with exactly one bank.

**Multiplicity is an *application domain constraint*!**

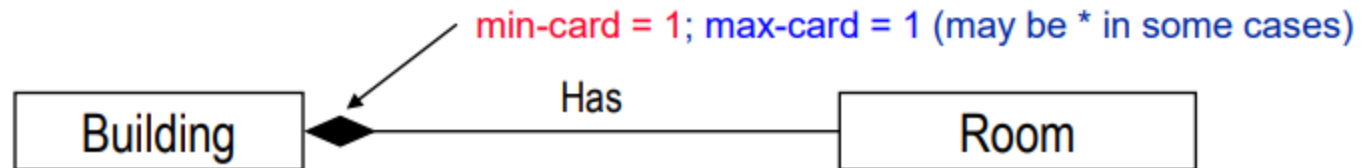
# Aggregation/Composition relationship



- A special type of association in which there is a “**part-of**” **relationship** between one class and another class.
- 👉 A component **may exist independent** of the aggregate object of which it is a part → aggregation. [◊ adornment]




- 👉 A component **may not exist independent** of the aggregate object of which it is a part → composition. [◼ adornment]



# When to use Aggregation /Composition



- Would you use the phrase “**part of**” to describe the association or name it “**Has**”?  
 **BUT BE CAREFUL!** Not all “Has” associations are aggregations.
- Is there an **intrinsic asymmetry** to the association where one object class is subordinate to the other(s)?
- Are operations on the **whole** automatically applied to the **part(s)**? → **composition**

The decision to use aggregation is a matter of **judgment**.  
It is a **design decision**.

It is **not wrong** to use association rather than aggregation!  
(In a real project, when in doubt, use association!)



