

# CHAPTER ONE: INTRODUCTION TO SE

---

# OBJECT ORIENTED SOFTWARE ENGINEERING

# Session - 1

# Requirements for this Course

- You may be **proficient in a programming language**, but you have no **or limited experience in analysis or design of a system**
- You want to learn more about the technical aspects of analysis and design of complex software systems

## Objectives of the Course

- Appreciate Software Engineering:
  - Build complex software systems in the context of frequent change
- Understand how to
  - produce a high quality software system within time while dealing with complexity and change
- Acquire technical knowledge (main emphasis)

# OUTLINE

---

- ❖ **The Nature of Software**
- ❖ **What is Software Engineering?.**
- ❖ **Software Engineering and the Engineering Profession**
- ❖ **Stakeholders in Software Engineering**
- ❖ **Software Quality**
- ❖ **Activities Common to Software Projects**
- ❖ **An Overview of Object Oriented Systems Development**
- ❖ **Object Basics**
- ❖ **Object Oriented Systems Development Life Cycle**

# 1.1. The Nature of Software

Software is intangible

- **Hard** to understand development effort

Software is easy to reproduce

- Cost is in its *development*
  - in other engineering products, manufacturing is the costly stage

The industry is labor-intensive

Hard to automate

# The Nature of Software ...

## **Untrained people can hack something together**

Quality problems are hard to notice

## **Software is easy to modify**

People make changes without fully understanding it

## **Software does not ‘wear out’**

It *deteriorates* by having its design changed: erroneously, or in ways that were not anticipated, thus making it complex

## **Conclusions**

- Much software has poor design and is getting worse
- Demand for software is high and rising
- We are in a perpetual ‘software crisis’
- We have to learn to ‘engineer’ software

# Types of Software

## **Custom**

For a specific customer

## **Generic**

Sold on open market

Often called COTS (Commercial Off The Shelf)

Shrink-wrapped

## **Embedded**

Built into hardware

Hard to change

# Types of Software

Differences among **custom**, **generic** and **embedded software**

	Custom	Generic	Embedded
Number of <i>copies</i> in use	low	medium	high
Total <i>processing power</i> devoted to running this type of software	low	high	medium
Worldwide annual <i>development effort</i>	high	medium	low



# Types of Software

## **Real time software**

E.g. control and monitoring systems

Must react immediately

Safety often a concern

## **Data processing software**

Used to run businesses

Accuracy and security of data are key

## 1.2 What is Software Engineering?

The process of solving customers' problems by the systematic development and evolution of large, high-quality software systems within cost, time and other constraints

### Other definitions:

IEEE: (1) the application of a systematic, disciplined, quantifiable approach to the development, operation, maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).

The Canadian Standards Association: The systematic activities involved in the design, implementation and testing of software to optimize its production and support.

# What is Software Engineering?...

## **Solving customers' problems**

This is the *goal* of software engineering

Sometimes the solution is to *buy, not build*

Adding unnecessary features does not help solve the problem

Software engineers must *communicate effectively* to identify and understand the problem

## **Systematic development and evolution**

An engineering process involves applying *well understood techniques* in a organized and *disciplined* way

Many well-accepted practices have been formally standardized

e.g. by the IEEE or ISO

Most development work is *evolution*

# What is Software Engineering?...

## Large, high quality software systems

- Software engineering techniques are needed because large systems *cannot be completely understood* by one person
- Teamwork and co-ordination are required
- Key challenge: Dividing up the work and ensuring that the parts of the system work properly together
- The end-product must be of sufficient quality

## 1.3 Software Engineering and the Engineering Profession

- The term Software Engineering was coined in 1968
  - People began to realize that the principles of engineering should be applied to software development
- **Engineering is a licensed profession**
  - **In order to protect the public**
  - Engineers design artifacts following well accepted practices which involve the application of science, mathematics and economics
  - Ethical practice is also a key tenet of the profession

# Software Engineering and the Engineering Profession

## Ethics in Software Engineering:

### Software engineers shall

- Act consistently with public interest
- Act in the best interests of their clients
- Develop and maintain with the highest standards possible
- Maintain integrity and independence
- Promote an ethical approach in management
- Advance the integrity and reputation of the profession
- Be fair and supportive to colleagues
- Participate in lifelong learning

## 1.4 Stakeholders in Software Engineering

### **1. Users**

Those who use the software

### **2. Customers**

Those who pay for the software

### **3. Software developers**

### **4. Development Managers**

# 1.5 Software Quality...

## **Usability**

Users can learn it and fast and get their job done easily

## **Efficiency**

It doesn't waste resources such as CPU time and memory

## **Reliability**

It does what it is required to do without failing

## **Maintainability**

It can be easily changed

## **Reusability**

Its parts can be used in other projects, so reprogramming is not needed



# 1.6 Activities Common to Software Projects...

## **Requirements and specification**

Includes

**Domain analysis**

**Defining the problem**

**Requirements gathering**

Obtaining input from as many sources as possible

**Requirements analysis**

Organizing the information

**Requirements specification**

Writing detailed instructions about how the software should behave

# Activities Common to Software Projects...

## Design

Deciding how the requirements should be implemented, using the available technology

Includes:

***Systems engineering:*** Deciding what should be in hardware and what in software

***Software architecture:*** Dividing the system into subsystems and deciding how the subsystems will interact

***Detailed design of the internals of a subsystem***

***User interface design***

***Design of databases***

# Activities Common to Software Projects...

## **Modeling**

Creating representations of the domain or the software

- Use case modeling
- Structural modeling
- Dynamic and behavioral modeling

## **Programming**

## **Quality assurance**

Reviews and inspections

Testing

## **Deployment**

## **Managing the process**

## 1.7 Difficulties and Risks in Software Engineering

- Complexity and large numbers of details
- Uncertainty about technology
- Uncertainty about requirements
- Uncertainty about software engineering skills
- Constant change
- Deterioration of software design
- Political risks

# Object-oriented analysis and design

- **Object-oriented analysis and design (OOAD) is a popular technical approach for**
  - analyzing, designing an application, system, or business
  - by applying the **object oriented paradigm** and
  - visual modeling throughout the development life cycles for better communication and product quality.
- **Object-oriented programming (OOP) is a method**
  - based on the concept of “objects”,
  - which are data structures that contain data,
  - in the form of fields,
  - often known as attributes;
  - and code, in the form of procedures,
  - often known as methods.

- **What is OOAD?**- Object-oriented analysis and design (OOAD) is a software engineering approach that models a system as a group of interacting **objects** .
- **Analysis** — understanding, finding and describing concepts in the problem domain.
- **Design** — understanding and defining software solution/objects that represent the analysis concepts and will eventually be implemented in code.
- **OOAD** - A software development approach that emphasizes a logical solution based on objects.
- Software development is dynamic and always undergoing major change.

- ❖ **System development** refers to all activities that go into producing information system solution.
- ❖ System development activities consist of
  - **system analysis,**
  - **modelling,**
  - **design,**
  - **implementation,**
  - **testing and maintenance.**
- ❖ **A software development methodology** → series of processes → can lead to the development of an application.
- ❖ Practices, procedures, and rules used to develop software, totally based on system requirements

# ORTHOGONAL VIEWS OF THE SOFTWARE

❖ Two Approaches,

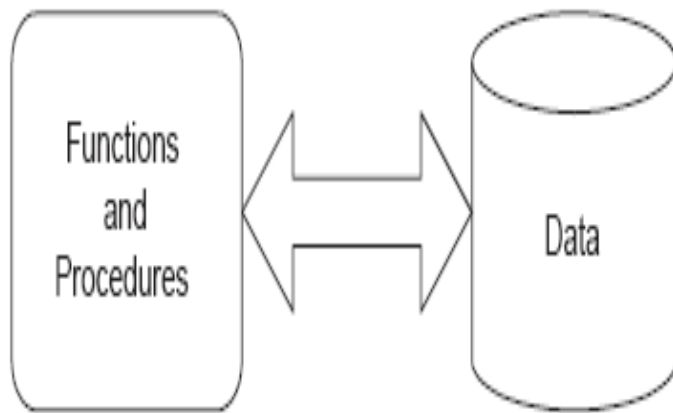
- **Traditional Approach**
- **Objected-Oriented Approach**

❖ **TRADITIONAL APPROACH**

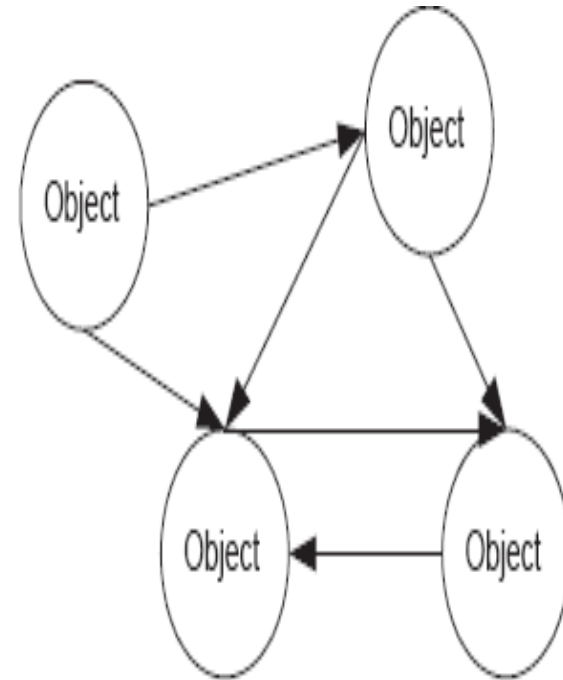
- Collection of **programs or functions**.
- A system that is designed for **performing certain actions**.
- **Algorithms + Data Structures = Programs**.
- Software Development Models (**Waterfall, Spiral, Incremental, etc..**)



