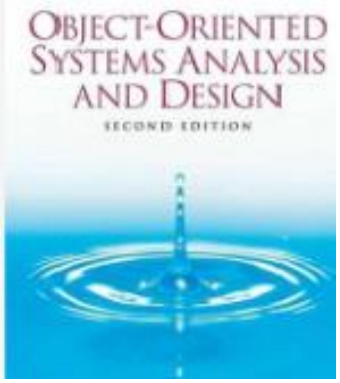
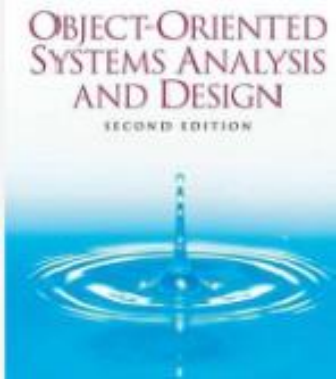
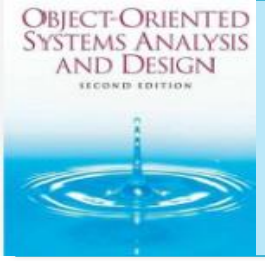


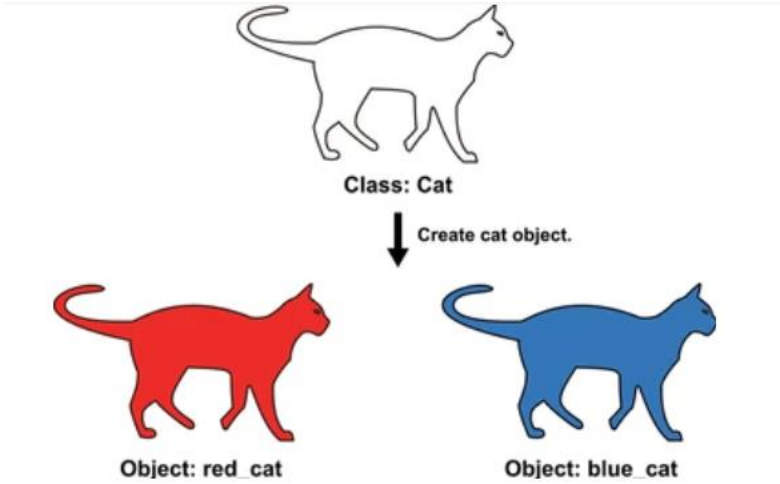


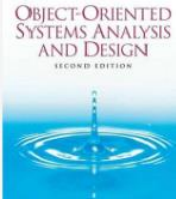
Object Oriented System Analysis and Design (OOSAD) INSY3063



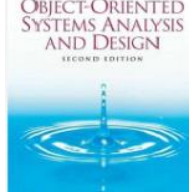
Chapter 2

Basic Object orientation Concepts, Object Oriented System Development Life Cycle Methodologies and Modeling with UML

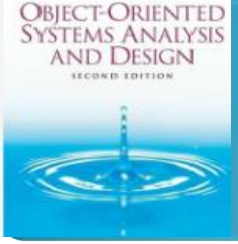




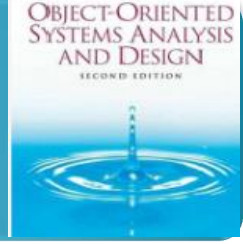
CHAPTER 2:



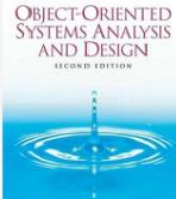
- 2.1.Basic Concepts of Object orientation
- 2.2. Object-oriented System Development life cycle(OOSDLC)
  - 2.2.1. Phases of SDLC
  - 2.2.2. Basic SDLC Methodologies
- 2.3. Unified Process
- 2.4. Unified Modeling Language
- 2.5. UML- Unified Modeling Language
  - Definition
  - UML Structure(Building Blocks)
  - UML- Architectural Views
  - UML –Diagram types (Structure, Behavior and Interaction)
  - UML Diagram Examples
- 2.6. A Modern Look at Modeling



Objectives



- To revise different basic object oriented concepts
- Describe the various benefits provided by OOP;
- Explain the programming applications of OOP.
- Differentiate stages of object oriented system development process.
- To differentiate common object oriented methodologies
- To describe unified process
- To define Unified Modeling Language
- To identify Unified process from other software process models
- To describe Unified Modeling Language (UML)
- To identify different UML diagrams
- To Understand OO modeling concepts in system/software development using Unified Modeling Language



UNDERSTANDING THE BASICS: OBJECT ORIENTED CONCEPTS

- **BASIC oo concepts**
  - **Abstraction, Encapsulation, and information hiding.**
  - **Inheritance, Association and Aggregation**
  - **Collaboration Persistence**
  - **Coupling and Cohesion**
  - **Polymorphism**
  - **Interfaces and Components**
  - **Patterns**

- Object Oriented Technology(OOT) is built up on a sound engineering foundation, the conceptual framework of which is the– "**Object Model**".
- The Object model encompasses the major elements (**essential**, a model without these is not **object oriented**).
  - **Abstraction**
  - **Encapsulation**
  - **Modularity**
  - **Hierarchy**

OO concepts from structured point of view /Objects and Classes

OBJECT-ORIENTED SYSTEMS ANALYSIS AND DESIGN SECOND EDITION

- Object Model Minor Elements(useful but not essential)
  - **Typing**
  - **Concurrency**
  - **Persistence**
- Other important concepts include
  - **Objects, Classes, Polymorphism, Message, Attributes, Methods**

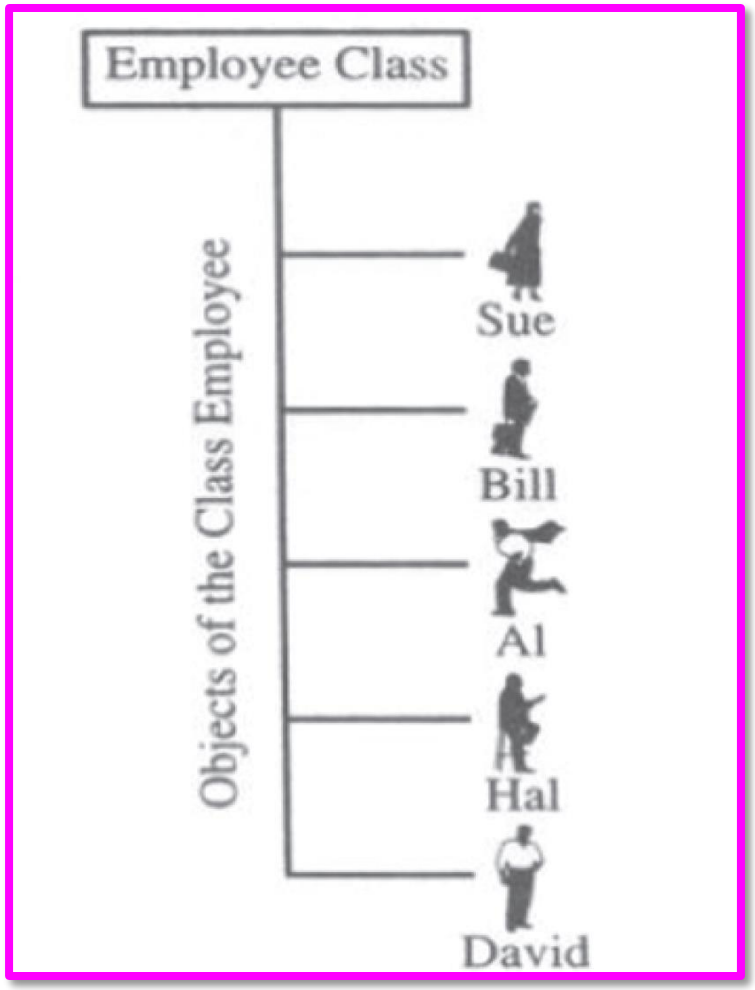
➢ OO concepts in the language of structured terminology

- **Class** is a software abstraction of an object, effectively, a template from which **objects** are created.
  - The definition of a class describes the layout, including both the data and the functionality, of the objects to be created from it.
- **Object** is a software construct that mirrors a concept in the real-world, e.g., **a person, place, thing, event, concept, screen, or report**
  - If a class can be thought of as a **table**, an object can be thought of as a **record** occurrence

OO concepts from structured point of view /Objects and Classes

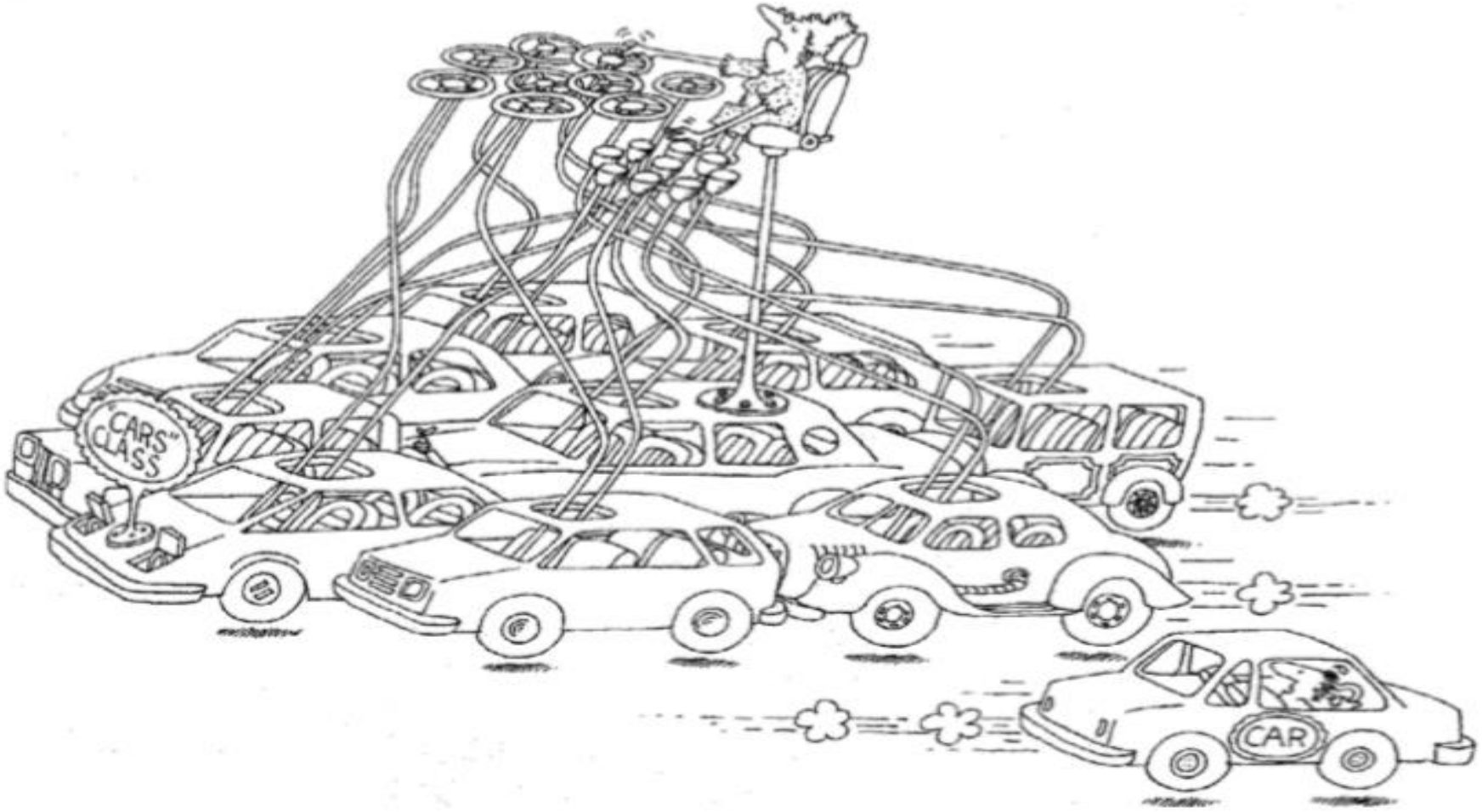
OBJECT-ORIENTED SYSTEMS ANALYSIS AND DESIGN
SECOND EDITION

It is a blueprint of data and member functions.



Employee class example

Class



Interface and Class Implementation example - java

```
interface Vehicle {
//all are the abstract methods.
   void changeGear(int a);
   void speedUp(int a);
   void applyBrakes(int a);
}
```

```
class Bicycle implements Vehicle
{
   int speed;
   int gear;
    // to change gear
   @Override
   public void changeGear(int newGear){
      gear = newGear;
   }
   // to increase speed
   @Override
   public void speedUp(int increment){
      speed = speed + increment;
   }
   // to decrease speed
   @Override
   public void applyBrakes(int decrement){
      speed = speed - decrement;
   }
   public void printStates() {
       System.out.println("speed: " + speed
           + " gear: " + gear);
   }
}
```

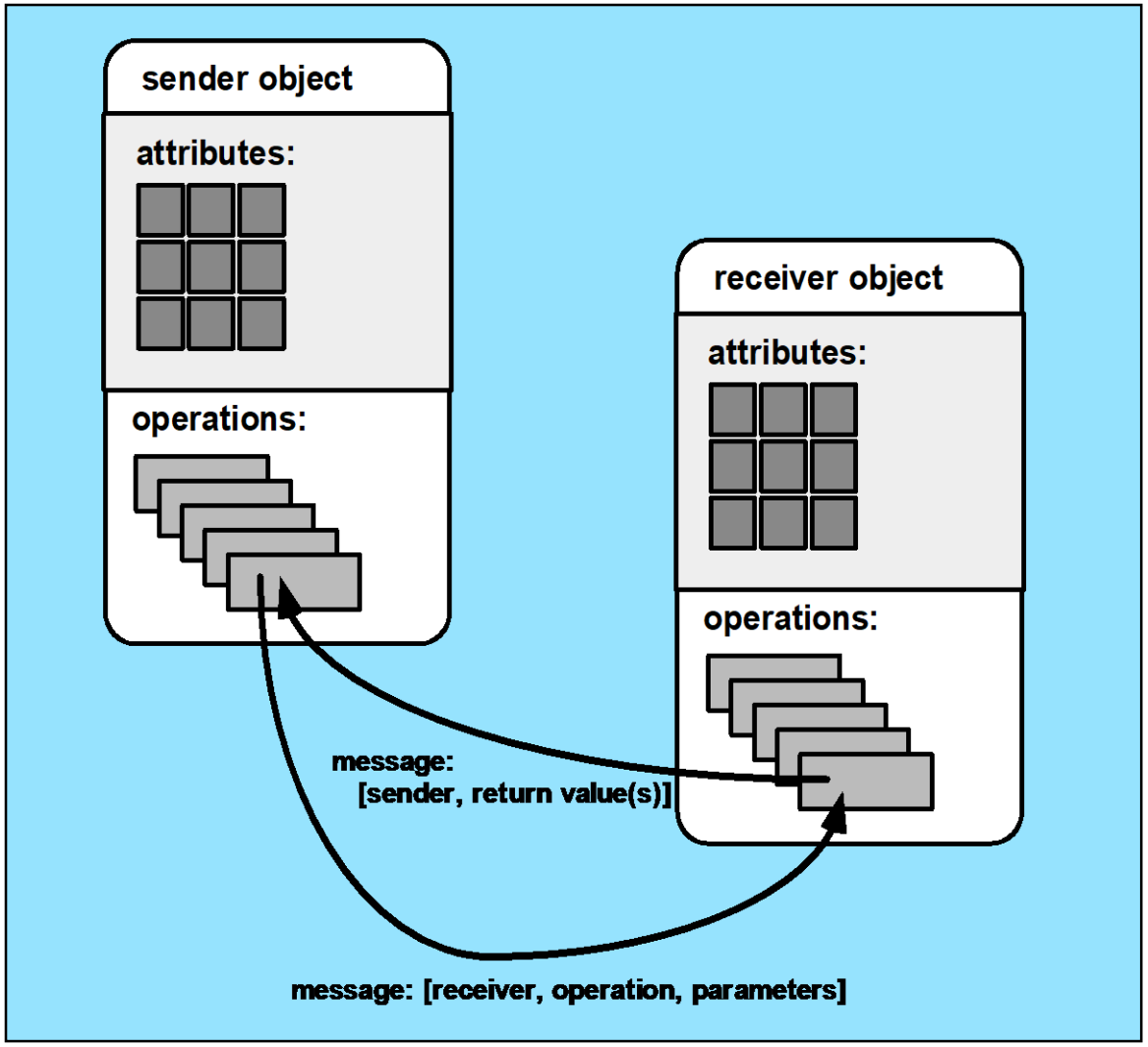
OO concepts in the language of structured terminology

- **Attribute**. An attribute is equivalent to a data element in a record. Attribute can be thought of as a local variable in the context of a programming language.
- **Method**. A method can be thought of as either a function or procedure.
  - Methods access and modify the attributes of an object.
  - Better yet, methods can do a whole bunch of stuffs that has nothing to do with attributes. (e.g. function that register students to a course.)
- An executable procedure that is encapsulated in a class and is designed to operate on one or more data attributes that are defined as part of the class.
- A method is invoked via **message passing**.

OO concepts from structured point of view /Objects and Classes

OBJECT-ORIENTED SYSTEMS ANALYSIS AND DESIGN SECOND EDITION

- A method is invoked
- via **message passing**.



What is an abstraction?

- **Abstraction** (from the Latin ***abs***, meaning ***away from*** and ***trahere*** , meaning ***to draw***) is the process of taking away or removing characteristics from something in order to reduce it to a set of essential characteristics.
- In **object-oriented programming**, abstraction is one of three central principles (along with **encapsulation** and **inheritance**).
- **Abstraction** is the classification of phenomena into concepts

What is an abstraction?

- **Abstraction meanings**
  - **Object** -> something in the world
  - **Class** -> collection of objects
  - **Superclass** -> collection of subclasses
  - **Operation** -> methods(objects have methods, classes have operations)
  - **Attributes** and relationships -> instance variables (local variables).

What is an abstraction?

- **Modeling** constructs an **abstraction** of the system focusing only on the **relevant aspects** and **ignoring all othe**r.
- **Modeling** enables developers to deal with complexity by reasoning with simpler concepts.
- In object-oriented programming theory, **abstraction** involves the facility **to define objects that represent abstract** "**actors**" that can perform work, report on and change their state, and "**communicate**" with other objects in the system.

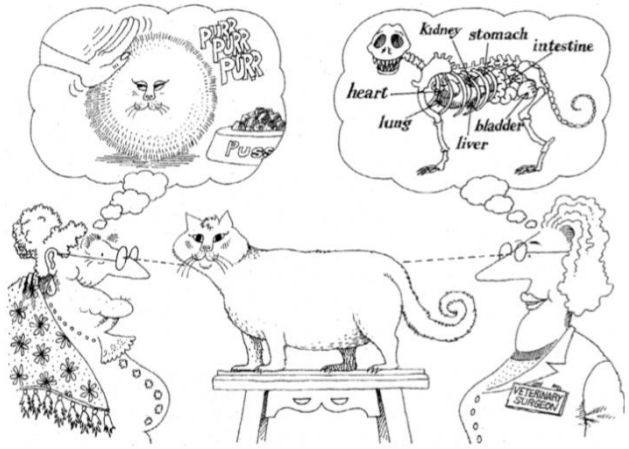
What is an abstraction?

- **Abstraction** denotes extracting essential characteristics of an object that distinguish it from all other kinds of objects
- Same real-world object may have **different abstractions depending on the problem domain** (e.g. **Abstraction of a person for a University System and a Police Investigation system are different**).
- **Abstraction** is one of the fundamental ways that we as humans cope with complexity.

OO concepts from structured point of view /Objects and Classes

OBJECT-ORIENTED SYSTEMS ANALYSIS AND DESIGN
SECOND EDITION

- **Abstraction** focuses on the **essential characteristics of some object, relative to the perspective of the viewer. Try to give other examples to express similar idea about abstraction**



- Abstraction of a cat Sensor

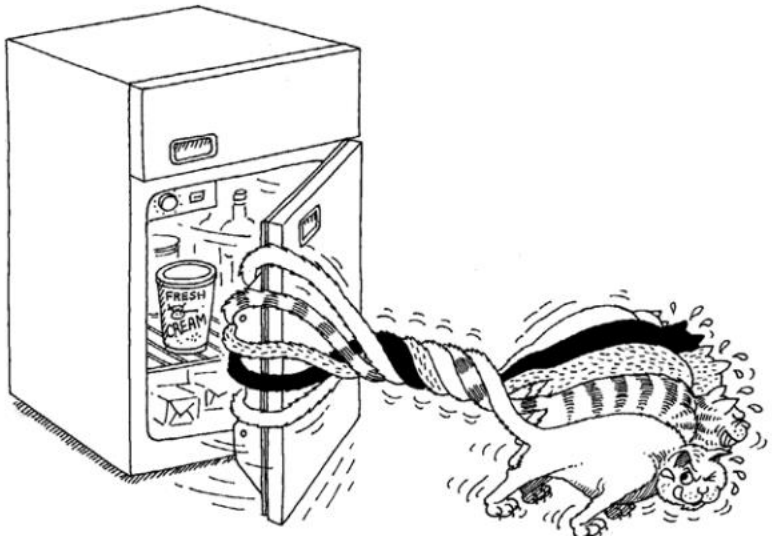
Abstraction: Temperature Sensor
Important Characteristics: temperature location
Responsibilities: report current temperature calibrate

Abstraction of a Temperature Sensor

- **From** the most to the least useful, **abstractions** include the following:

1. **Entity abstraction =>** An object that represents a useful model of a problem domain or solution domain entity
2. **Action abstraction =>** An object that provides a generalized set of operations, all of which perform the same kind of function
3. **Virtual machine abstraction =>** An object that groups operations that are all used by some superior level of control, or operations that all use some junior-level set of operations.
4. **Coincidental abstraction =>** An object that packages a set of operations that have no relation to each other.

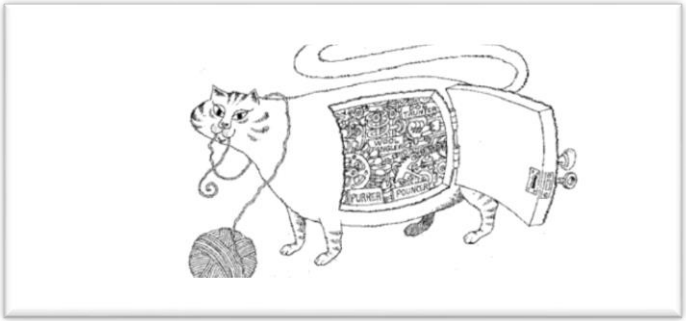
- the picture below shows **how objects cooperate with one another define the boundaries of each abstraction** and thus the **responsibilities and protocol of each object**.



Objects collaborate with other objects to achieve some behavior.

Encapsulation

- The term **encapsulation** refers to **the hiding of state details**, but extending the concept of ***data type*** from earlier programming languages to **associate *behavior*** most strongly with the data, and **standardizing** the way that different data types interact, is the beginning of **abstraction.**



Encapsulation hides the details of the implementation of an object

- **Abstraction** and **encapsulation** are complementary concepts: Abstraction ***focuses on the observable behavior of an object***, whereas encapsulation ***focuses on the implementation that gives rise to this behavior***.

Encapsulation

- It is hiding the inner workings of object's operations from the outside world and from other objects
  - Example : a Monitor and CPU
- Details can be hidden in classes
- For **abstraction** to work, implementations must be **encapsulated**

Encapsulation

- This gives rise to ***information hiding*:**
- Programmers do not need to know all the details of a class
- Encapsulation deals with the issue of how you intend to modularize the features of a system.
- In the object-oriented world, you modularize systems into classes, which, in turn, are modularized into methods and attributes.

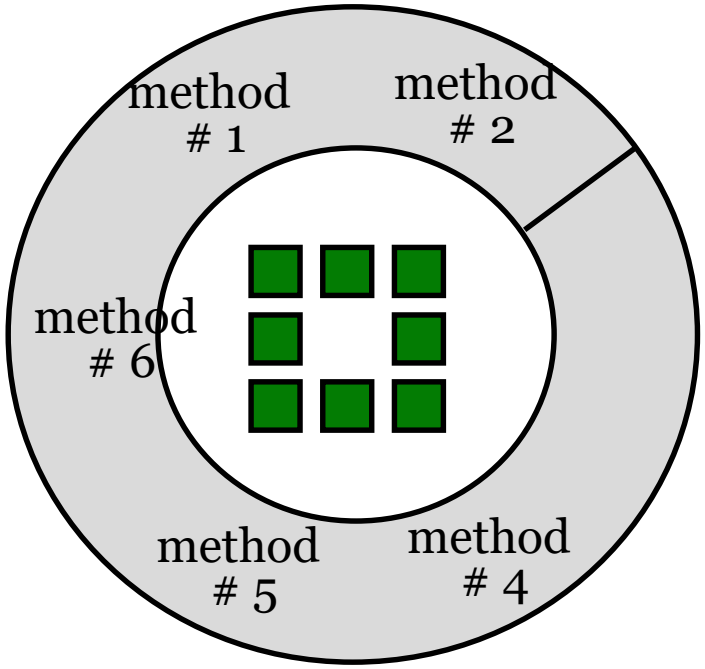
Encapsulation

- **Encapsulation** is the act of painting the box "**black**". Don't want to show how something is defined.
- Segments in the system compartmentalize their subsystems, which provide structure and behavior. Segments are black boxes to the other segments
- **Encapsulation** hides the details of the implementation of an object.