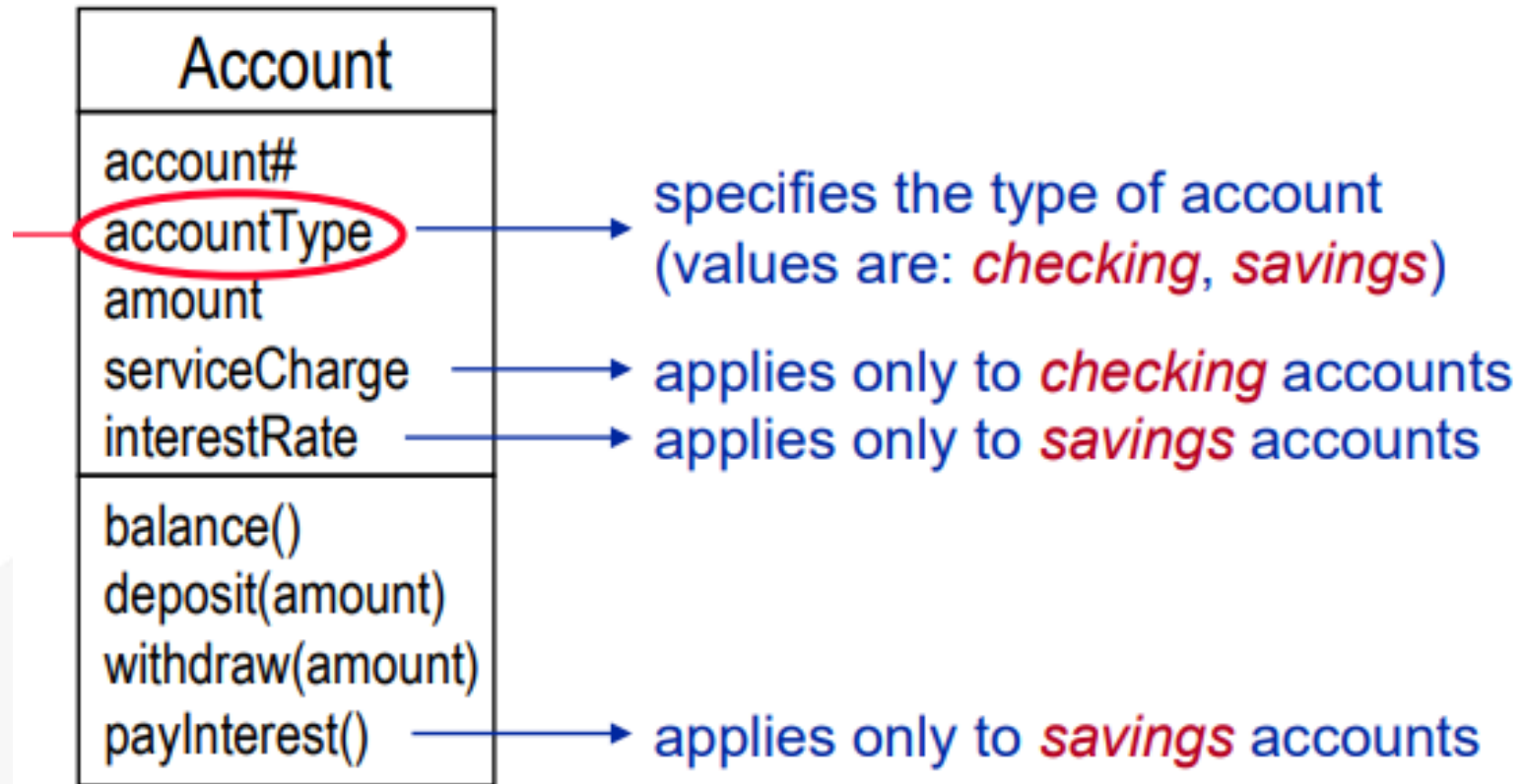


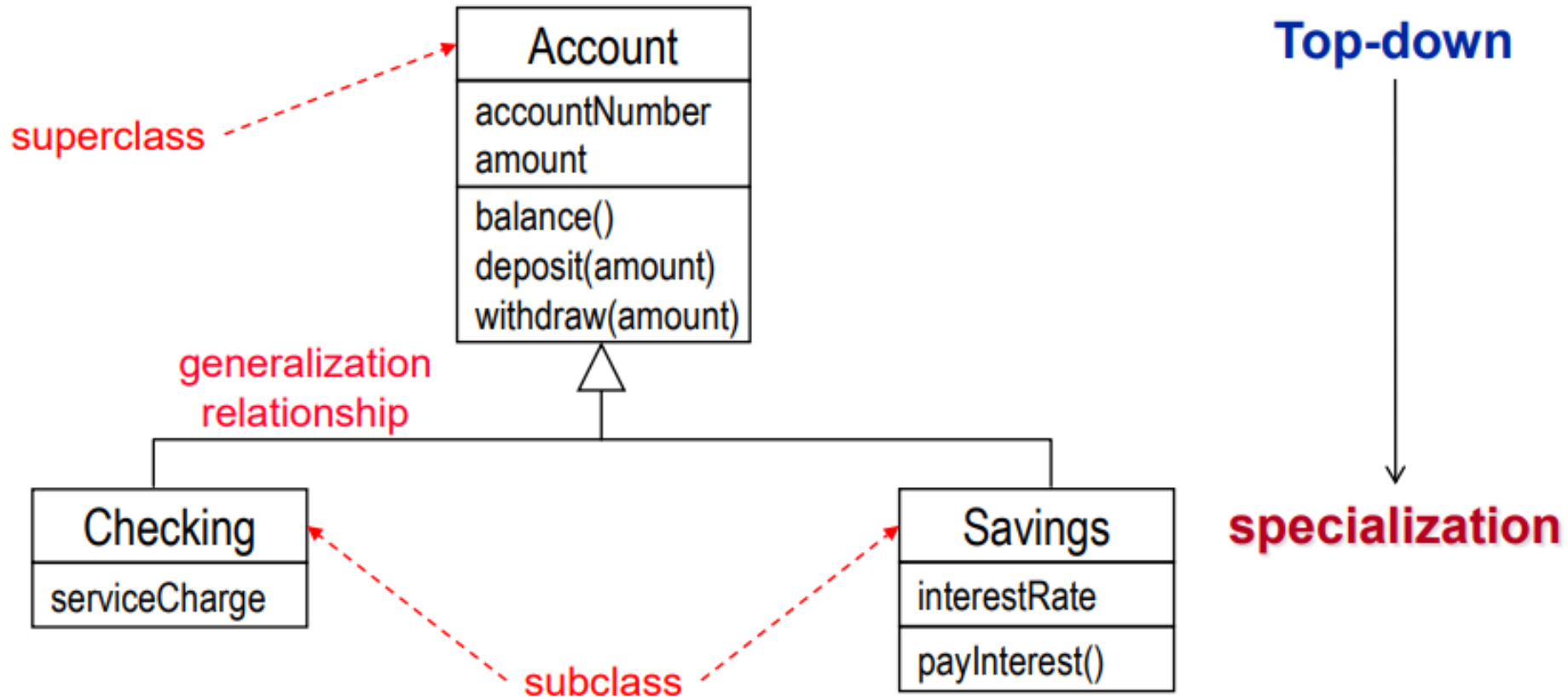
**A *generalization* is a relationship between classes of the same kind.**