# ORTHOGONAL VIEWS OF THE SOFTWARE



A Structured Application



An Object Application

# Difference between Traditional and Object Oriented Approach

<b>TRADITIONAL APPROACH</b>	<b>OBJECT ORIENTED SYSTEM DEVELOPMENT</b>
Collection of procedures(functions)	Combination of data and functionality
Focuses on function and procedures, different styles and methodologies for each step of process	Focuses on object, classes, modules that can be easily replaced, modified and reused.
Moving from one phase to another phase is complex.	Moving from one phase to another phase is easier.
Increases duration of project	decreases duration of project
Increases complexity	Reduces complexity and redundancy

# Object-oriented Approach

- Object Oriented Development is a **new way of thinking about software** based on abstractions that exist in the **real world** as well as in the program.
- Object Oriented Development is a method of design encompassing the process of object-oriented decomposition and a notation for depicting logical and physical as well as static and dynamic models of the system under design.

# OBJECT-ORIENTED APPROACH ...

An approach to the solution of problems in which all computations are performed in the context of **objects**.

- The objects are **instances** of classes, which:
  - are data abstractions
  - contain procedural abstractions that operate on the objects
- A **running program** can be seen as a collection of **objects** collaborating to perform a given task

## ❖ OBJECT ORIENTED APPROACH

- OO development offers a different model from the traditional software → **based on functions and procedures.**
- software is a collection of **discrete object that encapsulate their data as well as the functionality.**
- Each object has **attributes (properties) and method (procedures).**
- software by building self contained modules or objects that can be easily **REPLACED, MODIFIED AND REUSED.**
- Objects **grouped in to classes and object are responsible for itself.**

# EXAMPLES OF OBJECT ORIENTED SYSTEMS

- ❖ In OO system , **“everything is object”**.
- ❖ **A spreadsheet cell, bar chart, title in bar chart, report,**
- ❖ **numbers, arrays, records, fields, files, forms,**  
**an invoice, etc.**
- ❖ **A window object** is responsible for things like **opening, sizing, and closing itself.**
- ❖ **A chart object** is responsible for things like **maintaining data and labels even for drawing itself.**

# BENEFITS OF OBJECT ORIENTATION

- ❖ **Faster development,**
- ❖ **Reusability,**
- ❖ **Increased quality**
- ❖ **modeling the real world and provides us with the stronger equivalence of the real world's entities (objects).**
- ❖ **Raising the level of abstraction to the point where application can be implemented in the same terms as they are described.**

# WHY OBJECTORIENTATION

- ❖ OO Methods enables to develop **set of objects that work** together → software → similar to traditional techniques.
- ❖ It adapts to
  - **Changing requirements**
  - **Easier to maintain**
  - **More robust**
  - **Promote greater design**
  - **Code reuse**



❖ Others

- **Higher level of abstraction**
- **Seamless transition among different phases of software development.**
- **Encouragement of good programming technique.**
- **Promotion of reusability.**

## TRADITIONAL APPROACH

- The traditional view of a computer program is that of a process that has been encoded in a form that can be executed on a computer.
- This view originated from the fact that the first computers were developed mainly to automate a well-defined process (i.e., an algorithm) for numerical computation, and dates back to the first stored-program computers.
- Accordingly, the software creation process was seen as a translation from a description in some 'natural' language to **a sequence of operations that could be executed on a computer.**

# TRADITIONAL APPROACH

- The '**process-centred**' approach used to software development is called **top-down** functional decomposition.
- The first step in such a design was to recognise what the process had to deliver (in terms of input and output of the program), which was followed by decomposition of the process into functional modules.

## TRADITIONAL APPROACH....

- As many would argue, this paradigm is still the best way to introduce the notion of programming to a **beginner**, but as systems became more **complex**, its effectiveness in developing solutions became suspect.
- This change of perspective on part of the software developers happened over a period of time and was fuelled by several factors including the high cost of development and the constant efforts to find uses for software in **new domains(OOSE)**.

# TRADITIONAL APPROACH

- ❖ The traditional approach to software development tends toward **writing a lot of code to do all the things that have to be done.**
- ❖ **Algorithmic Centric Methodology** – only the algorithm that can accomplish the task.
- ❖ **Data-Centric Methodology** - think about the data to build a structure based on the algorithm
- ❖ You are the only active entity and the **code is just basically a lot of building materials.**

# OBJECTS AND CLASSES

# Objects

- The concepts of objects and classes are intrinsically linked with each other and form the foundation of object-oriented paradigm.
- Identity that distinguishes it from other objects in the system.
- State that determines the characteristic properties of an object as well as the values of the properties that the object holds.
- Behavior that represents externally visible activities performed by an object in terms of changes in its state.

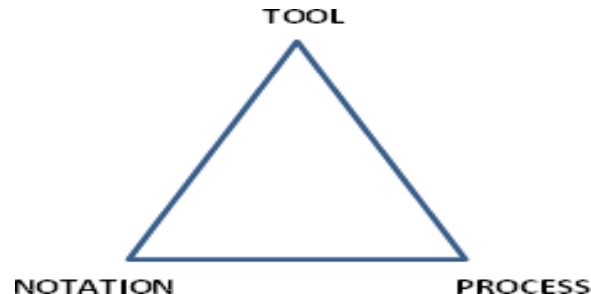
# Class

- A class represents a collection of objects having same characteristic properties that exhibit common behavior.
- Creation of an object as a member of a class is called **instantiation**.
- Thus, object is an instance of a class.
- Example: **Circle** – a class
  - x, to the center
  - a, to denote the radius of the circle
- Some of its operations can be defined as follows:
  - findArea(), method to calculate area
  - findCircumference(), method to calculate circumference



# Object oriented Methodologies

- Many methodologies have been developed for object oriented development.
- A methodology usually includes
  - **Notation** : Graphical representation of classes and their relationships with interactions.
  - **Process** : Suggested set of steps to carry out for transforming requirements into a working system.
  - **Tool** (CASE): Software for drawings and documentation



# SESSION - 2

# OVERVIEW OF UNIFIED APPROACH(UA)

- ❖ The **unified approach (UA)** is a methodology for **software development**.
- ❖ **Booch, Rumbaugh, Jacobson methodologies** gives the best practices, processes and guidelines for OO oriented software development.
- ❖ Combines with the OMT (Object Modeling Technique **in Unified Modelling Language(UML)**).
- ❖ UA utilizes the **unified modeling language (UML)** which is a set of **notations and conventions** used to describe and **model an application**.

## ❖ **Layered Architecture**

- UA uses **layered architecture to develop applications.**
- **Creates object that represent elements to the user through interface or physically stored in database.**
- The layered approach consists of **user interface, business, access layers.**
- This approach **reduces the interdependence of the user interface, database access and business control.**
- **More robust and flexible system.**

# UML – Unified modeling language: Introduction

- UML is a notation that resulted from the unification of OMT (Object Modeling Technique).
- The goal of UML is to provide a standard notation that can be used by all object-oriented methods and to select and integrate the best elements of precursor notations(OMT & OOSE).
  - For example, UML includes the **use case diagrams** introduced by OOSE and uses many features of the OMT **class diagrams**.
- UML also includes new concepts that were not present in other major methods at the time, such as extension mechanisms and a constraint language.
- UML has been designed for a broad range of applications.

# UML – Unified modeling language: Introduction

- Hence, UML provides constructs for a broad range of systems and activities (e.g., distributed systems, analysis, system design, deployment).
- System development focuses on three different models of the system.
- The **functional model**, represented in UML with **use case diagrams**, describes the functionality of the system from the user's point of view.
- The **object model**, represented in UML with **class diagrams**, describes the structure of the system in terms of **objects, attributes, associations, and operations**.
- The **dynamic model**, represented in UML with **interaction diagrams, state diagrams, and activity diagrams**, describes the internal behavior of the system.

# UML – Unified modeling language:

## Introduction

- UML focuses on standard modeling language and not a standard process.
- UML focuses the concept of Booch, Rumbaugh and Jacobson.
- The UML is a standard graphical design for object-oriented graphical design and a medium for presenting important analysis and design concepts.

# UML Diagrams

- Use Case Diagrams
- Class Diagrams
- Package Diagrams
- Interaction Diagrams
  - Sequence
  - Collaboration
- Activity Diagrams
- State Transition Diagrams
- Deployment Diagrams



# USE CASE DIAGRAMS

# Introduction

Use-cases are descriptions of the functionality of a system from a user perspective.

- Depict the behaviour of the system, as it appears to an outside user.
- Describe the functionality and users (actors) of the system.