Encapsulation

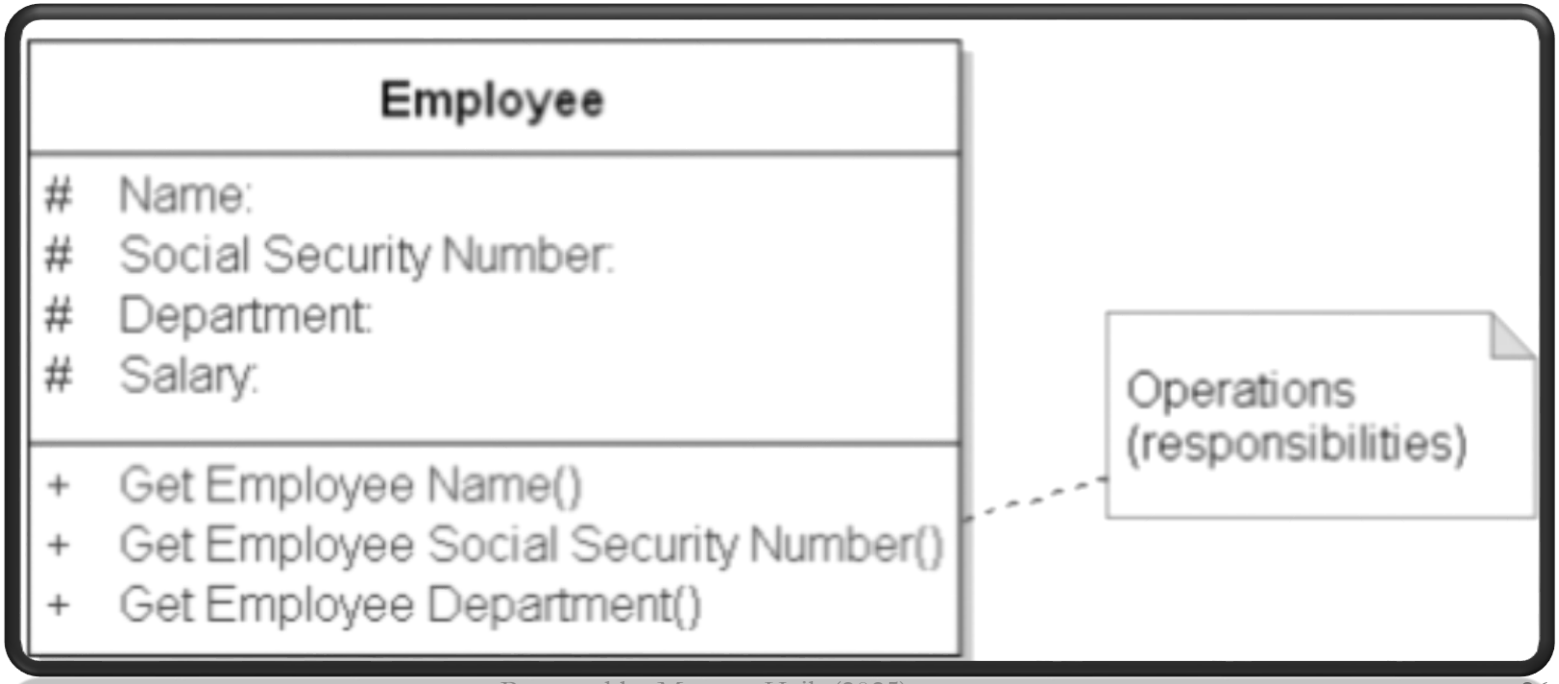
- The object encapsulates both data and the logical procedures required to manipulate the data

Achieves "information hiding"



Encapsulation .

- It is good engineering practice to encapsulate the state of an object rather than expose it using operation **abstraction**.



Polymorphism

- **Polymorphism** is another important OOP concept. Polymorphism, a Greek term, means the **ability to take more than one form.**
- An operation may exhibit different behavior is different instances.
- The behavior depends upon the types of data used in the operation.
  - ✓ For example, consider the operation of addition. For two numbers, the operation will generate a sum.
  - ✓ If the operands are strings, then the operation would produce a third string by concatenation.

Polymorphism

- The process of making an operator to exhibit different behaviors in different instances is known as **operator overloading.**
- **That is a single function name can be used to handle different number and different types of argument.**
- This is something similar to **a particular word having several different meanings depending upon the context.**
- Using a single function name to perform different type of task is known as function overloading.

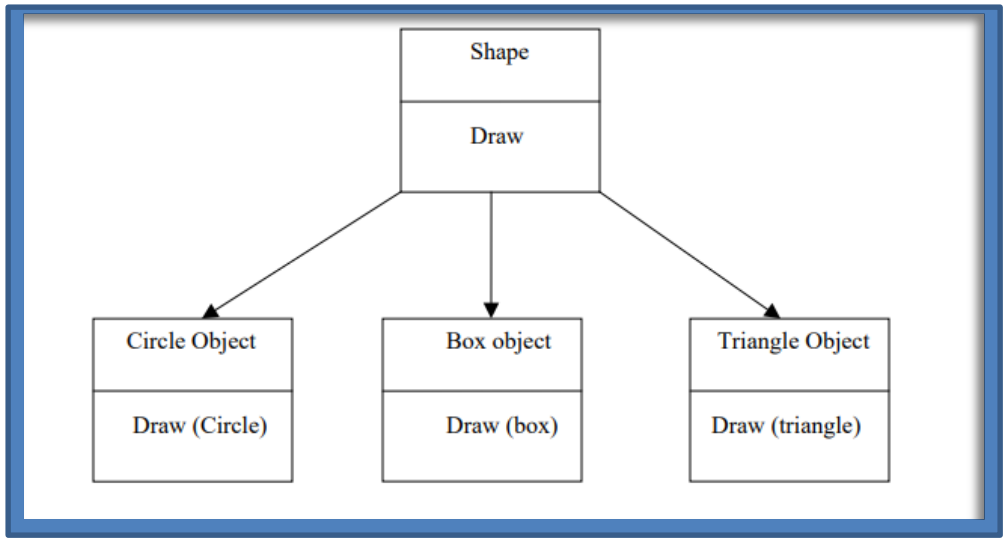
Polymorphism

- **Polymorphism** plays an important role in allowing **objects having different internal structures to share the same external interface.**
- This means **that a general class of operations may be accessed in the same manner even though specific action associated with each operation may differ.**
- **Polymorphism** is extensively used in **implementing inheritance**

Polymorphism

- In the next slid the figure illustrates that a single function name can be used to handle different number and different types of argument.
- This is something similar to a particular word having **several different meanings depending upon the context**

Polymorphism

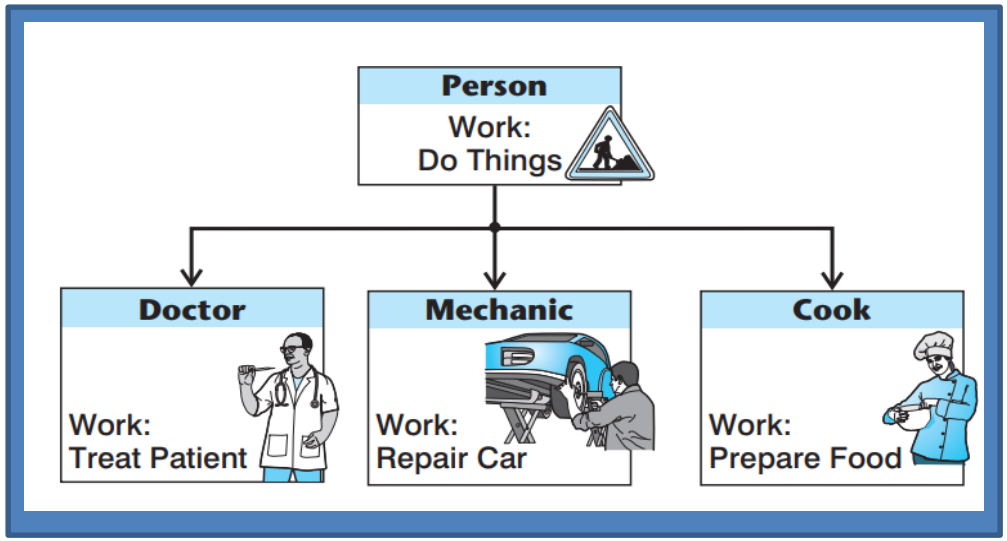


Polymorphism

- In the next slid the figure illustrates that a single function name can be used to handle different number and different types of argument.
- This is something similar to a particular word having **several different meanings depending upon the context**

The same message, **"Work,"** is implemented differently **depending on the nature of the object, not the nature of the message.**

Polymorphism



Person is a superclass with a **Work** responsibility. **Person** can have numerous subclasses, including **Doctor**, **Mechanic**, and **Cook**.

Inheritance, Association and Aggregation

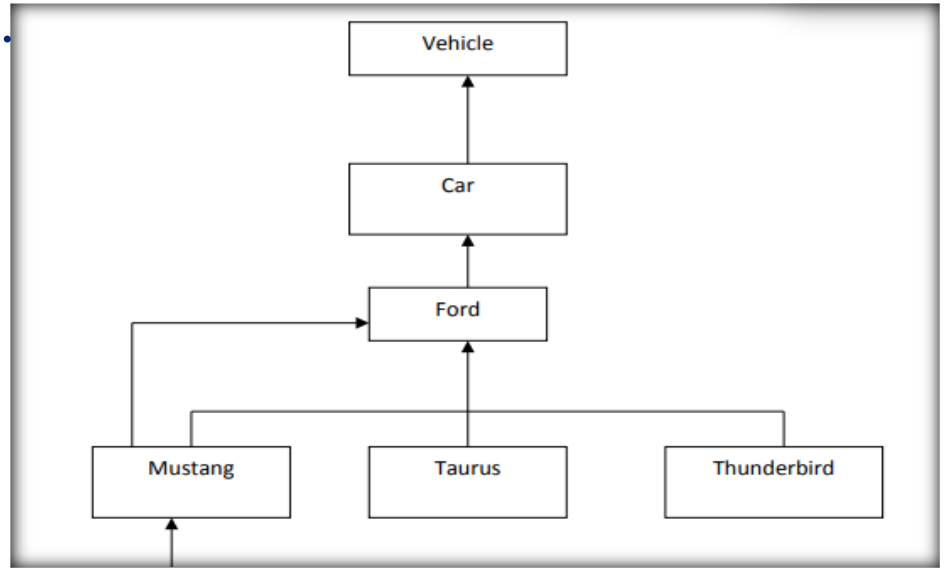
Inheritance

- **Inheritance**:- is the property of object- oriented systems that allows objects to be built from other objects.
- **Inheritance** allows explicitly taking advantage of the commonality of objects when constructing new classes.
- **Inheritance** is a relationship between classes where one class is the parent class of another (derived) class. The parent class is also known as the **base class or superclass**

Inheritance, Association and Aggregation

Inheritance

- For example the **car class defines the general behavior of cars**. The ford class inherits the general behavior from the car class and adds behavior specific to fords.



- **Inheritance** is the process by which objects of one class acquired the properties of objects of another classes. It supports the concept of hierarchical classification

Inheritance, Association and Aggregation

Inheritance

- For example, the bird, ‘robin’ is a part of class ‘flying bird’ which is again a part of the class ‘bird’.
- The principal behind this sort of division is that each derived class shares common characteristics with the class from which it is derived .
- In OOP, the concept of **inheritance** provides the idea of **reusability**.

Inheritance, Association and Aggregation

Inheritance

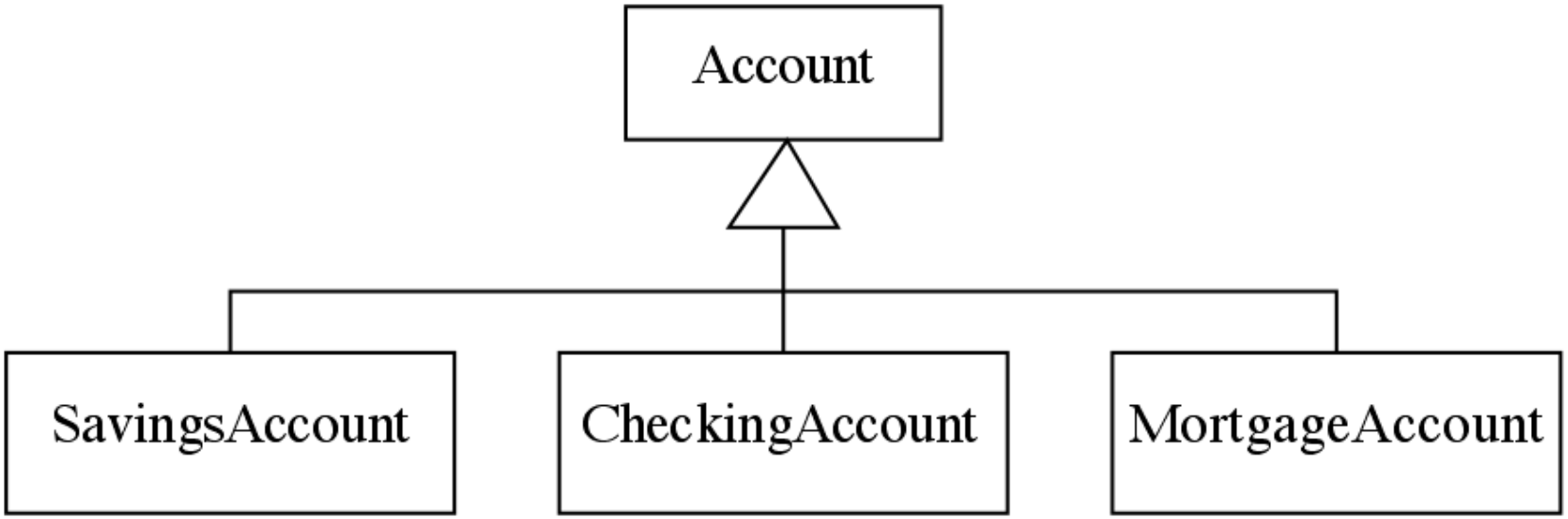
- This means that we can **add additional features** to an existing class without modifying it.
- This is possible by deriving **a new class from the existing one.**
- The new class will have the combined feature of both the classes.
- **Inheritance** is a relationship among classes wherein one class shares the structure and/or behavior defined in **one (single inheritance) or more (multiple inheritance) other classes**

➢ Inheritance, Association and Aggregation

Inheritance

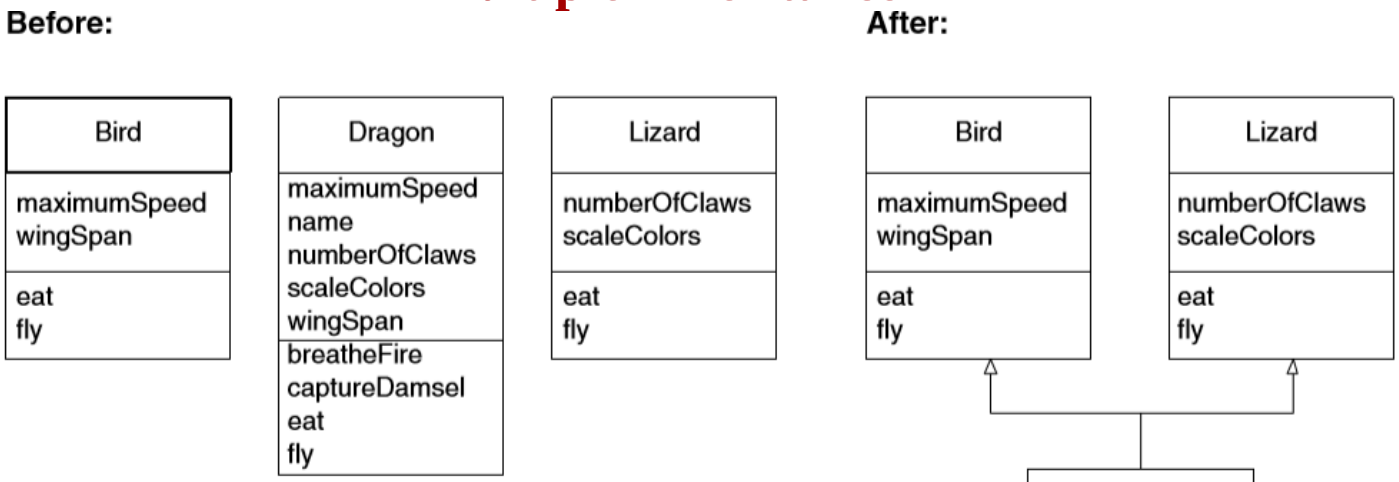
- The implicit possession by **all subclasses of features defined in its super classes.**

Single inheritance

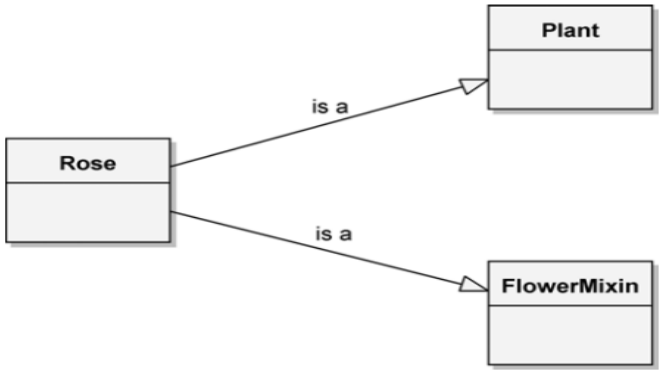


Inheritance, Association and Aggregation

Multiple Inheritance



The Rose Class, Which Inherits from Multiple Superclasses



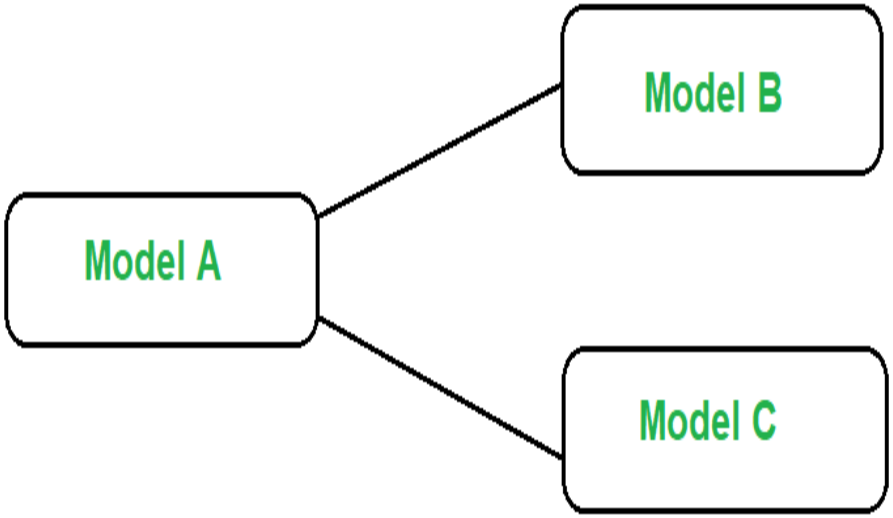
➢ Inheritance, Association and Aggregation

Association and Aggregation

- **Association** is a relationship between two classes where one class use another
- **An association** is defined as an organization of people with a common purpose and having a formal structure.
- It represents a binary relationship between two objects that describes an activity.
- It is a relationship between objects. For example, A doctor can be associated with multiple patients.

Inheritance, Association and Aggregation

Association and Aggregation



Association

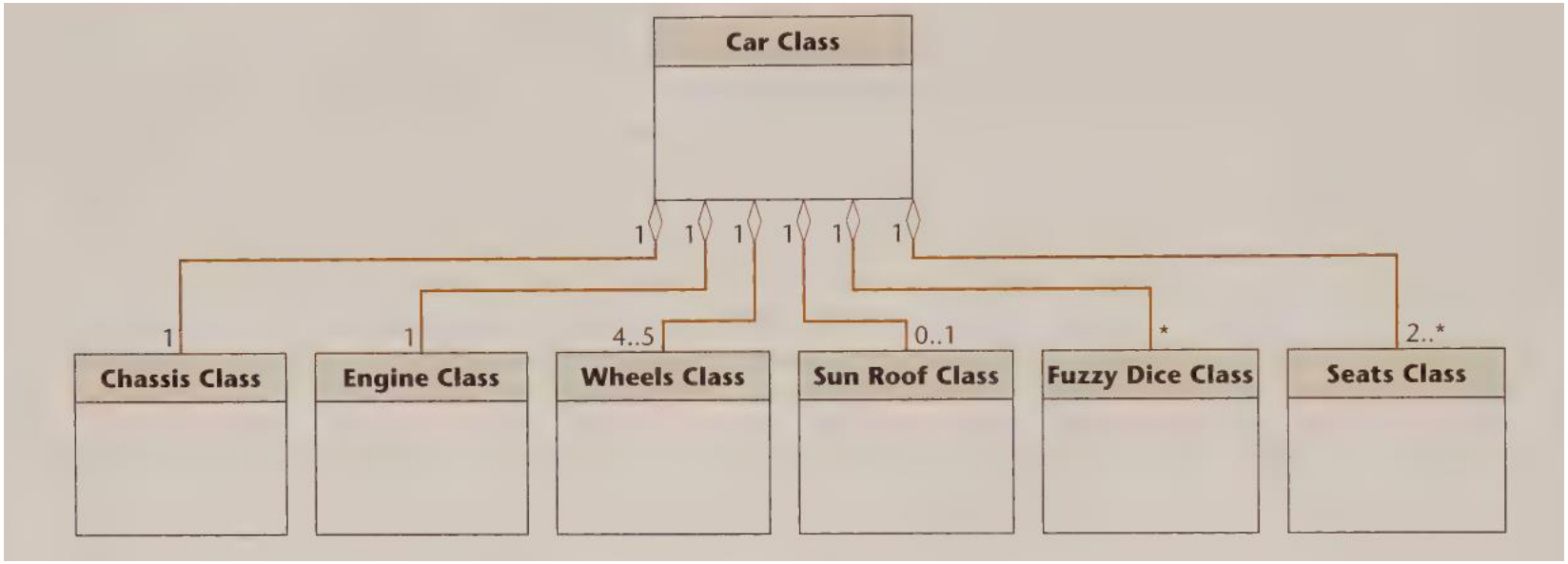
➢ Inheritance, Association and Aggregation

Aggregation

- **An aggregation** is a collection, or the gathering of things together.
- This relationship is represented by a "**has a**" relationship.
- In other words, aggregation is a **group, body, or mass composed of many distinct parts or individuals**
- For example, phone number list is an example of aggregation.

Inheritance, Association and Aggregation

Aggregation



Remaining concepts of Object orientation

OBJECT-ORIENTED SYSTEMS ANALYSIS AND DESIGN SECOND EDITION

Generalization and specialization

- **Generalization and specialization** represent a hierarchy of relationships between classes, where subclasses inherit from super-classes.
- In the **generalization process**, the common characteristics of classes are combined to form a class in a higher level of hierarchy,
- i.e., subclasses are combined to form a generalized super class.
- It represents an **"is – a or kind – of"** relationship.
- For example, "car is a kind of land vehicle", or "ship is a kind of water vehicle".

Remaining concepts of Object orientation

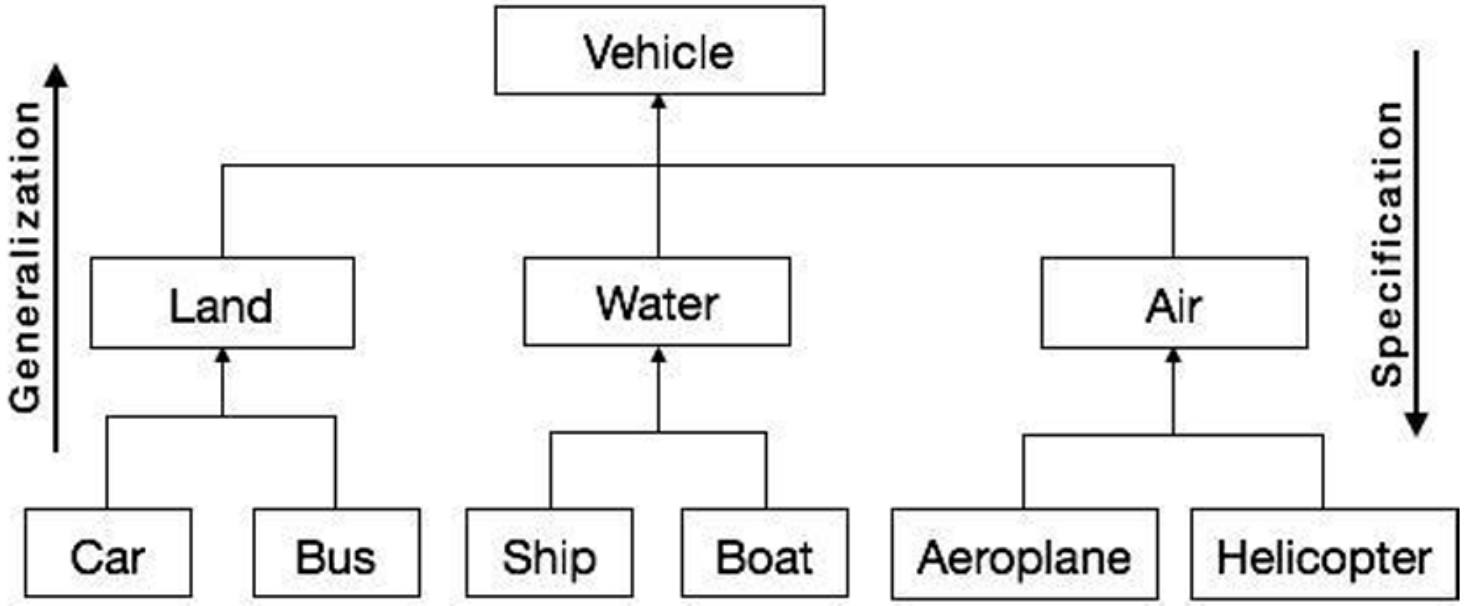
OBJECT-ORIENTED SYSTEMS ANALYSIS AND DESIGN SECOND EDITION

Generalization and specialization

- **Specialization**
- **Specialization** is the reverse process of generalization.
- Here, the distinguishing features of groups of objects are used **to form specialized classes from existing classes.**
- It can be said that the subclasses are the specialized versions of the super-class.
- The following figure shows an example of generalization and specialization.