- *If it is meaningful to do so*, we classify classes according to the **similarity** of their **attributes**, **operations** and **associations**.

 Look for “**kind-of**” **statements** that are **true in an application domain**.

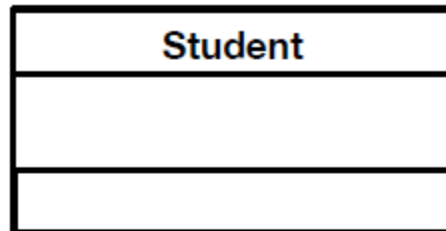




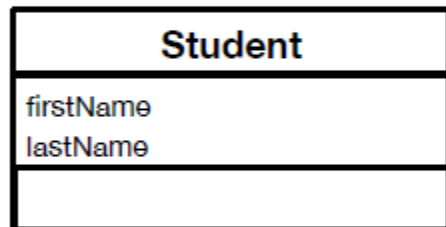




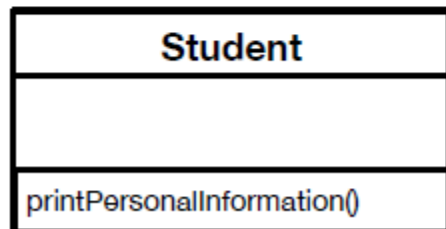
Class:




Attributes:





Methods:



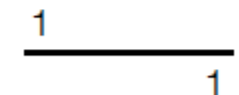
Associations:


Inheritance: 


Aggregation: 

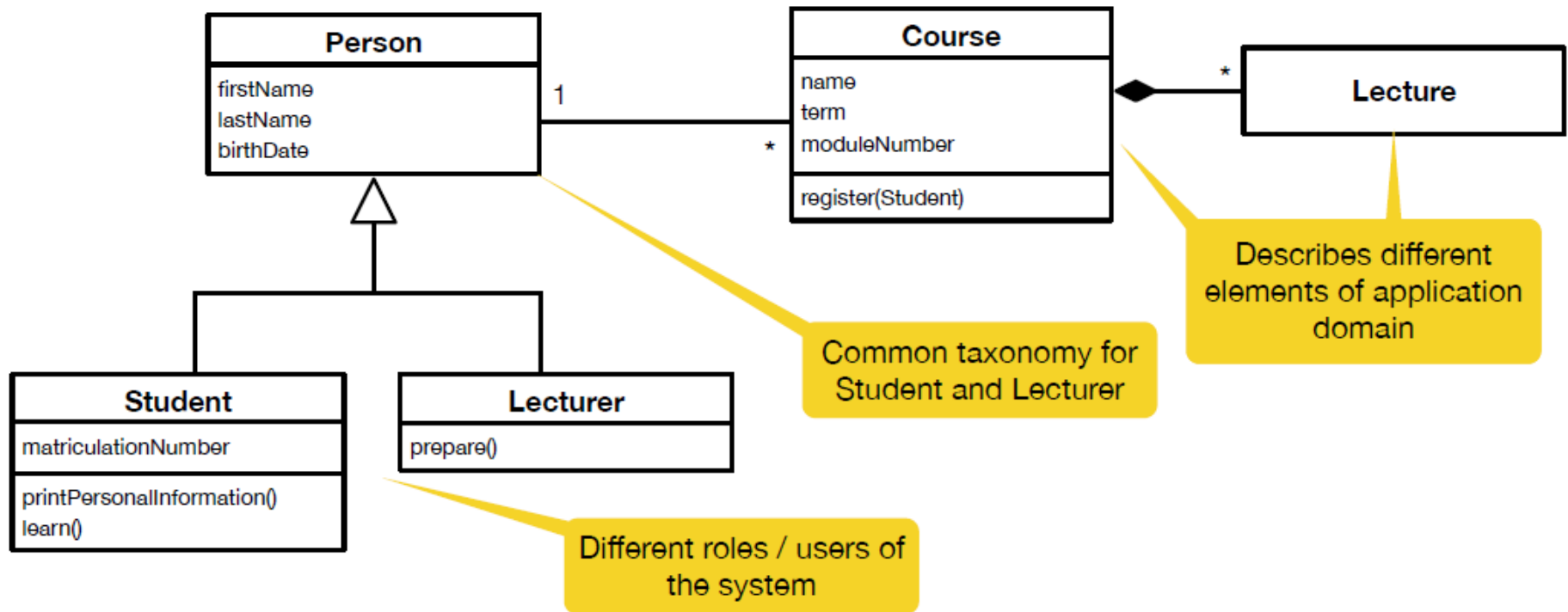
Composition: 