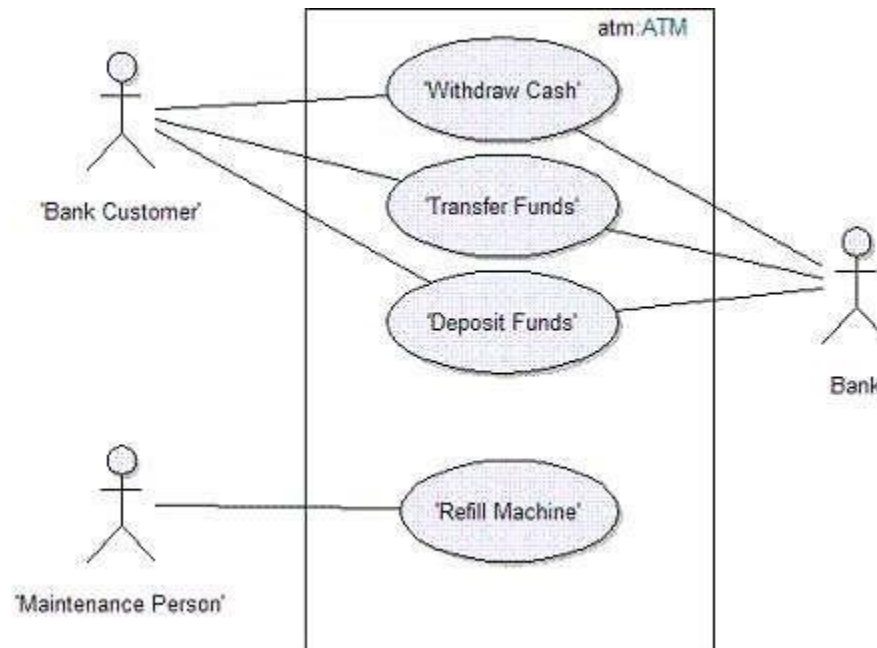
Show the relationships between the actors that use the system, the use cases (functionality) they use, and

relationship between different use cases.

- Document the scope of the system.
- Illustrate the developer's understanding of the user's requirements.

# Use Case Diagrams

- A **use case diagram** at its simplest is a representation of a user's interaction with the system that shows the relationship between the **user** and the different use cases in which the user is involved.



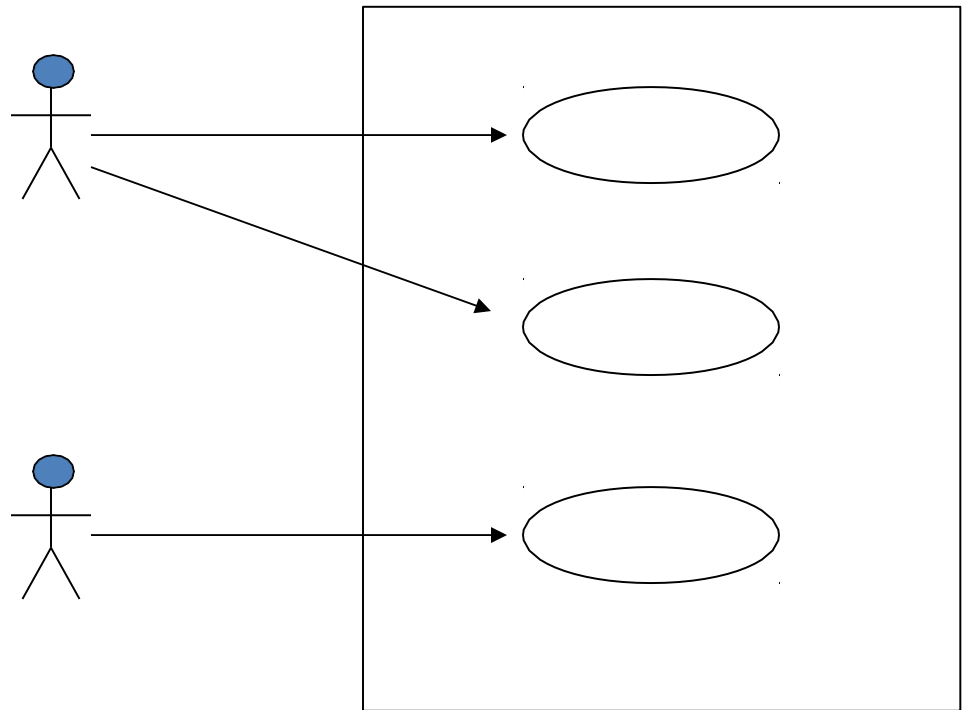
# Components of use case diagram

**1. Actor**

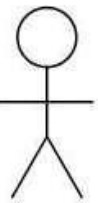
**2. Use case**

**3: System boundary**

**4: Relationship**



# Actor

something  outside the system that interacts with the system.

- An actor is anything that exchanges data with the system.
- An actor can be a **user**, **external hardware**, or another **system**.

# How to Find Actors

- Actors:
  - Supply/use/remove information
  - Use the functionality.
  - Will be interested in any requirement.
  - Will support/maintain the system.
  - The system's external resources.
  - The other systems will need to interact with this one.

# Documenting Actor Characteristics

## **Brief Description:**

- What or who the actor represents?
- Why the actor is needed?
- What interests the actor has in the system?

## **Actor characteristics might influence how the system is developed:**

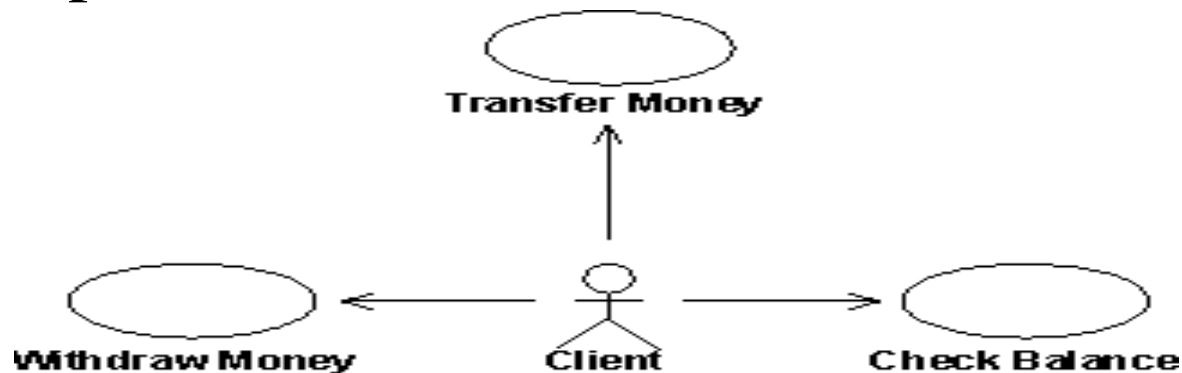
- The actor's scope of responsibility.
- The physical environment in which the actor will be using the system.
- The number of users represented by this actor.

# Use case

- A set of **scenarios** that describes an interaction between a user and a system, including alternatives.



- Example





# How to Find Use Cases

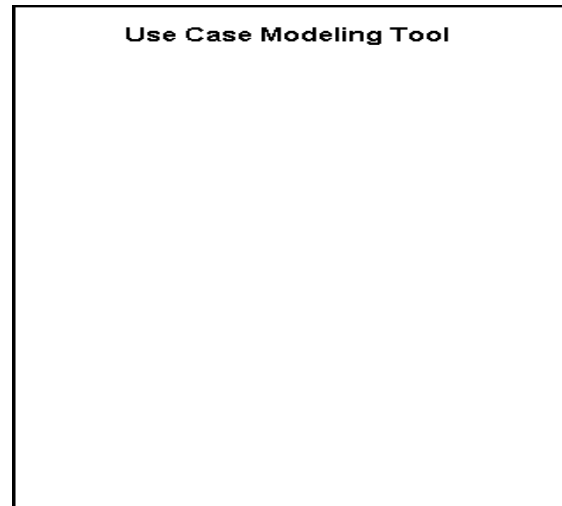
- What are the **system tasks** for each actor you have identified?
- Does the actor need to be informed about certain occurrences in the system?
- Will the actor need to inform the system about sudden, external changes?
- Does the system supply the business with the correct behavior?
- Can all features be performed by the use cases you have identified?
- What use cases will support and maintain the system?
- What information must be modified or created in the system?

# Use cases types:

- System **start** and **stop**.
- Maintenance of the system (add user, ...).
- Maintenance of data stored in the system.
- Functionality needed to modify behavior in the system.

# System Boundary

- It is shown as a rectangle.
- It helps to identify what is external versus internal, and what the responsibilities of the system are.
- The external environment is represented only by actors.