Remaining concepts of Object orientation

- **Generalization and specialization**



Remaining concepts of Object orientation

OBJECT-ORIENTED SYSTEMS ANALYSIS AND DESIGN SECOND EDITION

➢ Links and Association

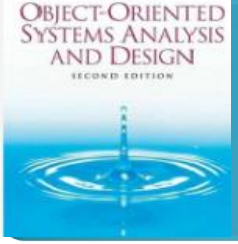
❑ Link

- ❖ **A link** represents a connection through which **an object collaborates with other objects.**
- ❖ **A link** is a physical or conceptual connection between objects".
- ❖ Through a link, **one object may invoke the methods or navigate through another object.** A link depicts the **relationship between two or more objects.**

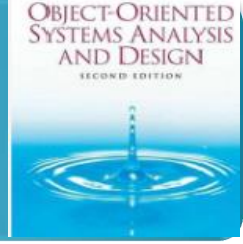
❑ Association

- ❖ **Association** is a **group of links having common structure and behavior**.
- ❖ **Association** depicts the **relationship between objects of one or more classes**.

☞ **A link** can be defined as an instance of an association.



Remaining concepts of Object orientation



- **Links and Association**
- **Degree of an Association.**

- **Degree of an association** denotes the number of classes involved in a connection.
  - A unary relationship connects objects of the same class.
  - A binary relationship connects objects of two classes
  - A ternary relationship connects objects of three or more classes.

Remaining concepts of Object orientation

OBJECT-ORIENTED SYSTEMS ANALYSIS AND DESIGN SECOND EDITION

Cardinality Ratios of Associations

- **Cardinality** of a binary association denotes the **number of instances participating** in an association.

- There are three types of cardinality ratios, namely:
  - **One–to–One** : A single object of class A is associated with a single object of class B.
  - **One–to–Many** : A single object of class A is associated with many objects of class B.
  - **Many–to–Many :** An object of class A may be associated with many objects of class B and conversely an object of class B may be associated with many objects of class A.

Remaining concepts of Object orientation

OBJECT-ORIENTED SYSTEMS ANALYSIS AND DESIGN SECOND EDITION

Aggregation or Composition

- **Aggregation or composition** is a relationship among classes by which a class can be made up of any combination of objects of other classes.
- It allows objects to be placed directly within the body of other classes.
- **Aggregation** is referred as **a "part–of" or "has–a" relationship**, with the ability to navigate **from the whole to its parts**.
- **An aggregate objec**t is an object that is **composed of one or more other objects**.
- **Example In the relationship**, "a car has–a motor", **car is the whole object** or **the aggregate**, and the **motor is a "part–of" the car**.
- **Aggregation** may denote:
  - **Physical containment**: Example, a computer is composed of **monitor, CPU, mouse, keyboard, and so on.**
  - **Conceptual containment**: Example, **shareholder has–a share**.

Persistence

- **Abstract and Concrete Classes**
  - Abstract Classes cannot be instantiated while concrete class can be
- **Persistence** focuses on **the issue of how to make objects available for future use.**
  - To make an object persistent, you must save the **values of its attributes** to permanent storage (such as a relational database or a file) as well as any information needed to maintain the **relationships (aggregation, inheritance, and association)** with which it is involved.
  - In addition **to *saving objects, persistence is also concerned with their retrieval and deletion*.**
  - **Business/domain classes are usually persistent**.
    - Need to store both attributes and associations
  - **User-interface classes are usually transitory**

Interfaces

- **An interface** is the definition of a collection of one or more methods, **and zero or more attributes.**
- **Interfaces** ideally define a **cohesive set of behaviors.**
- **Interfaces** are implemented by **classes and components.**
- To implement an interface, a **class or component must include the methods defined by the interface.**

Component

- **A component** is a modular, extensible unit, **with independent deployment** that has contractually specified interface(s) and explicitly defined dependencies, if any.
- **A component represents a reusable piece of software** that provides some meaningful aggregate of functionality.
  - At the lowest level, a **component is a cluster of classes that are themselves cohesive but are loosely coupled relative to other cluster**
- Ideally, components should be modular, extensible, and open.
  - **Modularity** implies a component contains everything it needs to fulfill its responsibilities;
  - **Extensibility** implies a component can be enhanced to fulfill more responsibilities than it was originally intended to, and
  - **Open** implies it can operate on **several platforms and interact with other components** through a single **programming interface**.

Pattern

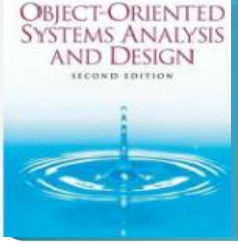
- **A pattern** is a solution to a common problem taking relevant forces into account, effectively supporting the reuse of proven techniques and approaches of other developers.
- Several types of patterns exist (Addressed in the respective chapters)
  - **Analysis patterns**: describe a solution to common problems found in the analysis/business domain analysis of an application.
  - **Design patterns**: describe a solution to common problems found in the design of systems
  - **Process patterns**: address software process model related issues.

Summary

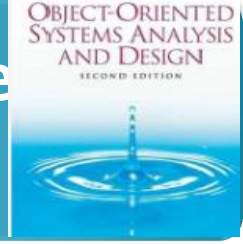
- **The maturation of software system analysis and design** has led to the development of object-oriented analysis, design, and programming methods, all of which address the issues of programming-in-the-large.
- There are several different programming paradigms: **procedure-oriented, object-oriented, logic-oriented, rule-oriented, and constraint-oriented.**
- **An abstraction** denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.
- **Encapsulation** is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation.

Summary

- **Modularity** is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.
- **Hierarchy** is a ranking or ordering of abstractions.
- **Typing** is the enforcement of the class of an object, such that objects of different types may not be interchanged or, at the most, may be interchanged only in very restricted ways.
- **Concurrency** is the property that distinguishes an active object from one that is not active.
- **Persistence** is the property of an object through which its existence transcends time and/or space

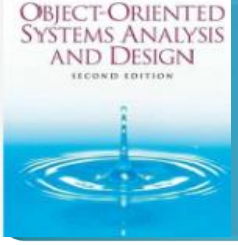


Object-oriented System Development life cycle

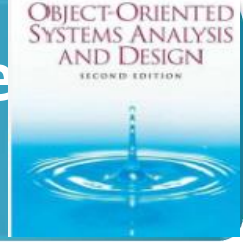


Software Development Process

- The **software development process** is normally long and tedious.
- However, **project managers** and **system analysts** can leverage software development life cycles to **outline**, **design**, **develop**, **test**, and eventually **deploy** information systems or software products with greater **regularity**, **efficiency**, and **overall quality.**
- **Software development life cycle (SDLC)** or a **software development process** is a **process of planning and managing software development**
- It typically involves dividing software development work into **smaller, parallel, or sequential steps or sub-processes to improve design and/or product management.**



Object-oriented System Development life cycle



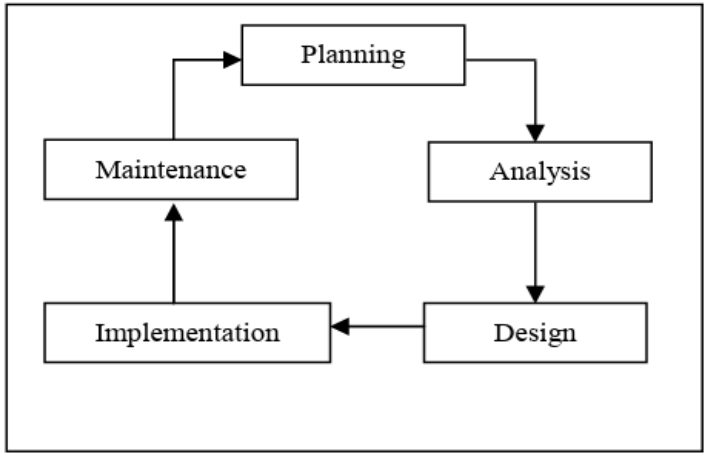
Software Development Process

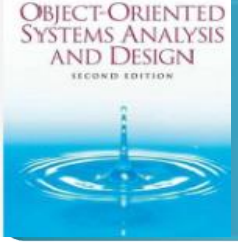
- The **methodology** may include the pre-definition of specific **deliverables** and **artifacts** that are created and completed by a project team to develop or maintain an application.
- A life-cycle "**model**" is sometimes considered a more general term for a category of **methodologies** and a software development "**process**" is a particular instance as adopted by a specific organization.
- For example, there are many specific software development processes that fit the **spiral life-cycle model**.
- The field is often considered a subset of the **systems development life cycle.**

Software Development Process

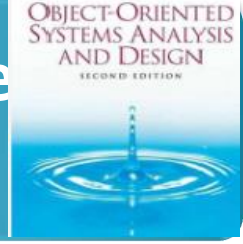
- **SDLC methodologies** view software development as primarily a project management process rather than a technical one.
- **Systems development methodology (SDM)** is a standard process followed in an organization to conduct **all the steps necessary to analyze, design, implement and maintain information systems.**
- **Systems development life cycle (SDLC)** is a **framework** composed of distinct steps or phases in the development of an Information System.
- **SDLC** consists of **five main stages** which include Planning, Analysis, Design, Implementation and Maintenance as shown in the figure below:

- **Systems Development Life Cycle=>**





Object-oriented System Development life cycle



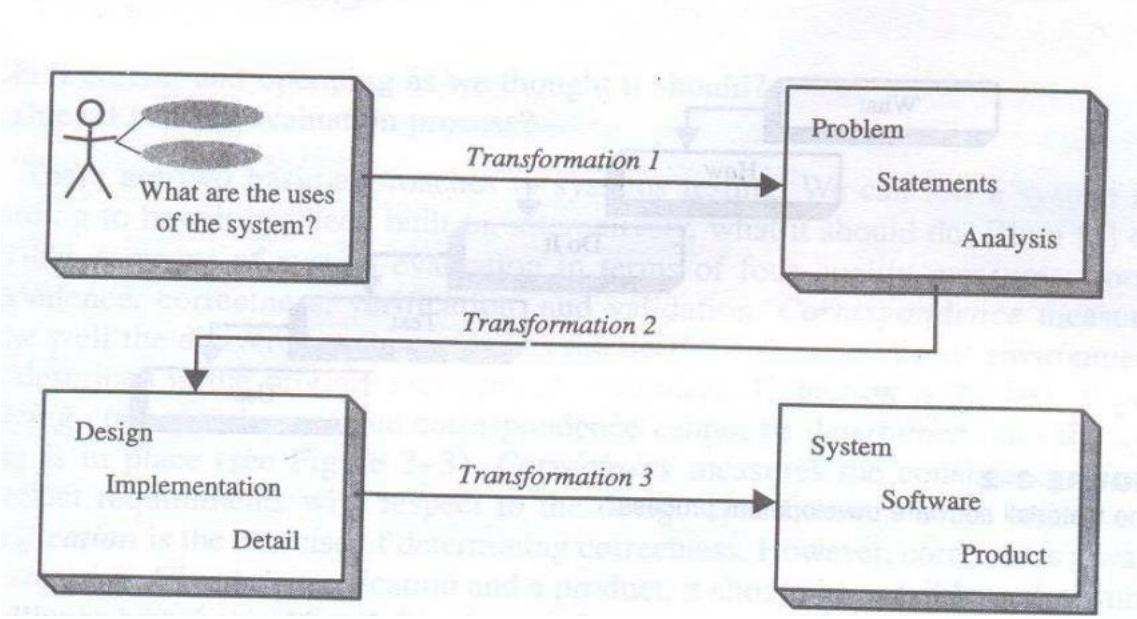
Software Development Process

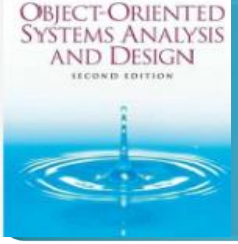
- Some books divide this stages in to 7
- Stage 1: Planning Stage
- Stage 2: Feasibility or Requirements of Analysis Stage
- Stage 3: Design and Prototyping Stage
- Stage 4: Software Development Stage
- Stage 5: Software Testing Stage
- Stage 6: Implementation and Integration
- Stage 7: Operations and Maintenance Stage

Object-oriented System Development life cycle

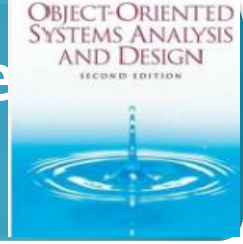
OBJECT-ORIENTED SYSTEMS ANALYSIS AND DESIGN SECOND EDITION

- **Software Development Process** can be considered as a process of transformation from one stage to another stage in the life cycle of software development





Object-oriented System Development life cycle

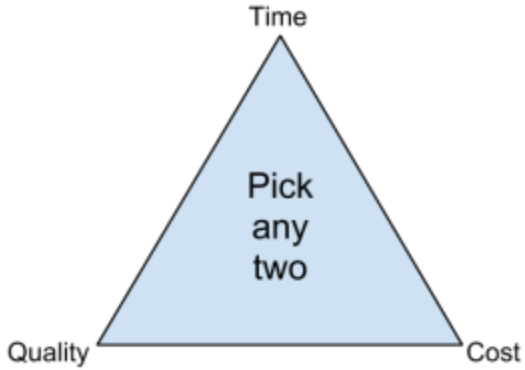


Software Development Process

- **A project** is an undertaking that **starts with a concept and ends with achieving a goal or a set of goals**.
- **SDLC** methodologies focus on software development as **a business project**, most aspects of which are independent of coding or technology.
- Hence the term "**life**," which indicates a beginning and an end. "**Cycle**" means that the process, in part or in whole, can be repeated.
- Beyond this basic similarity, the numerous SDLC methodologies part ways from each other.
- There is one concept we have to consider as a software developer. Which is the **quality triangle.**

Software Development Process

The Quality Triangle



- When developing software, or any sort of product or service, there exists a **tension** between the developers and the different stakeholder groups, such as management, users, and investors.
- This tension relates to how quickly the software can be developed (**time**), how much money will be spent (**cost**), and how well it will be built (**quality**).
- The quality triangle is a simple concept. It states that for any product or service being developed, you can only address two of the three: time, cost, and quality.

Software Development Process

Stages of software development

- We know that the Object-Oriented Modeling (OOM) technique visualizes things in an application by using models organized around objects.
- Any software development approach goes through the following stages:
  - Analysis,
  - Design, and
  - Implementation.

What is a software process model?

- **A software process model** is an abstraction of the software development process.
- **The models** specify the stages and order of a process. So, think of this as a representation of the order of activities of the process and the sequence in which they are performed.

Software Development Process

- **What is a software process model?**
- **A model will define the following:**
  - **The tasks** to be performed
  - **The input and output of each task**
  - **The pre and post-conditions for each task**
  - **The flow and sequence of each task**
- **The goal of a software process model** is **to provide guidance for controlling and coordinating the tasks** to achieve the end product and objectives as effectively as possible.

Software Development Process

What is a software process model?

- There are many kinds of process models for meeting different requirements.
- We refer to these as **SDLC models** (**Software Development Life Cycle models).**
- The most popular and important SDLC models are as follows:
  1. **The Waterfall Mode**
  2. **Prototyping**
  3. **Incremental & Iterative Approach**
  4. **The Spiral Mode**
  5. **Rapid Application Development (RAD)**
  6. **Agile Methodologies**

Software Development Process

SDLC models

1. **The Waterfall Mode**

- **The Waterfall Model** is the classic life cycle methodology and is closely tied to project management.
- This methodology was one of the earliest attempts to bring order to the chaos of software development, but it has been widely criticized for its **inflexible approach** to an undertaking that is extremely iterative
- **It's linear and straightforward** and requires development teams to finish one phase of the project completely before moving on to the next.
- **Each stage has a separate project plan** and takes information from the previous stage to avoid similar issues (if encountered).
- However, it **is vulnerable to early delays and can lead to big problems arising for development teams later down the road.**

Software Development Process

SDLC models

1. The Waterfall Mode

- The Waterfall Method views development activities as strictly defined steps or stages:

1. **Feasibility study.**
2. **System investigation (generally the same as gathering requirements**
3. **System analysis. This system analysis, however, does not correspond to the activity that we described earlier (under "What Do Methodologies Address?"). Instead, it has elements of domain analysis, and is basically "an attempt to understand all aspects of the present system and why it developed as it did, and eventually indicate how things might be improved by any new system.**
4. **System design.**
5. **Implementation.**
6. **Review and maintenance**

- shortcomings when applied in the field:

Software Development Process

SDLC models

1. **The Waterfall Mode**

It has the following shortcomings when applied in the field:

1. **Inflexibility**
2. **Overreliance on Documentation**
3. **Detachment from Technology**
4. **Detachment from Marketplace**
5. **Detachment from the Profession**

- You don't realize any value until the end of the project
- You leave the testing until the end
- You don't seek approval from the stakeholders until late in the day

This approach is **highly risky**, often more **costly** and generally **less efficient** than **Agile** approaches

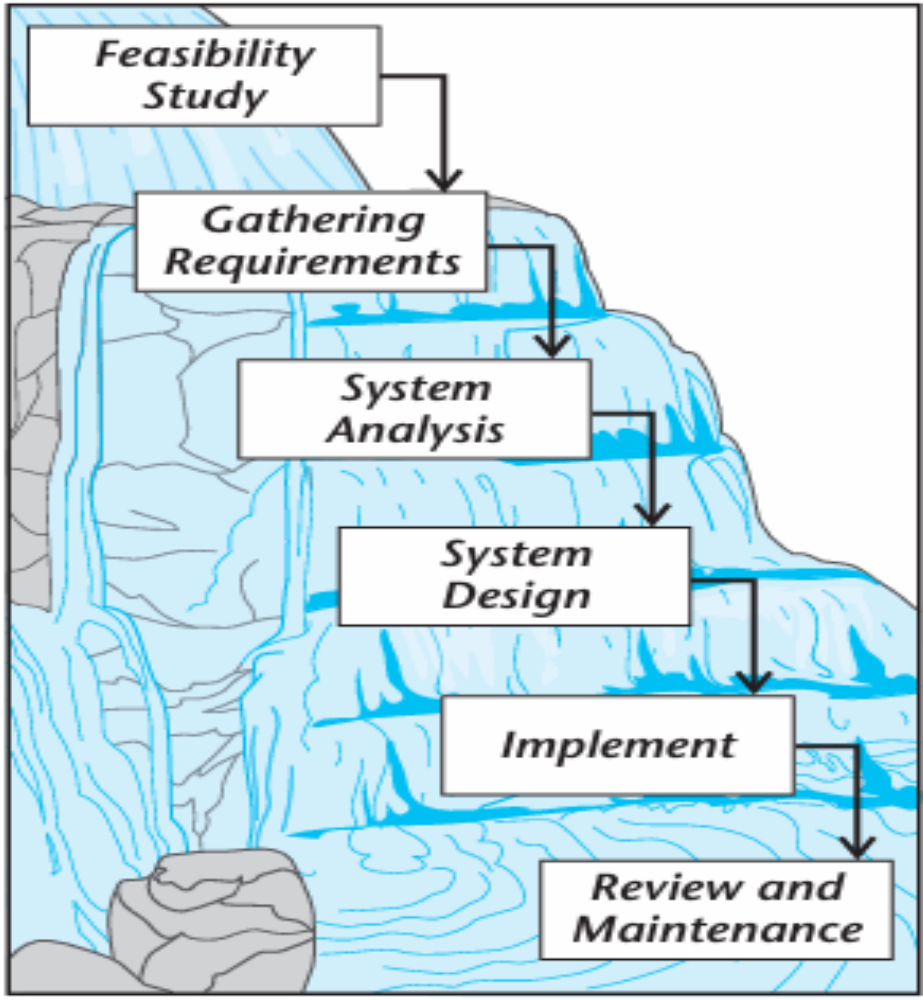
Object-oriented System Development life cycle

- **Software Development Process**
- **SDLC models**
  1. **The Waterfall Mode**

- **Three variations of linear sequential**
- **methods include**

Waterfall Model, V-Model and W-Model

(Read more to know the deference)



Software Development Process

SDLC models

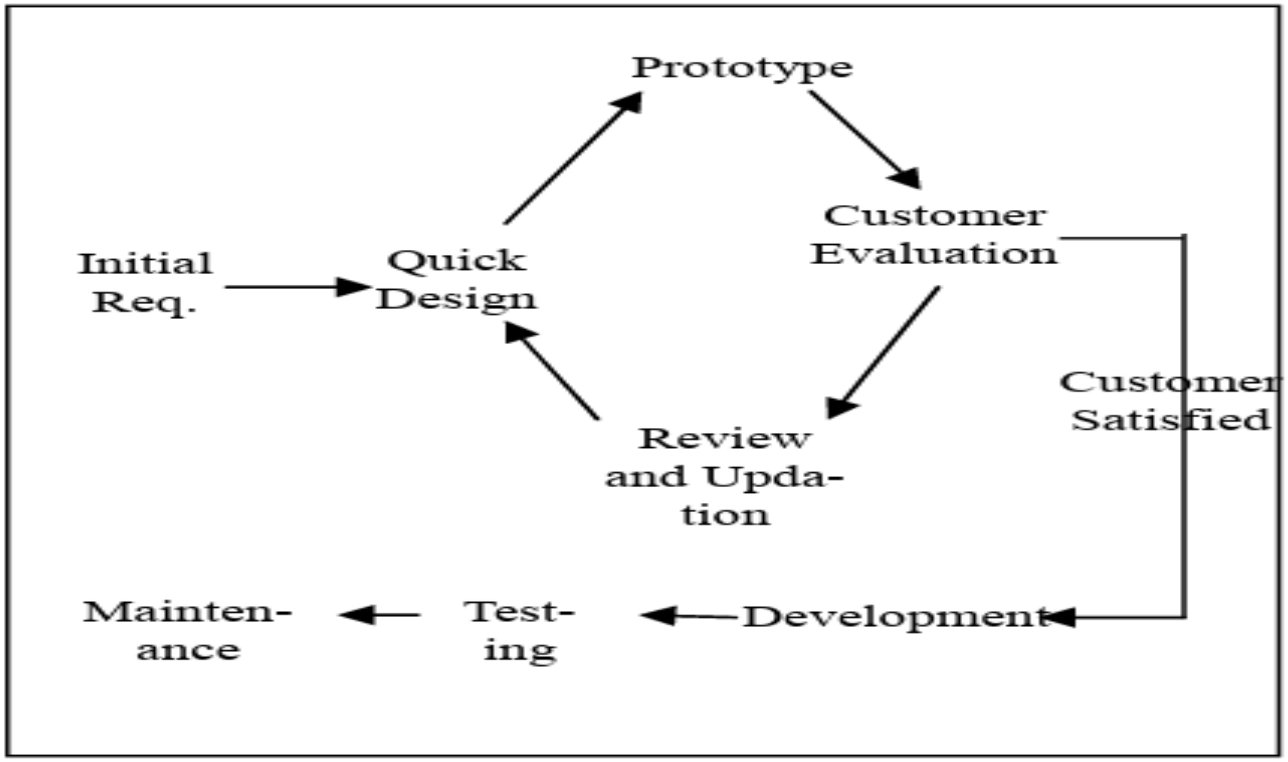
2. The Prototyping Model

- **Prototyping** is the creation of a working model with the essential features of the final product for testing and verification of requirements.
- This model is widely used when the customers are **unclear of their requirements.**
- The initial requirements are gathered from the customers followed by a quick design.
- **A prototype** is developed and shown to the customer for evaluation. Once the customer is **satisfied**, full-fledged systems development will be done as shown in Figure below.

➢ Software Development Process

❑ SDLC models

2. The Prototyping Model



Software Development Process

- **SDLC models**

2. The Prototyping Model

- The prototype developed is of two types – **Throwaway and Evolutionary**.
- If the prototype is developed **only to get acceptance from customer and discarded further**, then it is known as '**Throwaway Prototype**'.
- The Prototype which will be **further developed as the actual system** is referred as **'Evolutionary Prototype'.**
- **Incremental & Iterative Approach**

Software Development Process

SDLC models

2. The Prototyping Model

- **Advantages**:
  - Users are involved in the development process
  - Captures requirements in concrete form, rather than verbal/abstract form
- **Disadvantages**
  - Unbalanced Architecture
  - Insufficient analysis
  - User confusion of prototype and finished system
  - Developer misunderstanding of user objectives
  - Developer attachment to prototype
  - Excessive development time of the prototype
  - Expense of implementing prototyping
  - The Illusion of Completeness.
  - Diminishing Changeability

Software Development Process

SDLC models

3. Incremental & Iterative Approach

- In incremental development, the product is built through successive versions that are refined and expanded with each iteration.
- At first glance, the incremental approach may appear the same as prototyping.
- Whereas prototyping aims to build a model that implements all major requirements (in theory, at least),
- in the incremental development each version of the product fully implements a selected set of requirements.

Software Development Process

SDLC models

3. Incremental & Iterative Approach

- The incremental approach groups the requirements into layers.
- The requirements in the innermost layer, considered most essential to the mission of the software, are implemented first.
- In each successive iteration, the defects of the previous version are corrected, the design is adjusted accordingly, and the requirements in the next layer are undertaken.

➢ Software Development Process

❑ SDLC models

3. Incremental & Iterative Approach

❖ Three concepts underlie the incremental approach.

1. **The Initialization Step**. **A "base" version** of the software is created and presented to users to elicit critical response.
2. **The Control List**. A list specifying necessary changes to design as well as new requirements. This list is not static but changes continuously as a result of user interaction with successive versions and new analysis.
3. **The Iteration Step**. New versions are built from the most recent control list. The goal of each iteration is to be **clear-cut and modular** to allow for redesign.

