PS02CMCA32 Software Engineering

Unit 3

Software Design

The design of a system is a plan for a solution such that if the plan is implemented, the implemented system will satisfy the requirements of the system and will preserve its architecture. The design activity is a two-level process. The first level produces the system design which defines the modules needed for the system, and how the components interact with each other. The detailed design refines the system design, by providing more description of the processing logic of components and data structures. A design methodology is a systematic approach to creating a design. Most design methodologies concentrate on system design. During system design a module view of the system is developed, which should be consistent with the component view created during architecture design.

Design approaches

- Function-oriented design
- Object-oriented design

Function-Oriented Design

Design principles

- Problem partitioning and hierarchy
- Abstraction
- Modularity
- Top-down and bottom-up strategies

Module-level concepts

- Coupling
- Cohesion

Criteria for Quality of Software Design

Thumb Rules for design Evaluation

Quality Criteria

- **Correctness**
- **Completeness** – Implements all the specifications
- **Verifiable** design
- **Traceability** – All design elements can be traced to some requirements
- **Efficiency** – Concerned with the proper use of scarce/expensive resources
- **Simplicity / Understandability**- Produce designs that are simple to understand
- **Maintainability** – Requires thorough understanding of different modules and their interconnectivities
- **Best possible design** – There can be many correct designs. Select the best one.

Design Principles

Problem Partitioning and Hierarchy

- Solving large/complex problems
- "Divide and conquer"
- Divide into smaller pieces, so that each piece can be conquered separately.
- Goal : Divide the problem into manageably small pieces that can be solved separately.
- Basic rationale / belief : If the pieces of a problem are solvable separately, the cost of solving the entire problem is more than the sum of the costs of solving all the pieces.
- As the #components increases, the cost of partitioning also increases and the cost of added complexity increases.
- When the cost of added complexity is greater than the savings achieved by partitioning, we should stop partitioning.

Abstraction

- An abstraction of a component describes the external behavior of that component without worrying about the internal details that produce the behavior.
- A tool that permits the designer to consider a component at an abstract level without worrying about the details of implementation of the component.
- Any component / system – provides some services to its environment.
- Allows the designer to concentrate on one component at a time using the abstraction of other components.

Modularity

- A system is considered modular if it consists of discrete components so that each component can be implemented separately and a change to one component has minimal impact on other components.
- Modularity helps in system debugging – Isolating a system problem to a component is easier if the system is modular.
- Modularity helps in system building – A modular system can be easily built by putting its modules together.
- A system is partitioned into modules so that **modules are**
  - solvable separately
  - modifiable separately
  - compilable separately
  - implementable separately.

- Software is divided into separately named and addressable components (modules)

- Why do we prefer modularization ?

* To ease planning for implementation (coding)
* To easily accommodate changes
* To effectively test and debug programs
* To conduct long-term maintenance without serious side-effects

Module-Level Concepts

Coupling and Cohesion

Coupling :

- Coupling between modules is the strength of interconnections between modules or a measure of interdependence among modules.
- In general, the more we must know about module A in order to understand module B, the more closely connected A is to B.
- "Highly coupled" modules are joined by strong interconnections.
- "Loosely coupled" modules have weak interconnections.
- Independent modules have no interconnections.
- **To solve and modify a module separately, we would like a module to be loosely coupled with other modules.**
- The coupling between modules is largely decided during system design.
- Coupling increases with the complexity and obscurity of the interface between modules.

Factors affecting coupling

- Interface complexity
- Type of connection
- Type of communication

Coupling	**Interface complexity**	**Type of connection**	**Type of communication**
Low	Simple/Obvious	To module By name	Data Control
High	Complicated /Obscure	To internal elements	Hybrid

- Coupling is reduced when elements in different modules have little or no bonds between them.
- Coupling can be reduced if relationships among elements in different modules is minimized.
- Coupling increases if a module is used by other modules via indirect/obscure interface – like using shared variables.

- Complexity of an interface : refers to the #items being passed as parameters and the complexity of items.
- An interface is used to support communication between modules.

Cohesion of a module:

- Cohesion of a module refers to the relationship between elements of the same module. It indicates how closely the elements of a module are related to each other.
- Cohesion of a module represents how tightly bound the internal elements of a module are to one another.
- It is possible to strengthen the bonds between elements of the same module, by maximizing the relationship between elements of the same module.
- Cohesion and coupling are related.
- Usually, the greater the cohesion of each module, the lower the coupling between modules.

Types of Cohesion

- **Functional Cohesion:** Functional cohesion is said to exist, if different elements of a module cooperate to achieve a single function. For example, a module containing all the functions required to manage employees' payroll displays functional cohesion.
- **Sequential Cohesion:** An element outputs some data that becomes the input for other element, i.e., data flow between the parts. It occurs naturally in functional programming languages.
- **Communicational Cohesion:** A module is said to have communicational cohesion, if all the functions of the module refer to or update the same data structure, e.g., the set of functions defined on an array or a stack.
- **Procedural Cohesion:** Elements of procedural cohesion ensure the order of execution. Ex- calculate student GPA, print student record, calculate cumulative GPA, print cumulative GPA.
- **Temporal Cohesion:** When a module contains functions that are related by the fact that all the functions must be executed in the same time span, the module is said to exhibit temporal cohesion. The set of functions responsible for initialization, start-up, shut-down of some process, etc. exhibit temporal cohesion.
- **Logical Cohesion:** The elements are logically related and not functionally. Ex- A component reads inputs from tape, disk, and network. All the code for these functions is in the same component. Operations are related, but the functions are significantly different.
- **Coincidental Cohesion:** The elements are not related(unrelated). The elements have no conceptual relationship other than location in source code. It is accidental and the worst form of cohesion. Ex- print next line and reverse the characters of a string in a single component.

Types of Coupling

- Coupling is the measure of the degree of interdependence between the modules. A good software will have low coupling.

Types of Coupling:

- **Data Coupling:** If the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be data coupled. In data coupling, the components are independent to each other and communicating through data. Module communications don't contain tramp data. Example-customer billing system.
- **Stamp Coupling** Two modules are stamp coupled, if they communicate using a composite data item, such as a structure in C or a record in Pascal.
- **Control Coupling:** If the modules communicate by passing control information, then they are said to be control coupled. It can be bad if parameters indicate completely different behavior and good if parameters allow factoring and reuse of functionality. Example- sort function that takes comparison function as an argument.
- **External Coupling:** In external coupling, the modules depend on other modules, external to the software being developed or to a particular type of hardware. Ex- protocol, external file, device format, etc.
- **Common Coupling:** The modules have shared data such as global data structures. The changes in global data mean tracing back to all modules which access that data to evaluate the effect of the change. So it has got disadvantages like difficulty in reusing modules, reduced ability to control data accesses and reduced maintainability.
- **Content Coupling:** In a content coupling, one module can modify the data of another module or control flow is passed from one module to the other module. This is the worst form of coupling and should be avoided.

Design Concepts for
Object-Oriented Design

Information Hiding:

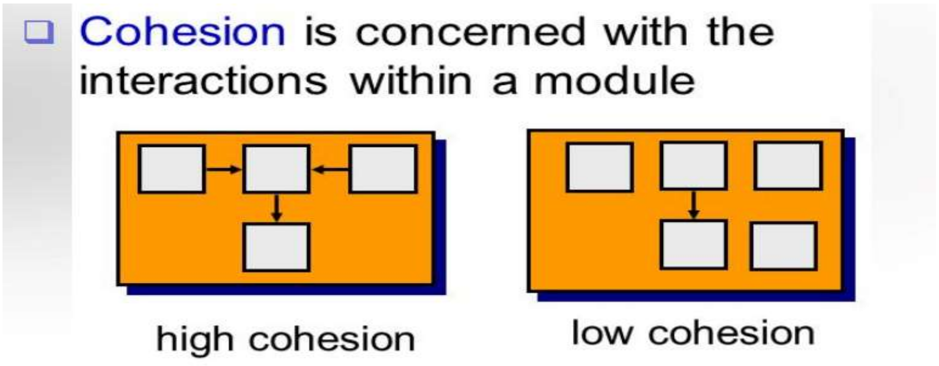
- Suggests that **software components** should be designed in such a way that **information** (data structure and algorithms) contained within a software component is **inaccessible to other software components** that have no need for such information.
- Implies that effective modularity can be achieved by defining a set of independent software components that communicate with one another only necessary information.

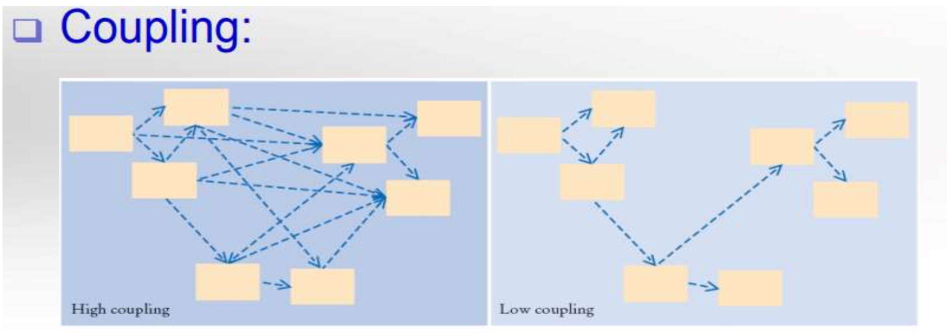
Consider the following example :

```
Class Time
    {
       public :
                  void Display();   // member function –declared public, accessible to all
       private :
                 int ticks;  // data member- declared private, external users cannot access it
    };
```

Functional Independence:

- Achieved by developing independent modules and aversion to excessive interaction with other modules.
- Module independence is measured by two quality criteria : cohesion and coupling.
- Cohesion is an indication of relative functional strength of a module. A cohesive module should ideally do just one thing.
- Coupling is an indication of relative inter-dependence among modules.
- Coupling depends on the interface complexity between modules (the point at which entry/reference is made to a module). It depends on what data pass across the interface.





High and Low Coupling between Modules

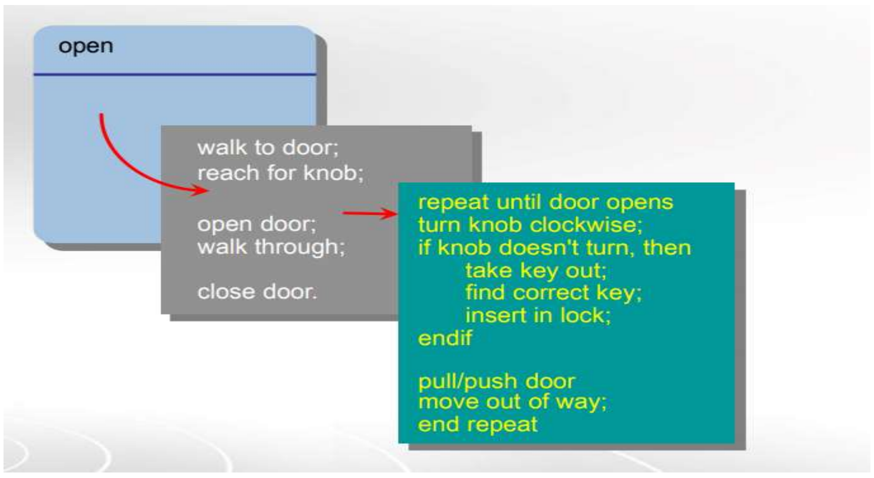
Refactoring :

- A process of changing a software system in such a way that it does not alter the external behavior of the code(design) yet improves the internal structure (Fowler, 1999).
- When a software is refactored, the existing design is examined for
  * redundancy
  * unused design elements
  * inefficient/unnecessary algorithms
  * poorly constructed/inappropriate data structures
  * or any other design failure that can be corrected to yield better design
- Result : Software that is easy to integrate/test/maintain

Refinement :

- A top-down design strategy where a program is developed by successively refining levels of procedural details in a hierarchical manner.
- It is an elaboration process that begins at a high level of abstraction.
- Abstraction and refinement are complementary.

While abstraction hides unnecessary details from outsiders, refinement helps to reveal low-level details and design progresses.



Object-oriented approach : Some important concepts and terms

Basic Mechanisms

- Objects
- Class
- Methods and messages
- Inheritance, Multiple inheritance
- Abstract class

Related Technical Terms :

- Persistence
- Agents
- Widget

Key Concepts

- Abstraction
- Encapsulation
- Polymorphism
- Composite objects
- Genericity
- Dynamic Binding

Object-Oriented Design

Overview of important concepts

Objects :

- In the object-oriented approach, ***a system*** is designed as ***a set of interacting objects***.
- An Object : may represent ***a tangible real-world entity***, such as an employee, a book, a library member, etc. An object sometimes may also represent ***some conceptual entity***, e.g. a scheduler, a controller, etc.
- When a system is analyzed, developed, and implemented in terms of natural objects occurring in it, it becomes easier to understand the design and implementation of the system.
- Each object consists of some ***data*** that are **private** to the object and ***a set of functions (or operations)*** that operate on these data.
- An object cannot directly access the data internal to another object. However, an object can indirectly access the internal data of other objects by invoking the operations (i.e. methods) supported by those objects. This mechanism is popularly known as ***data abstraction***.
- Data abstraction means that each object hides from other objects the exact way in which its internal information is organized and manipulated. It only provides a set of methods, which other objects can use for accessing and manipulating this priivate information of the object.

Class :

- Similar objects constitute a class.
- Objects possessing similar attributes and displaying similar behavior constitute a class.
- For example, a set of employees can constitute a class in an employee pay-roll system.
- A class serves as a template for object creation.

- Since each object is created as an instance of some class, classes can be considered as abstract data types (ADTs).

Methods and Messages :

- The **operations supported by an object** are called its ***methods***.
- Methods are the only means available to other objects for accessing and manipulating the data of another object.
- The methods of an object are invoked by sending messages to it.
- The set of valid messages to an object constitutes its protocol.

Inheritance :

- The inheritance feature allows us to define a new class by extending or modifying an existing class.
- The original class is called the ***base class (or superclass)*** and the new class obtained through inheritance is called the ***derived class (or subclass)***.
- The base class contains only those properties that are common to all the derived classes.
- Each derived class is a specialization of its base class because it modifies or extends the basic properties of the base class in certain ways.