# Relationship

- Relationship is an association between use case and actor.
- There are several Use Case relationships:

- Association



- Extend

<<extend>>

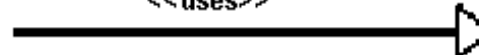


- Generalization



- Uses

<<uses>>



- Include

<<include>>



# Relationship

..... Continue

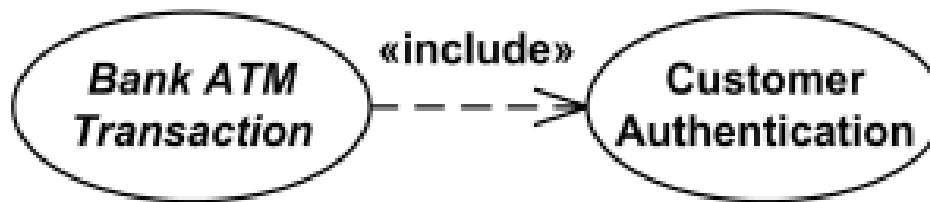
Association: communication between an **actor** and a **use case**; Represented by a solid line. \_\_\_\_\_

- Generalization: relationship between one **general use case** and **a special use case** (used for defining special alternatives)
- Represented by a line with a triangular arrow head toward the **parent use case**.



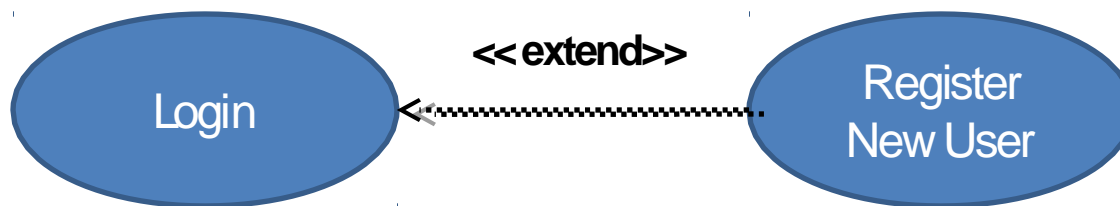
## Include Relationship

- Include relationships insert additional behavior into a base use case
- use cases that are included as parts of other use cases. Enable to factor common behavior.
- The base use case explicitly incorporates the behavior of another use case at a location specified in the base.



# Extend Relationship

- The extended relationship is used to indicate that use case completely consists of the behavior of another use case at one or specific point
- use cases that extend the behavior of other core use cases. Enable to factor variants
- The base use case implicitly incorporates the behavior of another use case at certain points called extension points
- It is shown as a dotted line with an arrow point and labeled <<extend>>



- Include: a dotted line labeled <<include>> beginning at **base use case** and ending with an arrows pointing to the **include use case**. The include relationship occurs when a chunk of behavior is similar across more than one use case. Use “include” in stead of copying the description of that behavior.

<<include>>  
----->

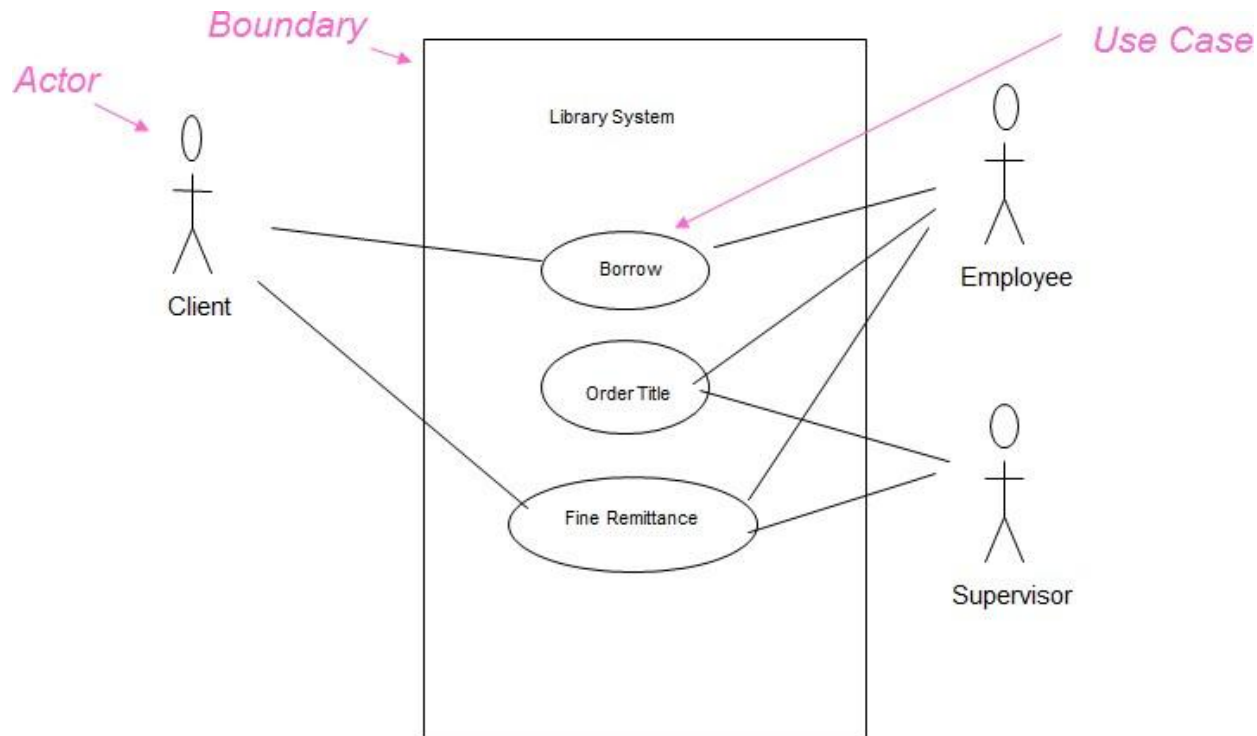
- Extend: a dotted line labeled <<extend>> with an arrow toward the **base case**. The extending use case may **add** behavior to the **base use case**. The base class declares “extension points”.

<<extend>>  
----->

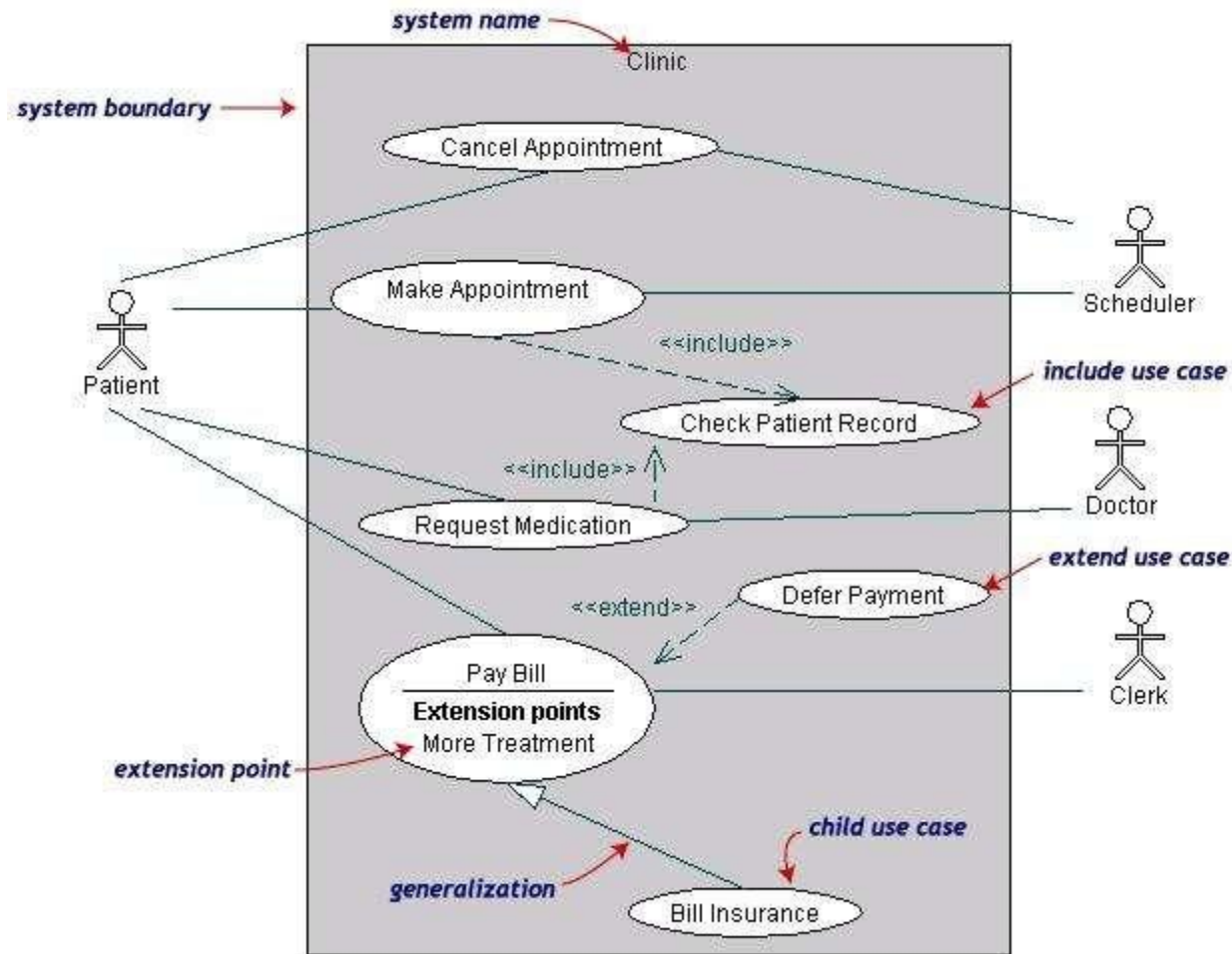


# Example: Library management System

- A generalized description of how a system will be used.
- Provides an overview of the intended functionality of the system



# Use Case Diagrams(cont.)



Continued...