Software Development Process

SDLC models

3. Incremental & Iterative Approach

- Furthermore, in each iteration, the development goes through a complete life cycle: requirements analysis, analysis, design, and implementation.
- Unlike the Waterfall Model, however, each iteration is relatively short.
- When the control list becomes empty, the product is complete
- **The incremental approach** considers the code, and the control list, as the main development documentation. Since the users interact with versions that fully implement a set of requirements.
- **One of the strongest objections** to the incremental approach is its overoptimistic assumption that the **architecture** of the platform, the information system, or the application can gracefully accommodate changes that each iteration requires

Software Development Process

SDLC models

4. The spiral model

- **The spiral model** is a risk-oriented life cycle model that breaks a software project up into mini-projects.
- **The Spiral Model** combines the **Waterfall Model** with **prototyping** and **iteration**.
- It can also accommodate **alternate life cycle methodologies**.
- The result is a complicated approach that has nevertheless been **very popular with government projects** since its **focus is on risk management**: any kind of risk including, but not limited to, risks associated with **quality, performance, architecture, cost, and so on.**
- The Spiral Model starts as a small project that spawns other small projects as it moves forward through a spiral iteration. Each iteration has six steps:

Software Development Process

- **SDLC models**

3. The spiral model

Six Steps:

1. Determine objectives, alternatives, and constraints
2. Identify and resolve risks
3. Evaluate alternatives
4. Develop the deliverables for that iteration, and verify that they are correct
5. Plan the next iteration
6. Commit to an approach for the next iteration (if you decide to have one)

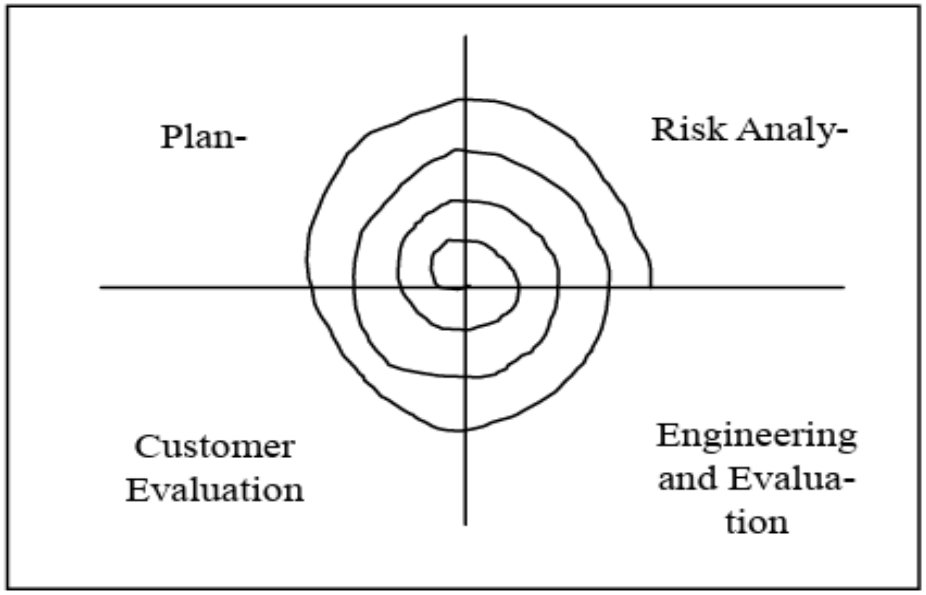
It is not suitable for small as it is expensive

Software Development Process

SDLC models

3. The spiral model

- Depending on the risks identified, you can combine the Spiral Model with other approaches and methods.



- Besides the complexity of the concept, the Spiral Model has been criticized in that it largely ignores specifications, milestones, reviews, and scheduling

Software Development Process

SDLC models

5. Rapid Application Development (RAD)

- **Rapid Application Development** is selecting techniques, methods, practices, and procedures that result in **faster development and shorter schedules.**
- The RAD definition begs the question: Who would not want "faster develop ment" and "shorter schedules"?
- The answer is that RAD is a triple-functionality concept.

1. First, it functions as a mostly advertising enticement: Buy this product and you will develop your software faster.
2. Second, as a general concept (not capitalized), it is a pragmatic approach, not a product or a step-by-step methodology: Select the fastest and most effective methods and practices and discard the ineffective and slow ones.

➢ Software Development Process

❑ SDLC models

5. Rapid Application Development (RAD)

❖ In other words, do not become a slave to everything that this or that methodology prescribes or proscribes. In this sense, any kind of rational and well-scheduled development plan is RAD.

3. The third function is when RAD is offered as a "named" and formal methodology (this time capitalized). As such, it has **its own characteristics and guidelines,**

The most important of which are the following four phases:

1. Requirements Planning.
2. Design.
3. Implementation.
4. Enhancements & Maintenance

Software Development Process

- **SDLC models**

5. Rapid Application Development (RAD)

- In other words, do not become a slave to everything that this or that methodology prescribes or proscribes. In this sense, any kind of rational and well-scheduled development plan is RAD.
- It **increases customer involvement**, **encourages development of prototypes** and **extensively employs computer aided software engineering** (CASE) **tools**.

Software Development Process

SDLC models

6. Agile Methodologies

- **Agile methodologies** aim at being **adaptive** rather than **predictive**.
- Out of the object-oriented approach emerged the latest view of systems development, collectively called the **agile methodologies**.
- Through this work we have come to value:
  1. Individuals and interactions over processes and tools
  2. Working software over comprehensive documentation
  3. Customer collaboration over contract negotiation
  4. Responding to change over following a plan

Software Development Process

- **SDLC models**

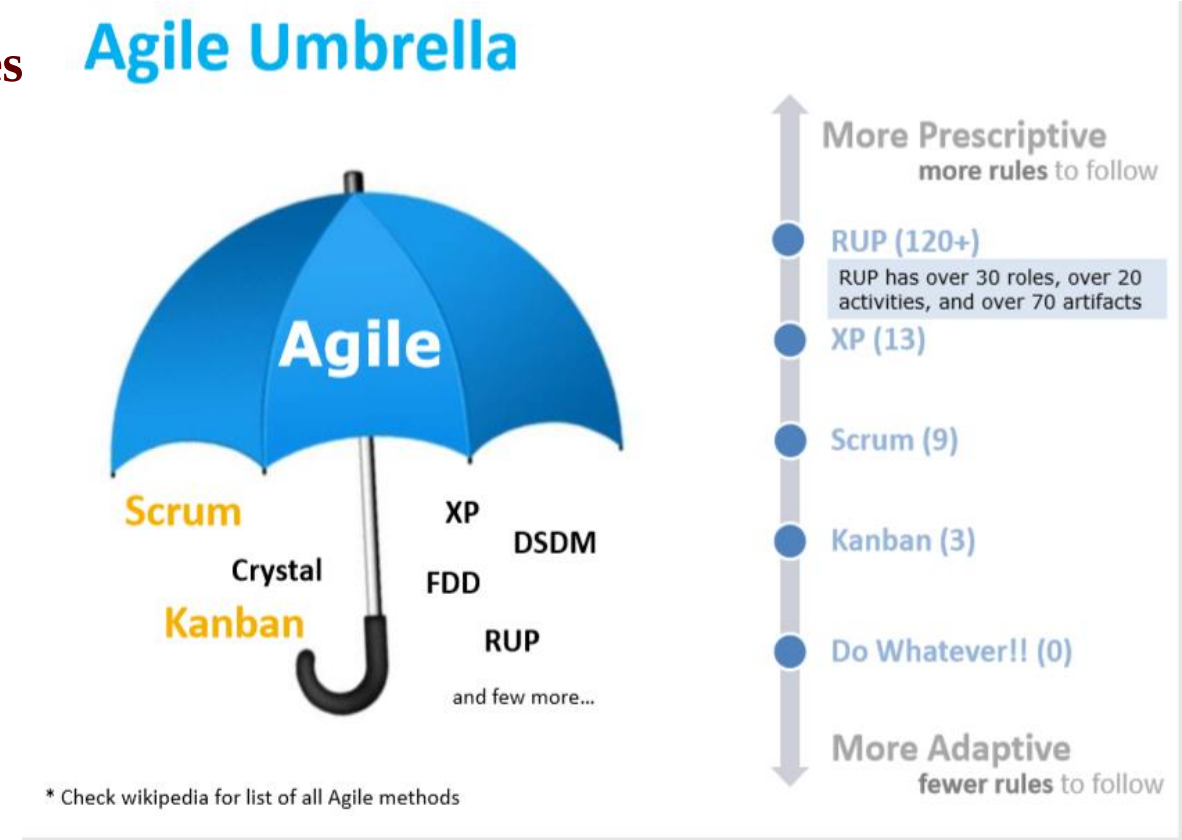
5. Agile Methodologies

- Through this work we have come to value: The agile methodologies share three key principles:
  1. A focus on adaptive rather than predictive methodologies,
  2. Individuals and interactions over processes and tools
  3. A focus on people rather than roles, and (3) a self-adaptive process.
- Agile Methods are lightweight Methods
- Different Agile Development Methodologies

➢ Software Development Process

❑ SDLC models

5. Agile Methodologies



- **Software Development Process**
- **SDLC models**

5. Agile Methodologies

Agile Manifesto

Individuals and interactions over **processes and tools**

Working software over **comprehensive documentation**

Customer collaboration over **contract negotiation**

Responding to change over **following a plan**

- Software Development Process
- SDLC models
  5. Agile Methodologies
  - Different Agile Development Methodologies
    - RUP
    - SCRUM
    - XP
    - FDD
    - DSDM
    - Kanban
    - Crystal

- **Software Development Process**
- **SDLC models**
- **The need for standardization**

  - There are so many methods and notations competing with each other that users are distracted by the decisions they need to make.
  - Existing methods are already converging since these methods pick up ideas from other sources.
  - **A single**, **common** language is desirable because it can be used for all development methods, used throughout the project lifecycle, and used for different application technologies.

Software Development Process

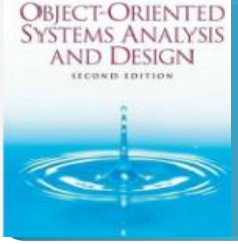
Unified Process

- **The Unified Process**, the methodology that we will be using for object-oriented analysis and design.
- **The Unified Process** is an iterative and incremental methodology
- **The Unified Process** is a modeling technique. A model is a set of UML diagrams that represent one or more aspects of the information system we want to develop.
- **The Unified Process** can be viewed as an adaptable methodology. That is, it has to be modified for the specific information system to be developed.

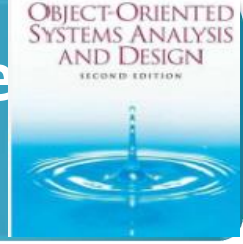
Software Development Process

Unified Process.

- **The Unified Process** is more than just a series of steps that, if followed, will result in the construction of an information system.
- In fact, **no such single "one size fits all" methodology could exist**, because there is such a wide variety of different types of information systems.
- The name Unified Software Development Process (or USDP) was used. The term Unified Process is generally used today, for brevity.



Object-oriented System Development life cycle



- **Software Development Process**
- **Unified Process.**
  - **This process is based in the following principles:**

1. **Use case driven:** The development is planned and organized over **a list of use cases.**
2. **Architecture centered**: The development process leads **to the construction of a system architecture that allows the implementation of the requirements**. That architecture is based on the identification of a structure that is iteratively built from a conceptual model.
3. **Iterative and incremental**: Development is divided into **iterations or development cycles**. At each iteration, new features are added to the system architecture, or corrected/refined, leaving it more complete and closer to the final desired system.
4. **Risk oriented**: The elements of greater risk for a project are addressed early. For instance, **critical use cases are identified, detailed, and implemented before the others**

- **Software Development Process**
- **Unified Process**.
  - **Phases of Unified Process:**

1. **The Inception Phase:-** to determine whether the proposed information system is economically viable.
   - **The deliverables of the inception phase include:-**
     i. The initial version of **the domain model.**
     ii. The initial version of **the business model.**
     iii. The initial version of **the requirements artifacts** (especially the use cases).
     iv. A preliminary version of **the analysis artifacts**.
     v. A preliminary version of **the architecture.** The initial list of risks.
     vi. The initial ordering of **the use cases**.
     vii. The plan for **the elaboration phase.**
     viii. The initial version of **the business case.**

- **Software Development Process**
- **Unified Process**.
  - **Phases of Unified Process:**

2. **The Elaboration Phase**:- to refine the initial requirements (use cases), refine the architecture, monitor the risks and refine their priorities, refine the business case, and produce the project management plan.

- **The deliverables of the elaboration phase include :-**
  - i. The completed domain model.
  - ii. The completed business model.
  - iii. The completed requirements artifacts.
  - iv. The completed analysis artifacts.
  - v. An updated version of the architecture.
  - vi. An updated list of risks.
  - vii. The project management plan (for the remainder of the project).
  - viii. The completed business case.

- **Software Development Process**
- **Unified Process**.
  - **Phases of Unified Process:**

3. **The Construction Phase** :- is to produce the first operational-quality version of the information system, sometimes called the beta release.

- **The deliverables of the construction phase include:-**

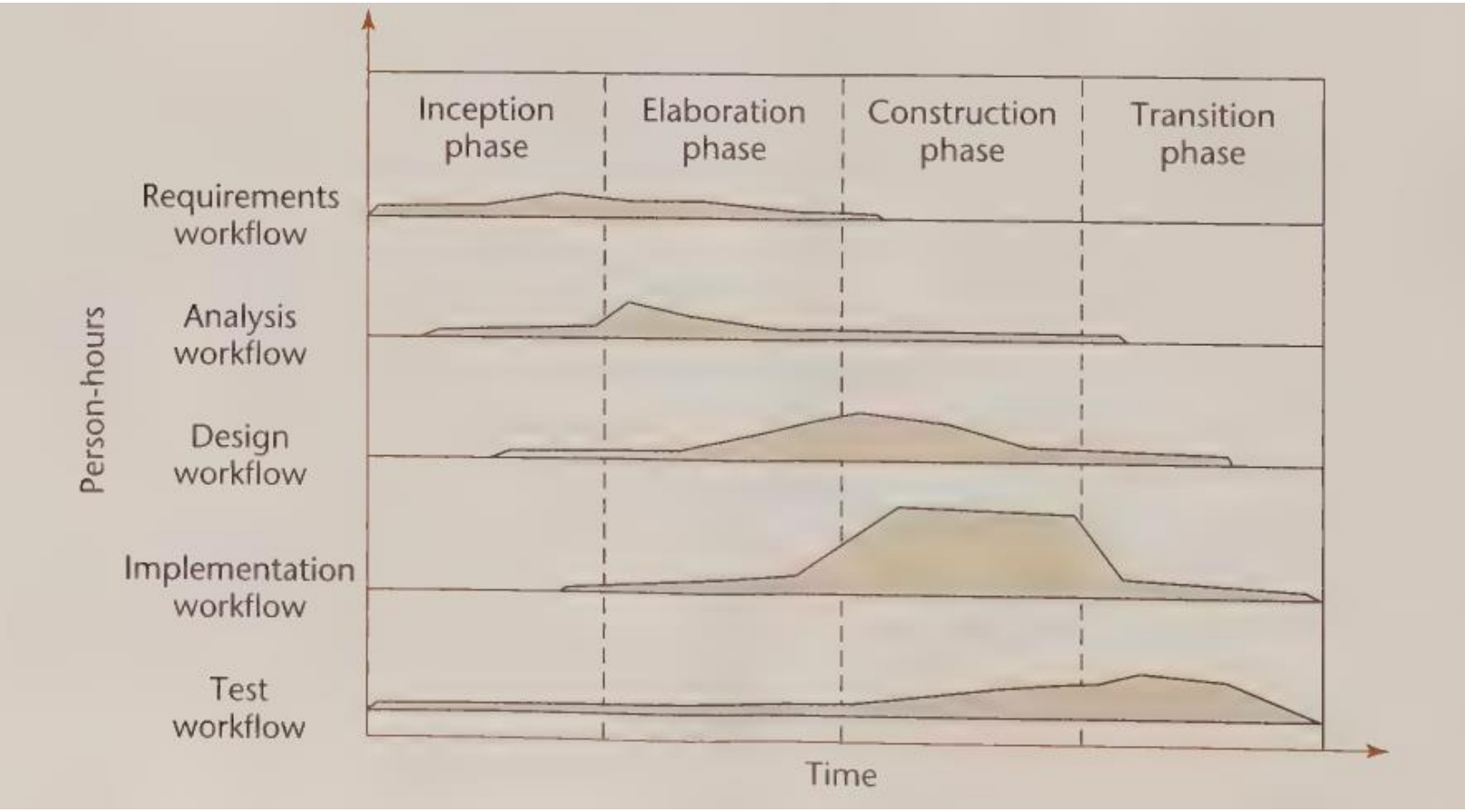
i. The initial user manual and other manuals, as appropriate.
ii. All the artifacts (beta release versions).
iii. The completed architecture.
iv. The updated risk list.
v. The project management plan (for the remainder of the project).
vi. If necessary, the updated business case.

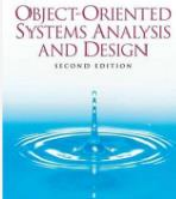
- **Software Development Process**
- **Unified Process**.
  - **Phases of Unified Process:**
    4. **The Transition Phase**:- The aim of the transition phase is to ensure that the client's requirements have indeed been met.
  - **The deliverables of the construction phase include:**
    i. All the artifacts (final versions).
    ii. The completed manuals.

Object-oriented System Development life cycle

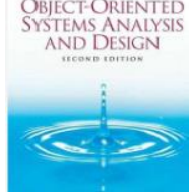
OBJECT-ORIENTED SYSTEMS ANALYSIS AND DESIGN SECOND EDITION

- **Software Development Process**
- **The Core Workflows and Phases of the Unified Process**

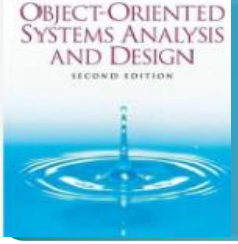




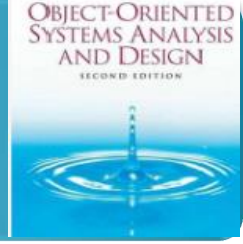
UNDERSTANDING THE BASICS: MODELING WITH UML



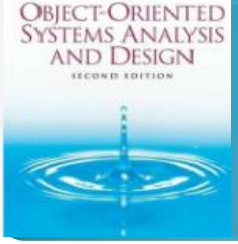
- **Definition**
- **UML Structure(Building Blocks)**
- **UML- Architectural Views**
- **UML –Diagram types (Structure, Behavior and Interaction)**
- **UML Diagram Examples**



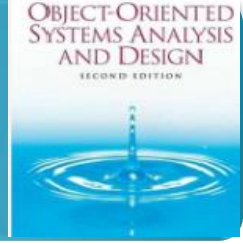
Unified Modeling Language(UML)



- **What is Unified Modeling Language**
- **Definition**
  - The Unified Modeling Language (UML) is a family of graphical notations, backed by single meta-model, that help **in describing and designing software systems, particularly software systems built using the object-oriented style.**
  - **A language** is simply a tool for expressing ideas.
  - **UML** is a language that can be used to describe things.

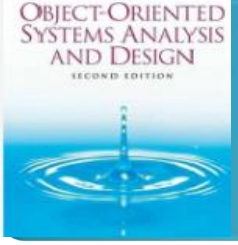


Unified Modeling Language(UML)

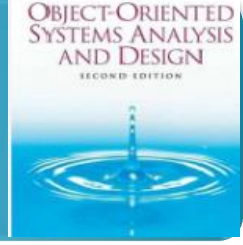


- **What is Unified Modeling Language**
- **Definition**

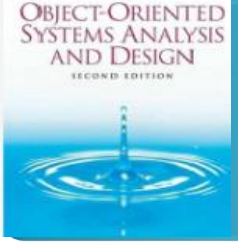
  - As a language, UML can be used to describe information systems developed using the traditional paradigm or any of the many versions of the object-oriented paradigm, including the Unified Process.
  - **UML** is a modeling language for object-oriented system **analysis**, **design**, and **deployment**.
  - **UML** is a language for object-oriented modeling.
  - **UML's notation**—its system of figures and symbols—is designed to represent object-oriented concepts.
  - **UML** is not a product, nor a process nor a methodology.



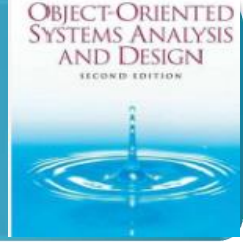
Unified Modeling Language(UML)



- **What is Unified Modeling Language**
- **Definition**
  - **The Unified Modeling Language(UML)** is a language that can be used to **describe things**.
  - **UML** is a family of **graphical notations**, backed by single meta-model, that help in **describing and designing software systems, particularly software systems built using the object-oriented style.**
  - **UML** is a modeling language for object-oriented system **analysis**, **design**, and **deployment**.

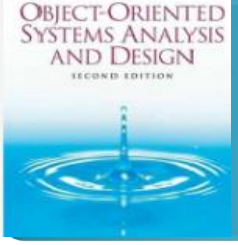


Unified Modeling Language(UML)

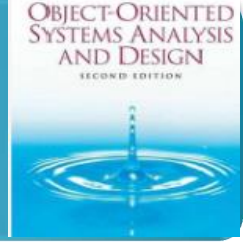


What is Unified Modeling Language

- **UML** is used to:
  - **model object-oriented designs**
  - **Shows overall design of a solution**
  - **Shows how classes interact with each other**
  - **Shows class specifications**
- **UML** diagrams use specific icons and notations
- It is language independent
- **UML** is the standard diagrammatic notation for drawing picture related to software.

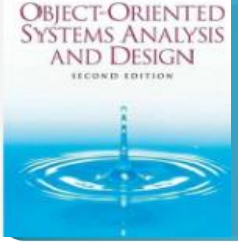


Unified Modeling Language(UML)

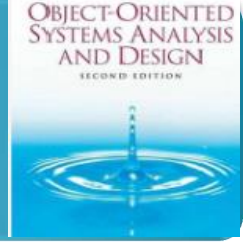


What is Unified Modeling Language

- UML is a language for object-oriented modeling. UML's notation—its system of figures and symbols—is designed to represent object-oriented concepts.
- **The UML** offers a standard way to write **a system's blueprints**, including **conceptual things** such as **business processes** and **system functions** as well as concrete things such as **programming language statements, database schemas, and reusable software components.**
- The UML is a language for
  - **Visualizing**
  - **Specifying**
  - **Constructing**
  - **Documenting the artifacts of a software-intensive system**.



Unified Modeling Language(UML)

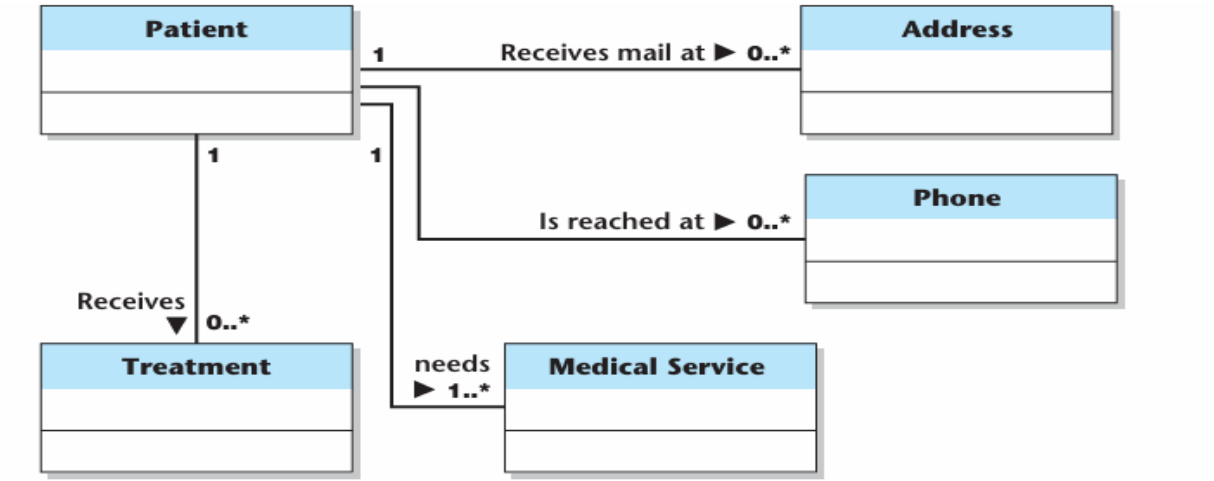


What is Unified Modeling Language

- **Visualizing:-**UML diagrams visualize system components, their relationships, and their interactions.
- It provides of **a set of graphical elements** that are combined to form diagrams. Each **diagram is a visual presentation or view of the system** and satisfies one or more broad but overlapping types of modeling: **Behavioral , Structural and Interaction**
  - **Specifying:-** UML **provides precise and complete models** for the three major activities of system development: **analysis, design, and implementation.**

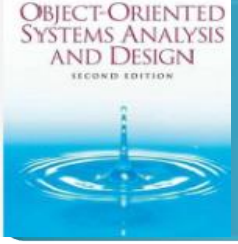
What is Unified Modeling Language

- **Constructing:-** UML models are compatible with object-oriented languages.

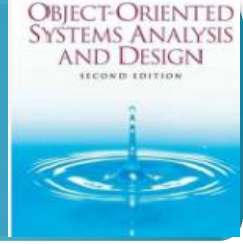


A UML Diagram: Modeling Object-Oriented Concepts

- **Documenting the artifacts of a software-intensive system.**



Unified Modeling Language(UML)



What is Unified Modeling Language

- The UML offers a standard way to write **a system's blueprints**, including **conceptual things** such as **business processes** and **system functions** as well as concrete things such as **programming language statements, database schemas, and reusable software components.**
- **The UML language** has been under development since James Rumbaugh and Grady Booch joined forces **at Rational Software and started to unify their already well-known diagrammatic notations and processes**.

Why We Use UML ?

- **Standardized notation** without sacrificing specialized model data
- **Common language** that can be used **from product conception to delivery**, **from system to detailed design levels.**
- **Reduced learning curve** across projects
- **Increased domain and design model reuse**
- **Increased customer involvement** /understanding of problem translation to product solution.

Rules of UML

- UML's building blocks should be put together to develop a **well-formed** model.
- **well-formed** model is a model that is semantically self-consistent and in harmony with all its related models.
- **The UML has semantic rules** for
  - Names: What you can call things, relationships, and diagrams
  - Scope: The context that gives specific meaning to a name.
  - Visibility: How those names can be seen and used by others
  - Integrity: How things properly and consistently relate to one another.
  - Execution: What it means to run or simulate a dynamic model.