Multiplicities:

one to one: 

one to many: 

many to many: 



# Question?

Consider the following sentence: "A square is a polygon". From an analysis of the sentence, identify the relations between the two classes Square and Polygon that can be inferred from the sentence.

Consider the following sentence: "Students live in hostels". From an analysis of the sentence, identify the relation between the two classes Student and Hostel that can be inferred from the sentence.

# Chapter 5. System Models

5.1. System Modelling

5.2. Functional Models

5.3. Structural models

5.4. Behavioral models





- **Behavioral models** are **models of the dynamic behavior** of a system as it is executing.
- They show **what happens or what is supposed to happen** when a **system responds to a stimulus/Input from its environment**.
- You can think of these stimuli as being of two types:
  - ❑ **Data:** Some data arrives that has to be processed by the system.
    - ❖ These systems are Data-Processing Systems
  - ❑ **Events:** Some event happens that triggers system processing
    - ❖ These systems are event-driven systems
- **Sequence Diagram** can be used **as an Behavioral Model**

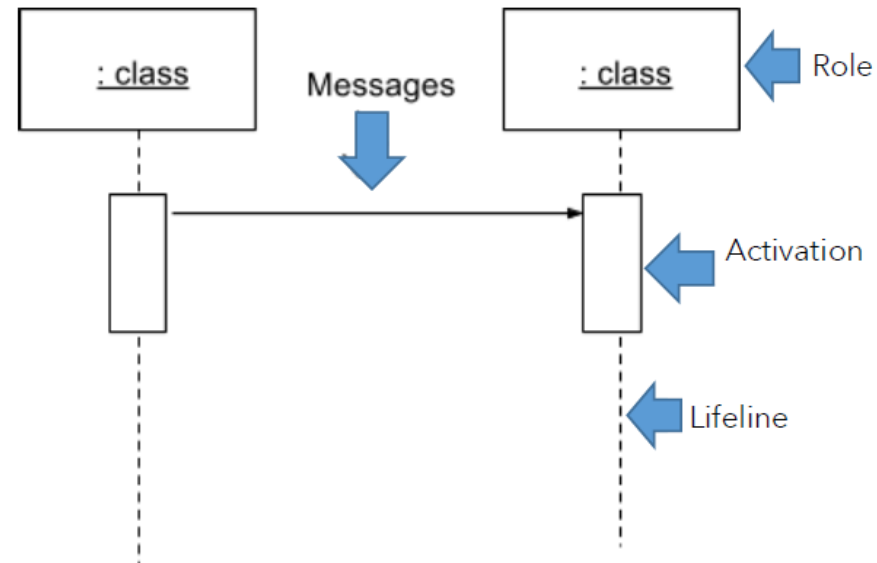
# Sequence Diagrams

- **Sequence diagrams** are part of the UML and are used to model the interactions between the actors and the objects within a system.
- A **sequence diagram** shows the sequence of interactions that take place during a particular use case or use case instance.
- Sequence diagrams are used to show a design team how objects in a program interact with each other to complete tasks.
- In simple terms, **a sequence diagram is like a map of conversations between different people**, with the messages sent from person to person outlined.