- **Pay Bill** is a parent use case and **Bill Insurance** is the child use case. (generalization)
- Both **Make Appointment** and **Request Medication** include **Check Patient Record** as a subtask.(include)
- The **extension point** is written inside the base case
- **Pay bill**; the extending class **Defer payment** adds the behavior of this extension point. (extend)

# Use case Description

:Each use case may include all or part of the following

- Title or Reference Name - meaningful name of the UC
- Author/Date - the author and creation date
- Modification/Date - last modification and its date
- Purpose - specifies the goal to be achieved
- Overview - short description of the processes
- Cross References - requirements references
- Actors - agents participating
- Pre Conditions - must be true to allow execution
- Post Conditions - will be set when completes normally
- Normal flow of events - regular flow of activities
- Alternative flow of events - other flow of activities
- Exceptional flow of events - unusual situations
- Implementation issues - foreseen implementation problems

# Example- Money Withdraw

**Use Case:** Withdraw Money

**Author:** Group3

**Date:** 20-03-2020

**Purpose:** To withdraw some cash from user's bank account

**Overview:** *The use case starts when the customer inserts his card into the system. **The system requests the user PIN.** The system validates the PIN. If the validation succeeded, the customer can choose the withdraw operation else alternative 1 – validation failure is executed. **The customer enters the amount of cash to withdraw.** The system checks the amount of cash in the user account, its credit limit. If the withdraw amount in the range between the current amount + credit limit the system dispense the cash and prints a withdraw receipt, else alternative 2 – amount exceeded is executed.*

**Cross References:** R1.1, R1.2, R7

# Example- Money Withdraw ---continue

**Actors:** Customer

## **Pre Condition:**

- The ATM must be in a state ready to accept transactions
- The ATM must have at least some cash on hand that it can dispense
- The ATM must have enough paper to print a receipt for at least one transaction

## **Post Condition:**

- The current amount of cash in the user account is the amount before the withdraw minus the withdraw amount
- A receipt was printed on the withdraw amount
- The withdraw transaction was audit in the System logfile

# Example- Money Withdraw ---continue

## Typical course of events

Actor Actions	System Actions
1. Begins when a Customer arrives at ATM	
2. Customer inserts a Credit card into ATM	3. System verifies the customer ID and status
5. Customer chooses “Withdraw” operation	4. System asks for an operation type
7. Customer enters the cash amount	6. System asks for the withdraw amount
	8. System checks if withdraw amount is legal
	9. System dispenses the cash
	10. System deduces the withdraw amount from account
	11. System prints a receipt
13. Customer takes the cash and the receipt	12. System ejects the cash card

# Example- Money Withdraw ---continue

- **Alternative flow of events:**

- **Step 3:** Customer authorization failed. Display an error message, cancel the transaction and eject the card.
- **Step 8:** Customer has insufficient funds in its account. Display an error message, and go to step 6.
- **Step 8:** Customer exceeds its legal amount. Display an error message, and go to step 6.

- **Exceptional flow of events:**

- Power failure in the process of the transaction before step 9, cancel the transaction and eject the card



# SESSION - 3

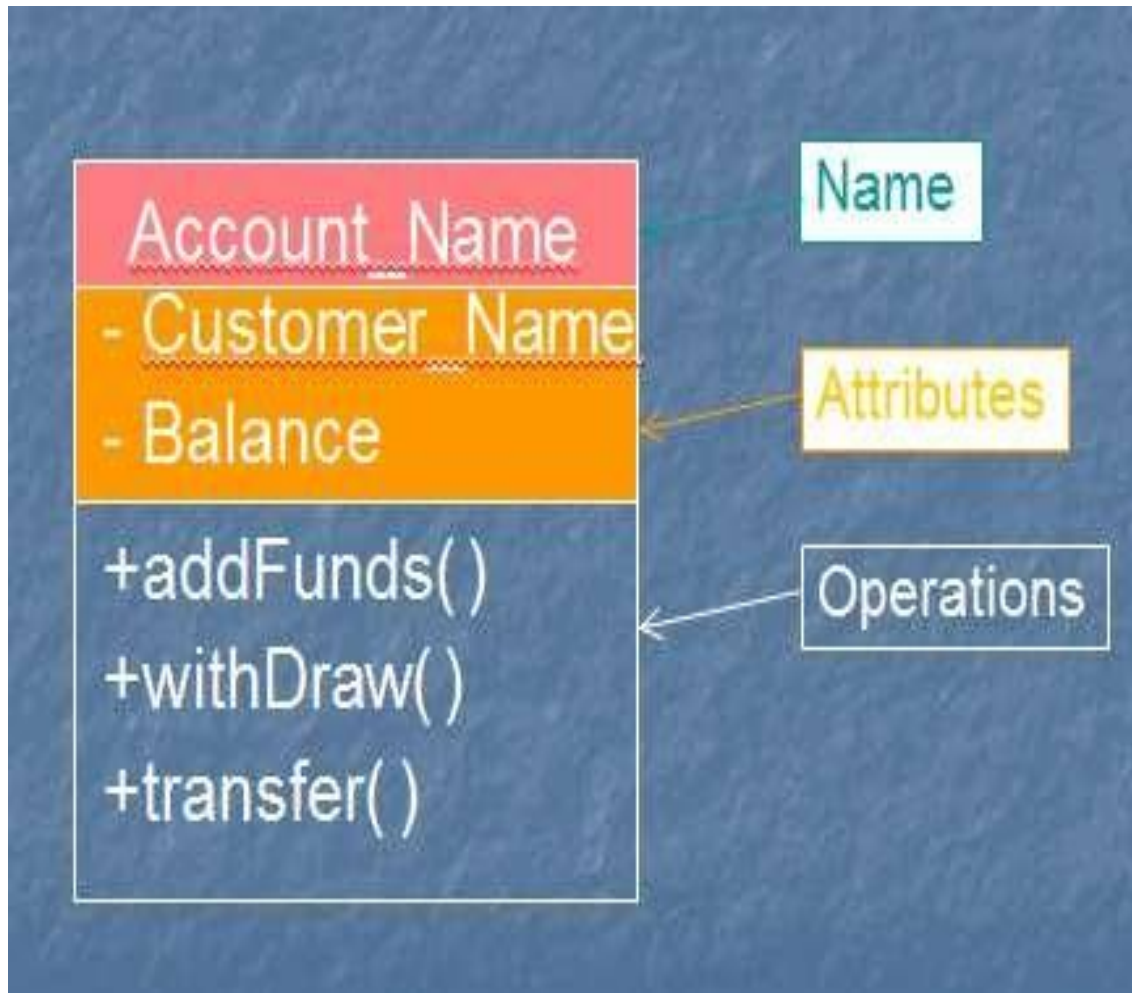
# Class diagram

- Used for describing **structure and behavior** in the use cases
- Provides a conceptual model of the system in terms of entities and their relationships
- Used for requirement capture, end-user interaction
- Detailed class diagrams are used for developers

# Class representation

- Each class is represented by a rectangle subdivided into three compartments
  - Name
  - Attributes
  - Operations
- Modifiers are used to indicate visibility of attributes and operations.
  - '+' is used to denote *Public* visibility (everyone)
  - '#' is used to denote *Protected* visibility (friends and derived)
  - '-' is used to denote *Private* visibility (no one)
- By default, attributes are hidden and operations are visible.

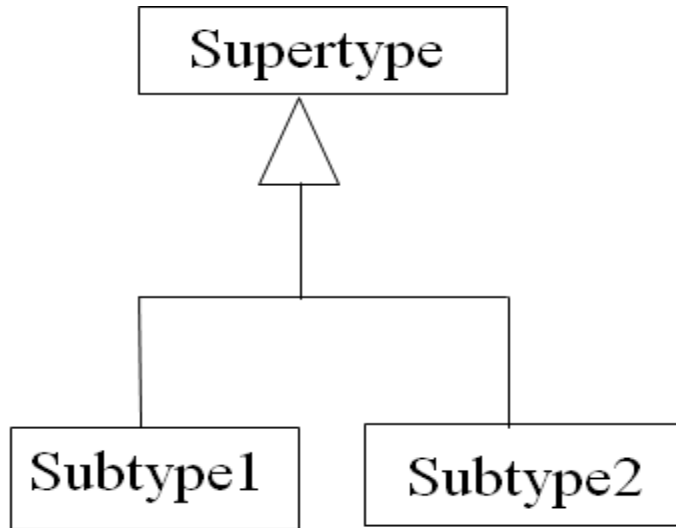
# An example of Class



# OO Relationships

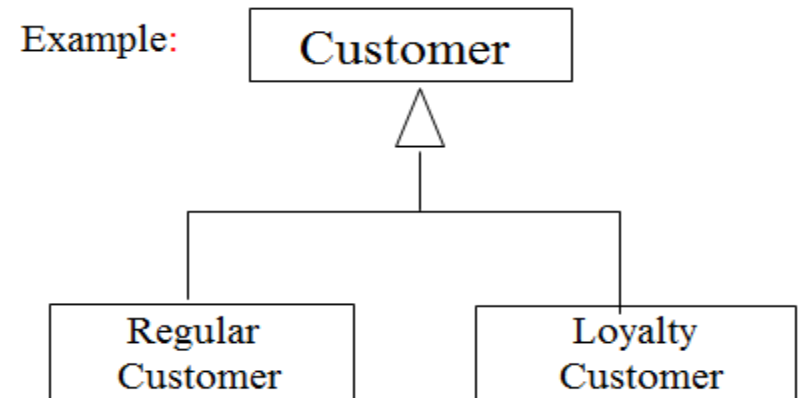
- There are two kinds of Relationships
  - Generalization (parent-child relationship)
  - Association (student enrolls in course)
- Associations can be further classified as
  - Aggregation
  - Composition