- In addition to inheriting all the properties of the base class, a derived class can define new properties. That is, a derived class can define new data and methods.
- A derived class can even give new definitions to methods which already exist in the base class.
- Redefinition of the methods which existed in the base class is called method overriding.

Multiple inheritance

- A mechanism by which a subclass can inherit attributes and methods from more than one base classes.

Abstract class

- Classes that are not intended to produce instances of themselves.
- Code reuse - enhanced.
- Effort required to develop software is decreased.

Polymorphism

Polymorphism literally means poly (many) morphism (forms).

- The same message can result in different actions when received by different objects. This is also referred to as static binding. This occurs when multiple methods with the same name exist.
- When we have an inheritance hierarchy, an object can be assigned to another object of its ancestor class. A method call to the ancestor object results in the invocation of the appropriate method of the object of the derived class. Since the exact method to which a

method call would be bound cannot be known at compile time, and is dynamically decided at the runtime, this is also known as dynamic binding.

<u>Unified Modeling Language (UML)</u>

- UML is a modeling language
  - Not a system design or development methodology
- Used to document object-oriented analysis and design
- Independent of any specific design methodology.
- Developed in early 1990s
  - To standardize the large number of object-oriented modeling notations that existed.

Different software development houses were using different notations to document their object-oriented designs

- Current version is UML 2.0
- Adopted by Object Management Group (OMG) as a de facto standard in 1997.
  OMG is an association of industries which tries to facilitate early formulation of

standards.

- Provides a set of notations (e.g. rectangles, lines, ellipses, etc.) to create models of systems.
- Models - very useful in documenting the object-oriented analysis and design results obtained using some methodology.
- UML contains an extensive set of notations and suggests construction of many types of diagrams.

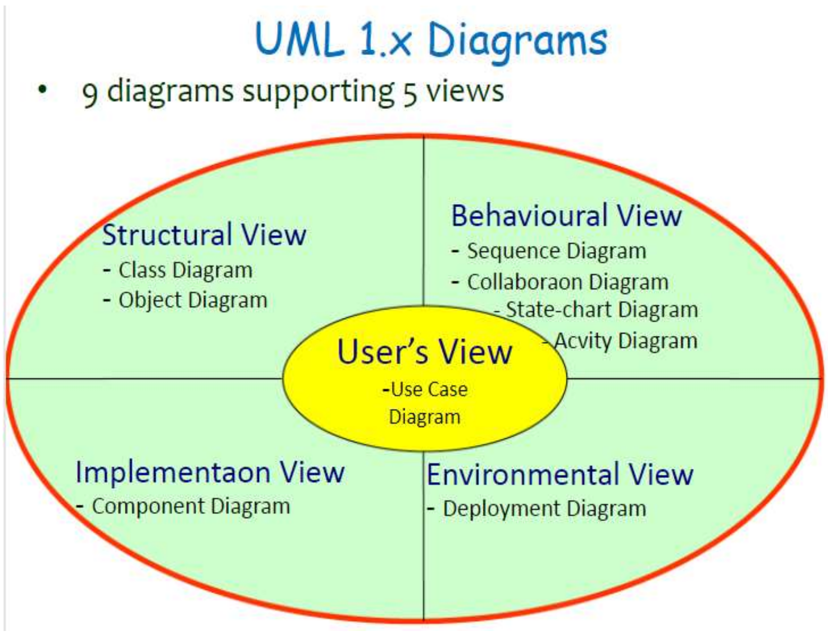
What is a model ?

- A model captures aspects important for some applications while omitting (or abstracting) the rest.
- A model in the context of software development can be ***graphical, textual, mathematical, or program code-based***.
- Graphical models – very popular, easy to understand and construct.
- It helps in managing complexity.
- Can be used for a variety of purposes during software development, including the following :

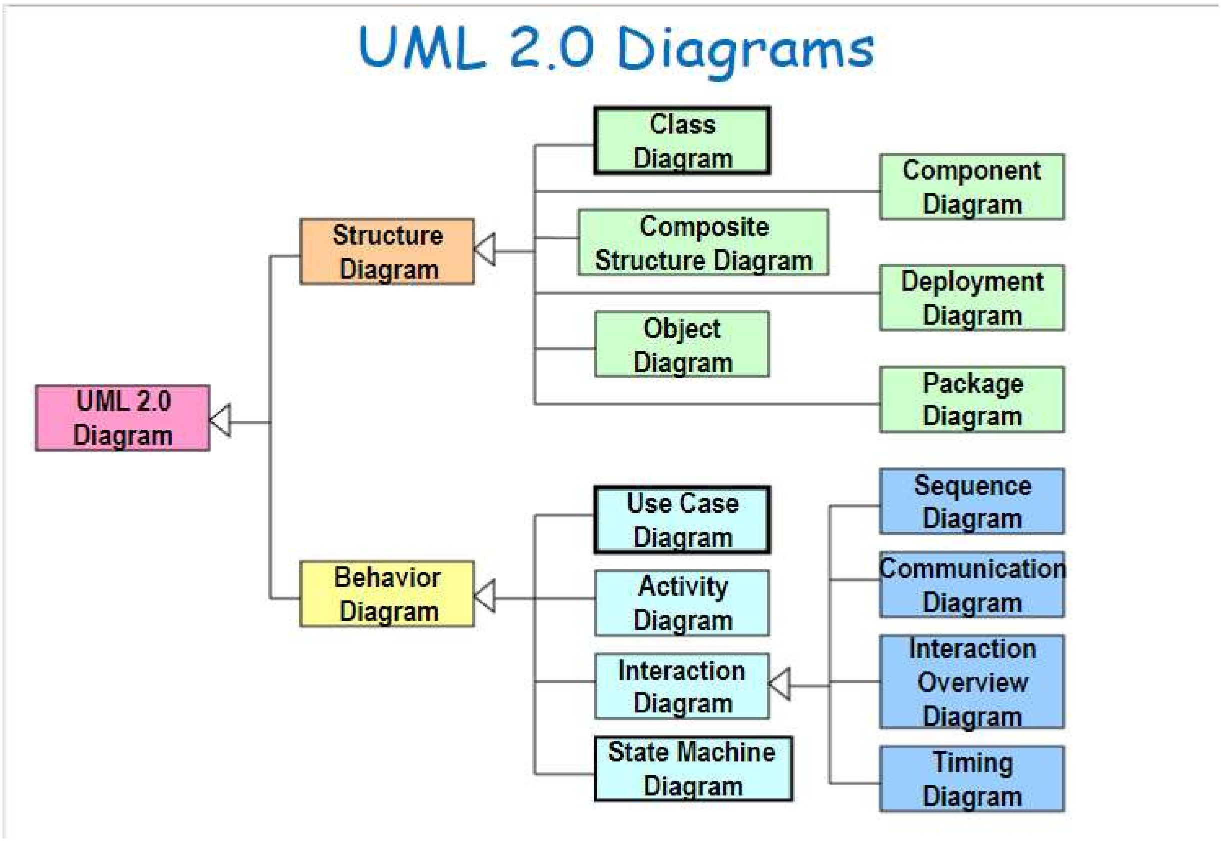
  Analysis, Specification, Design, Code generation, Testing, Understanding the problem, and Working of a system.

UML Diagrams

- UML can be used to construct ***9 different types of diagrams*** to capture different views of a system.
- Different UML diagrams ***provide different perspectives*** of the software system to be developed.
- ***Facilitate comprehensive understanding of the system***.
- The UML diagrams can capture the following views of a system :
  - * Structural view
  - * Behavioral view
  - * Implementation view
  - * Environmental view


UML 1.x Diagrams

- 9 diagrams supporting 5 views



<u>Use Case Model</u>

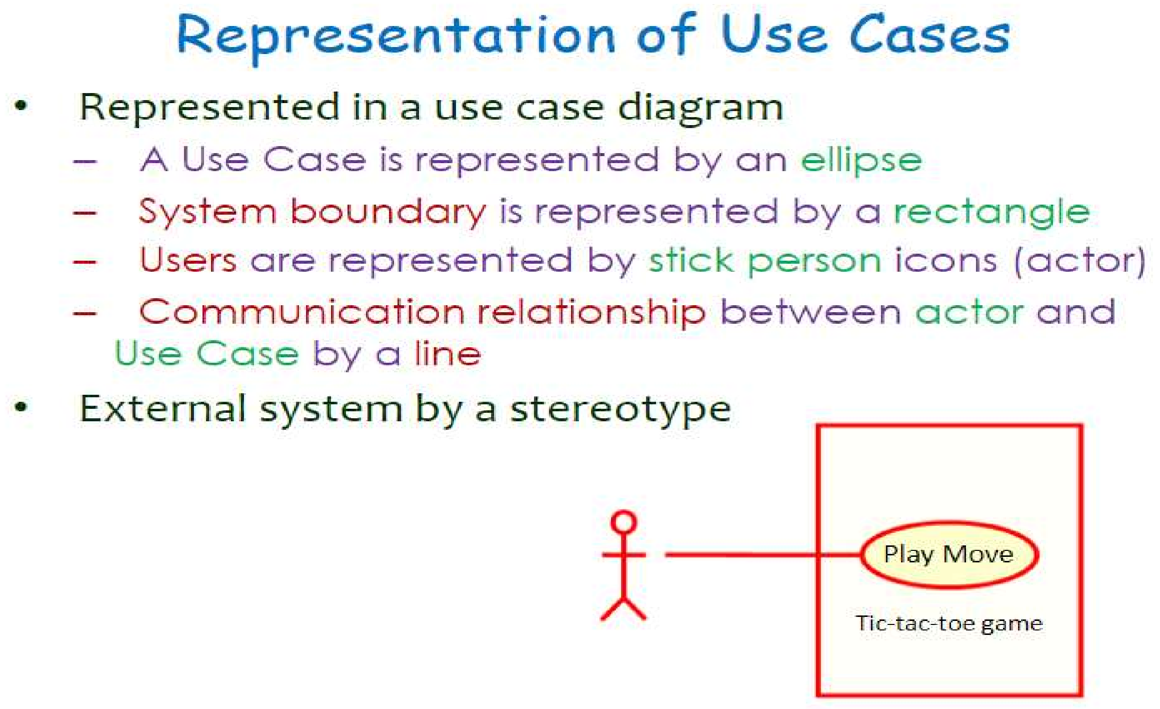
- The use case model for any system consists of a set of "use cases".
- <u>Use cases</u> represent <u>different ways in which a system can be used by the users.</u>
- A simple way to find all the use cases of a system :

  Ask : "What the users can do using the system ?"
- The use cases partition the ***system behavior*** into ***transactions***, such that each transaction performs ***some useful action*** from the ***user's point of view***. Each transaction may involve either a single message or multiple ***message exchanges*** between the <u>***user and the system***</u> to complete itself.

- The use case model is an important <u>***analysis and design***</u> artifact.
- Other UML models must conform to the use case model in any <u>***use case-driven***</u> or <u>***user-centric analysis and development approach***</u>.
- The use case model represents a <u>***functional or process model***</u> of a system.

- ***Use cases*** can be represented by drawing a ***use case diagram*** and writing an ***accompanying text*** elaborating the drawing.
- Each ***use case*** is ***represented by*** an ***ellipse*** with the name of the use case written inside the ellipse.
- ***All the ellipses (i.e. use cases) of a system are enclosed within a rectangle which represents the system boundary.***
- The name of the system being modeled (e.g. Library Information System) appears inside the rectangle.
- The ***different users*** of the system are represented by using the ***stick person icon***.
- Each ***stick person icon*** is normally referred to as an ***actor***.

- An actor is a ***role played by a user*** with respect to the system use.
- It is possible that the same user may play the role of multiple actors.
- Each actor can participate in one or more use cases.
- The ***line*** connecting the ***actor*** and the ***use case*** is called the communication relationship.
- It indicates that ***the actor makes use of the functionality provided by the use case.***
- Both the ***human users*** and ***external systems*** can be represented by ***stick person icons.***
- When a stick person icon represents an external system, it is annotated by the stereotype <<external system>>.
- ***Stereotyping*** can be used to give a ***special meaning*** to any ***basic UML construct***.