The UML Diagram Types

- Behavior diagrams
- Interaction diagrams
- Structure diagrams

- **Behavioral Diagrams** – focus on **dynamic aspects of the software system** – **Use-case, Interaction, State Chart, Activity**
- In a **behavior diagram**, **individual aspects of a system and their changes are displayed at runtime**.
- A **behavior diagram** is **intended to provide clarity,** for example, about **internal processes, business processes or the interaction of different systems.**
- They emphasize **what must happen in the system or business process**.
- They are used to describe the **functionality of the system.**

The UML Diagram Types

Behavior or Functional diagrams

- Depending on the diagram used, a selected aspect is shown.

1. **Use Case Diagram(Functional)** – high-level behaviors of the system, user goals, external entities: Actors
2. **Sequence Diagram** – focus on time ordering of messages
3. **Collaboration Diagram** – focus on structural organization of objects and messages
4. **State Chart Diagram** – event driven state changes of system
5. **Activity Diagram** – flow of control between activities

The UML Diagram Types

Structure diagrams

- **Structure diagram** focus on static aspects of the software system
  - **Class, Object, Component, Deployment**
- The elements in a structure diagram represent **the meaningful concepts of a system, and may include abstract, real world and implementation concepts.**
- **Behavior diagrams** show the dynamic behavior of the objects in a system, which can be described as a series of changes to the system over time.

The UML Diagram Types

Structural Diagrams

1. **Class Diagram** – set of classes and their relationships. Describes interface to the class (set of operations describing services)
2. **Object Diagram** – set of objects (class instances) and their relationships
3. **Component Diagram** – logical groupings of elements and their relationships
4. **Deployment Diagram** - set of computational resources (nodes) that host each component.

The UML Diagram Types

Interaction diagram

- **Interaction diagram** focus on describing the **flow of messages within a system**, providing context for one or more lifelines within a system.
- As its name might suggest, an interaction diagram is a type of UML diagram that's **used to capture the interactive behavior of a system**.
- In addition, interaction diagrams can be **used to represent the ordered sequences within a system and act as a means of visualizing real-time data via UML.**

The UML Diagram Types

Interaction diagrams

- There is one (or more) **Interaction** diagram per use case
  - ✓ Represent a sequence of interactions
  - ✓ Made up of objects, links, and messages

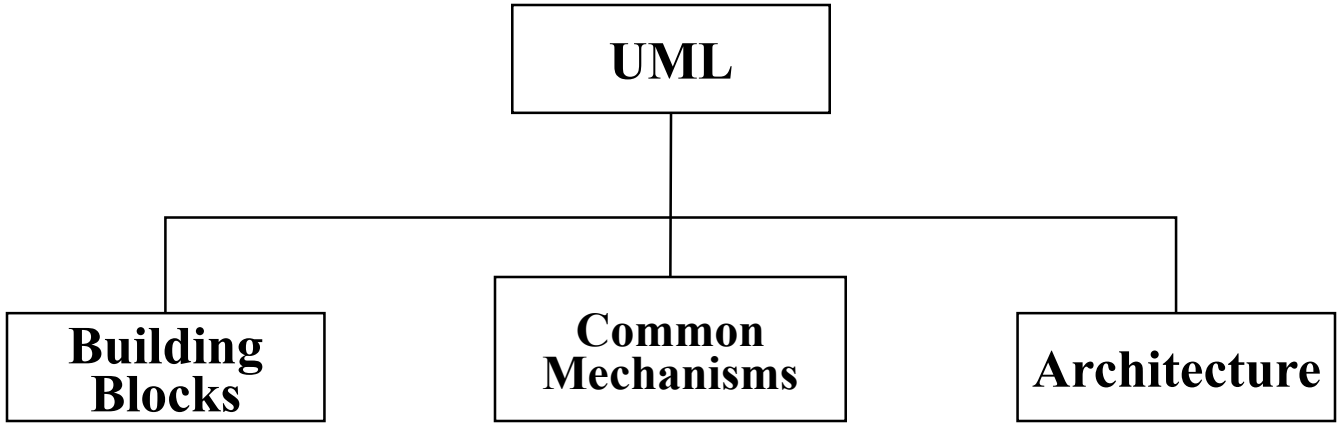
1. Sequence diagrams

- ✓ Models flow of control by time ordering
- ✓ Emphasizes passing messages write time
- ✓ Shows simple iteration and branching

2. Collaboration diagrams

- ✓ Models flow of control by organization
- ✓ Structural relationships among instances in the interaction
- ✓ Shows complex iteration and branching

UML STRUCTURE



- **Building blocks**
  - Things
  - Relationships
  - Diagrams

- **Common Mechanisms**
  - Specifications
  - Adornments (decoration)
  - Common divisions
  - Extensibility mechanisms

- **Architecture**
  - use-case view
  - logical view
  - process view
  - implementation view
  - deployment view

➢ BUILDING BLOCKS OF UML

❖ **UML has three building blocks:**

1. **Things**, the objects.
2. **Relationships**, the glue that holds things together.
3. **Diagrams**, categorized as either structure or behavioral.

BUILDING BLOCKS OF UML

1. **Things – the abstractions**
   1. **Structural things** – nouns, static, represent conceptual or physical elements: ***Class***, ***interface***, ***collaboration***, ***use case***, ***active class***, ***component***, and ***node***
   2. **Behavioural things** – verbs, dynamic, represent behaviour over time and space: ***Interaction*** and ***state machine***
   3. **Grouping things** – organizational parts of UML: ***Packages***
   4. **Annotational things** – explanatory parts of UML: ***Note***

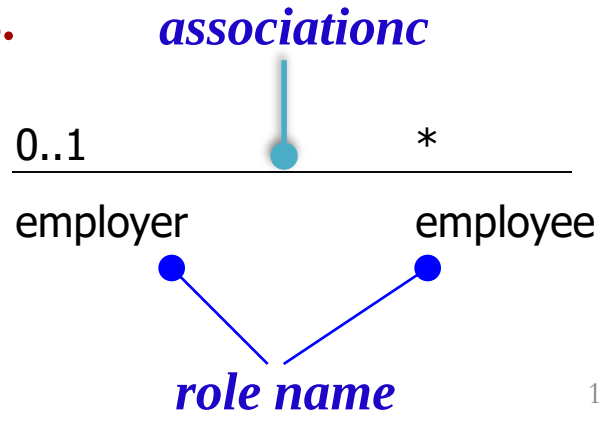
➢ BUILDING BLOCKS OF UML

2. Relationships – tie things together

A. **Dependency** (uses) – a semantic relationship between two things in which a change to one thing (**the independent thing)** may affect the semantics of the other thing (**the dependent thing**)

   A. Eg: A car uses a fuel

B. **Association** – a structural relationship that describes **a set of links, a link being a connection among objects.**



➢ BUILDING BLOCKS OF UML

2. Relationships – tie things together

c. Generalization (is-a) –

a specialization/generalization relationship in which objects of the specialized element (**the child**) are substitutable for objects of the generalized element (**the parent**)

➢ BUILDING BLOCKS OF UML

3. Diagram

- The ***graphical*** representation of a set of elements
- Help to ***visualize*** a system from different perspectives
- May contain any ***combination of things and relationships.***

BUILDING BLOCKS OF UML

UML DIAGRAMS

Structure diagrams:

Diagrams used to describe structure

1. Class diagram
2. Object diagram
3. Component diagram
4. Deployment diagram
5. Package diagram

Behavior diagrams

Diagrams used to describe behavior and Function

6. Use Case diagram (**functional**)
7. State chart diagram
8. Activity diagram

Interaction diagrams

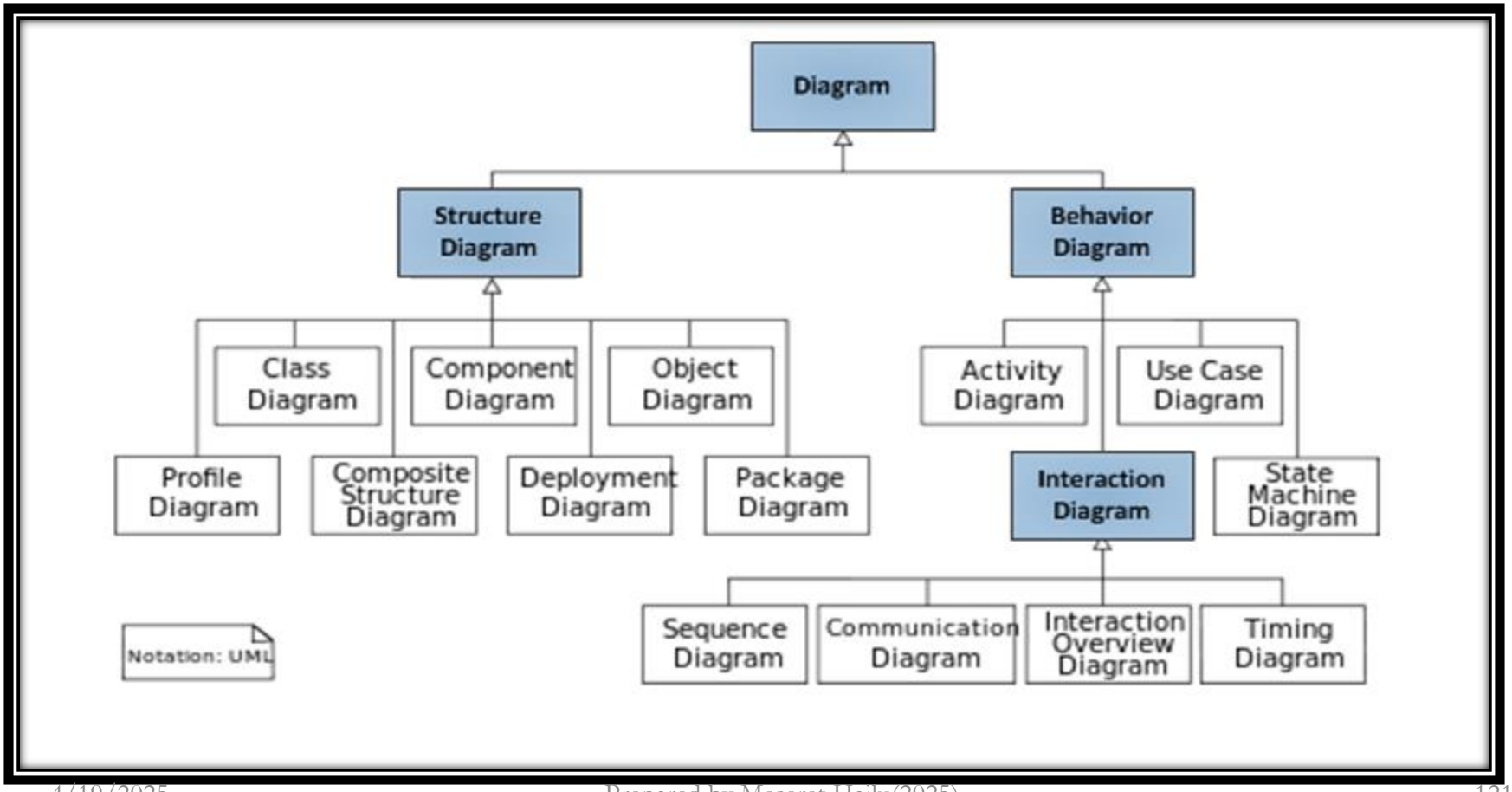
10. Sequence diagram
11. Communication diagram
12. **Interaction overview diagram (*)**
13. **Timing diagram (*)**

(*) not existing in UML 1.x, added in UML 2.0

- Collaboration diagrams

BUILDING BLOCKS OF UML

UML DIAGRAMS



➢ The most commonly used UML diagrams

1. Use case diagram

- ❖ **Use case diagrams** describe **how the system is used**.
- ❖ It is the **starting** point for UML modeling.
- ❖ **A use-case diagram** includes a set of **use cases and actors**, and the **relationships** among them.
- ❖ **A use case** is **a description of a set of sequences of actions that a system performs** that yields an observable result of value to **a particular actor**.
- ❖ **An actor** is a role that **a user plays with respect to the system.**

The most commonly used UML diagrams

Components of a Use Case

- **A use case** has **four components: a goal, stakeholders, a system, and a scenario.**
- Not every story that describes a system's behavior is a use case.

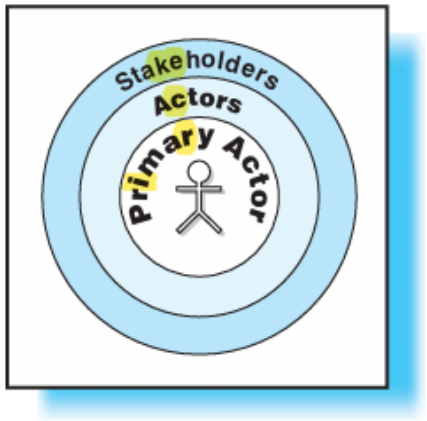
A use case must consist of four well-defined components to qualify as such:

1. **A goal is** the outcome of a successful use case.
2. **Stakeholders** whose interests are affected by the outcome of the use case, including **actors** who **interact with the system to accomplish the goal**.
3. **A system** that provides the services that the actors need to reach the objective.
4. **A scenario** that the actors and the system follow to accomplish the goal.

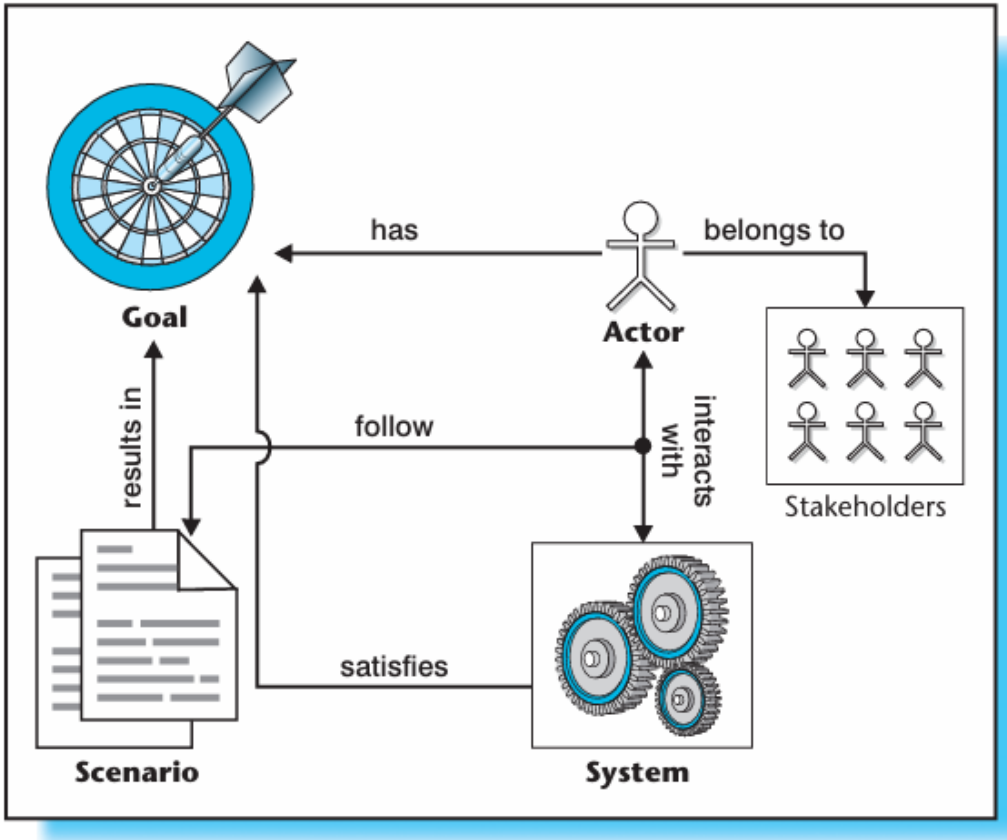
The most commonly used UML diagrams

Components of a Use Case

- **A use case** itemizes the interaction between a system and the actor(s) to achieve a goal.
- This goal must be **meaningful, a logically complete function, and of measurable value**
- **Stakeholders** are those entities whose interests are affected by the **success or the failure of the use case**
- The goal of the primary actor is specified by the name of the use case
- **An actor** is in fact **not a specific person or group of persons, but a role**. The same person may appear in different roles.
- **Actors are stakeholders** who interact with the system. The primary actor is an actor whose goal is accomplished by the use case



- **The most commonly used UML diagrams**
  - **Components of a Use Case**



Components of a Use Case: Actor(s), System, Goal, Scenario.

- **The goal must result in measurable value, but the value is a matter of judgment.**
- **A use case is successful only if its stated goal is completely achieved.**
- ***Suppose the supermarket customer picks up items and then leaves the supermarket without paying for them. Then use case is not successful.***

- **The most commonly used UML diagrams**
  - **Use case diagram**,
  - **A use case's name is its goal. The name must be active, concise, and decisive.**
  - **A use case's name** must show action.
  - It must have **one transitive verb, simple or compound**, and **one grammatical object, simple or compound.**
  - The following examples are acceptable:
    - **Verify Credit Card:** compound object.
    - **Set Up Tent**: compound verb.
  - ***To accomplish a goal, a set of steps must be taken, but: It is the goal that decides the relevance of activities in a use case.***

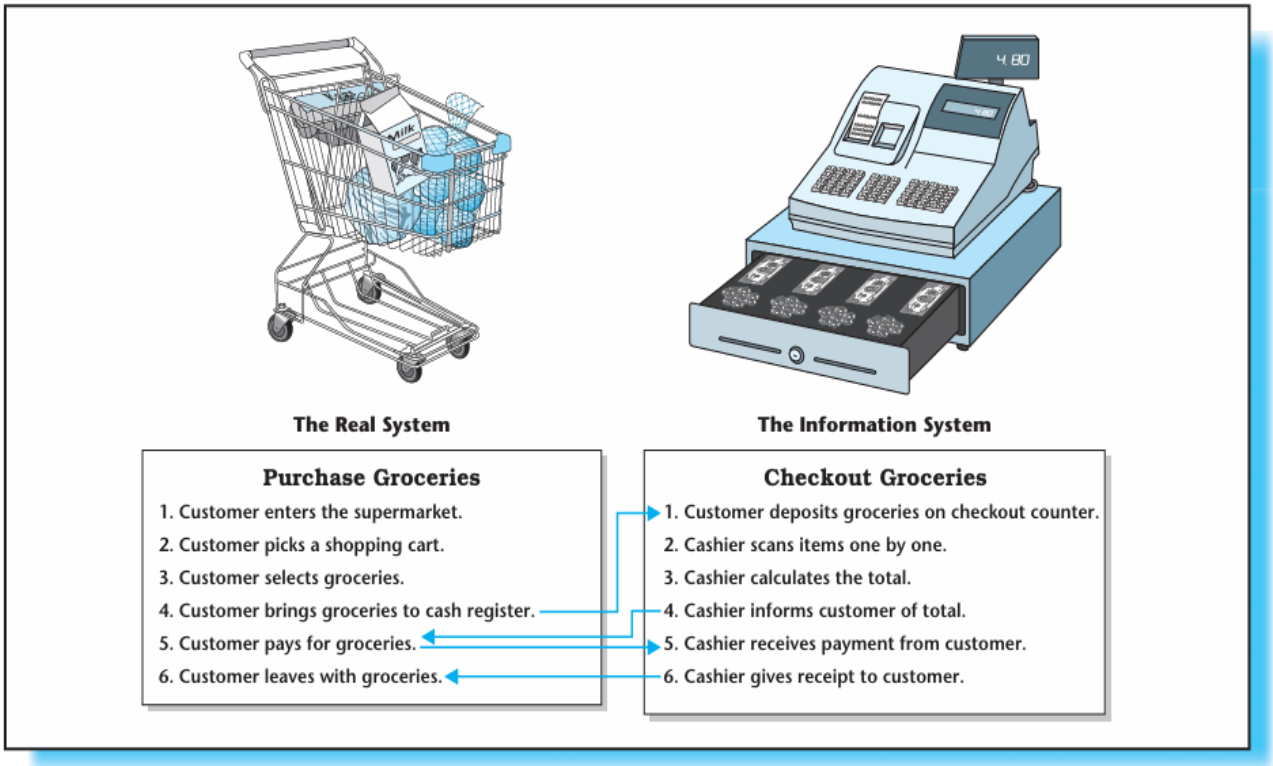
- **The most commonly used UML diagrams**
  - **Use case diagram**,
    - **A use case's name is its goal. The name must be active, concise, and decisive.**
    - **A use case** must enforce the interests of all stakeholders.
    - **An actor** is identified by **a unique name** that describes **a unique role.**
    - **The name** must be unique **across the whole enterprise, across the whole system,** and **not just within the scope of one use case, a set of use cases, or one domain or subsystem.**

- **The most commonly used UML diagrams**
  - **Use case diagram**,
    - The commercial division of a bank may believe that a **Customer is a corporation only (Commercial Customer)**, while the co**nsumer division may consider only an individual (Individual Customer)** as such.
    - Both are correct within their own spheres of activity. The system defines the boundaries of a use case.
    - **The system** defines the **boundaries of a use case.** The scope of the system constrains the scope of the use case.

The most commonly used UML diagrams

1. Use case diagram,

For Example: One Supermarket, Two Systems:



The System Is the Boundary of the Use Case

- Does this mean that there is an insurmountable wall between them, that one use case cannot call on the services of other systems outside its scope?
- A use case cannot leave a system, but it can reach across its boundaries.

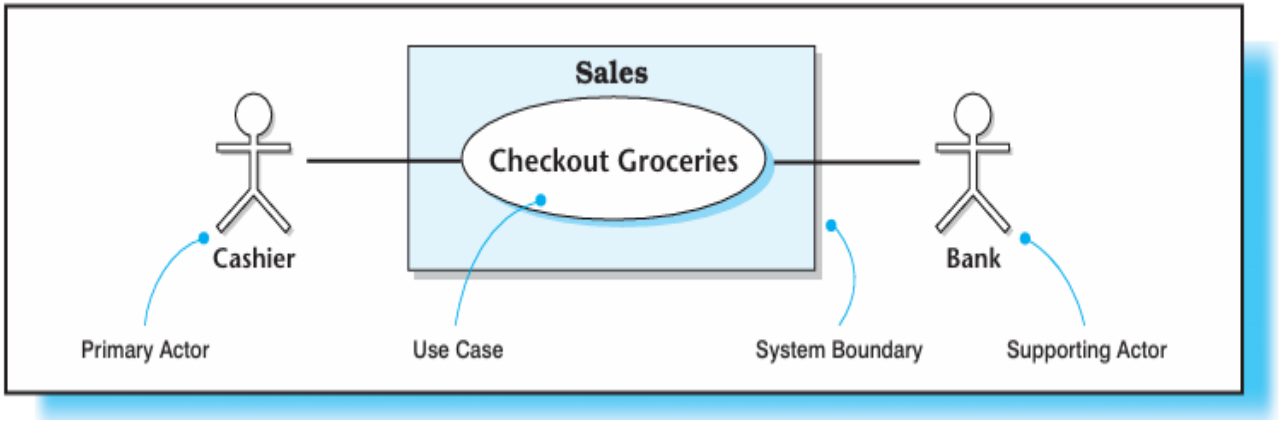
The most commonly used UML diagrams

1. Use case diagram,

For Example: One Supermarket, Two Systems:

Use Case Diagram:

- ☞ **The Interaction Between the Actors and the System**

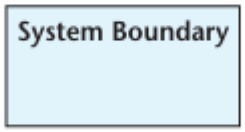


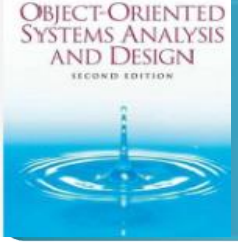
The most commonly used UML diagrams

OBJECT-ORIENTED SYSTEMS ANALYSIS AND DESIGN SECOND EDITION

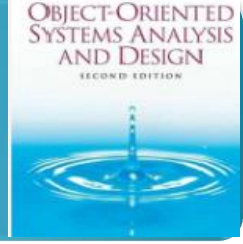
1. Use case diagram,

- The Basic Elements of the Use Case Diagram

 Actor	The stick figure identifies an **actor.** It is the same for any actor, be it a person or a system. More than one actor can be associated with a use case, and one actor can be associated with more than one use case.
 System Boundary	The rectangle is the ***system boundary*** or the ***scope.*** For the use case, any entity outside this boundary can exist only as an actor.
 Use case	The ellipse (the oval) represents the **use case.** It resides inside the system boundary. In the diagram, the only textual information about the use case is its name.
 Association	The simple line on the left represents ***association.*** It shows the communication between an actor and a use case.



The most commonly used UML diagrams

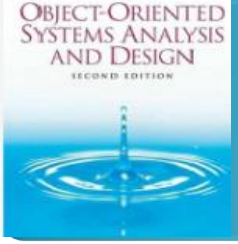


1. **Use case diagram**,

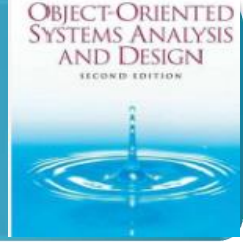
- **The scenario** is an ordered sequence of interactions between the actor(s) and the system to accomplish a goal.
- Therefore, **the steps** in the scenario must be sketched out carefully.
- ***Use case modeling captures the behavioral requirements of a system in terms that must be readily understood by all stakeholders***
- **The steps** fall into four categories.

1. **Normal Flow.** Normal flow is the best-case scenario, the ideal flow of events. If every step in the normal flow goes smoothly, the goal of the use case is accomplished.

In Purchase Groceries, the customer finds all desired items, brings them to the checkout counter, pays for them, and carries them out. Normal flow is the only category that a use case must have, at least explicitly. (It is difficult to imagine a use case that will never go wrong.)