# OO Relationships: Generalization

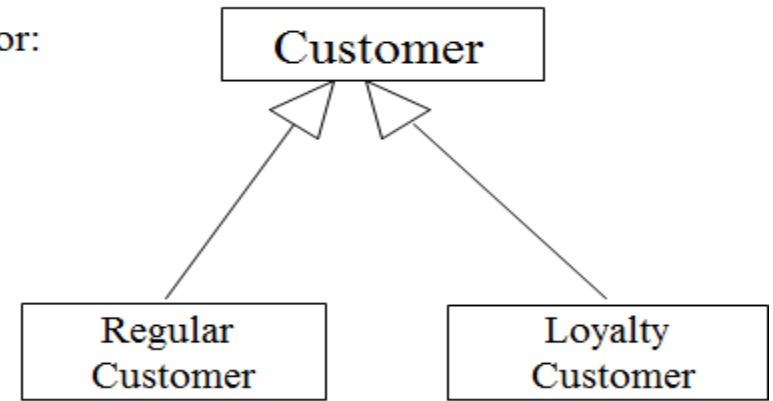


- Generalization expresses a parent/child relationship among related classes.

- Used for abstracting details in several layers



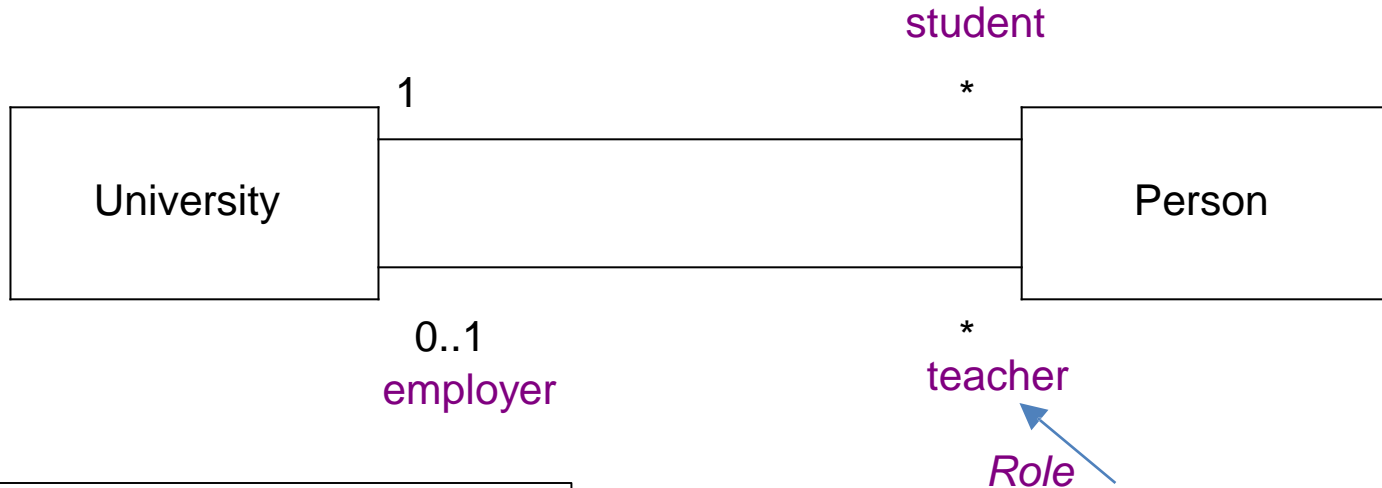
or:



# OO Relationships: Association

- Represent relationship between instances of classes
  - Student enrolls in a course
  - Courses have students
  - Courses have exams
  - Etc.
- Association has two ends
  - Role names (e.g. enrolls)
  - Multiplicity (e.g. One course can have many students)
  - Navigability (unidirectional, bidirectional)

# Association: Multiplicity and Roles



Multiplicity	
Symbol	Meaning
1	One and only one
0..1	Zero or one
M..N	From M to N (natural language)
N	From zero to any positive integer
0..*	From zero to any positive integer
1..*	From one to any positive integer

**Role**

*“A given university groups many people; some act as students, others as teachers. A given student belongs to a single university; a given teacher may or may not be working for the university at a particular time.”*



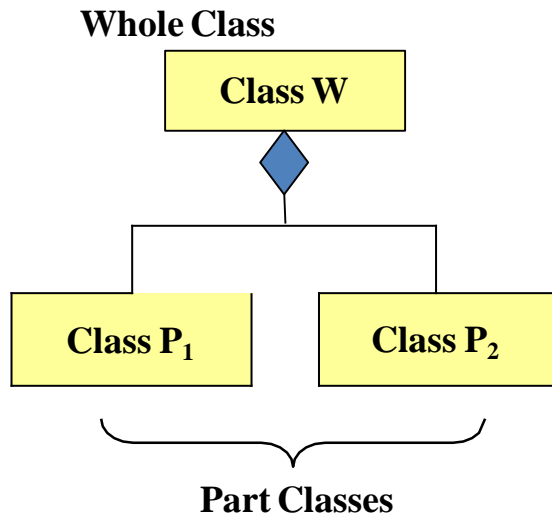
# Association: Model to Implementation



```
Class Student {  
    Course enrolls[4];  
}
```

```
Class Course {  
    Student have[];  
}
```

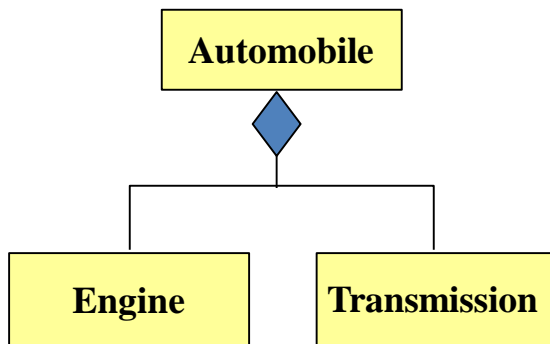
# OO Relationships: Composition



**Composition:** expresses a relationship among instances of related classes. It is a specific **kind of Whole-Part** relationship.

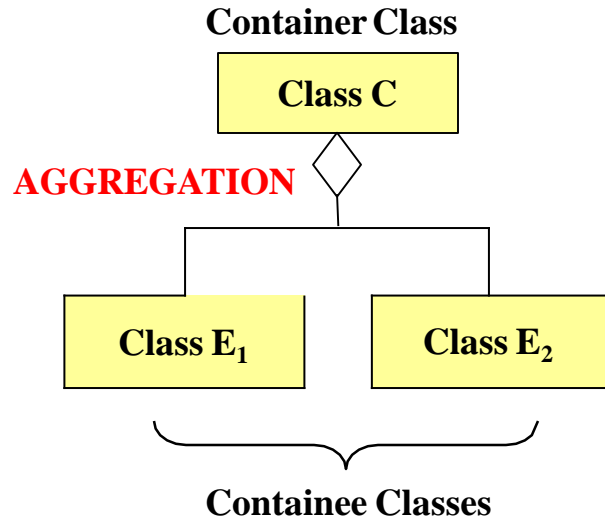
It expresses a relationship where an instance of the Whole-class has the responsibility to **create and initialize instances** of each Part-class.

## Example



It may also be used to express a relationship where instances of the Part-classes have **privileged access or visibility** to certain attributes and/or behaviors defined by the Whole-class.

# OO Relationships: Aggregation

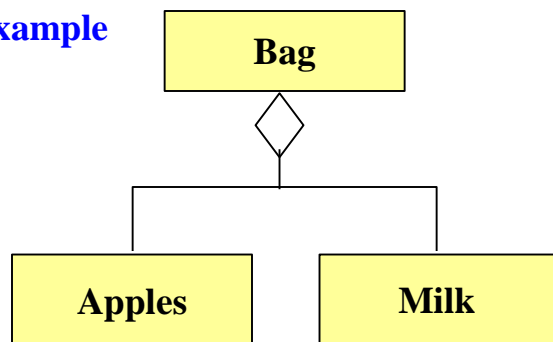


**Aggregation:** expresses a relationship among instances of related classes. It is a specific kind of Container-Containee relationship.

It expresses a relationship where an instance of the Container-class has the responsibility to hold and maintain instances of each Containee-class that have been created outside the Container-class.