Representation of Use Cases

- Represented in a use case diagram
  - A Use Case is represented by an ellipse
  - System boundary is represented by a rectangle
  - Users are represented by stick person icons (actor)
  - Communication relationship between actor and Use Case by a line
- External system by a stereotype

Play Move
Tic-tac-toe game

- A ***use case*** typically represents ***a sequence of interactions between a user and a system.***
- These interactions consist of one mainline sequence.
- Mainline sequence : represents the normal interaction between a user and a system. The mainline sequence is the most frequently occurring sequence of interactions.
- For example, in the ***mainline sequence*** of the ***withdraw-cash***, the ***use case*** supported by a ***bank ATM*** would be :
  * insert a card
  * enter a password
  * select the amount withdraw option
  * enter the amount to be withdrawn
  * complete the transaction
  * get the amount

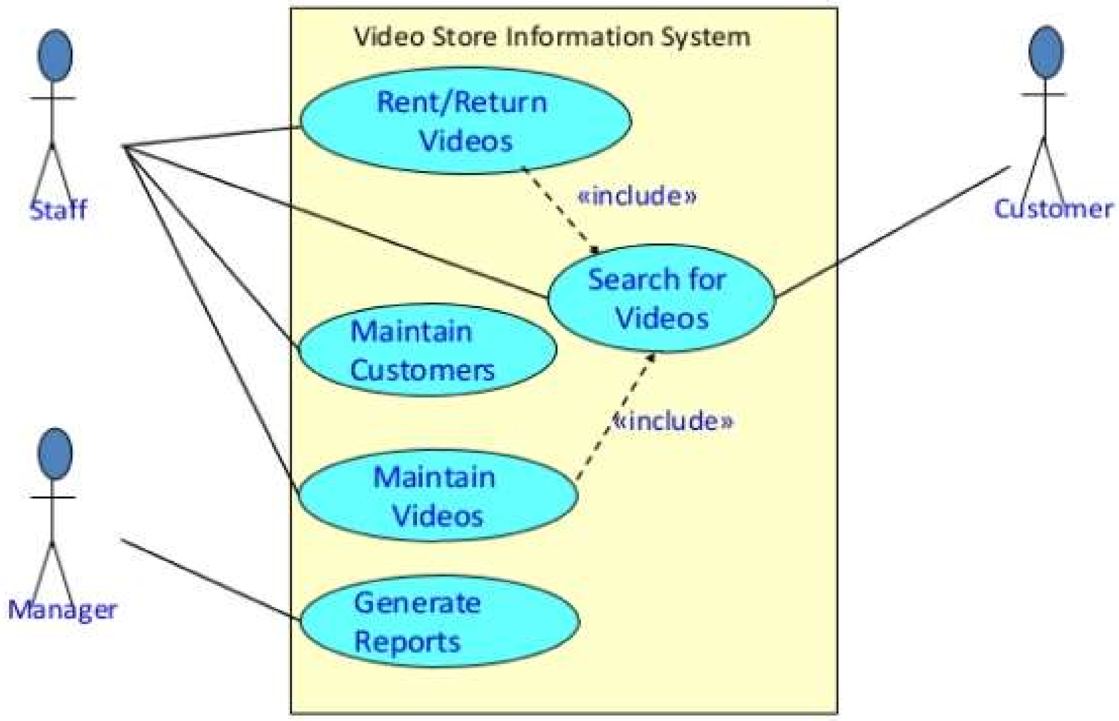
- Several variations to the mainline sequence may also exist.
- A ***variation*** from the mainline sequence occurs ***when some specific conditions hold***.
- These ***variations*** are also called ***alternate scenarios*** or ***alternate paths***.
- For the bank ATM example, variations or alternate scenarios may occur, if the password is invalid or the amount to be withdrawn exceeds the account balance.
- A use case can be viewed as a set of related scenarios tied together by a common goal.
- ***The mainline sequence and each of the variations are called scenarios or instances of the use case.***
- Each scenario is a single path of user events and system activity through the use case.

Use Case terms

Actor : A person or a system which uses the system being built for achieving some goal.
Primary actor : The main actor for whom a use case is initiated and whose goal satisfaction is the main objective of the use case.
Scenario : A set of actions that are performed to achieve a goal under some specified conditions.
Main success scenario: describes the interaction if nothing fails and all steps in the scenario succeed.
Extension scenario: Describe the system behavior if some of the steps in the main scenario do not complete successfully.

Example

- Video Store Information System supports the following business functions:
  - Recording information about videos the store owns
    - This database is searchable by staff and all customers
  - Information about a customer's borrowed videos
    - Access by staff and also the customer. It involves video database searching.
  - Staff can record video rentals and returns by customers. It involves video database searching.
  - Staff can maintain customer, video and staff information.
  - Managers of the store can generate various reports.

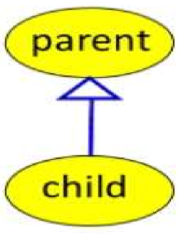


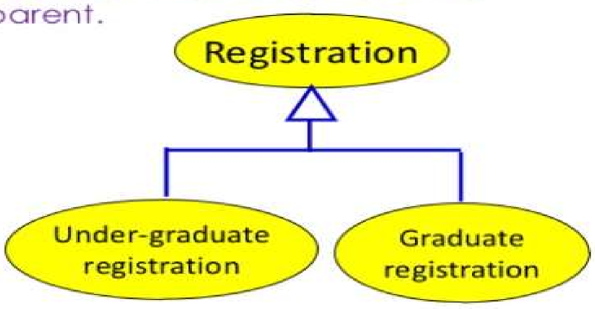
Factoring Use Cases

- Two main reasons for factoring:
  - Complex use cases need to be factored into simpler use cases
  - To represent common behaviour across different use cases
- Three ways of factoring:
  - Generalization
  - Include
  - Extend

Generalization

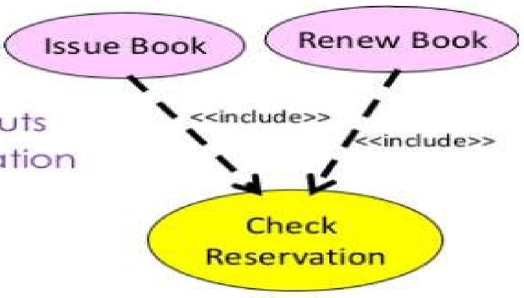
- The child use case inherits the behaviour of the parent use case.
  - The child may add to or override some of the behavior of its parent.

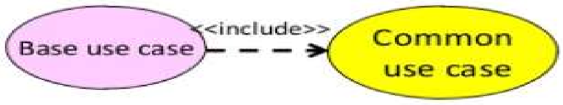




Include

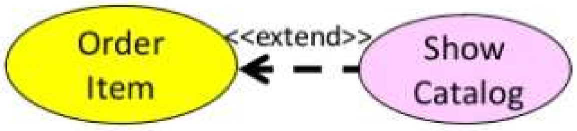
- When you have a piece of behaviour that is similar across many use cases
  - Break this out as a separate use-case and let the other ones "include" it
- Examples of use case include
  - Validate user interaction
  - Sanity check on sensor inputs
  - Check for proper authorization



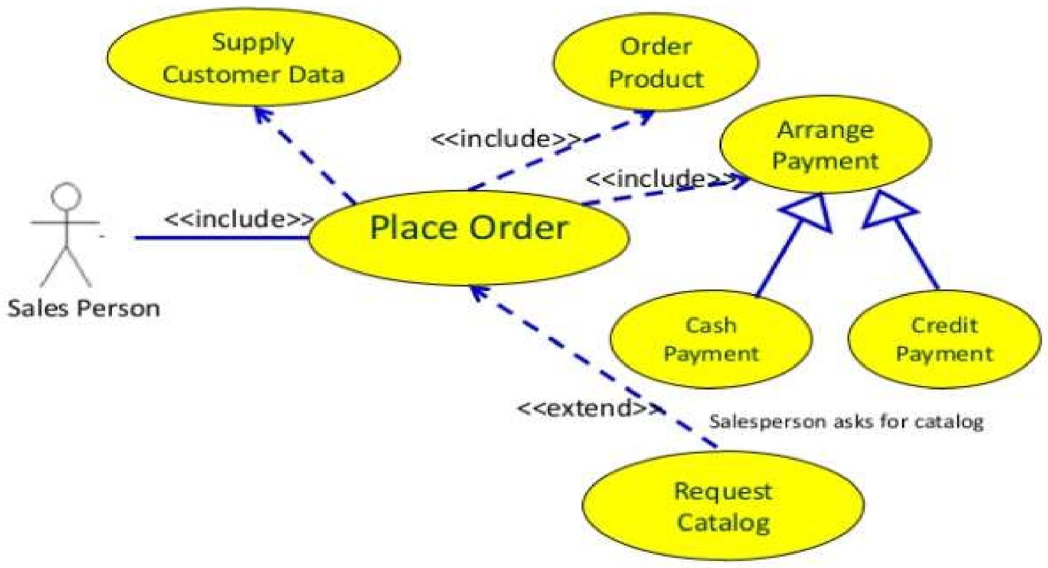


Extends

- Use when a use-case optionally can do a little bit more:
  - Capture the normal behaviour
  - Capture the extra behaviour in a separate use-case
  - Create extends dependency
- Makes it a lot easier to understand



Example



Class Diagrams

- A ***class diagram*** describes the ***static structure*** of a system.
- It ***shows how a system is structured*** rather than *how it behaves.*
- The static structure of a system consists of a number of class diagrams and their *dependencies*.
- The main constituents of a class diagram are ***classes and their relationships*** : generalization, aggregation, association and various kinds of dependencies.
- Concepts necessary to understand the UML syntax for representation of the classes and their relationships :

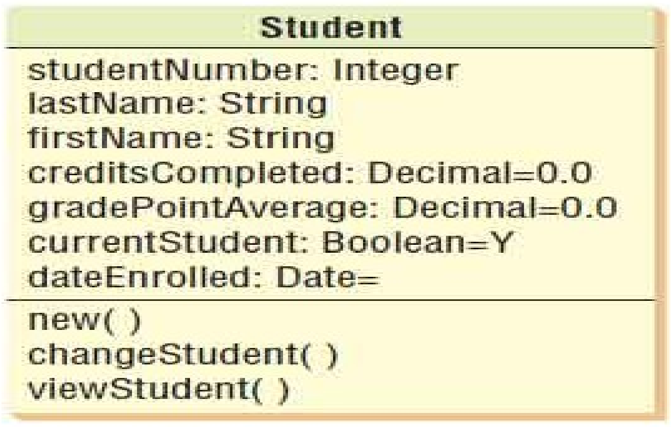
  * Classes * Attributes * Operation * Association
  * Aggregation * Composition * Inheritance * Dependency
  * Constraints * Object diagrams

- Object-oriented methodologies work to discover ***classes, attributes, methods, and relationships among classes***. Because programming occurs at the class level, defining classes is one of the most important object-oriented analysis tasks. ***Class diagrams*** show the ***static features*** of the system and do not represent any particular processing. A class diagram also shows the ***nature of the relationships between classes***.

- **A classes** is represented by a ***rectangle*** in a class diagram.
- In the simplest format, the ***rectangle*** may include only the ***class name***, but may also include the ***attributes*** and ***methods***. Attributes are what the class knows about characteristics of the objects, and methods (also called operations) are what the class knows about *how to do things*. Methods are small sections of code that work with the attributes.

Example : Representation of a Student Class

An extended Student class that shows the type of data, and in some cases, its initial value or default value.

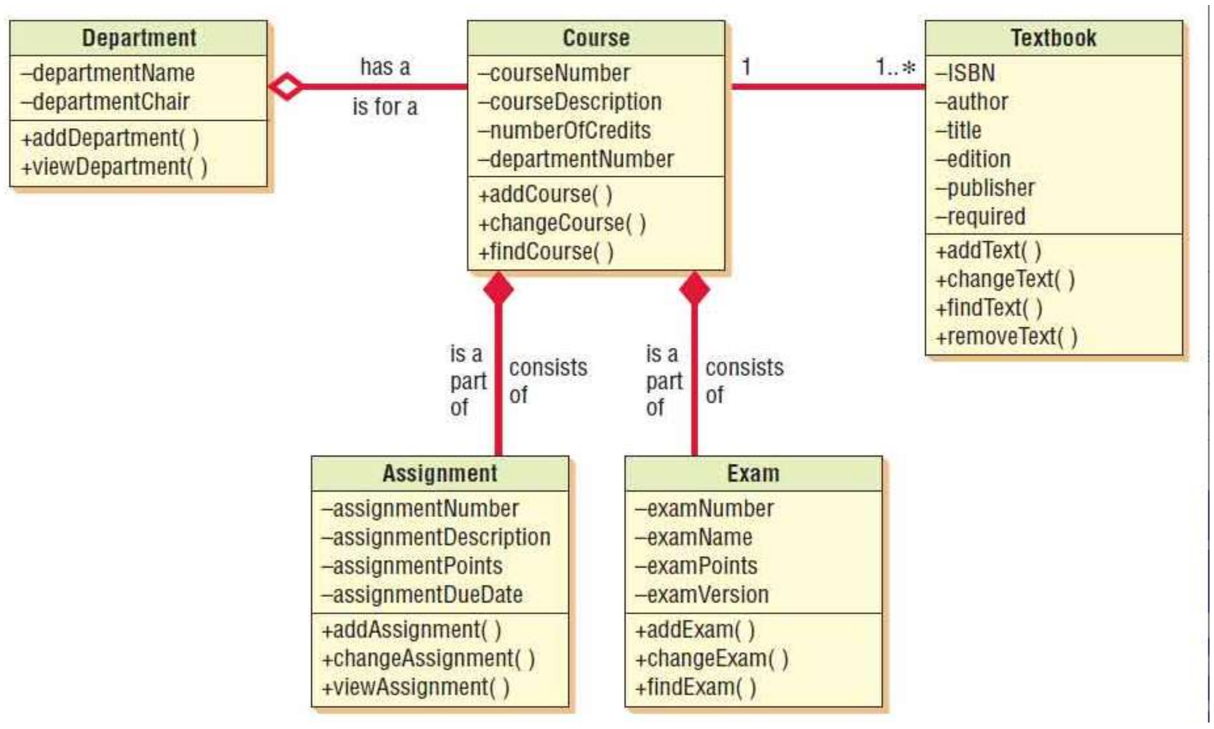


Class Diagrams

- A class in the diagram may show just the ***class name***; or the ***class name and attributes***; or the ***class name, attributes, and methods***. Showing only the class name is useful when the diagram is very complex and includes many classes. If the diagram is simpler, attributes and methods may be included.
- When attributes are included, there are *three ways* to show the ***attribute information***. The simplest is to include only the ***attribute name***, which takes the least amount of space.
- The ***type of data*** (such as string, double, integer, or date) may also be included on the class diagram. The most complete descriptions would include an equal sign (=) after the type of data followed by the ***initial value for the attribute.*** If the attribute must take on one of a finite number of values, such as a student type with values of F for full-time, P for part-time, and N for non-matriculating, these may be included in curly brackets separated by commas: **studentType:char{F,P,N}**.

- The ***attributes*** (or properties) are usually designated as ***private***, or only available in the object. This is represented on a class diagram by a ***minus sign*** in front of the attribute name.
- On a class diagram, ***public*** messages (and any ***public attributes***) are shown with a ***plus sign (+)*** in front of them.
- Attributes may also be ***protected***, indicated with a different symbol (#). These attributes are hidden from all classes except immediate subclasses. Under rare circumstances, an attribute is public, meaning that it is visible to other objects outside its class.
- Making attributes private means that the attributes are available to outside objects through the class methods only, a technique called ***encapsulation***, or ***information hiding***.

A class diagram for course offerings.
The filled-in diamonds show aggregation
and the empty diamond shows a whole-part relationship.



- The figure illustrates a ***class diagram for course offerings***. Notice that the ***name is centered at the top of the class, usually in boldface type***. The area directly below the name shows the ***attributes***, and the ***bottom portion lists the methods***. The class diagram shows data storage requirements as well as processing requirements.
- Meaning of the diamond symbols

- UML syntax for representation of the classes and their relationships.