The most commonly used UML diagrams

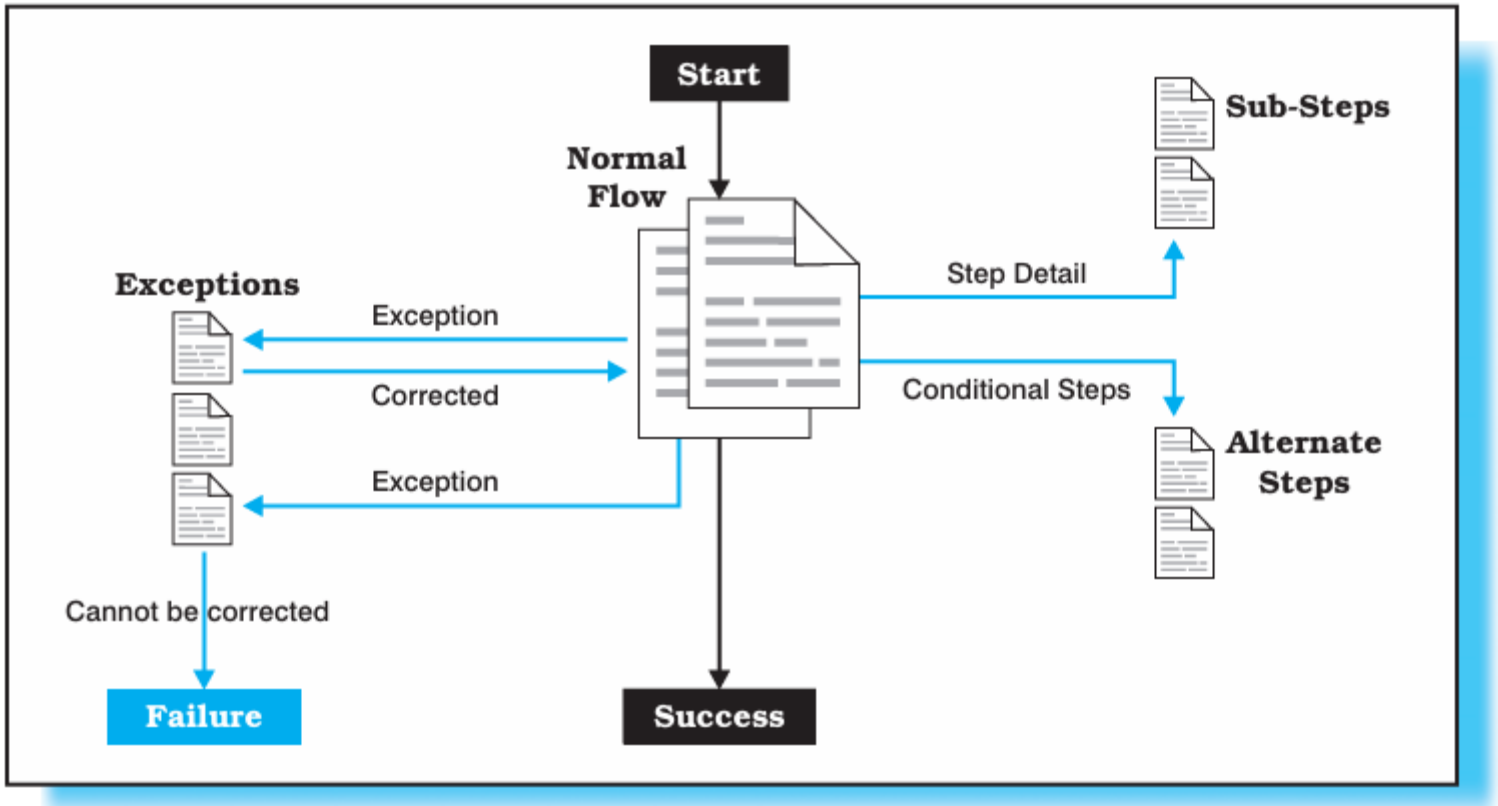


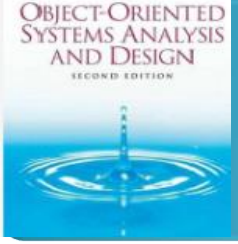
1. **Use case diagram**,

2. **Alternate Flow**. Alternate flow is composed of steps that are **conditional**: When a step is not part of the normal flow and is needed only if a condition is true or false, then it belongs to the **alternate flow.**
3. **Sub-Flows.**
   Sub-flows detail steps in the normal flow that consist of discrete sub-steps.
4. **Exceptions.**
   Exceptions are those events that prevent certain steps, or the entire use case, from completing successfully.

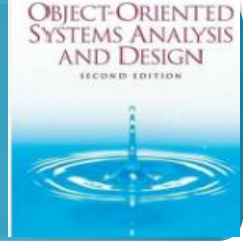
The most commonly used UML diagrams

1. **Use case diagram**,





The most commonly used UML diagrams



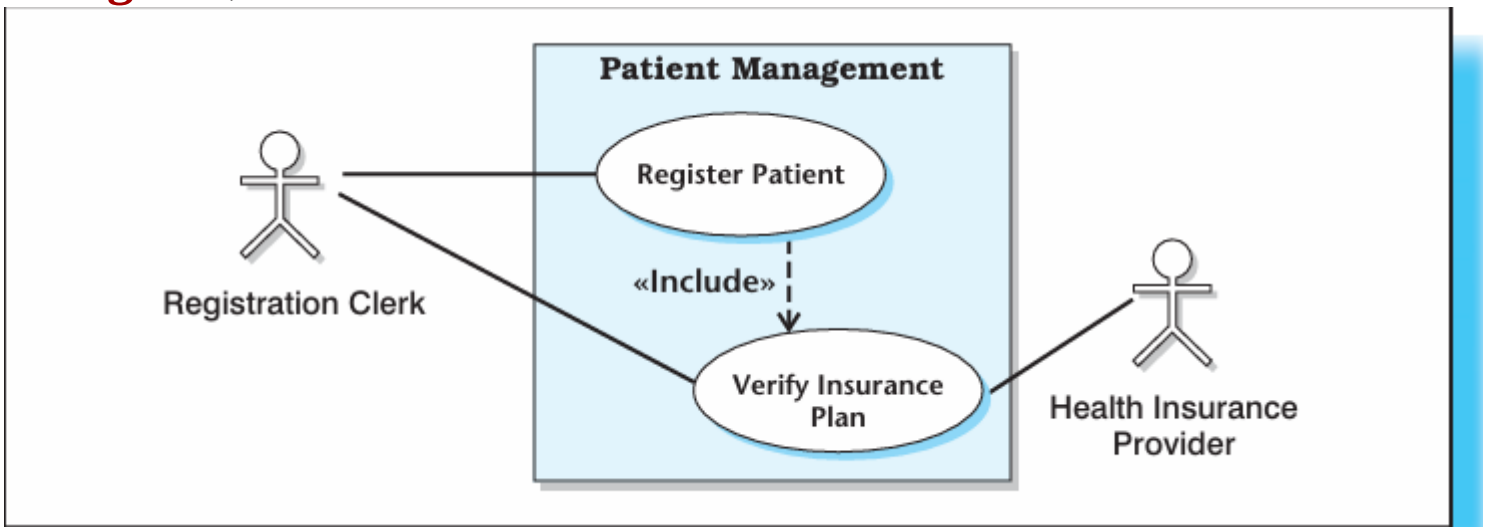
1. **Use case diagram**,

There are five types of relationships in a use case diagram:

1. **Association** between an actor and a use case
2. **Generalization** of an actor
3. **Extend** relationship between two use cases:- An extend relationship extends a use case by adding **new behaviors or action. I**t is shown as a dotted-line arrow pointing toward the use case that has been extended and labeled with the <<extends>> symbol.
4. **Include** relationship between two use cases:- which arises when one use case uses another use case. An include relationship also is shown diagrammatically as a dotted-line arrow pointed toward the use case that is being used. The line is labeled with the <<include>> symbol.
5. **Generalization** of a use case
6. **Precondition**: Required state of the system before the use case can start, if applicable.
7. **Post-Condition:** The state of the system resulting from the successful completion of the use case.

The most commonly used UML diagrams

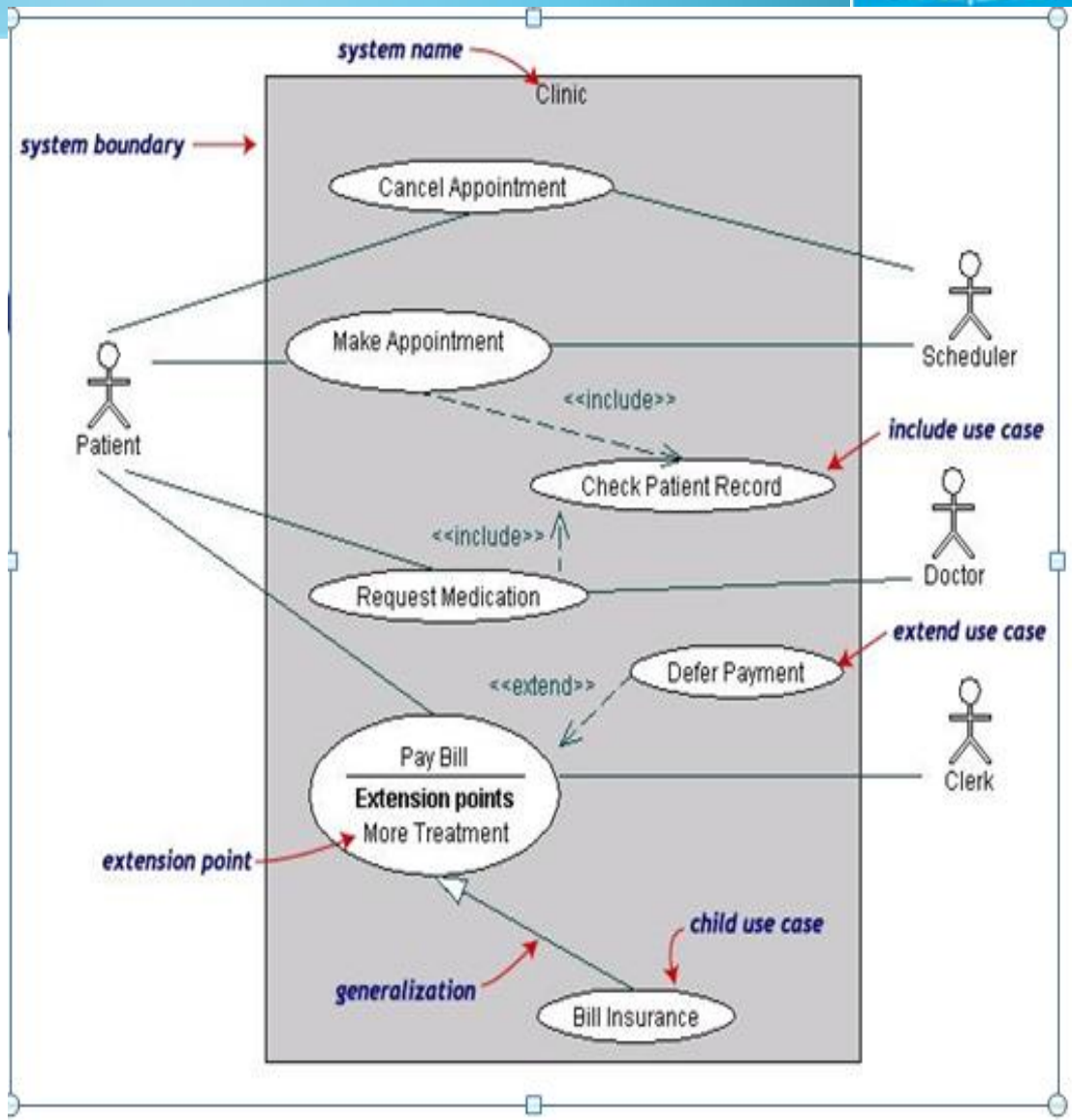
1. **Use case diagram**,



The most commonly used UML diagrams

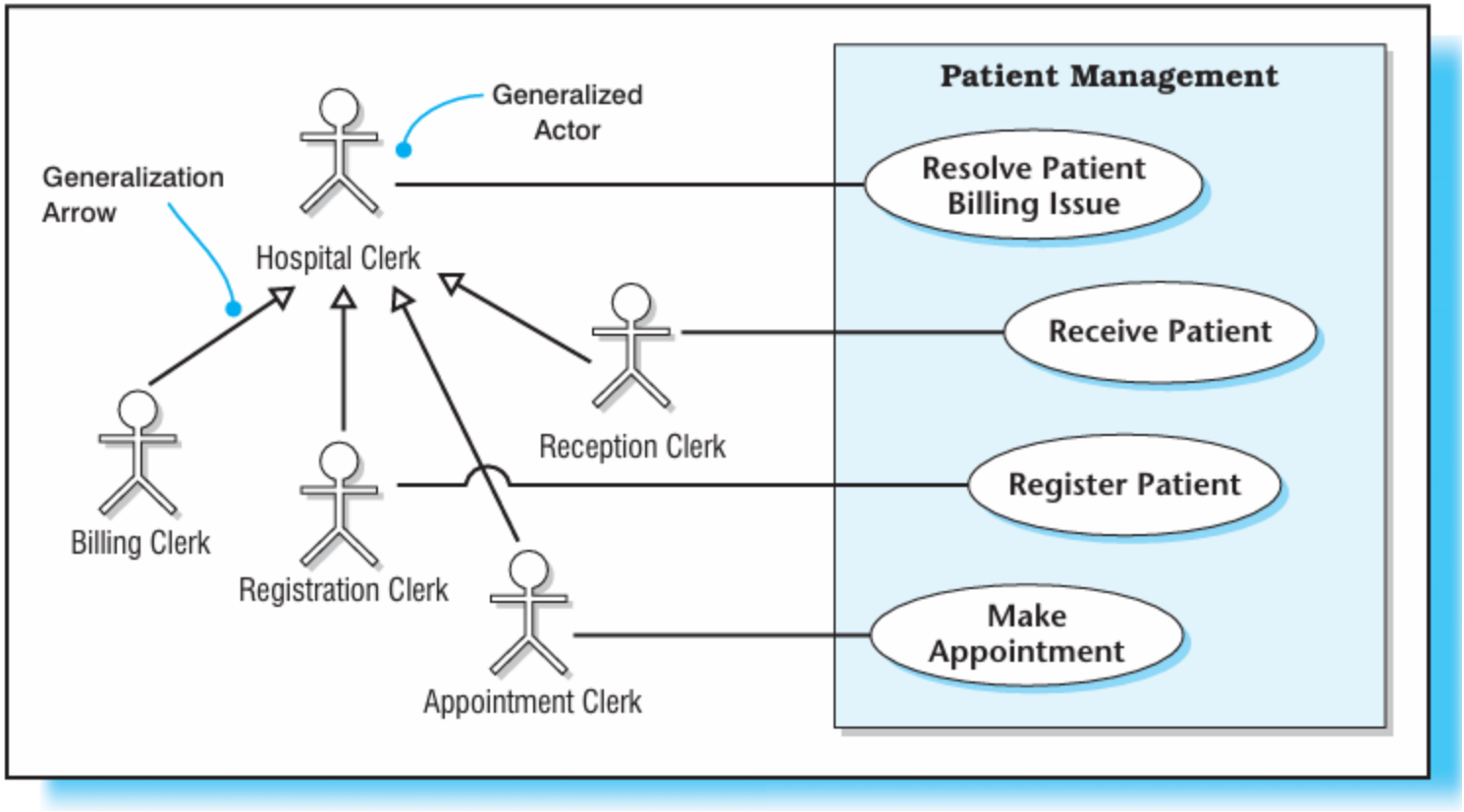
1. **Use case diagram**,

- 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)
- **Pay Bill** is a parent use case and **Bill Insurance** is the child use case. (generalization)



- **The most commonly used UML diagrams**

1. **Use case diagram (Generalization)**



➢ The most commonly used UML diagrams

Assignment

➢ Take a simplified system of the automatic teller machine (ATM).

The ATM offers the following services:

1. Distribution of money to every holder of a smartcard via a card reader and a cash dispenser.
2. Consultation of account balance, cash and cheque deposit facilities for bank customers who hold a smartcard from their bank. Do not forget either that:
3. All transactions are made secure.

- **The most commonly used UML diagrams**

Assignment

4. It is sometimes necessary to refill the dispenser, etc. From these four sentences, we will work through the following activities:

   i. Identify the actors,

   ii. Identify the use cases,

   iii. Construct a use case diagram,

   iv. Write a textual description of the use cases,

   v. Complete the descriptions with dynamic diagrams,

   vi. Organise and structure the use cases.

➢ The most commonly used UML diagrams

2. Activity diagram

- **Activity diagrams:-** are operation upon the states of an object that requires some time period.
- **An activity diagram** describes the business and operational step-by-step workflows of components in a system.
- **An activity diagram** shows the overall flow of control
- **Activities** are shown in activity diagrams that portray **the flow from one activity to another.**
- **An action** is an **atomic** operation that executes as a result of certain events.

➢ The most commonly used UML diagrams

2. Activity diagram

- By **atomic**, it is meant that **actions are un-interruptible**,
- i.e., **if an action starts executing**, it runs into completion **without being interrupted by any event**.
- **An action** may operate upon an object on which an event has been triggered or on other objects that are visible to this object.
- **A set of actions comprise an activity**.

➢ The most commonly used UML diagrams

2. Activity diagram

- **Entry and Exit Actions**
- **Entry action** is the **action that is executed on entering a state**, irrespective of the transition that led into it.
- **Exit action is** the action that is executed while leaving a state, irrespective of the transition that led out of it.
- **Scenario** is a description of a specified **sequence of actions**. It depicts the behavior of objects undergoing a specific action series.
- **The primary scenarios** depict the **essential sequences** and the **secondary scenarios** depict the **alternative sequences**.

➢ The most commonly used UML diagrams

2. Activity diagram

- **Activity diagram** is the tool that UML provides for business process modeling.
- **Activity diagram** is a tool that assists us in navigating complicated logical flows.
- **An activity diagram** depicts the flow from activity to activity. It presents a visual, dynamic view of the system and its components.
- **A use case is a flow of activity in words**. If the flow is simple and linear, it can be understood on its own.

➢ The most commonly used UML diagrams

2. Activity diagram

- ❖ **A use case diagram**, contrary to the expectations that its name raises, does nothing to help us in **visualizing what goes inside a use case.**
- ❖ **Each use case may create one activity diagram** based on its complexity.
- ❖ Its strong point is to provide a view from outside, between actors and a set of use cases and between use cases themselves.
- ❖ **A use case has to make assumptions.** If it does not, the explanations and the descriptions would overwhelm the flow.
- ❖ **The question is: How safe is an assumption?**

➢ The most commonly used UML diagrams

2. Activity diagram

- If the flow is **complex and iterative**, words will not suffice.
- In a model-driven development process, however, **no one model has to stand alone.**
- Where words fail us, **an activity diagram can paint a clearer picture**.
- Activity diagram is **a flexible tool and its application is not limited to use cases.**
- It can be attached **to any dynamic model or to any operation whose flow needs clarification.**

➢ The most commonly used UML diagrams

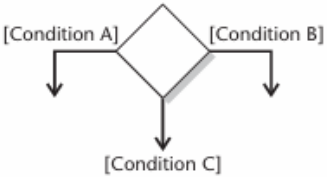
2. Activity diagram

- If the flow of a use case is too complex for one clear activity diagram, we may use the technique of hiding a set of activities under a label and expanding it in another diagram.
- **UML's activity diagram** is based on a flow chart that has been around for a long time.
- Its longevity is a testament to its effectiveness.

- **The most commonly used UML diagrams**

2. Activity diagram

The Building Blocks of an Activity Diagram

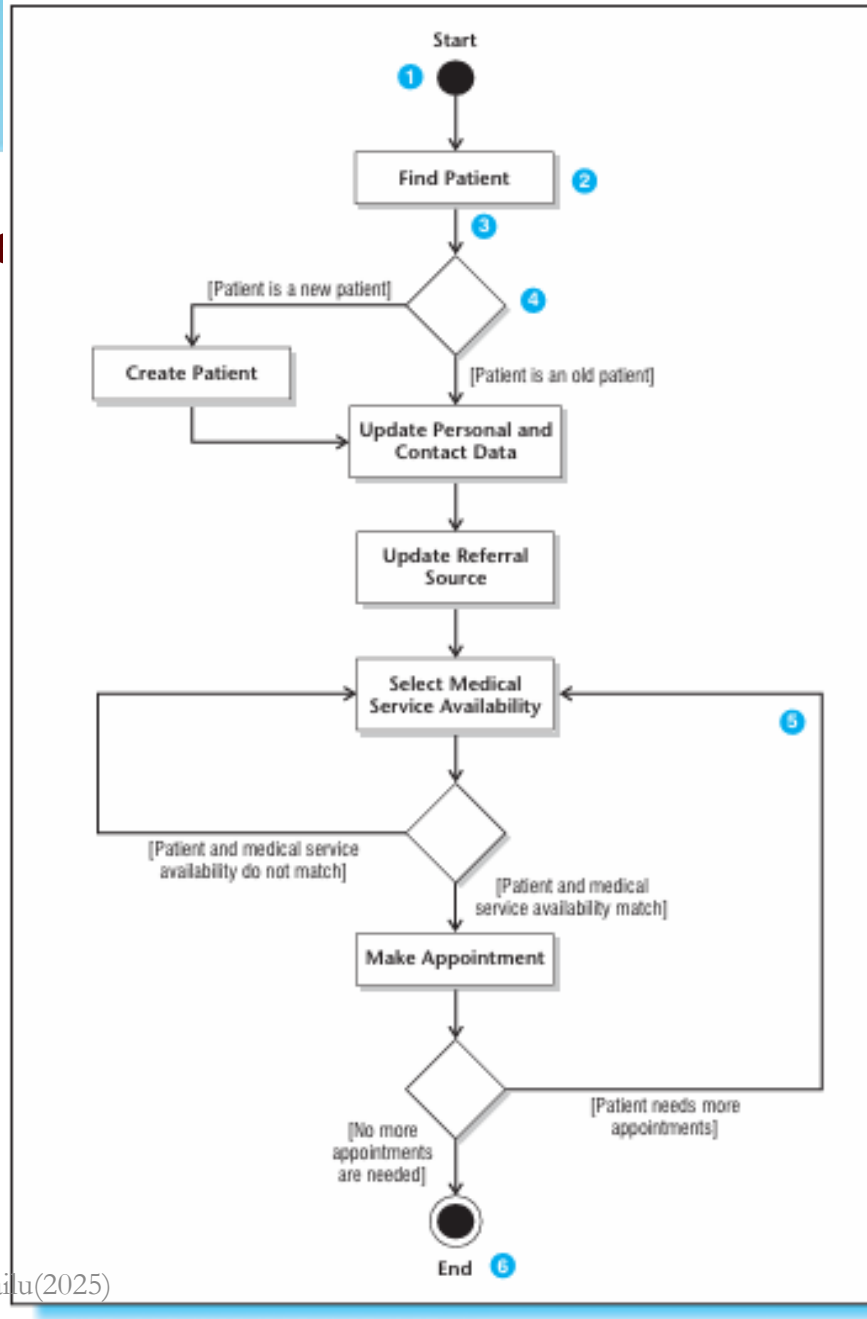
❶ 	**Start State.** Corresponds to **precondition** in use cases. An activity diagram may have only one starting point.
❷ 	**Activity.** The same as an activity (or a ***step***) in a use case, identified by a short label in active voice. You may also include the step number (e.g., 5 for normal flow, 4.2 for sub-flow, or 5.b for an alternate) in the activity label to make it easier to match diagram activities with those in a use case. An activity can be a placeholder for an included or an extending use case.
❸ 	**Transition.** The arrow identifies the transition of one activity to another.
❹ 	**Branching.** The diamond is a ***decision point***. Depending on the outcome of a logical test, the action branches to two or three activities or new decision points. The branches are labeled (between square brackets) with the condition that causes them. A branch usually—but not always—matches an alternate activity. (Sometimes one branch belongs to the activity in the normal flow.)
❺ 	**Looping.** Iteration or looping is presented by a transition arrow that doubles back from a decision point (a diamond) to an earlier activity.
❻ 	**End.** Normally, this icon would end in the post-condition of the use case. But an activity diagram can have more than one end, as does the use case: Exceptions can cancel the entire process and result in *abnormal* endings. Abnormal endings should be displayed by a separate end icon.

- **The most commonly used UML**

2. Activity diagram

- **Example 1: The Logical Flow of Make Appointment**

- In many ways **UML activity diagrams** are the **object-oriented equivalent of flowcharts and Data Flow diagrams** from **structured development**.