Aggregation should be used to express a more informal relationship than composition expresses. That is, it is an appropriate relationship where the Container and its Containees

Example



Aggregation is appropriate when Container and Containees have no special access privileges to each other.

# Aggregation vs. Composition

- **Composition** is really a strong form of **aggregation**
  - components have only one owner
  - components cannot exist independent of their owner
  - components live or die with their ownere.g. Each car has an engine that can not be shared with other cars.
- **Aggregations** may form "part of" the aggregate, but may not be essential to it. They may also exist independent of the aggregate.  
e.g. Apples may exist independent of the bag.

# Session 3

Interaction Diagram -  
Sequence diagram  
Collaboration Diagram

# Interaction Diagram

- From the name *Interaction* it is clear that the diagram is used to describe some type of **interactions** among the different elements in the model.
- So this interaction is a part of dynamic behavior of the system.
- This interactive behavior is represented in UML by two diagrams known as ***Sequence diagram*** and ***Collaboration diagram***.
- The basic purposes of both the diagrams are **similar**.

# Interaction Diagram

Sequence diagram emphasizes on **time sequence** of messages and collaboration diagram emphasizes on the **structural organization** of the objects that send and receive messages.

- The purposes of interaction diagram can be describes as:
  - To capture **dynamic behavior** of a system.
  - To describe the **message flow** in the system.
  - To describe **structural organization** of the objects.
  - To describe **interaction among** objects.



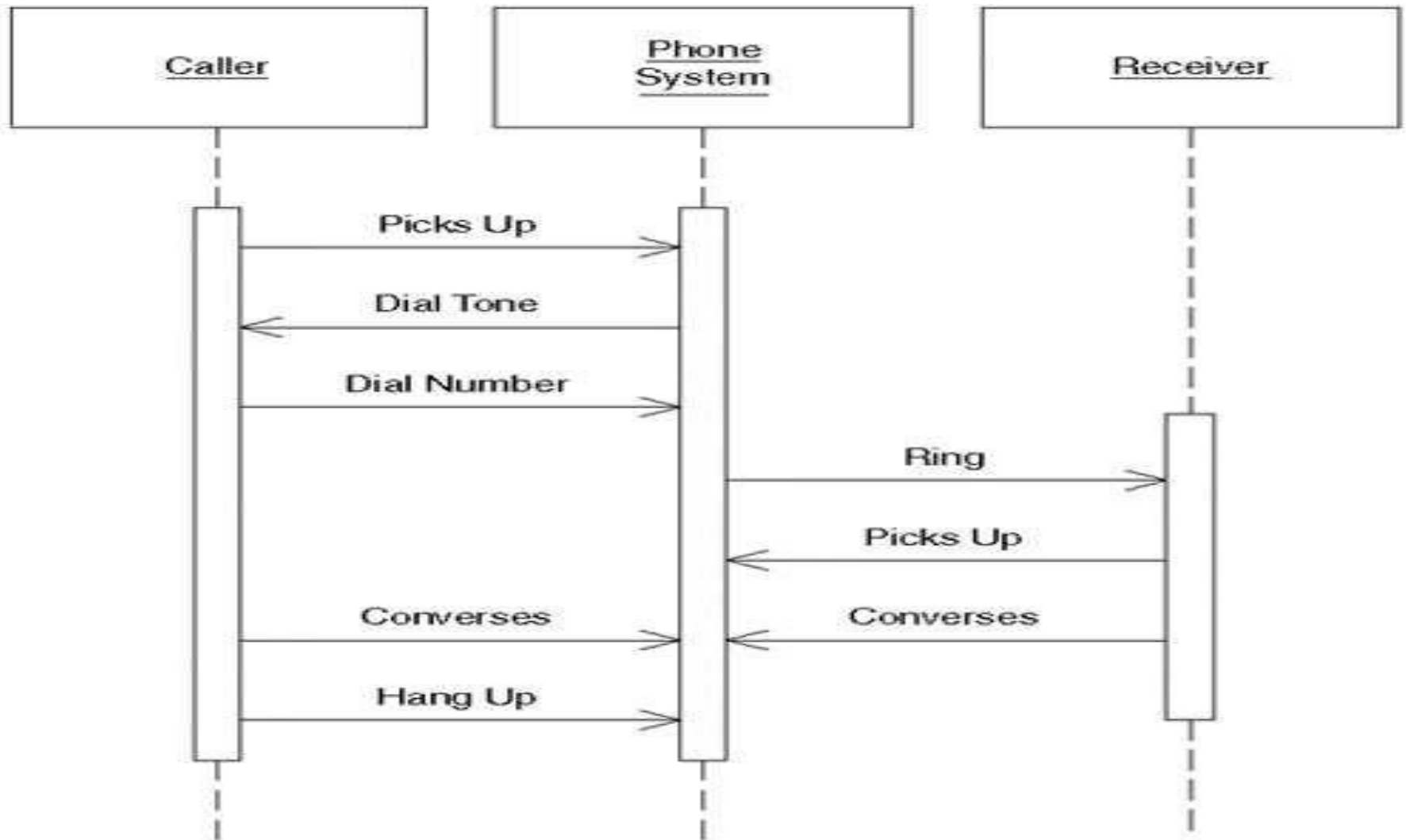
The following factors are to be identified clearly before drawing the interaction diagram:

- Objects taking part in the interaction.
- Message flows among the objects.
- The sequence in which the messages are flowing.
- Object organization.

Example :

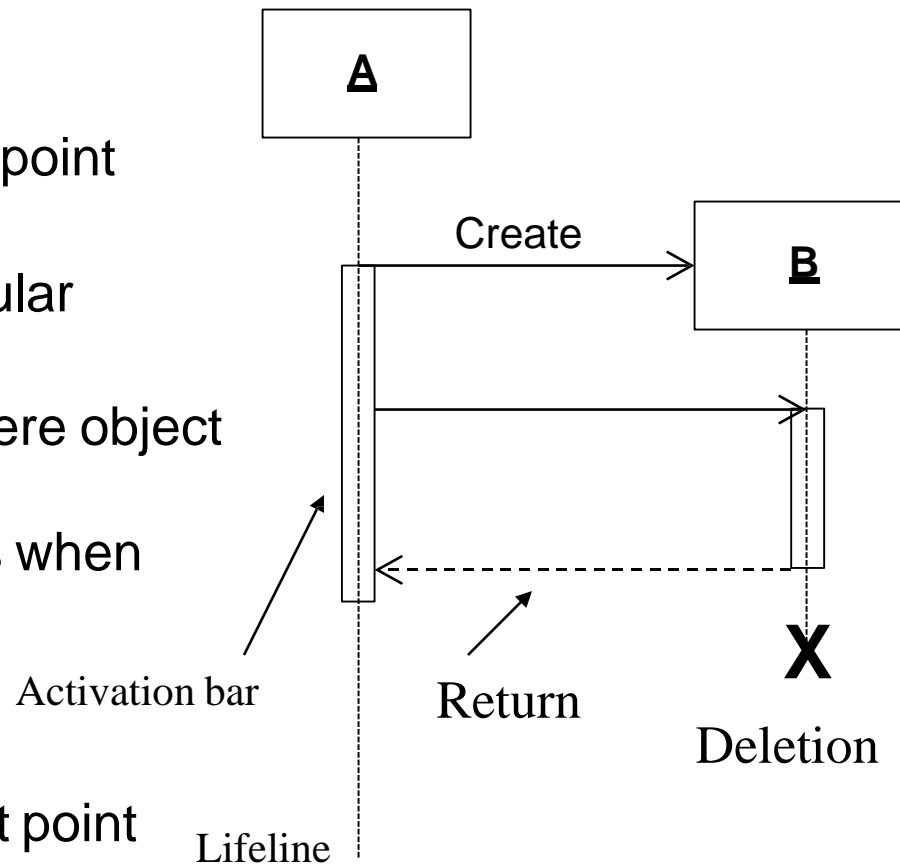
Making a phone call.