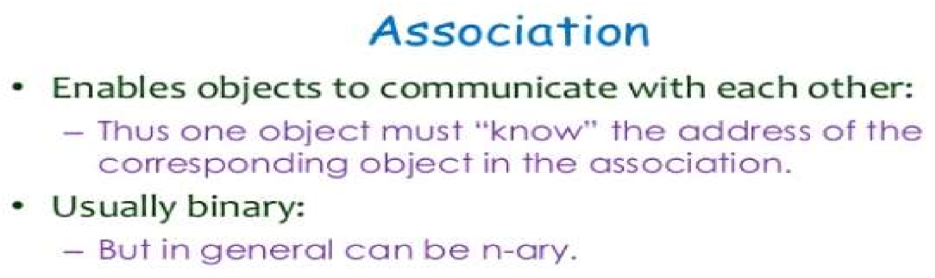
Classes

- The ***classes*** represent entities with common features, i.e. attributes and operations.
- ***Classes*** are represented as ***solid outline rectangles*** with ***compartments***.
- Classes have a mandatory name compartment where the name is written centered in boldface.
- The ***class name*** is usually written using mixed case convention and ***begins with an uppercase***.
- The class names are usually chosen to be singular nouns.

Association

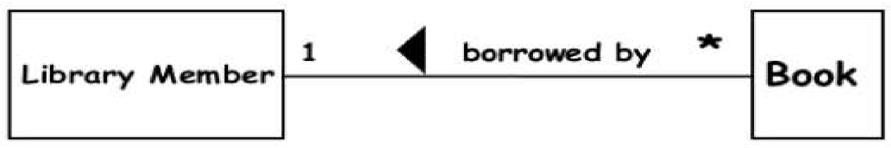
- Associations are needed to enable objects to communicate with each other.
- An ***association*** describes a ***connection between classes***.
- The relation between two objects is called object connection or link.
- For example, suppose Mahesh has ***borrowed*** the book Compiler Design. Here, borrowed is the ***connection*** between the objects Mahesh and the Compiler Design book.
- ***Association between two classes*** is represented by drawing a ***straight line*** between the ***concerned classes***. A graphical representation of the association relation is illustrated in the figure.

- The name of the association is written on the association line.
- An ***arrowhead*** may be placed on the association line to indicate the reading direction of the association.
- On each side of the association relation, the ***multiplicity*** is noted as ***an individual number*** or as a ***value range***.
- The ***multiplicity*** indicates ***how many instances of one class are associated with the other***.
- ***Value ranges*** of multiplicity are noted by specifying the minimum and maximum value, separated by two dots. For example, 1..8.
- An ***asterisk*** is a ***wild card*** and means ***many (zero or more)***.



Association

- Enables objects to communicate with each other:
  - Thus one object must "know" the address of the corresponding object in the association.
- Usually binary:
  - But in general can be n-ary.



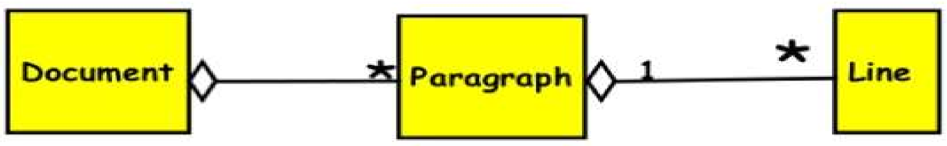
Example : "Many books may be borrowed by a Library Member."

Aggregation

- Aggregation is a special type of association where the involved classes represent a ***whole-part relationship***.
- The aggregate takes the responsibility of forwarding messages to the appropriate parts.
- If an instance of one class contains instances of some other classes, then aggregation (or composition) relationship exists between the ***composite object*** and the ***component object***.
- ***Aggregation is represented by the diamond symbol at the composite end of a relationship.***
- The number of instances of the component class aggregated can also be shown.
- The aggregation relationship ***cannot be reflexive (i.e. recursive)***. That is, ***an object cannot contain objects of its own class.***
- The aggregation relation is ***not symmetric***. ***That is, two classes A and B cannot contain instances of each other.***
- However, the aggregation relationship ***can be transitive***.

Aggregation

- Represents whole-part relationship
- Represented by a diamond symbol at the composite end
- Cannot be reflexive(i.e. recursive)
- Not symmetric
- It can be transitive

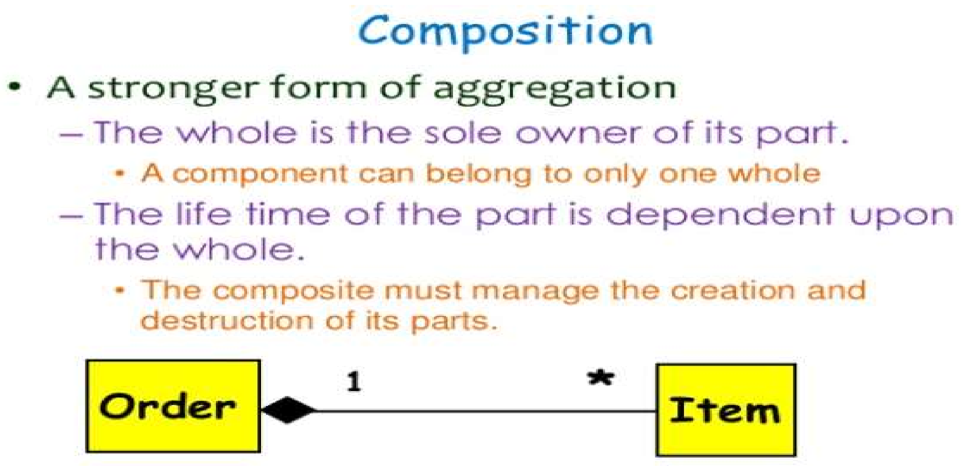


Composition

- ***Composition*** is a stricter form of aggregation, in which ***the parts are existence-dependent on the whole***.
- The life of each part is closely tied to the life of the whole.
- ***When the whole is created, the parts are created and when the whole is destroyed, the parts are destroyed.***
- The ***composition relationship*** is represented as ***a filled diamond drawn at the composite end***.

- Example : an invoice object with invoice items.

As soon as the invoice object is created, all the invoice items in it are created and as soon as the invoice object is destroyed, all invoice items in it are also destroyed.



Composition

- A stronger form of aggregation
  - The whole is the sole owner of its part.
    - A component can belong to only one whole
  - The life time of the part is dependent upon the whole.
    - The composite must manage the creation and destruction of its parts.

Order
1
*
Item

Dependency

- Dependency is a form of association between two classes.
- A ***dependency relation between two classes*** shows that ***a change in the independent class requires a change to be made to the dependent class***.
- ***Dependencies*** are shown as ***dotted arrows***.
- Dependencies may have various causes.
- Important reasons for dependency among classes :

  * A class invokes the methods provided by another class.

  * A class uses a specific interface of another class. If the properties of the class that provides the interface are changed, then a change becomes necessary in the class that uses that interface.

Constraints

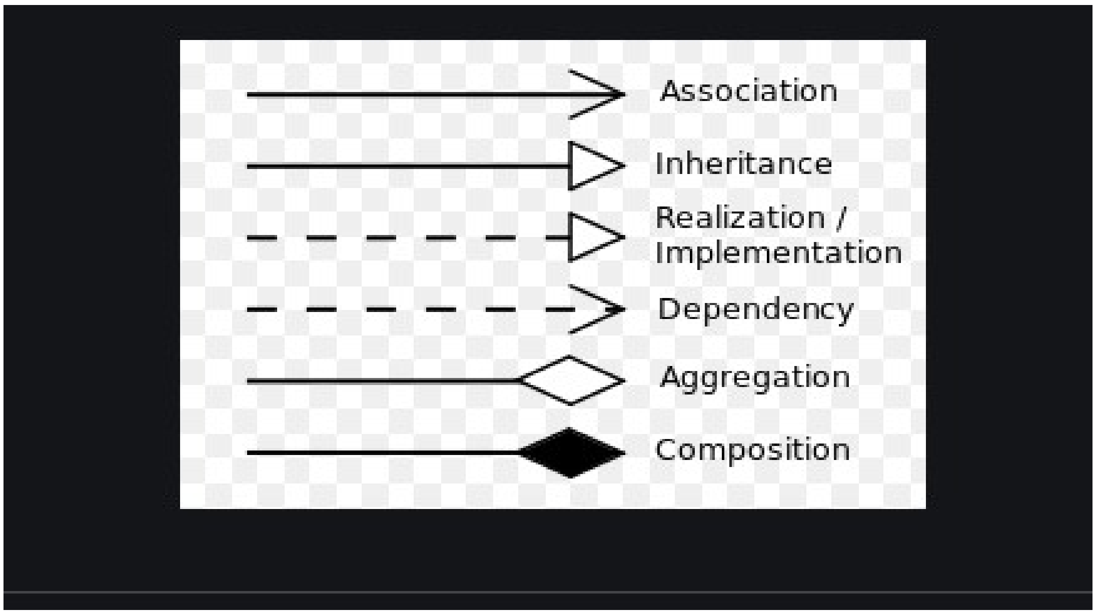
- ***Constraints*** describe ***a condition*** or ***an integrity rule***.
- It can describe the ***permissible set of values of an attribute***, specify ***pre- and post-conditions for operations***, define a ***specific order***, etc.
- For example, **{Constraint}** UML allows us to use any free form expression to describe constraints.

- ***Enclosed between braces.***
- Constraints ***can be expressed*** using ***informal English***.
- UML also provides ***Object Constraint Language (OCL)*** to specify constraints. [Rumbaugh 1999]

Inheritance

- The ***inheritance relationship*** is represented by means of an empty arrow pointing from the subclass to the superclass.
- ***The arrow is directly drawn from the subclass to the superclass.***
- Alternatively, the inheritance arrows from the subclasses may be combined into a single line.

Class Diagrams



Interaction Diagrams

- ***Interaction diagrams*** are models that ***describe how groups of objects collaborate to realize some behavior***.
- Typically, each interaction diagram realizes the behavior of a single use case.
- An interaction diagram ***shows*** a ***number of example objects*** and ***messages*** that are *passed between the objects* ***within the use case.***
- There are two kinds of interaction diagrams :
  - ***sequence diagrams***
  - ***collaboration diagrams***
- One diagram can be derived automatically from the other.
- They ***portray different perspectives of behavior of the system***.

Sequence Diagrams

- A sequence diagram ***shows interaction among objects*** as a two-dimensional chart.
- The chart is **read from top to bottom**.
- The ***objects participating in the interaction*** are ***shown at the top of the chart as boxes attached to a vertical dashed line***.
- ***Inside the box*** the ***name of the object*** is written with a ***colon*** separating it from the ***name of the class***. Both the ***name of the object and the class are underlined***.
- The objects appearing at the top signify that the objects already existed when the use case execution was initiated.
- However, if some object is created during the execution of the use case, and participates in the interaction (e.g. a method call), then that object should be shown at an appropriate place in the diagram where it was created.

- The ***vertical dashed line*** is called the ***object's lifeline***.
- The lifeline indicates the existence of the object at any particular point of time.
- The rectangle drawn on the lifeline is called the activation symbol and indicates that the object is active as long as the rectangle exists.
- Each message is indicated as an arrow between the lifelines of two objects.
- The messages are shown in chronological order from the top to the bottom.
- Reading the diagram from the top to the bottom shows the sequence in which the messages occur.

- Each message is labeled with a message name. Some control information can also be included.
- Each message is labeled with a message name. Some control information can also be included.
- Two types of control information are particularly valuable.

  * A condition (e.g. [invalid]) indicates that a message is sent, only if the condition is true.

  * An iteration marker (*) shows that a message is sent many times to multiple receiver objects as would happen when you are iterating over a collection or the elements of an array. You can also indicate the basis of the iteration, for example, [for every book object].

- **Unified Modeling Language (UML)** is a modeling language in the field of software engineering which aims to set standard ways to visualize the design of a system. UML guides the creation of multiple types of diagrams such as interaction , structure and behavior diagrams.
- A **sequence diagram** is the most commonly used interaction diagram.
- **Interaction diagram**

An interaction diagram is used to show the **interactive behavior** of a system. Since visualizing the interactions in a system can be a cumbersome task, we use different types of interaction diagrams to capture various features and aspects of interaction in a system.

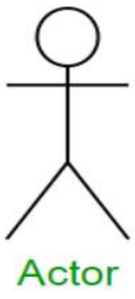
Sequence Diagrams

- A sequence diagram simply <u>depicts interaction between objects in a sequential order i.e. the order in which these interactions take place.</u> We can also use the terms event diagrams or event scenarios to refer to a sequence diagram.
- Sequence diagrams describe how and in what order the objects in a system function. These diagrams are widely used by businessmen and software developers to document and understand requirements for new and existing systems.

<u>Sequence Diagram Notations</u>

- **Actors :** An actor in a UML diagram represents a ***<u>type of role</u>*** where it interacts with the system and its objects. It is important to note here that an actor is always outside the scope of the system we aim to model using the UML diagram.

We use actors to depict various roles including human users and other external subjects. We represent an actor in a UML diagram using a stick person notation. We can have multiple actors in a sequence diagram



- **Lifelines** – A lifeline is a named element which depicts an individual participant in a sequence diagram. So basically each instance in a sequence diagram is represented by a lifeline.
- Lifeline elements are located at the top in a sequence diagram.
- The standard in UML for naming a lifeline follows the following format – Instance Name : Class Name.
- We display a lifeline in a rectangle called head with its name and type. The head is located on top of a vertical dashed line (referred to as the stem) as shown here. If we want to model an unnamed instance, we follow the same pattern except now the portion of lifeline's name is left blank.