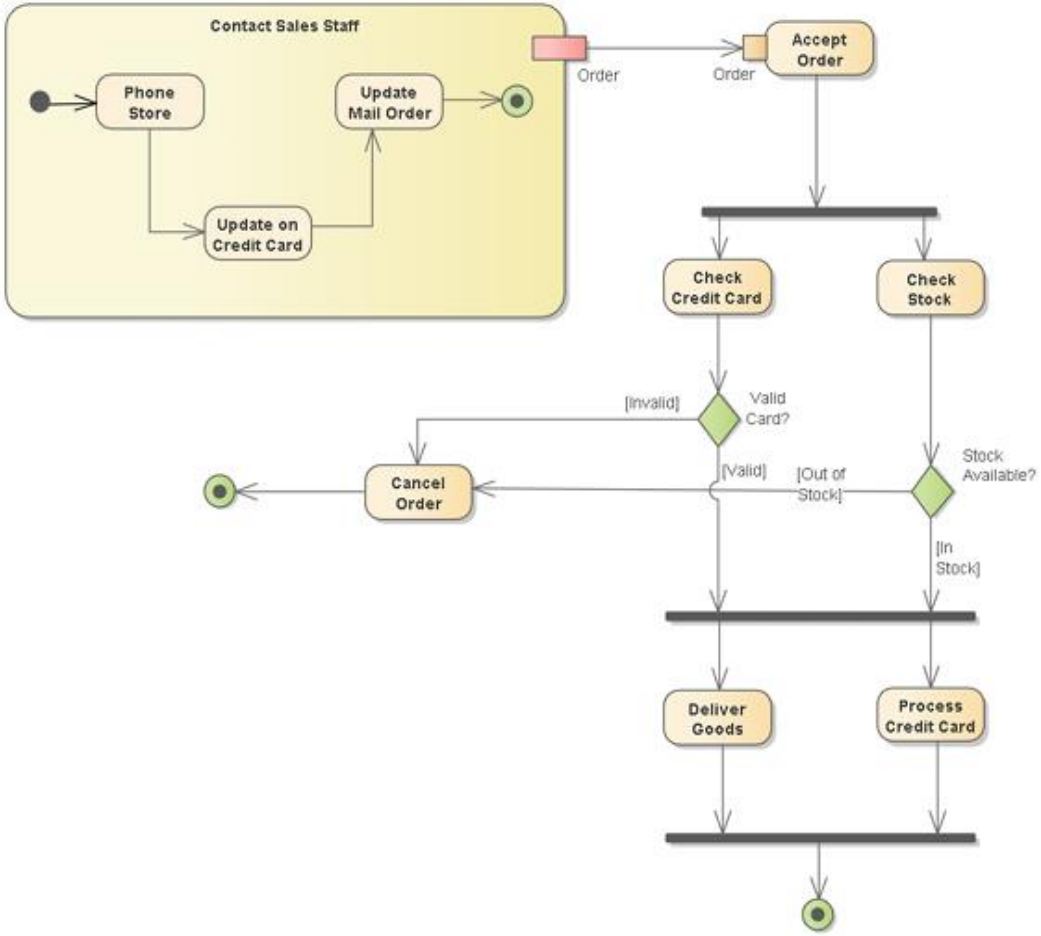


➢ The most commonly used UML diagrams

2. Activity diagram

Example 2:

☞This diagram illustrates some of the features of Activity diagrams, including **Activities, Actions, Start Nodes, End Nodes and Decision points.**

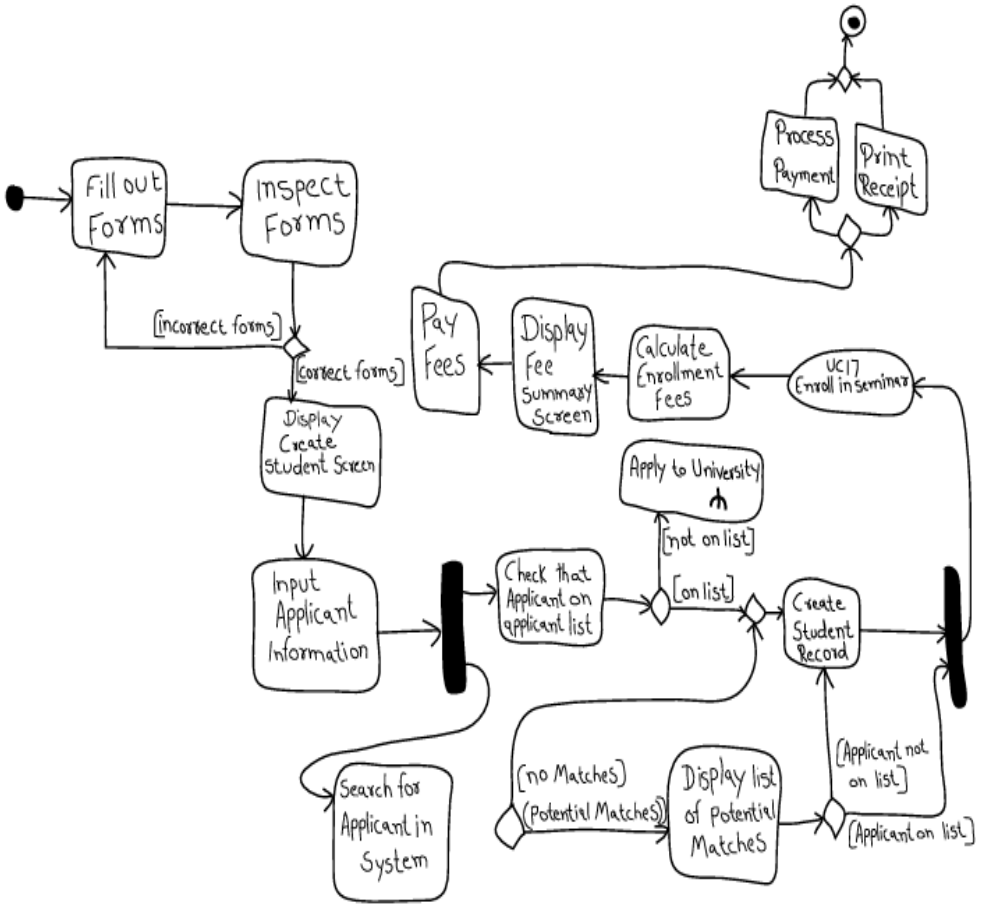


➢ The most commonly used UML diagrams

2. Activity diagram

Example 3:

☞This diagram illustrates A UML activity diagram for enrolling in university for the first time.



- **The most commonly used UML diagrams**

2. Activity diagram

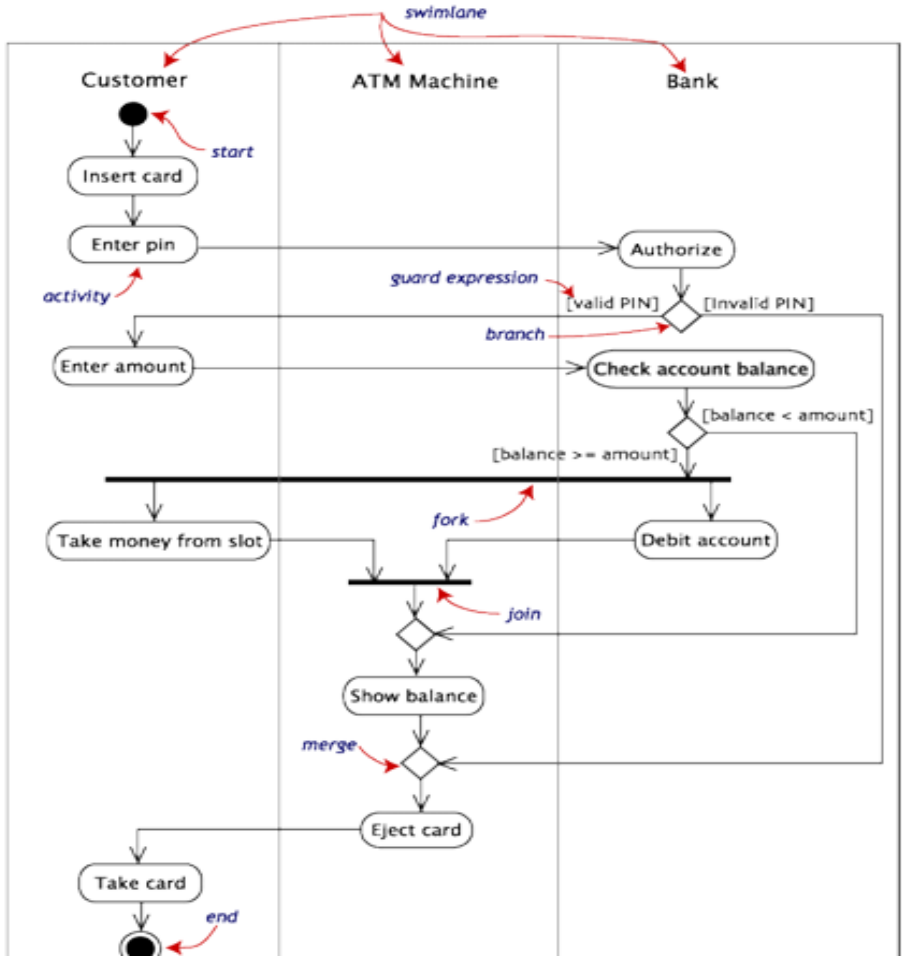
☞ Assignment 2:

☞ **Draw an activity Diagram for the case of assignment One (use case diagram) considering Customer, ATM machine and Bank as objects, Customer as a starting point and end point in the process of using ATM machine.**

- **The most commonly used UML diagrams**

2. Activity diagram

☞ **Solution for the Assignment 2**



➢ The most commonly used UML diagrams

3. Sequence diagram

- **Interaction diagrams** are used for modeling:
  - the control flow by time ordering using **sequence diagrams.**
  - the control flow of organization using **collaboration diagrams.**
- **Sequence diagram,** showing the sequence of activities and class relationships.
  - **Each use case may create one or more sequence diagrams**.
- It shows **how objects communicate with each other in terms of a sequence of messages.**

➢ The most commonly used UML diagrams

3. Sequence diagram

- Also indicates the **life span of objects relative to those messages.**
- It represents the temporal ordering of messages in a tabular manner.
- **Sequence diagrams** are **interaction diagrams** that illustrate the ordering of messages according to time.
- **A Sequence diagram** is a structured representation of behavior as a series of sequential steps over time.

➢ The most commonly used UML diagrams

3. Sequence diagram

❖ You can use it to:

1. **Depict workflow**, Message passing and how elements in general cooperate over time to achieve a result
2. **Capture the flow of information and responsibility** throughout the system, early in analysis; Messages between elements eventually become method calls in the Class model
3. **Make explanatory models for Use Case scenarios**; by creating a Sequence diagram with an Actor and elements involved in the Use Case, you can model the sequence of steps the user and the system undertake to complete the required tasks

➢ The most commonly used UML diagrams

3. Sequence diagram

❖ **A sequence diagram** is composed of **a timeline, objects that interact across this timeline, and the messages** that they exchange.

- **A timeline**
- **Objects**
- **Messages**

The Building Blocks of Sequence Diagrams

- **The most commonly used UML diagrams**

3. Sequence diagram

- **Notations**: These diagrams are in the form of **two-dimensional charts**. **The objects that initiate the interaction are placed on the x–axis. The messages that these objects send and receive are placed along the y–axis**, in the order of increasing time from top to bottom.
- **One primary advantage of the sequence diagram** is that it allows us to gradually and smoothly move from **"problem domain" into "system domain"** by discovering actions and objects that enable the system to support its behavior and properties.

➢ The most commonly used UML diagrams

3. Sequence diagram

- There are 5 kinds of actions that the UML explicitly supports: -
  - **Call a**nd **Return**
  - **Create** and **Destroy**
  - **Send**

➢ The most commonly used UML diagrams

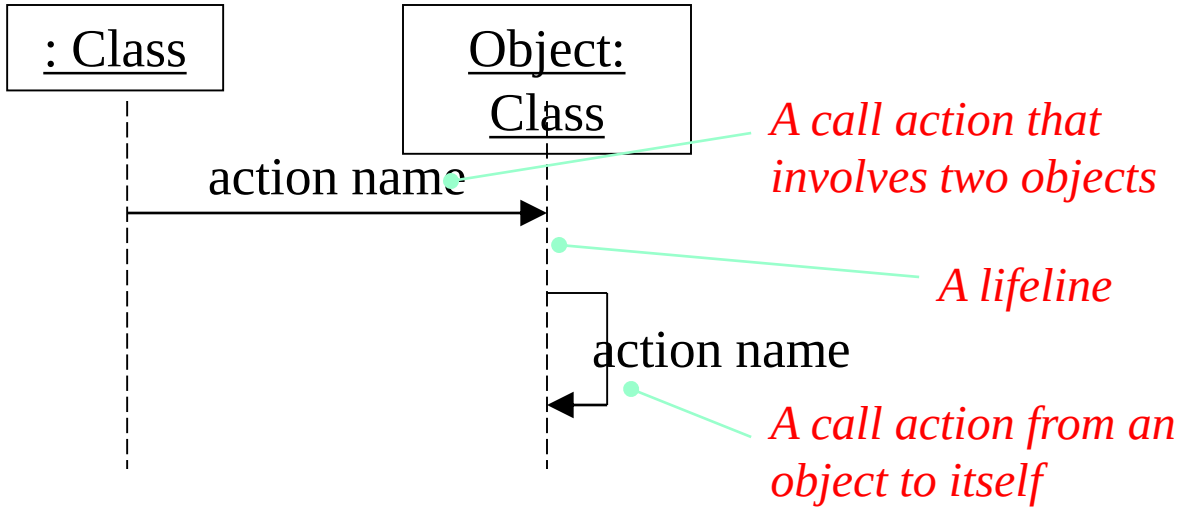
3. Sequence diagram

❖ **Sequence--Diagram Call Action**

- **A call action** invokes an operation on an object
- It is synchronous, meaning that
  - the sender expects that the receiver is ready to accept the message,
  - the sender waits for a response from the receiver before proceeding
- The UML represents **a call action** as an arrow **from the calling object to the receiving object**

❖ Sequence--Diagram Call Action

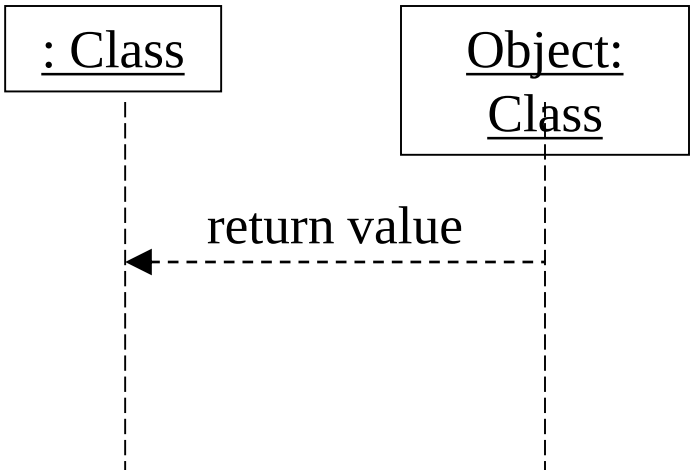
- **A call action** invokes an operation on an object
- It is synchronous, meaning that
  - the sender expects that the receiver is ready to accept the message,
  - the sender waits for a response from the receiver before proceeding
- The UML represents **a call action** as an arrow **from the calling object to the receiving object**



❖ Sequence Diagram – Return Value

➢ Return action

- A return action is the return of a value to the caller, in response to a call action
- The UML represents a return action as a dashed arrow from the object returning the value to the object receiving the value



Dinner Sequence Diagram

- In developing a sequence diagram, and consequently distributing the responsibilities, the following guidelines need to be considered:

1. Employ the use-case diagram to determine the use cases. Determine the actors that interact with the use case. Do not model interactions among actors. Actors are outside the scope of the system.
2. Consider one use case at a time. Obtain the text-based, detailed description of each use case.
3. For a given use case, specify an analysis boundary class between the use case and each actor. Thus, if there are two actors, there will be two boundary classes.

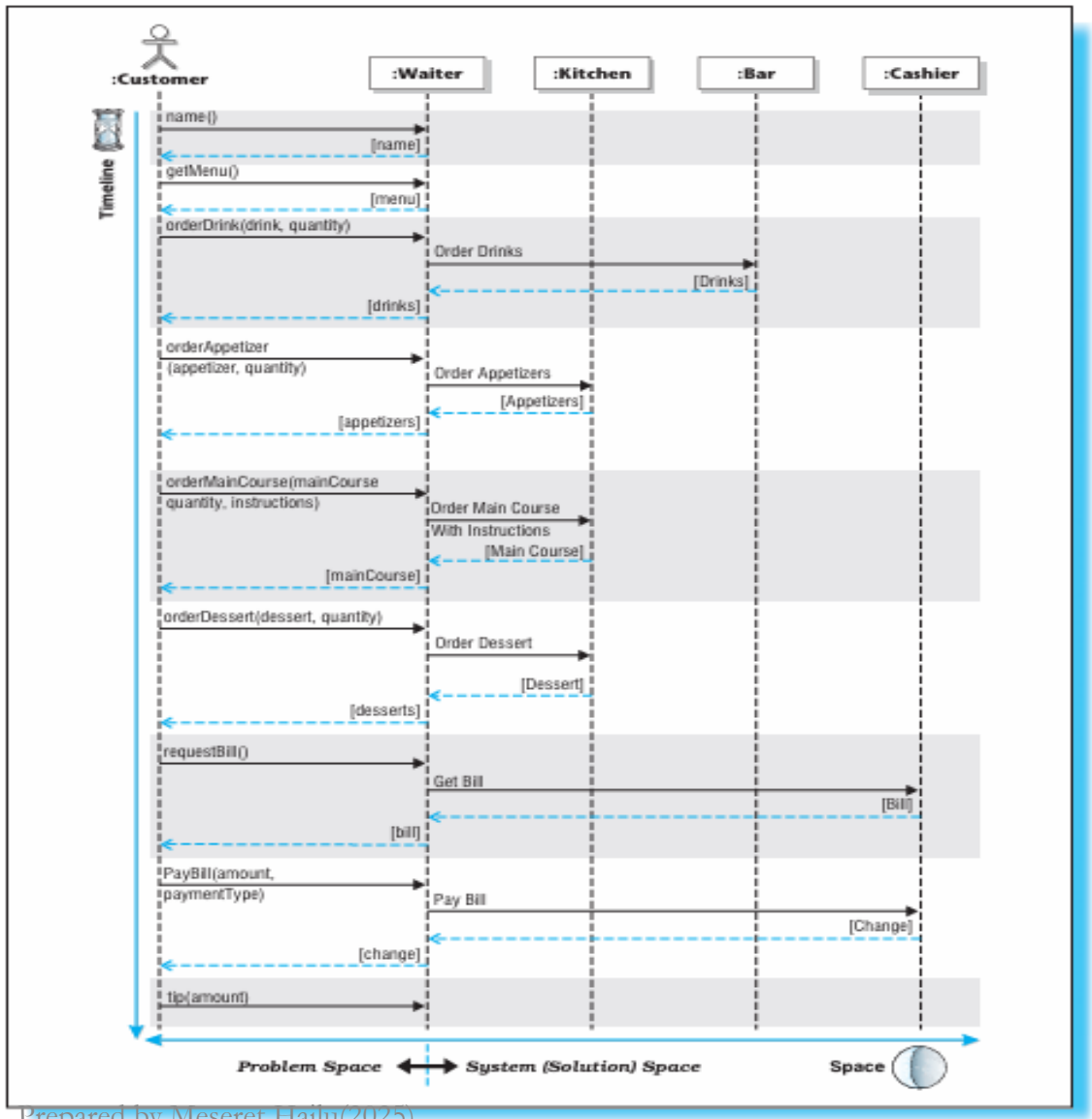
- **Dinner Sequence Diagram**

4. For a given use case, specify one control class.
5. Derive the entity classes from object relations for the use case.
6. Interactions in a sequence diagram are among objects of the classes, not among the classes themselves. You should use **the notation :Customer** to indicate an object of the class Customer.

OO concepts from structured point of view /Objects and Classes

OBJECT-ORIENTED SYSTEMS ANALYSIS AND DESIGN SECOND EDITION

- **Dinner Sequence Diagram**



- **The most commonly used UML diagrams**

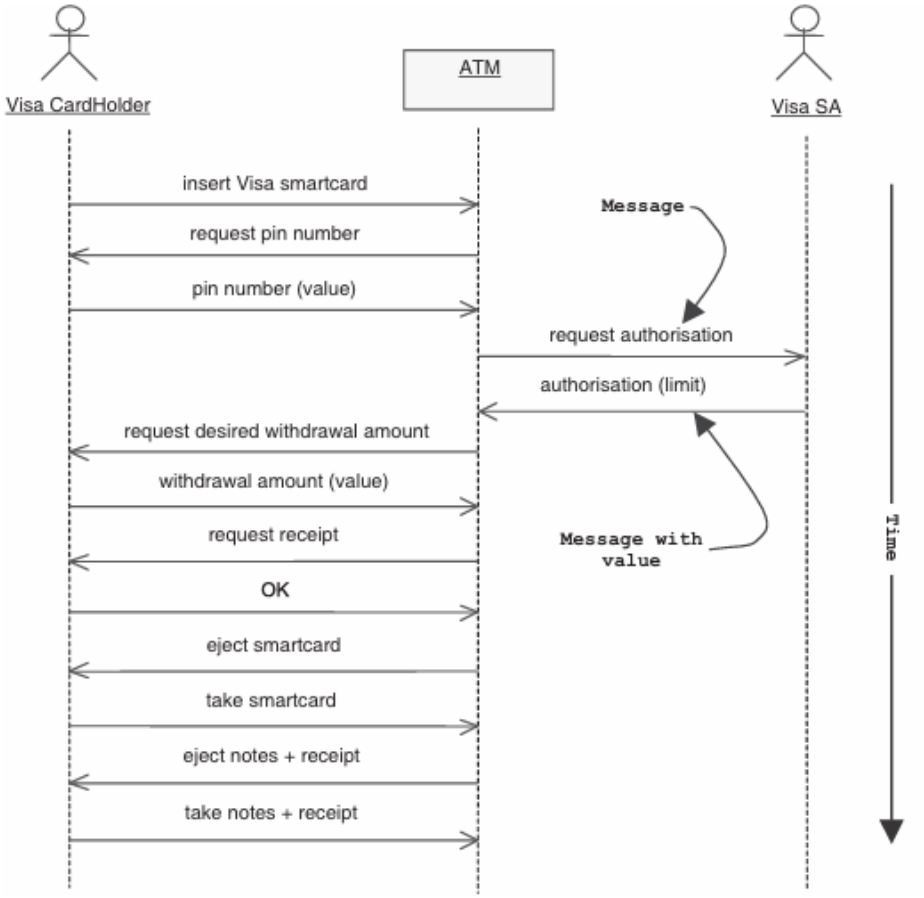
3. Sequence diagram

- **Assignment 3**
- Create a system sequence diagram that describes the main success scenario of the Withdraw money using a Visa card use case.

- **The most commonly used UML diagrams**

3. Sequence diagram

- **Solution for Assignment 3**



- **The most commonly used UML diagrams.**

4. A collaboration diagram

- **A collaboration diagram** is an alternative to a sequence diagram. We will consider Collaboration diagram in Chapter 6

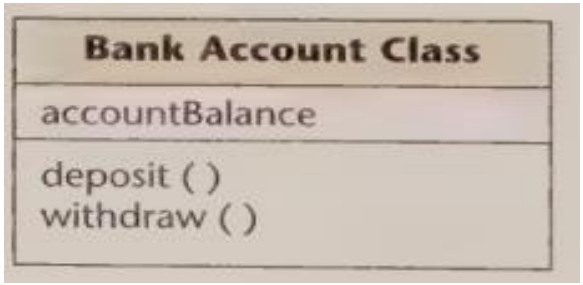
5. Class Diagram:-

- **A class diagram** depicts classes and their interrelationships.
- The Class Diagram of the figure below with an **Attribute** and Two **Operations** Added
- **Class Diagram: -** Describes the structure of a system by showing the system's classes, their attributes, methods and the relationships among

- **The most commonly used UML diagrams.**

5. Class Diagram:-

- **A class diagram** depicts classes and their interrelationships.
- The Class Diagram of the figure below with an **Attribute** and Two **Operations** Added
- **Class Diagram: -** Describes the structure of a system by showing the **system's classes, their attributes, methods and the relationships among the classes**



- **The most commonly used UML diagrams.**

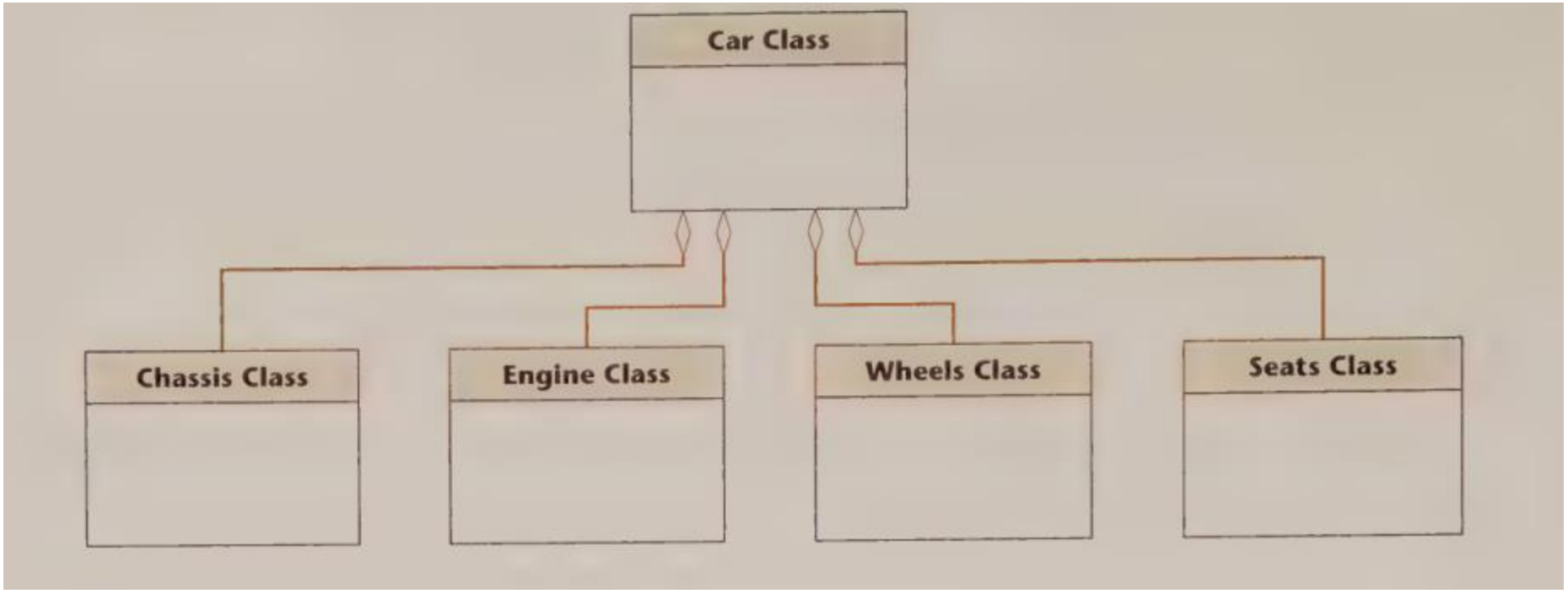
5. Class Diagram:-

- **bank account : Bank Account Class**
- In the above expression **bank account** is an **object**, **an instance** of a **class Bank Account Class**.
- In more detail, the underlining denotes that we are dealing with an object, **the colon denotes "an instance of,"** and the bold face and initial uppercase letters in **Bank Account Class denote this is a class.**
- **: Bank Account Class** means "an instance of class Bank Account Class,"

- **The most commonly used UML diagrams.**

5. Class Diagram:-

- **An Aggregation Example**



➢ The most commonly used UML diagrams.