# Sequence Diagram(Telephone call)



# Sequence Diagrams – Object Life Spans

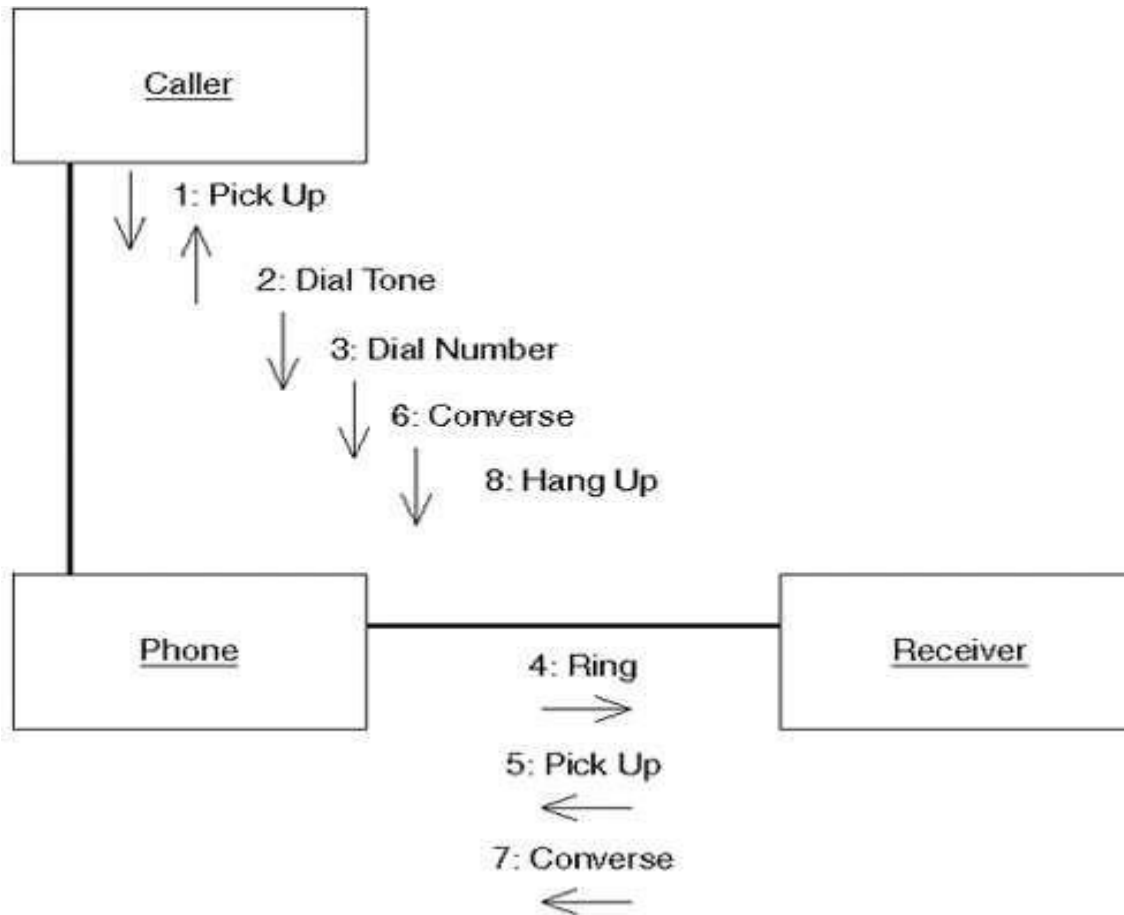
- **Creation**
  - Create message
  - Object life starts at that point
- **Activation**
  - Symbolized by rectangular stripes
  - Place on the lifeline where object is activated.
  - Rectangle also denotes when object is deactivated.
- **Deletion**
  - Placing an 'X' on lifeline
  - Object's life ends at that point



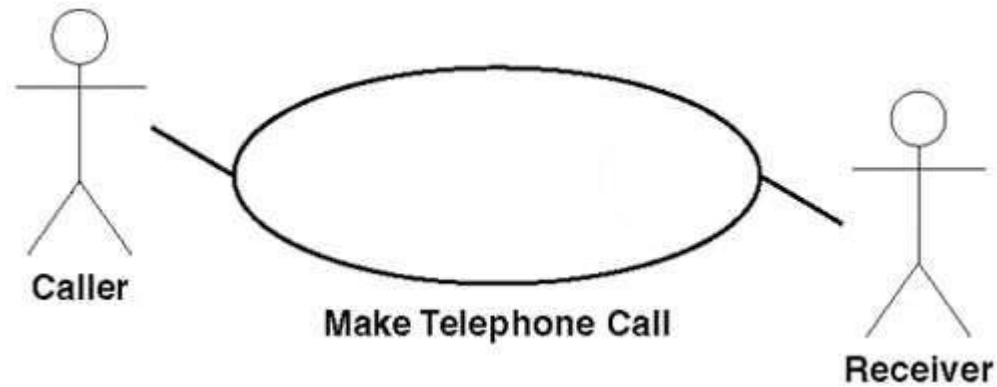
# Collaboration Diagram

- A *collaboration diagram* also shows the passing of messages between objects, but focuses on the objects and messages and their **order** instead of the time sequence.
- The sequence of interactions and the concurrent threads are identified using **sequence numbers**.
- A collaboration diagram shows an interaction the relationships among the objects playing the different roles.
- The *UML Specification* suggests that collaboration diagrams are better for **real-time specifications** and for **complex scenarios** than sequence diagrams.

# Collaboration Diagram(Telephone call)



# Use Case diagram – Telephone Call



# Activity Diagram

# Activity Diagram

- *Activity Diagram* – a special kind of State chart diagram, but shows the flow from activity to activity
- *Activity state* – *non-atomic* execution, ultimately result in some action; a composite made up of other activity/action states; can be represented by other activity diagrams
- *Action state* – *atomic* execution, results in a change in state of the system or the return of a value (i.e., calling another operation, sending a signal, creating or destroying an object, or some computation); non-decomposable



continued...

- Activity diagram is basically a flow chart to represent the flow from one activity to another activity.
- The activity can be described as an operation of the system.
- So the control flow is drawn from one operation to another.
- This flow can be sequential, branched or concurrent.
- Activity diagrams deal with all types of flow control by using different elements like fork, join etc.
- The basic purposes of activity diagrams are similar to other four diagrams.

continued...

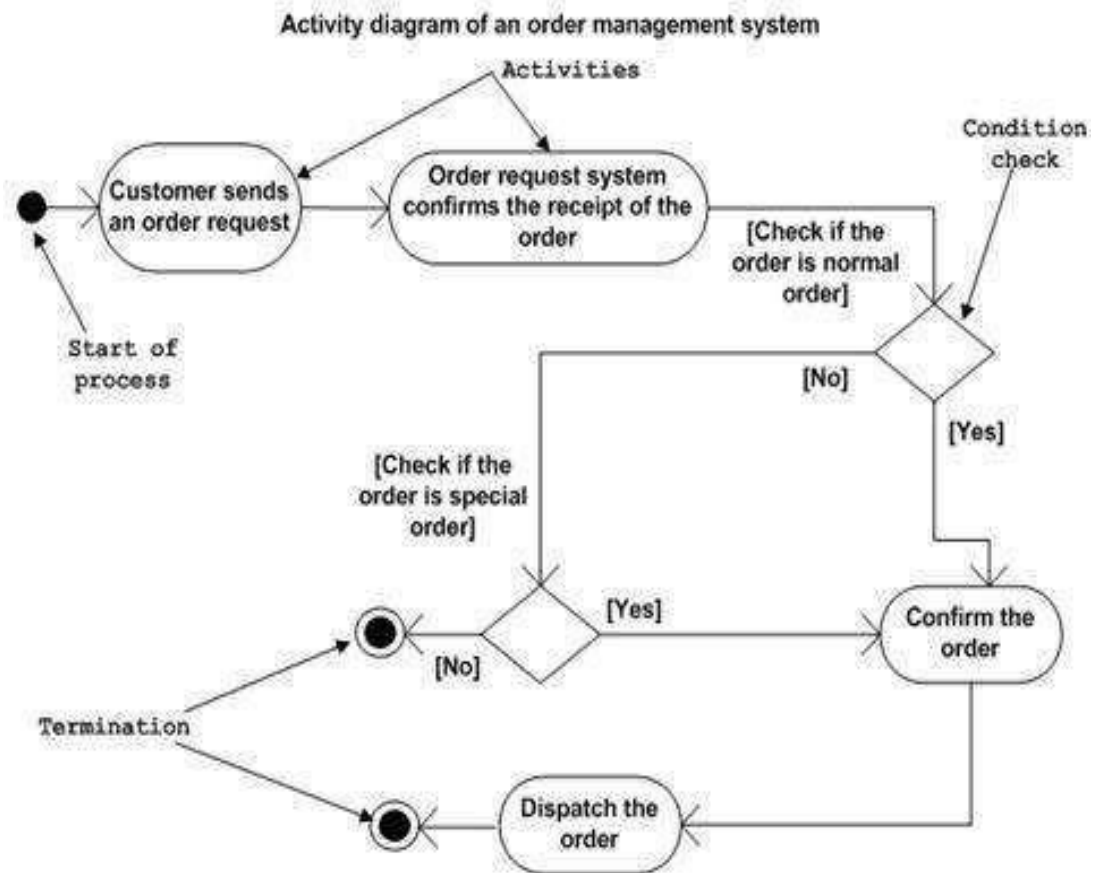
- It captures the dynamic behavior of the system.
- Other four diagrams are used to show the message flow from one object to another but activity diagram is used to show message flow from one activity to another.
- Activity diagrams are not only used for visualizing dynamic nature of a system but they are also used to construct the executable system by using forward and reverse engineering techniques.
- The only missing thing in activity diagram is the message part.

# How to draw Activity Diagram?

- Activity diagrams are mainly used as a flow chart consists of activities performed by the system.
- So before drawing an activity diagram we should identify the following elements:
  - Activities
  - Association
  - Conditions
  - Constraints

# Example : order management system

- Send order by the customer
- Receipt of the order
- Confirm order
- Dispatch order



# Uses of Activity Diagrams

- Modeling work flow by using activities.
- Modeling business requirements.
- High level understanding of the system's functionalities.
- Investigate business requirements at a later stage.

# Example : ATM System