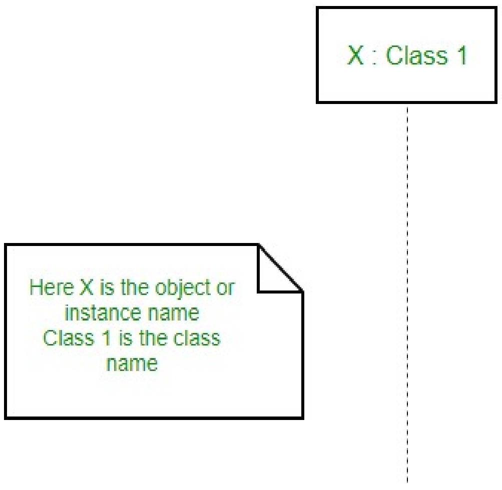

X : Class 1
Here X is the object or instance name
Class 1 is the class name

Figure – lifeline

- **Messages –** Communication between objects is depicted using messages.
- The messages appear in a sequential order on the lifeline.
- We represent messages using arrows. Lifelines and messages form the core of a sequence diagram.
- Messages can be classified into various **categories** :

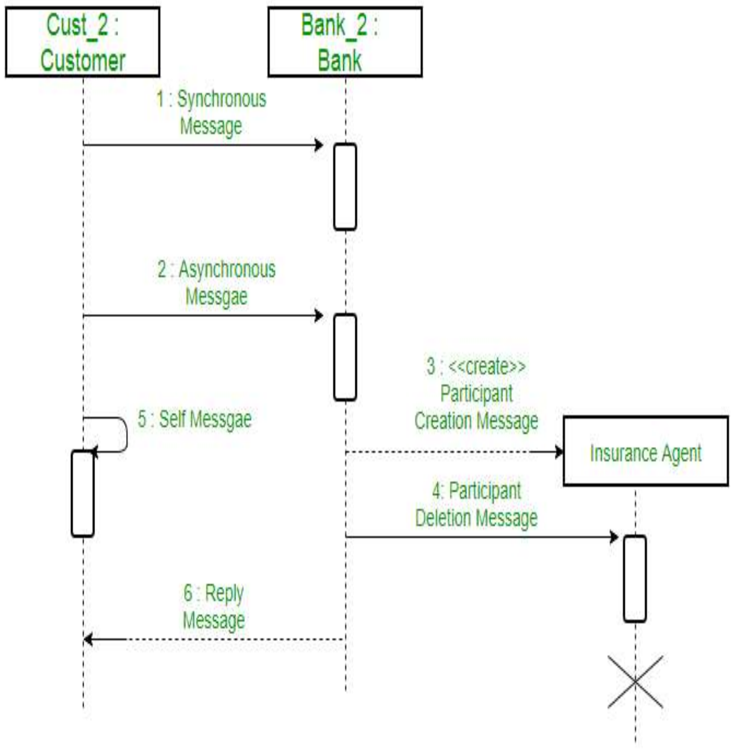


Figure - a sequence diagram with different types of messages

Synchronous messages –

- ***A synchronous message waits for a reply before the interaction can move forward***. The sender waits until the receiver has completed the processing of the message.
- The caller continues only when it knows that the receiver has processed the previous message i.e. it receives a reply message.
- A large number of calls in object oriented programming are synchronous.
- We use a ***solid arrow head*** to represent a synchronous message.

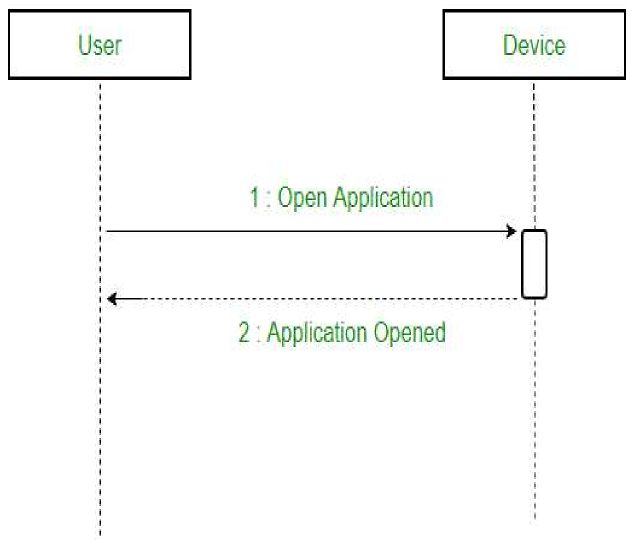
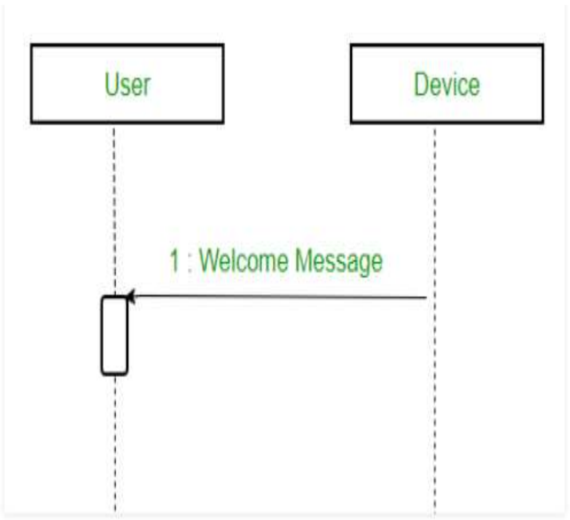


Figure - a sequence diagram using a synchronous message

Asynchronous Messages :

- An asynchronous message does not wait for a reply from the receiver.
- The interaction moves forward irrespective of the receiver processing the previous message or not.
- We use a ***lined arrow head*** to represent an asynchronous message.



Create message – We use a Create message to instantiate a new object in the sequence diagram. There are situations when a particular message call requires the creation of an object.

- It is represented with a ***dotted arrow*** and create word labeled on it to specify that it is the create Message symbol.
- For example – The creation of a new order on an e-commerce website would require a new object of Order class to be created.

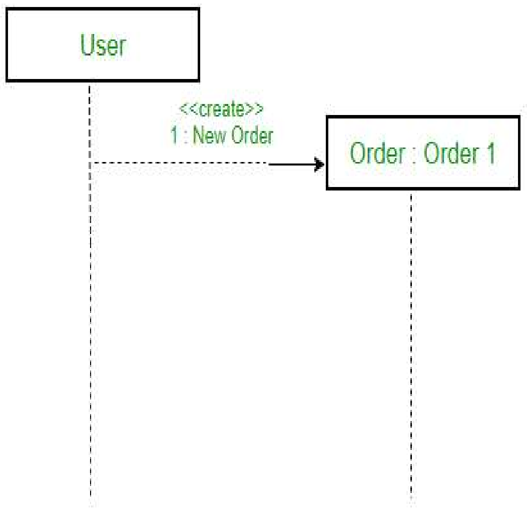


Figure - a situation where create message is used

Delete Message :

- We use a Delete Message to delete an object. When an object is deallocated memory or is destroyed within the system we use the Delete Message symbol.
- It destroys the occurrence of the object in the system.
- It is represented by an arrow terminating with a x.
- For example – In the scenario shown here when the order is received by the user, the object of order class can be destroyed.

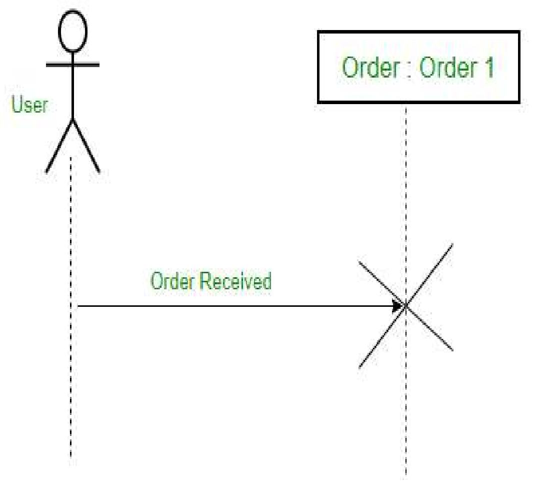


Figure - a scenario where delete message is used

Self Message :

- Certain scenarios might arise where the object needs to send a message to itself. Such messages are called Self Messages and are represented with a U shaped arrow.

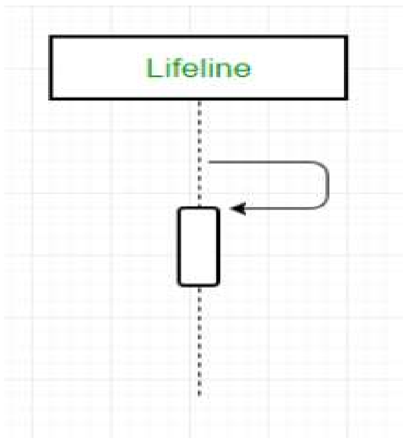


Figure - self message

Reply Message :

- Reply messages are used to show the message being sent from the receiver to the sender.
- We represent a return/reply message using an open arrowhead with a dotted line.
- The interaction moves forward only when a reply message is sent by the receiver.

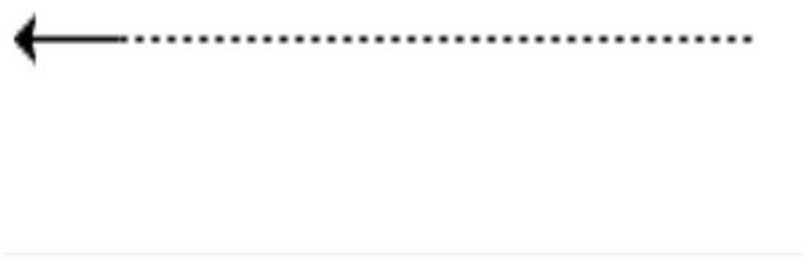
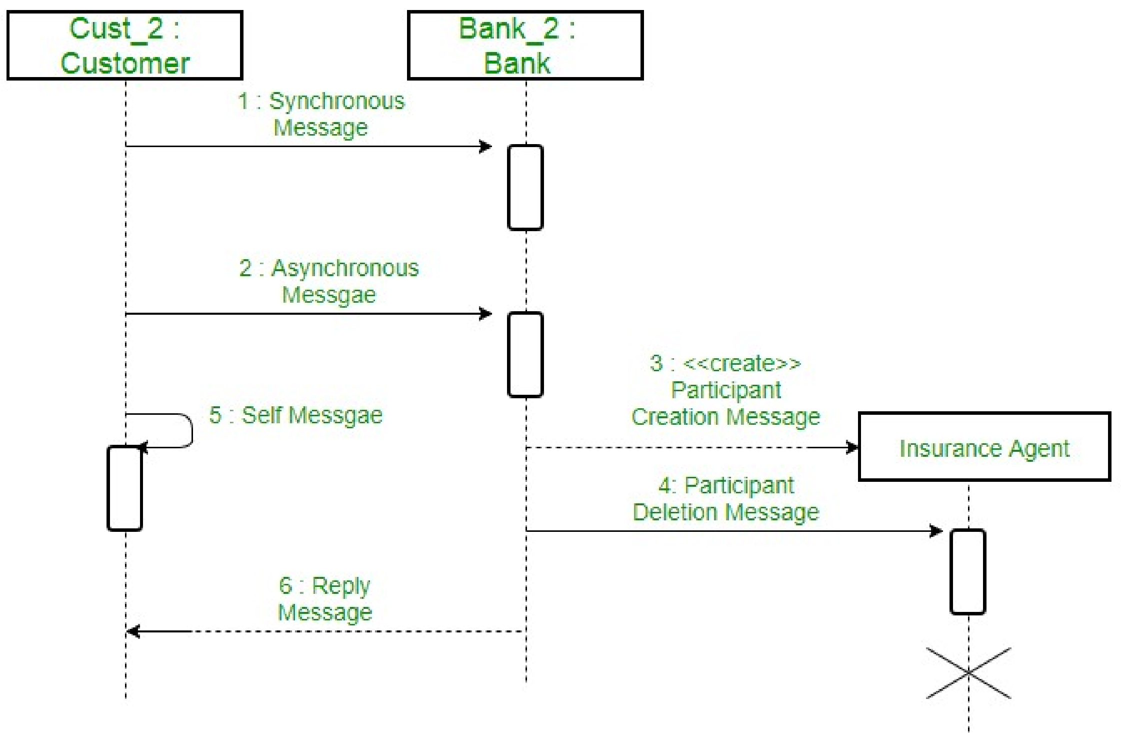


Figure – reply message

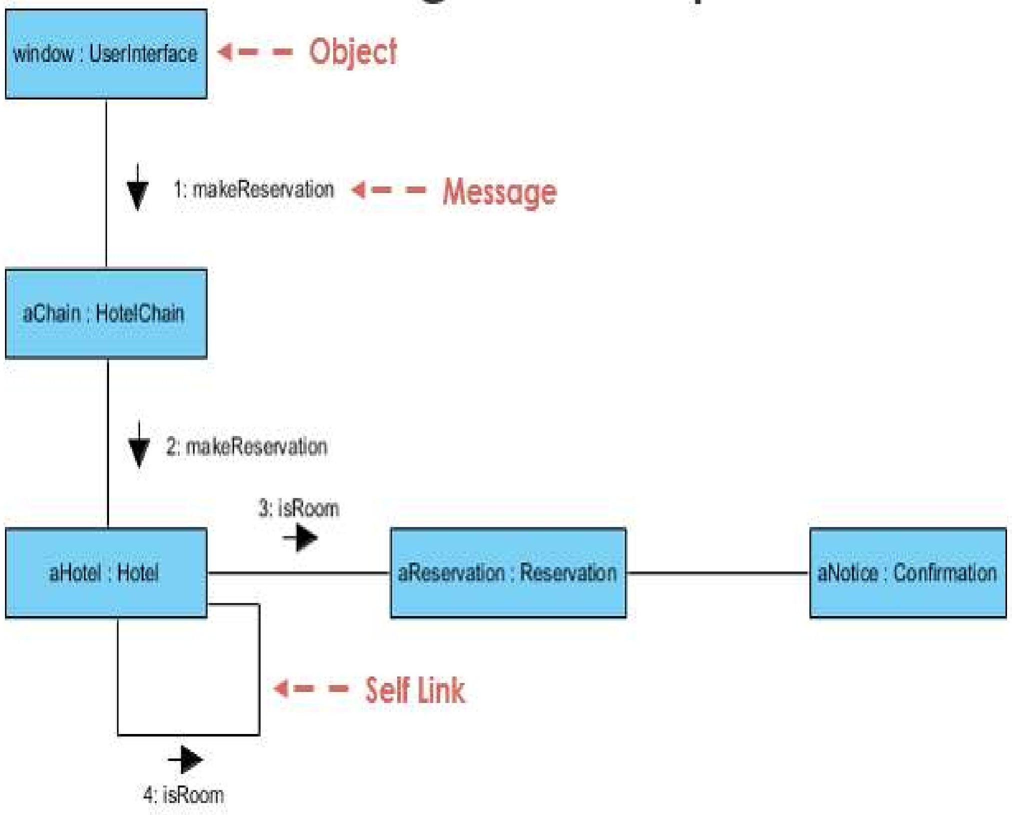
A sequence diagram with different types of messages
Source :https://www.geeksforgeeks.org/unified-modeling-language-uml-sequence-diagrams/