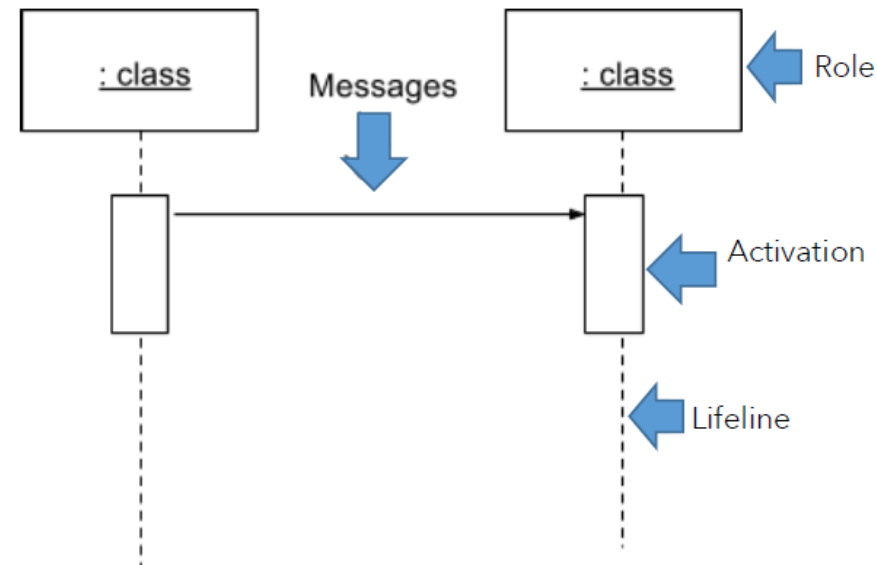
# Sequence Diagrams

- ❑ Boxes are used to represent a role played by an object. The **role** is typically named after the class for the object.
- ❑ “Lifelines,” which are vertical dotted lines, are used in the diagram to indicate an object as time goes by.
- ❑ Solid line arrows are used to show **messages** that are sent from one object to another, or a sender to a receiver.
- ❑ Receivers are at the pointed end of an arrow. A short description of the message is usually included above the arrow.