5. Class Diagram:-

- ❖ Among all UML diagrams, **the class diagram** is the most indispensable.
- ❖ We use **sequence, state chart, and activity diagrams to discover and/or refine classes,** but **class diagrams** are **the definitive blueprints** from which developers must build the final product.
- ❖ **A class diagram** shows a set of classes and their associations. The associations can be presented from **multiple viewpoints** and every viewpoint can be used **to model the user interface**:

- **The most commonly used UML diagrams.**

5. Class Diagram:-

- **Relationship**, or the nature of the connection between classes.
- For example, the Find Patients form displays the Patient Profile form to display detailed information about a patient. In fact, to a modest degree, class diagrams can be used **to model navigation.**
- **Multiplicity**, or how many instances of one class can associate with instances of another class.
- For example, the Find Patients UI needs multiple instances of the Dialog Box class to display messages and exceptions.

- **The most commonly used UML diagrams.**

5. Class Diagram:-

- In addition, since Find Patients might display multiple patients, its relationship with forms such as Patient Profile and Appointments is one-to-many. **(The "many," however, is not limitless: The resources of a workstation are limited and developers usually limit the number of forms that can be displayed at the same time.)**
- **Aggregation and composition**, which represent the relationship of whole to its parts.

- **The most commonly used UML diagrams.**

5. Class Diagram:-

- Even the simplest form, such as a dialog box that displays error messages, is composed of multiple objects: a displayer, an icon perhaps, and a button to close the form.
- **Generalization**, or the abstraction of common elements shared by a set of classes into a super class.

- **The most commonly used UML diagrams.**

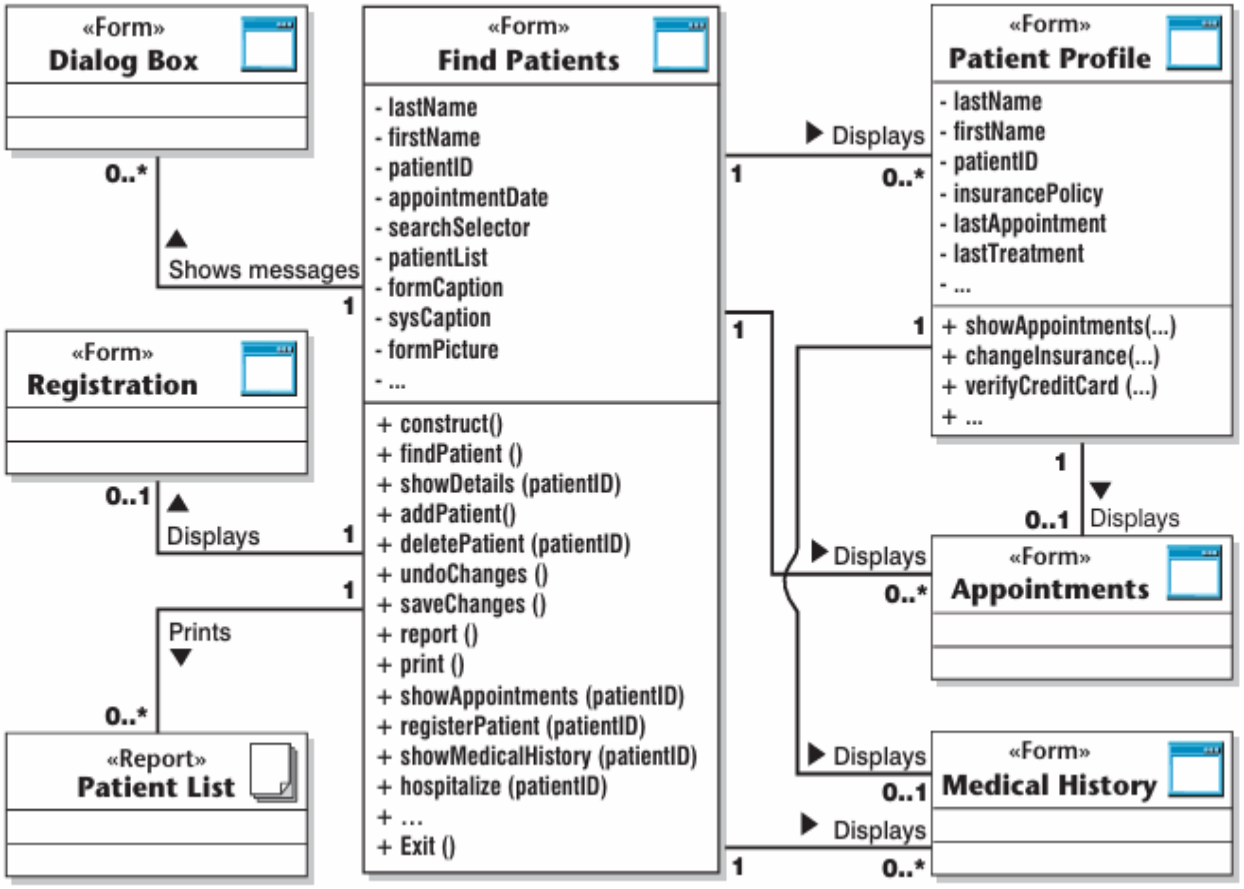
5. Class Diagram:-

- **Specialization**, or the creation of a **subclass** from an existing class by defining elements that are too specific for the parent class.
- Modern development tools allow developers to inherit general purpose components and specialize them, or create templates that can be reused to provide common functionality and harmonious appearance across the application

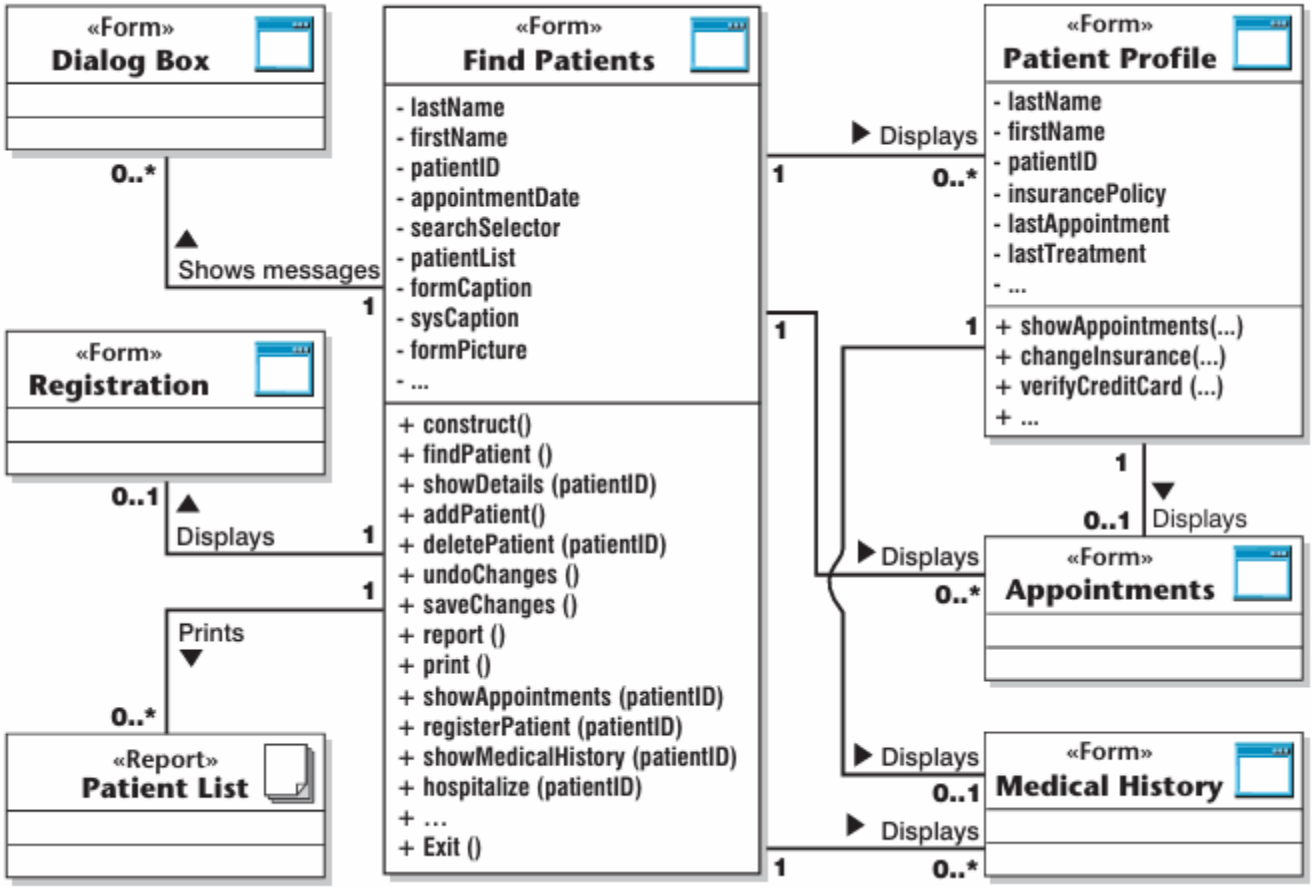
- **The most commonly used UML diagrams.**

5. Class Diagram:-

- **Examples of system Class Diagram**
- The UI Class Diagram:
- Modeling the Structure of the User Interface

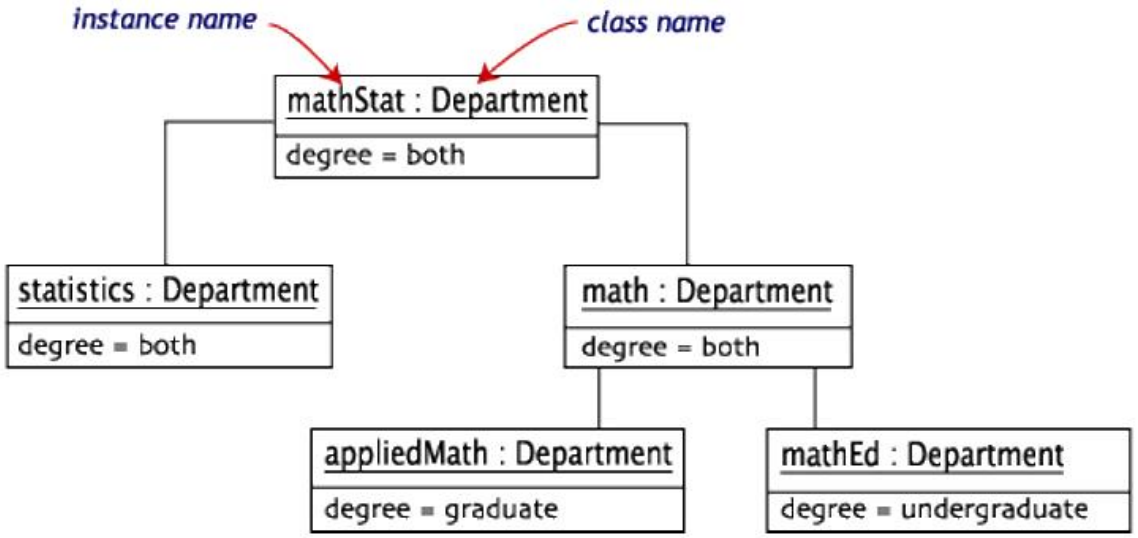


5. Class Diagram

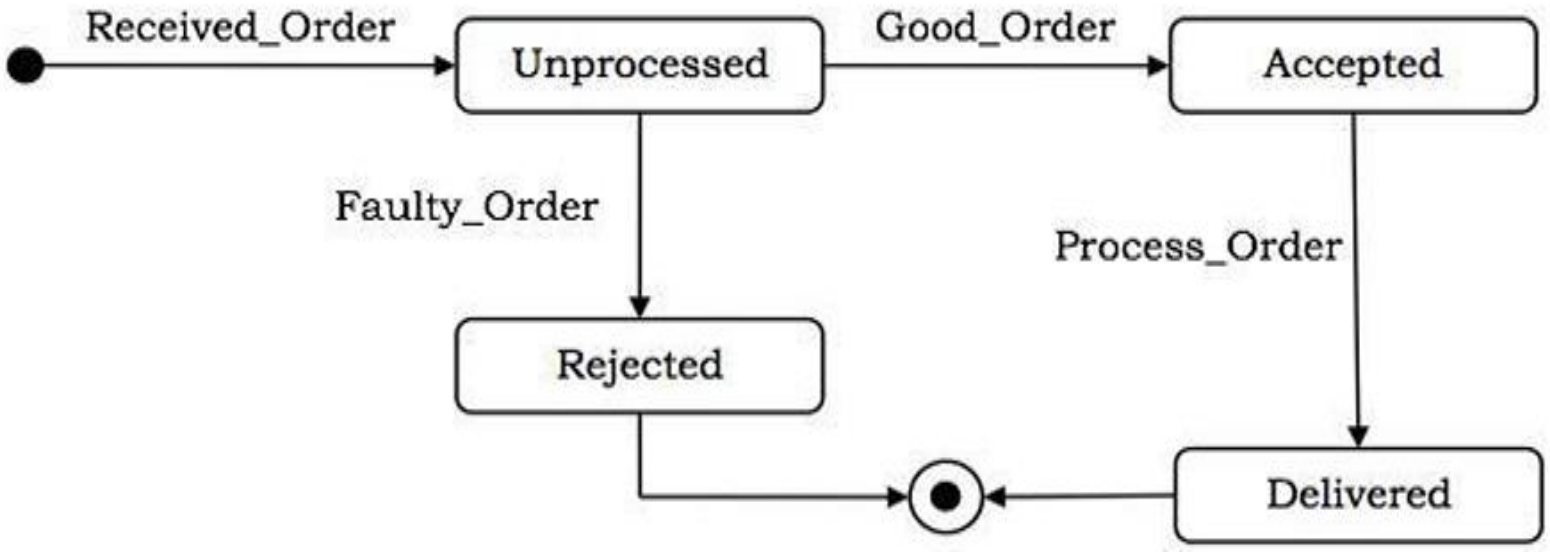


➢**Example 2:** **System Class diagram for Patient management System**

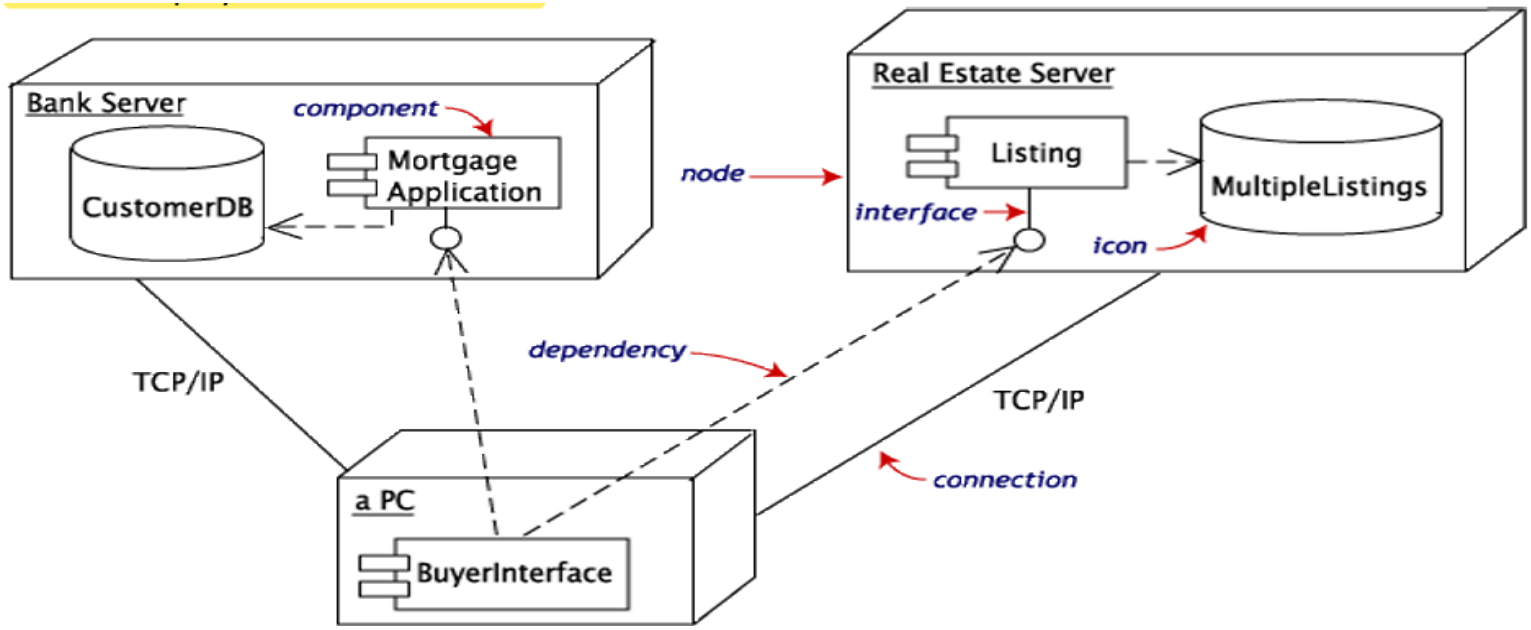
1. **An object diagram** shows a set of objects and the relationships among the objects. An object diagram is used to illustrate a specific instance of a class diagram.



2. **A state diagram** shows the transitioning of an object from state to state in response to various events. A state diagram captures the dynamic aspects of a system



3. **A deployment diagram** shows the configuration of run-time processing nodes and the components that live on them. It describes the hardware used in system implementations and the execution environments and artifacts deployed on the hardware

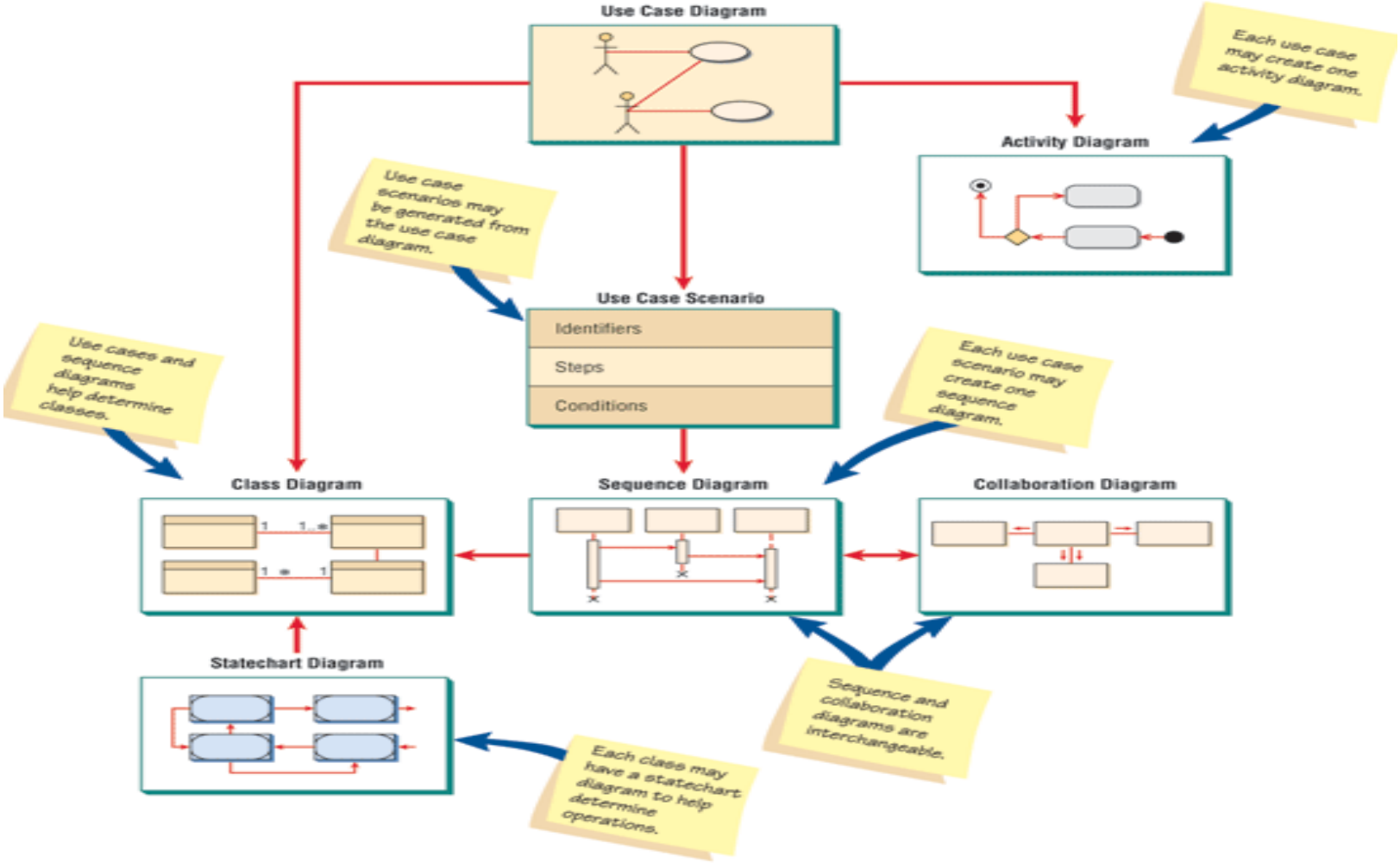


4. **A composite structure diagram** is used to show how a composite whole is made from its parts
5. **A package diagram** is used to show logically grouped analysis or design elements.
6. **A component diagram** shows the software components or modules and their dependencies.

OO concepts from structured point of view /Objects and Classes

OBJECT-ORIENTED SYSTEMS ANALYSIS AND DESIGN
SECOND EDITION

➢ Overview of UML Diagrams

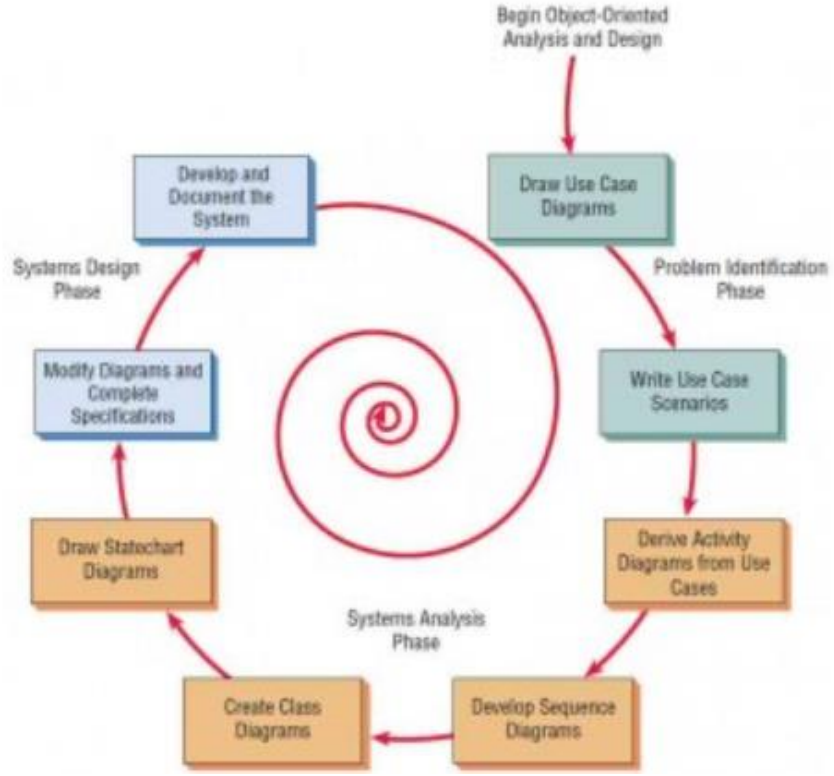


Assignment

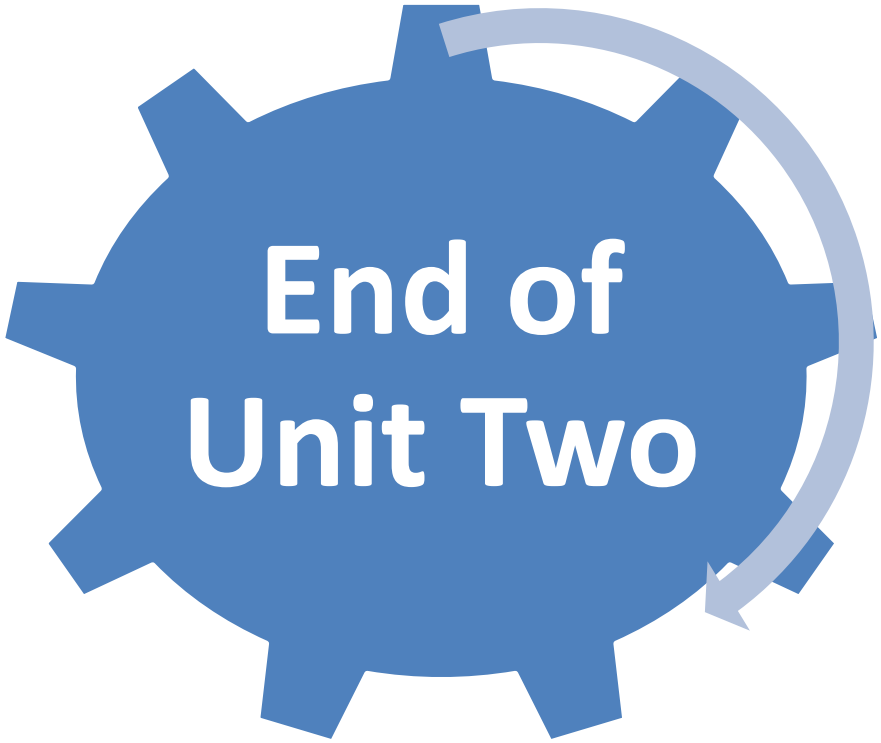
Reading and workout Assignment

1. What are the four fundamental principles of the Unified Process?
2. What is the goal of each of the four phases of the Unified Process?
3. How do the phases of UP differ from waterfall's phases?
4. Draw a UML Class Diagram for Online Shopping system.
5. Draw a UML Class diagram in the System having the classes **Bank, Branch, Account, Savings Account, Current Account, Loan, and Customer. Think about the natural relation ships of the classes and their multiplicity.**

Modern Development Technologies



Steps in the UML development process


End of
Unit Two