Collaboration Diagrams

- ***A <u>collaboration diagram</u> shows both the <u>structural and behavioral aspects</u> explicitly***. This is unlike a sequence diagram which shows only the behavioral aspects.
- The ***structural aspect*** of a collaboration diagram consists of ***objects*** and the ***links*** existing between them.
- In this diagram, an ***object*** is also called a ***collaborator***. The ***behavioral aspect*** is ***described by the set of messages exchanged among the different collaborators***.
- The ***<u>link</u>*** between objects is shown as a ***<u>solid line</u>*** and can be used to send messages between two objects.
- The ***<u>message</u>*** is shown as a ***<u>labeled arrow placed near the link</u>***. Messages are ***prefixed with sequence numbers*** because that is the only way to describe the relative sequencing of the messages in this diagram.
- It helps us in ***determining which classes are associated with which other classes.***

An Example of a Collaboration Diagram

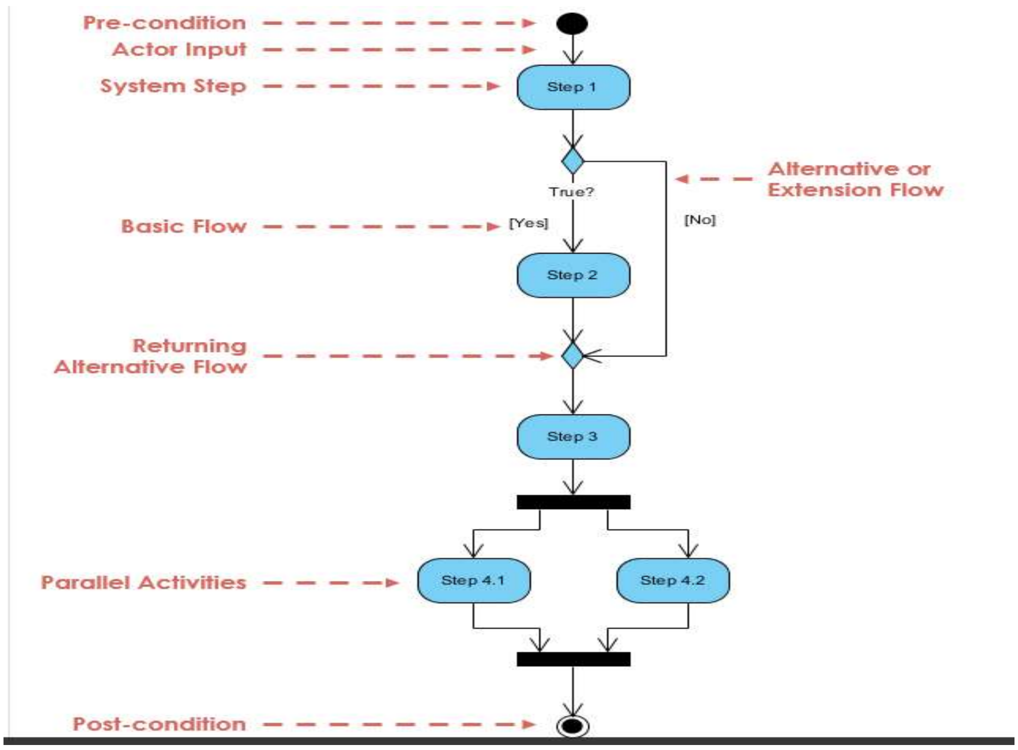


Activity Diagrams

- The activity diagram is possibly one modeling element which was not present in any of the predecessors of UML. No such diagrams were present either in the works of Booch, Jacobson, or Rumbaugh.
- The activity diagram focuses on ***representing activities or chunks of processing*** which may or may not correspond to the methods of classes.
- An ***activity*** is a state with an internal action and one or more outgoing transitions which automatically follow the termination of the internal activity. If an activity has more than one outgoing transitions, then these must be identified through conditions.

- Activity diagrams are similar to the procedural flow charts. The difference is that ***activity diagrams support description of parallel activities and synchronization aspects involved in different activities.***
- An interesting feature of the activity diagrams is the ***swim lanes***. Swim lanes ***enable you to group activities based on who is performing them***, for example, academic department vs. hostel office. Thus swim lanes ***subdivide activities based on the responsibilities of some components***. The activities in swim lanes can be assigned to some model elements, for example, classes or some component, etc.

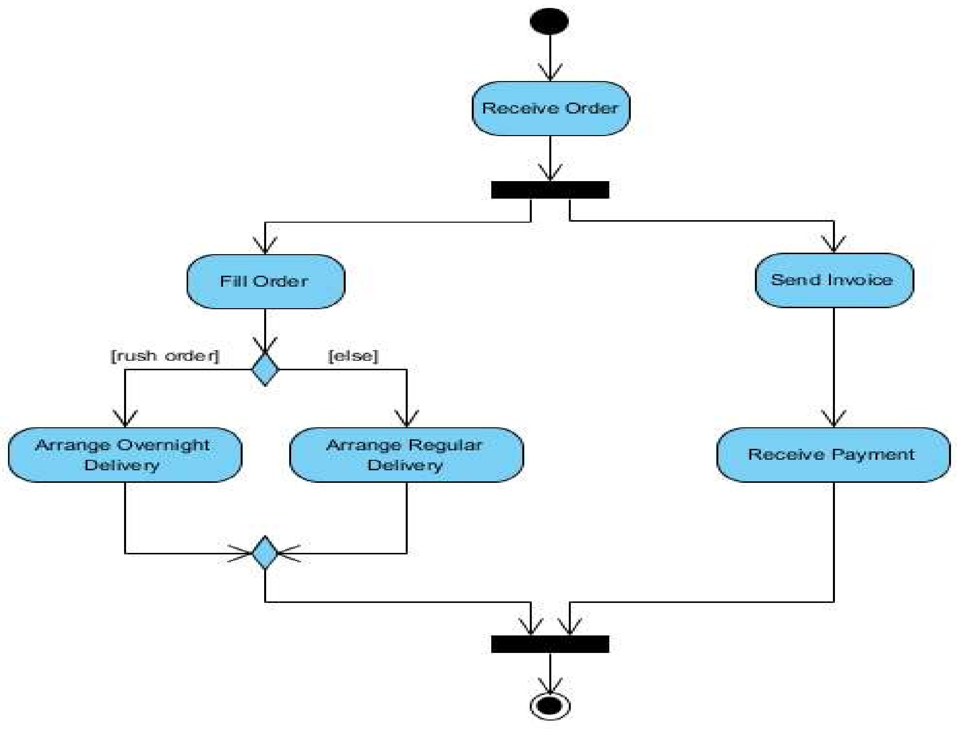
- Activity diagrams can be ***very useful to understand complex processing activities involving many components.***
- ***Activity diagrams*** are normally employed in ***business process modeling***. This is carried out during the initial stages of requirements analysis and specification.
- Later these diagrams can be used to develop interaction diagrams which help to allocate activities (responsibilities) to classes.



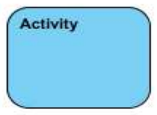
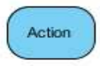
<u>Example : Activity Diagrams</u>

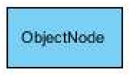
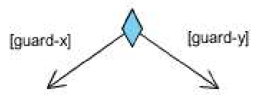
Process Order - Problem Description

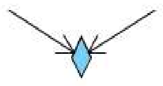
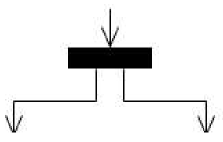
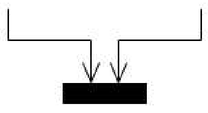
- Once the order is received, the activities split into two parallel sets of activities. One side fills and sends the order while the other handles the billing.
- On the Fill Order side, the method of delivery is decided conditionally. Depending on the condition either the Overnight Delivery activity or the Regular Delivery activity is performed.
- Finally the parallel activities combine to close the order.



Activity Diagram Notation Summary

Notation Description	UML Notation
Activity Is used to represent a set of actions	
Action A task to be performed	
Control Flow Shows the sequence of execution	→
Object Flow Show the flow of an object from one activity (or action) to another activity (or action).	→

Initial Node Portrays the beginning of a set of actions or activities	●
Activity Final Node Stop all control flows and object flows in an activity (or action)	◉
Object Node Represent an object that is connected to a set of Object Flows	
Decision Node Represent a test condition to ensure that the control flow or object flow only goes down one path	

Merge Node Bring back together different decision paths that were created using a decision-node.	
Fork Node Split behavior into a set of parallel or concurrent flows of activities (or actions)	
Join Node Bring back together a set of parallel or concurrent flows of activities (or actions).	

State Chart Diagrams

- A state chart diagram is normally used ***to model how the state of an object changes in its lifetime***.
- State chart diagrams are good at ***describing how the behavior of an object changes across several use case executions***.
- However, if we are interested in modeling some behavior that involves several objects collaborating with each other, the state chart diagram is not appropriate.
- State chart diagrams are ***based on the finite state machine (FSM) formalism***.
- An FSM consists of a ***finite number of states*** corresponding to those of the object being modeled. The **object undergoes state changes when specific events occur**. The FSM formalism existed long before the object-oriented technology, and has since been used for a wide variety of applications. Apart from modeling, it has even been used in theoretical computer science as a generator for regular languages.

- A major disadvantage of the FSM formalism is the ***state explosion*** problem. The number of states becomes too many and the model too complex when used to model practical systems. This ***problem is overcome in UML by using state charts***. The state chart formalism ***was proposed by David Harel [1990].***
- A state chart is a hierarchical model of a system and ***introduces the concept of a composite state (also called a nested state)***.

- ***Actions are associated with transitions*** and are considered to be ***processes that occur quickly*** and are ***not interruptible***.
- Activities are associated with states and can take longer.
- ***An activity can be interrupted by an event***.

The basic elements of the state chart diagram are as follows :

- ***Initial state*** : It is represented as a ***filled circle***.
- ***Final state*** : It is represented by a ***filled circle inside a larger circle***.
- ***State*** : It is represented by a ***rectangle with rounded corners***.
- ***Transition*** : A transition is shown as

an ***arrow between two states***. Normally, the ***name of the event*** which causes the transition is placed along side the arrow. You can also assign a ***guard*** to the transition. A guard is a ***Boolean logic condition***. The transition can take place only if the guard evaluates to true. The ***syntax for the label of the transition*** is shown in three parts :

event[guard]/action

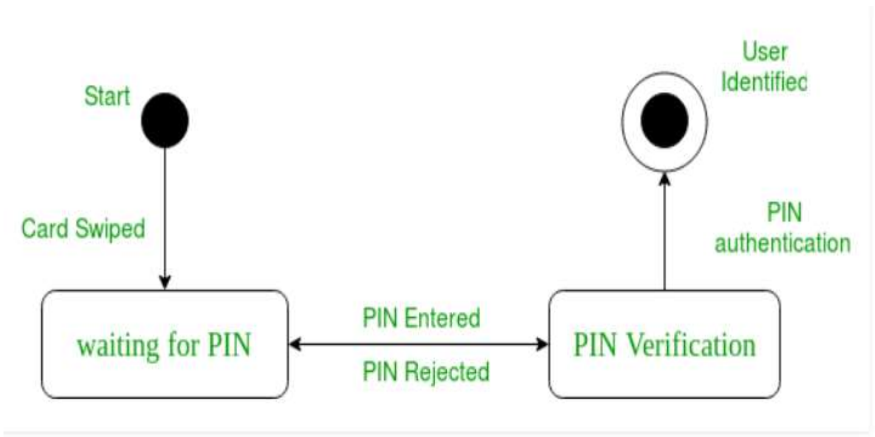


Figure - a state diagram for user verification

Unit 4   Coding and Testing

Coding

- Goal : To implement the design in the best possible manner.
- Coding activity : affects both testing and maintenance profoundly.
- Time spent in coding : a small percentage of the total software cost, while testing and maintenance consume the major percentage.
- Goal during coding : should not be to reduce the implementation cost, but the ***goal should be to reduce the cost of later phases***. Goal during the coding phase is not to simplify the job of the programmer. Rather, the ***goal should be to simplify the job of the tester and the maintainer***.
- During coding, the programs should not be constructed so that they are easy to write, but so that they are ***easy to read and understand***.
- Criteria for judging a ***program*** :
  - ***readability and understandability***
  - ***size of the program***
  - ***execution time***
  - ***required memory***

Programming Principles and Guidelines

Programming Principles and Guidelines

- How to write a high quality code ?
- How to avoid errors ?
- Writing code quickly
- Good programming (producing correct and simple programs) is a practice independent of the target programming language.

1. Common Coding Errors

Much of effort in developing software goes in identifying and removing bugs. How to reduce occurrence of bugs ? Educate programmers about most common types of errors.

1.1 Memory leaks

* A situation where the memory is allocated to the program which is not freed subsequently.