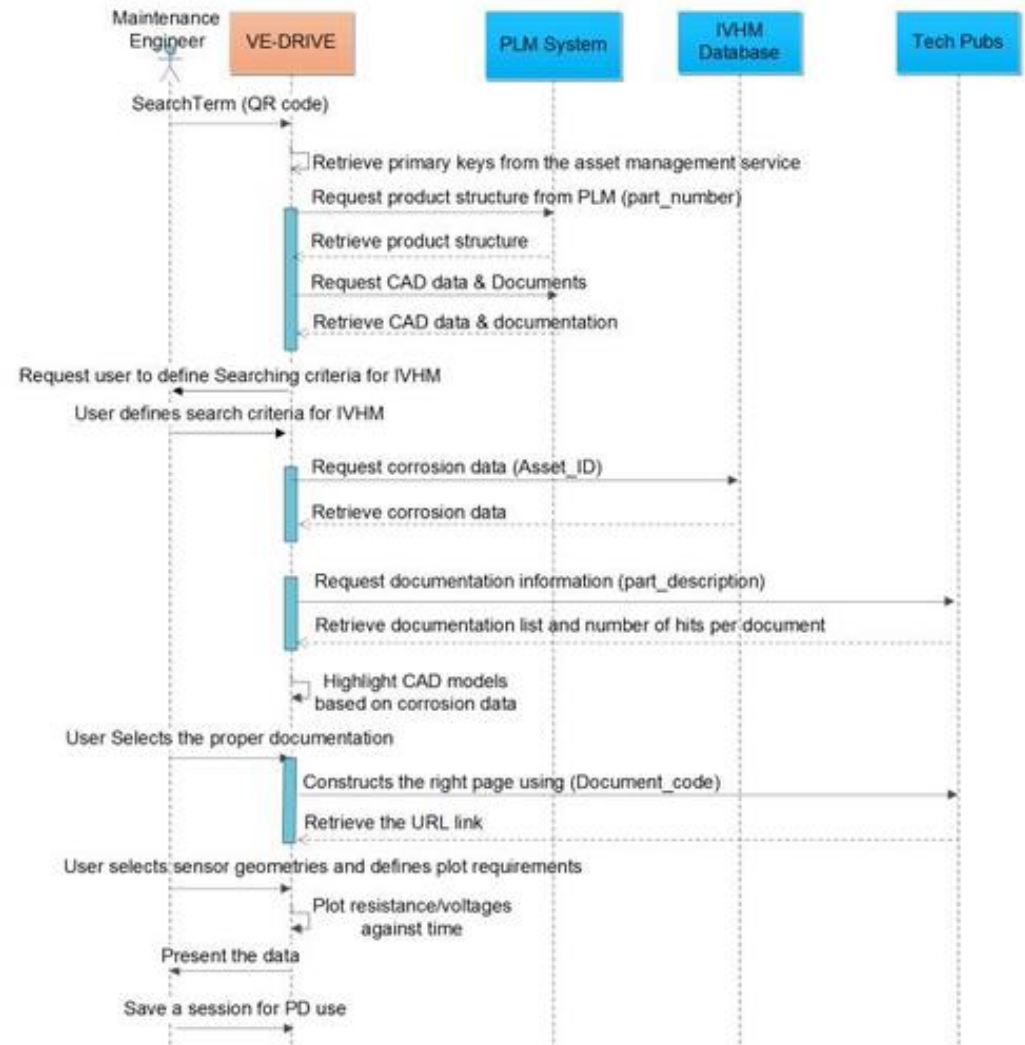
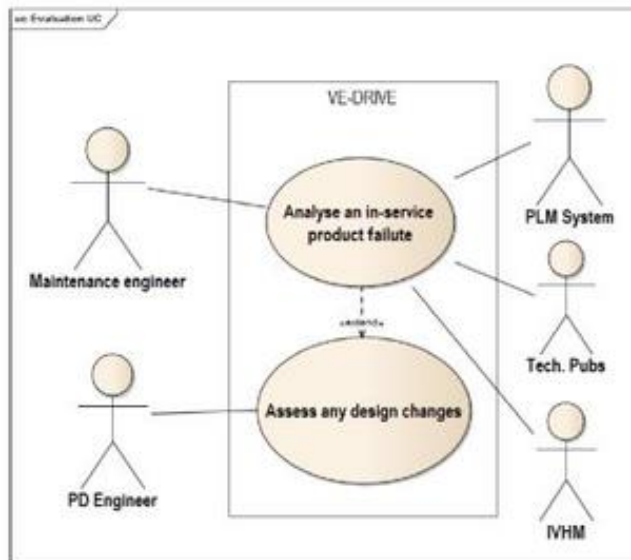


# Sequence Diagrams

- ❑ Dotted line arrows are used to show a return of data and control back to initiating objects. A short description of the return of data or control is usually included above the arrow.
- ❑ Small rectangles along an object's life line denote a method **activation**. You activate an object whenever an object sends, receives, or is waiting for a message.
- ❑ People, or **actors**, may also be included in sequence diagrams if they use or interact with objects. Actors are typically represented with stick figures.

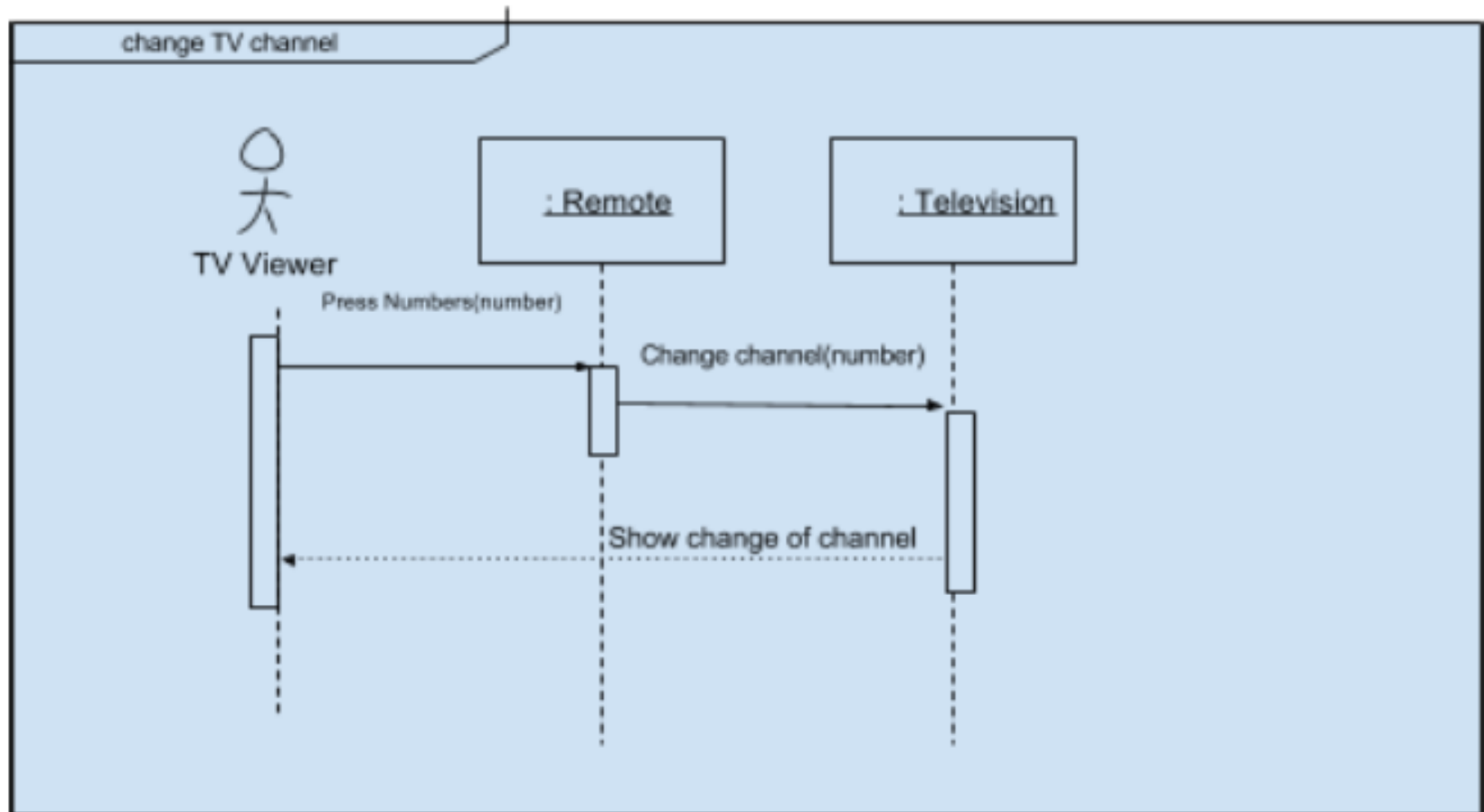


# Sequence Diagram for View patient information



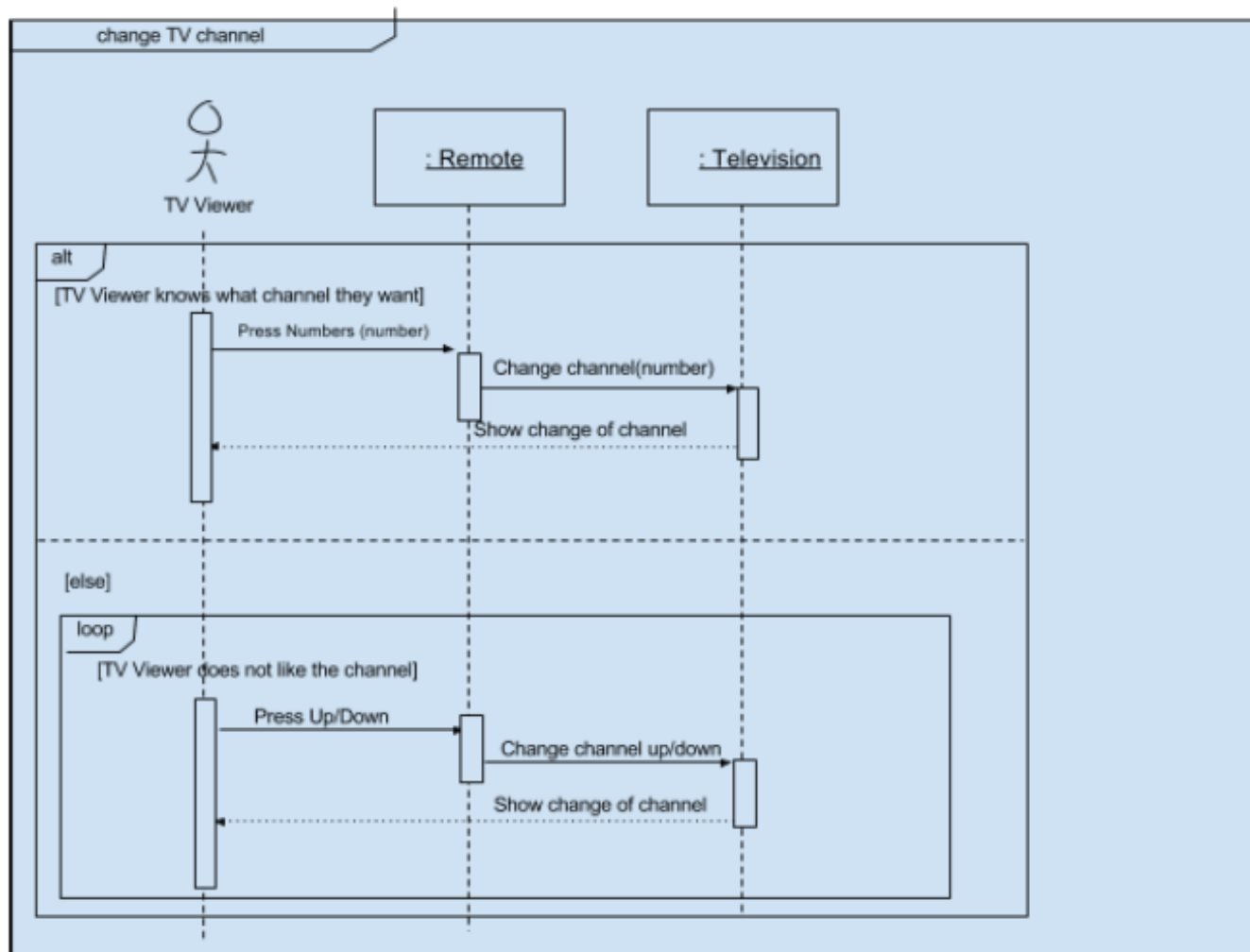
# Sequence Diagram for Transfer Data

- An example of a sequence diagram for changing the channel of your television using a remote control, with all of the elements described



# Sequence Diagram for Transfer Data

- As you design software, your sequence diagrams may get much more complicated. can also be demonstrated in a sequence diagram.



# Question?

Which one of the following is true about a UML Sequence Diagram?

- a. It describes the behavior of many Use Cases.
- b. It describes the behavior of a single Use Case.
- c. It describes the behavior of a single object.
- d. It describes the state behavior of several objects.

Which one of the following can be said of a sequence diagram?

- a. It is used to model the behavior of a single object when many use cases are executed
- b. It is used to model the behavior of several objects when a single use case is executed
- c. It is used to model the behavior of a single object when a single use case is executed
- d. It is used to model the behavior of several objects when many use cases are executed