* occurs frequently in languages which do not support automatic garbage collection (like C, C++)

1.2 Freeing an already freed resource

This error occurs when a programmer tries to free the already freed resource.

```
main()
{
    char  *str;
    str = (char *) malloc (10);
    if  (global == 0)
        free (str);
    free (str);  /* str is already freed */
}
```

1.3 NULL Dereferencing

* This error occurs when we try to access the contents of a location that points to NULL.
* An attempt to access uninitialized memory.
* A commonly occurring error. It may bring the software system down.
* Sometimes NULL dereferencing may occur only in some paths and only under certain situations. Often improper initialization in different paths leads to the NULL reference statement.
* It can also be caused because of aliases. For example, two variables refer to the same object, and one is freed and an attempt is made to dereference the second.

```
switch (i)
{
     case  0 :  s = OBJECT_1;  break;
 case  1 :  s = OBJECT_2;  break;
 }
 return (s);    /*  s  is  not  initialized for values other than 0 and 1*/
```

1.4 Lack of unique addresses

* Aliasing may create many problems.
* For example, in the string concatenation function, we expect source and destination addresses to be different.

```
strcat (src, dest)
```

In the above function, if src is aliased to dest, then we may get runtime error.

1.5 Synchronization error

* In a parallel program, where there are multiple threads possibly accessing some common resources, synchronization errors are possible.
* Different categories of synchronization errors
  - **Deadlocks :** The threads in a deadlock wait for resources which are in turn locked by some other thread.
  - **Race conditions :** occur when two threads try to access the same resource and the result of execution depends on the order of execution of the threads.
  - **Inconsistent synchronization :** A situation in which there is a mix of locked and unlocked accesses to some shared variables.

1.6 Array Index Out of Bounds

* Array index values should not exceed their bounds and should not be negative.

1.7 Arithmetic Exceptions

* These include errors like ***divide by zero*** and ***floating point exceptions***.

* Getting unexpected results or termination of the program.

1.8 Off by One

* This is one of the most common errors which can be caused in many ways.

* For example, starting at 1 when we should start at 0 or vice versa, writing <= N instead of < N or vice versa, and so on.

1.9 Enumerated data types

* Overflow and underflow errors can easily occur while working with enumerated types. Care must be taken when assuming the values of enumerated data types.

Example :
```
typedef enum {A, B, C, D} grade;
  void f1 (grade x)
  {
     int i, j;
     i = GLOBAL_ARRAY[x-1];    /* Underflow possible */
     j = GLOBAL_ARRAY[x+1];    /* Overflow possible */
}
```

1.10 Illegal use of & instead of &&

* This bug arises if non short circuit logic (like & or |) is used instead of short circuit logic (&& or ||). Non short circuit logic will evaluate both sides of the expression. But short circuit operator evaluates one side, and based on the result, it decides if it has to evaluate the other side or not.

Example :
```
if (object != null & object.getTitle() != null)
```
/ Here second operation can cause a null dereference */*

1.11 String handling errors : There are a number of ways in which string handling functions like strcpy, sprintf, gets etc. can fail.

Examples : one of the operands is NULL, the string is not NULL terminated, the source operand may have greater size than destination, etc.

1.12 Buffer overflow : It is a frequent cause of software failures.

* It is a security flaw – can be exploited by a malicious user for executing arbitrary code.

* By giving a large input, a malicious user can overflow the buffer. The return address can get rewritten to whatever the malicious user has planned. So, when a function call ends, the control goes to a place planned by the malicious user, where typically there may be some malicious code to take control of the computer or do some harmful action.

Example :
```
void mygets(char *str) {
         int ch;
```

```
            while (ch=getchar()  != '\n'  &&  ch != '\0')
                  *(str++)=ch;
            *str='\0';
    }
    void main()  {  char  ss[5];   mygets(ss);   }
```

2. Structured Programming

- Structured programming practice ***helps develop programs that are easier to understand.***
- Often regarded as "goto-less" programming.
- The key property of a structured statement is that it has a single-entry and a single-exit.
- During execution, the execution of the structured statement starts from one defined point and the execution terminates at one defined point.
- With single-entry and single-exit statements, we can view a program as a sequence of structured statements.
- If all statements are structured statements, then during execution, the sequence of execution of these statements will be the same as the sequence in the program text.
- By using single-entry and single-exit statements, the correspondence between the static and dynamic structures can be obtained.
- The most commonly used single-entry and single-exit statements are :

Selection : if C then S1 else S2
if C then S1

Iteration : while C do S
repeat S until C

Sequencing : S1; S2; S3;…

- A program has a static structure as well as a dynamic structure.
- Static structure of a program : structure of the text of the program.
- Dynamic structure : the sequence of statements executed during program execution.

Coding Standards and Guidelines

General coding standards

- ***Rules for limiting the use of globals :*** These rules list what types of data can be declared global and what cannot.
- ***Contents of the headers preceding codes for different modules :*** The information contained in the headers of different modules should be standard for an organization. The exact format in which the header information is organized can also be specified. The following are **some standard header data :**

* Name of the module
* Date on which the module was created
* Author's name

* Modification history
* Synopsis of the module
* Different functions supported along with their i/o parameters
* Global variables accessed/modified by the module

- ***Naming conventions for global variables and constant identifiers :*** A possible naming convention can be that global variable names always start with a capital letter, local variable names are made of small letters, and constant names are always capital letters.
- ***Error return conventions and exception handling mechanisms :*** The way error conditions are reported by different functions in a program and the way common exception conditions are handled, should be standard within an organization. For example, different functions while encountering an error condition should either return a 0 or return a 1, consistently.

Some Programming Practices

- Writing simple and clear code with few bugs
- Control constructs : It is desirable to use a few standard control constructs rather than using a wide variery of constructs, just because they are available in the language.
- Gotos : avoid as far as possible. If used, forward transfers are more acceptable.
- Information hiding : should be supported wherever possible. Only the access functions for the data structures should be made visible while hiding the data structure behind these functions.
- User-defined types : should be exploited wherever applicable. E.g. enumerated types.
- Nesting : deep nesting of if-then-else should be avoided.
- Module size : Large modules will not be functionally cohesive. There can be no hard-and-fast rule. The guiding principle should be cohesion and coupling.
- Module interface : should be simple. Any module whose interface has more than five parameters should be carefully examined and broken into multiple modules with simpler interfaces.
- Side effects : should be avoided. For example, modifying global variables. Side effects should be properly documented.
- Robustness : Exceptional conditions should be properly handled. Produce meaningful messages and exit gracefully, if required.
- Switch case with default : It is a good practice to always include a default case to avoid unpredictable results.
- Empty catch block : Whenever exceptions are caught, it is a good practiceto take some default action, even if it is just printing an error message.
- Empty if, while statement : A condition is checked but nothing is done based on the check ? Useless checks should be avoided. Empty finally, try, synchronized, empty static method, etc. are also not good.

- Read Return to be checked : The return value from reads should be checked before processing. For example, read from scanf() may be more than expected, may cause buffer overflow.
- Return from finally block : may cause a problem if there is a return in the try block also. How to distinguish between the two ?
- Correlated parameters : It is desirable to do counter checks on implicit assumptions about parameters.
- Trusted data sources : Counter checks should be made before accessing the input data.
- Give importance to exceptions : Exceptional paths often cause software systems to fail.

Representative coding guidelines

- ***Do not use a coding style that is too clever or too difficult to understand***
- ***Avoid obscure side effects***
- ***Do not use an identifier for multiple purposes***
- ***The code should be well-documented***
- ***The length of any function should not exceed 10 source lines***
- ***Do not use goto statements***

Coding Standards

- ***Naming Conventions :***

Some of the standard naming conventions that are followed often :

* Package names should be in lower case (e.g., mypackage, edu.iitk.maths)

* Type names should be nouns and should start with uppercase (e.g., Day, DateOfBirth, EventHandler)

* Variable names should be nouns starting with lowercase (e.g., name, amount)

* Constant names should be all uppercase (e.g., PI, MAX_ITERATIONS)

* Method names should be verbs starting with lowercase (e.g., getValue())

* Private class variables should have the _ suffix (e.g., "private int value_")

* Variables with a large scope should have long names; variables with a small scope can have short names; loop iterators should be named i, j, k, etc.

* The prefix ***is*** should be used for boolean variables and methods to avoid confusion (e.g., isStatus should be used instead of status); negative boolean variable names (e.g., isNotCorrect) should be avoided.

* The term ***compute*** can be used for methods where something is being computed; the term ***find*** can be used where something is being looked up (e.g., computeMean(), findMean()).

* Exception classes should be suffixed with Exception (e.g., OutOfBoundsException)

- ***<u>Files</u>***

There are conventions on how files should be named, and what files should contain, so that a reader can get some idea about what a file contains.

Some examples of these conventions :
* Java source files should have the extension .java - this is enforced by most compilers and tools.
* Each file should contain one outer class and the class name should be same as the file name.
* Line length should be limited to less than 80 columns and special characters should be avoided. If a line is longer, it should be continued and the continuation should be made very clear.

- ***<u>Statements</u>***

These guidelines are for the declaration and executable statements in the source code.
Some examples : (not everyone may agree. Organizations generally develop their own guidelines without restricting the flexibility of programmers)
* Variables should be initialized where declared, and they should be declared in the smallest possible scope.
* Declare related variables together in a common statement. Unrelated variables should not be declared in the same statement.
* Class variables should never be declared public.
* Use only loop control statements in a for loop.
* Loop variables should be initialized immediately before the loop.
* Avoid the use of *break* and *continue* in a loop.
* Avoid the use of *do ... while* construct.
* Avoid complex conditional expressions – introduce temporary boolean variables instead.
* Avoid executable statements in conditionals.

<u>Commenting and Layout</u>

- ***<u>Comments are textual statements that are meant for the program reader to aid the understanding of code.</u>***
- Providing comments for modules is most useful, as modules form the unit of testing, compiling, verification and modification.
- Comments for a module are often called ***prologue*** for the module, which describes the functionality and the purpose of the module, its public interface and how the module is to be used, parameters of the interface, assumptions about the parameters, and any side effects it has.
- Layout guidelines focus on how a program should be indented, how it should use blank lines, white spaces, etc.- readability.

Some guidelines :
* Single line comments for a block of code should be aligned with the code they are meant for.
* There should be comments for all major variables explaining what they represent.
* A block of comments should be preceded by a blank comment line with just "/*" and ended with a line containing just "*/".
* Trailing comments after statements should be short, on the same line, and shifted far enough to separate them from statements.

Coding Process

- The coding activity starts when some form of design has been done and the specifications of the modules to be developed are available.
- With the design, modules are assigned to developers for coding.
- In a top-down implementation, we start by assigning modules at the top of the hierarchy and proceed to the lower levels.
- In a bottom-up implementation, the development starts with first implementing the modules at the bottom of the hierarchy and proceeds up.

Some processes that developers use during coding :

1. An Incremental Coding Process :
2. Test Driven Development
3. Pair Programming
4. Source Code Control and Build

1. An Incremental Coding Process :

- Generally developers write the code for the currently assigned module, perform unit testing on it and fix the bugs found. However, a better process for coding is to develop the code incrementally.
- Code is built incrementally by the developers, testing it as it is built.
- Write code for implementing only part of the functionality of a module. This code is compiled and tested with some quick tests to check the code written so far.
- When the code passes these tests, the developer proceeds to add further functionality to the code, which is then tested again.
- Advantage of developing code incrementally with testing being done after every round of coding is to facilitate debugging - an error found in some testing can be safely attributed to code that was added since last successful testing.
- If automated test scripts are used, testing can be easily done frequently.

2. Test Driven Development (TDD)

- A programmer first writes the test scripts, and then writes the code to pass the tests.
- Writing test cases before the code is written – makes the development usage-driven.
- The whole process is done incrementally, with tests being written based on the specifications and code being written to pass the tests.
- Adopted in the extreme programming (XP) methodology.
- In this approach, developers write just enough code to pass the tests.
- By following this, the code is always in sync with the tests.
- By encouraging that code is written only to pass the tests, the responsibility of ensuring that required functionality is built, is being passed to the activity of writing the test cases.
- Prioritization for code development : first few tests are likely to focus on using the main functionality.

3. Pair Programming :

- Code is not written by individual programmers but by a pair of programmers.
- The process envisaged is that one person may type a program while the other may actively participate and constantly review what is being typed.
- When needed, the pair may discuss the algorithms, data structures, or strategies to be used for code development. The roles are rotated frequently.
- Incremental code reviewing
- Potential drawback of pair programming : It may result in loss of productivity by assigning two people for a programming task. There are also issues of accountability and code ownership, particularly when pairs are not fixed and rotate.

4. Source Code Control and Build :

- Many developers are generally involved in project development.
- Each programmer creates different source files, which are eventually combined together to create executables.
- Programmers keep changing their source files as the code evolves, and often make changes in other source files as well.
- In order to keep control over the sources and their evolution, source code control is almost always used in projects using tools like CVS on UNIX or visual source safe (VSS) on Windows.

- A modern source code control system contains a ***repository***, which is essentially ***a controlled directory structure***, which keeps the ***full revision history of all the files***.
- For efficiency, a ***file history*** is generally kept as deltas or increments from the base file. This allows any ***older version of the file*** to be recreated, thereby giving a flexibility to easily discard a change, should the need arise.

- For a project, a ***repository*** has to be set up with permissions for different people in the project.
- The files the repository will contain are also specified – these are the files whose evolution the repository maintains.
- Programmers use the repository to make their source files changes available, as well as obtain other source files.
- Some of the types of commands used by programmers :
  - * Get a local copy
  - * Make changes to file(s)
  - * Update a local copy
  - * Get Reports

<u>Refactoring</u>

- <u>Refactoring</u> is a technique to improve existing code and prevent the design decay with time.
- Refactoring is part of coding.
- <u>It is performed during the coding activity, but it is not regular coding.</u>

<u>Testing</u>

- <u>Testing :</u> An activity where the errors remaining from all the previous phases must be detected.
- Testing performs a very <u>*critical role for ensuring quality*</u>.
- During testing, the software to be tested is executed with a set of test cases, and the behavior of the system for the test cases is evaluated to determine if the system is performing as expected.
- The success of testing in revealing errors depends critically on the test cases.
- System testing vs testing of individual programs
- Test case selection, criteria for selecting test cases, and their effect on testing.
- Two basic approaches to testing :
  - * black box or functional testing
  - * white-box or structural testing
- Testing process

<u>Testing Fundamentals:</u>

- <u>Error, Fault, and Failure :</u>
- The term ***error*** is used in two different ways.
- **Error :** refers to the discrepancy between a computed, observed, or measured value and the true, specified, or theoretically correct value. That is, error refers to the <u>difference between the actual output of a software and the correct output</u>. In this interpretation, error is essentially a measure of the difference between the actual and the ideal.
- **Error** is also used to refer to <u>human action that results in software containing a defect or fault.</u> This definition is quite general and encompasses all the phases.

- **<u>Fault</u>** is a ***<u>condition that causes a system to fail in performing its required function.</u>***
- A ***fault*** is the basic reason for software malfunction and is synonymous with the commonly used term ***bug***.
- The term error is also often used to refer to defects.
- There is no wear and tear in software. The only faults that a software has are "design faults".
- ***<u>Failure is the inability of a system or component to perform a required function according to its specifications.</u>***
- A software failure occurs if the behavior of the software is different from the specified behavior.

- Failures may be caused due to functional or performance reasons.
- A failure is produced only when there is a fault in the system.

However, presence of a fault does not guarantee a failure.

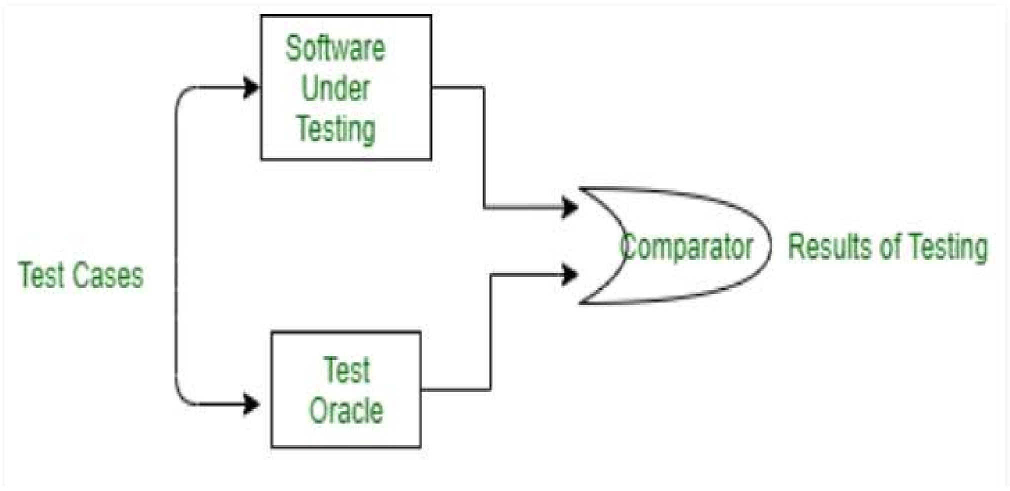
- Faults have a potential to cause failures and their presence is a necessary but not a sufficient condition for a failure to occur.
- Failure – is dependent on the project. Its exact definition is often left to the tester or project manager.
- For example, is a misplaced line in the output a failure or not ? Clearly, it depends on the project. Some will consider it a failure and others will not.

A defect might be recorded, and even corrected later, but its occurrence might not be considered a failure.

- Another example, if the output is not produced within a given time period, is it a failure or not ? For a real-time system this may be viewed as a failure, but for an operating system it may not be viewed as a failure.
- This means that there can be no general definition of failure, and it is up to the project manager or end user to decide what will be considered a failure.

Test Oracles

- A ***test oracle*** is a mechanism, different from the program itself, that can be used to check the correctness of the output of a program for the test cases.
- The test cases are given to the test oracle and the program under testing. The output of the two is then compared to determine if the program behaved correctly for the test cases.
- To test any program, we need to have a description of its expected behavior and a method of determining whether the observed behavior conforms to the expected behavior. For this we need a test oracle.
- Test oracles are necessary for testing.
- Ideally we would like an automated oracle, which always gives a correct answer. However, often the oracles are human beings, who can make mistakes.



- The human oracles generally use the specifications of the program to decide what the "correct" behavior of the program should be. However, the specifications may contain errors, be imprecise, or contain ambiguities.
- Testing requires some specifications against which the given system is tested.
- There are some systems where oracles are automatically generated from specifications of programs or modules. With such oracles, we are assured that the output of the oracle is consistent with the specifications.
- An oracle generated from the specifications will only produce correct results if the specifications are correct.
- Systems that generate oracles from specifications are likely to require formal specifications, which are frequently not generated during design.

Test Cases and Test Criteria

- ***Used for revealing the presence of faults***.
- Testing is as good as its test cases : Only for the set of inputs that exercise a fault in the program, will the output of the program deviate from the expected behavior. Sometimes even if there is a fault in a program, the program can still provide the expected behavior for many inputs.
- Ideally, we would like to determine ***a set of test cases*** such that ***successful execution of all of them implies that there are no errors in the program***.
- Each test case costs money, as effort is needed to generate the test case, machine time is needed to execute the program for that test case, and more effort is needed to evaluate the results.

- Two **fundamental goals** of **testing activity** :

  * ***Maximize the number of errors detected***
  * ***Minimize the number of test cases (i.e., minimize the cost)***

- ***Selecting test cases*** : primary objective is to ***ensure*** that ***if there is an error or fault in the program, it is detected by one of the test cases***.
- An ideal test case set is one that succeeds (meaning that its execution reveals no errors) only if there are ***no errors*** in the program.
- Exhaustive testing : set of test cases that includes all the possible inputs to the program. (ideal but impractical/infeasible).
- Test selection criterion (Test Criterion) specifies the conditions that must be satisfied by a set of test cases T for a given program P and its specifications S.
- Test criterion becomes the basis for test case selection.
- For example, if the criterion is that all statements in the program be executed at least once during testing, then a set of test cases T satisfies this criterion for a program P if the execution of P with T ensures that each statement in P is executed at least once.

Test Cases and Test Criteria

- Two fundamental properties for a ***testing criterion*** : ***reliability*** and ***validity***.
- A criterion is ***reliable*** if all the sets (of test cases) that satisfy the criterion detect the same errors. Every set will detect exactly the same errors.
- A criterion is ***valid*** if for any error in the program there is some set satisfying the criterion that will ***reveal the error***.
- A fundamental theorem of testing is that *if a testing criterion is valid and reliable, if set satisfying the criterion succeeds (revealing no faults), then the program contains no errors.*
- It has been shown that no algorithm exists that will determine a valid criterion for an arbitrary program.
- Generating test cases to satisfy a given criterion is not simple.
- Getting a criterion that is reliable and valid and that can be satisfied by a manageable number of test cases is usually not possible.

Testing Process

- The basic goal of the software development process is to produce software that has no errors or very few errors.
- Testing is the last phase before the final software is delivered.
- Testing has enormous responsibility of detecting any type of error that may be in the software.
- Testing should not be done on-the-fly. The process of testing should be carefully planned and the plan should be properly executed.
- In order to detect errors soon after they are introduced, each phase ends with a verification activity such as a review. But most of the verification activities in the early phases of software development are based on human evaluation and cannot detect all the errors.
- High responsibility on testing - due to unreliability of the quality assurance activities in the early part of the software development cycle.
- Software typically undergoes changes even after it has been delivered.
- In order to validate that a change has not affected some old functionality of the system, ***regression testing*** is done.
- In regression testing, old test cases are executed with the expectation that the same old results will be produced.
- Need for regression testing places additional requirements on the testing phase. It must provide the "old" test cases and their outputs.

Levels of Testing

- Testing is usually relied upon to detect the faults remaining from earlier stages, in addition to the faults introduced during coding itself.
- Different levels of testing are used in the testing process.
- Each level of testing aims to test different aspects of the system.

- Basic levels of testing :
  - * unit testing
  - * integration testing
  - * system testing
  - * acceptance testing

- Different levels of testing attempt to detect different types of faults.

- The first level of testing is called ***unit testing.***
- ***Unit testing : Different modules are tested against the specifications produced during design for the modules.***
- Unit testing is essentially for verification of the code produced during the coding phase, and hence the ***goal is to test the internal logic of the modules.***
- Unit testing : typically done by the programmer of the module.
- A module is considered for integration and use by others, only after it has been unit-tested satisfactorily.

- The next level of testing is ***integration testing***.
- In ***integration testing***, many unit-tested modules are combined into subsystems, which are then tested. The goal here is to see if the modules can be integrated properly.
- Integration testing : the ***emphasis*** is on ***testing interfaces between modules.***
- The next levels are ***system testing*** and ***acceptance testing***. Here the ***entire software system is tested***. The reference document for this process is the ***requirements document***, and the goal is ***to see if the software meets the requirements***. This is essentially a validation exercise.

- ***Acceptance testing*** is sometimes performed with ***realistic data*** of the client to demonstrate that the software is working satisfactorily. Testing here focuses on the ***external behavior of the system***. The internal logic of the program is not emphasized. Consequently, mostly ***functional testing*** is performed at these levels.
- These levels of testing are performed when a system is being built from the components that have been coded.
- There is another level of testing, called ***regression testing***.

- ***Regression testing is performed when some changes are made to an existing system.***
- Software systems undergo changes generally. Frequently, changes are made to "upgrade" the software by adding new features and functionalities.
- Clearly, the modified software needs to be tested to ***make sure*** that the ***new features*** that are added do ***indeed work***.
- As ***modifications*** have been made to an existing system, testing also has to be done to make sure that the modification has not had ***any undesirable side effect*** of making some of the earlier services faulty.
- Besides ensuring the desired behavior of the new services, regression testing has to ensure that the desired behavior of the old services is maintained.

- For **regression testing**, some test cases that have been executed on the old system are maintained, along with the output produced by the old system. **These test cases are executed again on the modified system and its output compared with the earlier output to make sure that the system is working as before on these test cases.**
- The test cases for systems should be properly documented for future use in regression testing.
- For many systems that are frequently changed, regtression testing scripts are used, which automate the process of regression testing. A regression testing script executes a suit of test cases.

Basic approaches to Testing :

There are two basic approaches to testing:

* Black-box testing

* White-box testing

Black-Box Testing

- Black-box testing is also called ***functional or behavioral testing***.
- Black-box testing is concerned with **functionality** rather than implementation of a program.
- ***Black-box testing is concerned with the function that the program is supposed to perform, and does not deal with the internal structure of the program responsible or actually implementing that function.***

- In black-box testing, the structure of a program is not considered.
- In black-box testing, the tester only knows the **inputs** that can be given to the system and what **output** the system should give.
- **Test cases** are decided solely on the basis of the **requirements** or **specifications** of the program or module, and the internals of the module or the program are not considered for selection of test cases.

- The basis for deciding test cases in ***functional testing*** is the **requirements** or **specifications** of the system or module.
- The most obvious functional testing procedure is exhaustive testing, which is impractical.
- Criterion for generating test cases ?
  - * generate them randomly ? (*This strategy has a little chance of resulting in a set of test cases that is close to optimal. i.e., that detects maximum errors with minimum test cases.*)
- There are no formal rules for designing test cases for functional testing. There are no precise criteria for selecting test cases. However, there are a number of techniques or heuristics that can be used to select test cases.

White-Box Testing

- **White-Box Testing** is a software testing technique in which internal structure, design and coding of software are tested to verify flow of input-output and to improve design, usability and security. In white-box testing, code is visible to testers so it is also called Clear box testing, Open box testing, Transparent box testing, Code-based testing and Glass box testing.
- It is one of two parts of the Box Testing approach to software testing. Its counterpart, black-box testing, involves testing from an external or end-user type perspective. On the other hand, white-box testing in software engineering is based on the inner workings of an application and revolves around internal testing.

- The clear box or white-box name symbolizes the ability to see through the software's outer shell (or "box") into its inner workings. Likewise, the term "black-box" symbolizes not being able to see the inner workings of the software so that only the end-user experience can be tested.

What do we verify in White Box Testing?

White box testing involves the testing of the software code for the following:

- Internal security holes
- Broken or poorly structured paths in the coding processes
- The flow of specific inputs through the code
- Expected output
- The functionality of conditional loops
- Testing of each statement, object, and function on an individual basis

White-Box Testing

- White-box testing is concerned with testing the implementation of a program.
- White-box testing is also called ***structural testing***.
- The intent of white-box testing is not to exercise all the different input and output conditions (although that may be a by-product) but to exercise the different programming structures and data structures used in the program.
- To test the structure of a program, structural testing aims to achieve test cases that force the desired coverage of different structures.
- The criteria for structural testing are generally quite precise as they are based on program structures.

- Three different approaches to structural testing :

  * control flow-based testing
  * data flow-based testing
  * mutation testing

Alpha Testing

- **Alpha Testing** is a type of acceptance testing performed to identify all possible issues and bugs before releasing the final product to the end users.
- Alpha testing is carried out by the testers who are internal employees of the organization.
- The main goal is to identify the tasks that a typical user might perform and test them.
- Alpha testing is done near the end of the development of the software, and before beta testing. The main focus of alpha testing is to simulate real users by using black box and white box techniques.

Beta Testing

- **Beta Testing** is performed by "real users" of the software application in "real environment" and it can be considered as a form of external User Acceptance Testing.
- It is the final test before shipping a product to the customers.
- Direct feedback from customers is a major advantage of Beta Testing.
- This testing helps to test products in customer's environment.
- Beta version of the software is released to a limited number of end-users of the product to obtain feedback on the product quality.
- Beta testing reduces product failure risks and provides increased quality of the product through customer validation.

Alpha Testing Vs Beta Testing

- Alpha Testing is performed by the Testers within the organization whereas Beta Testing is performed by the end users.
- Alpha Testing is performed at Developer's site whereas Beta Testing is performed at Client's location.
- Reliability and Security testing are not performed in-depth in Alpha Testing while Reliability, Security and Robustness are checked during Beta Testing.
- Alpha Testing involves both Whitebox and Blackbox testing whereas Beta Testing mainly involves Blackbox testing.
- Alpha Testing requires testing environment while Beta Testing doesn't require testing environment.
- Critical issues and bugs are addressed and fixed immediately in Alpha Testing whereas issues and bugs are collected from the end users and further implemented in Beta Testing.