**Chapter 2: Unified Modeling language**

**2.1 Introduction to UML and UML history**

* An industry standard graphical language for specifying, visualizing, constructing and documenting the artifacts of software systems, as well as for business modeling.
* The UML uses mostly graphical notations to express the OO analysis and design of software projects.
* Simplifies the complex process of software design.
* UML stands for **Unified Modeling Language**.
* UML is different from the other common programming languages such as C++, Java, COBOL, etc.
* UML is a pictorial language used to make software blueprints.
* UML can be described as a general purpose visual modeling language to visualize, specify, construct, and document software system.
* Although UML is generally used to model software systems, it is not limited within this boundary. It is also used to model non-software systems as well. For example, the process flow in a manufacturing unit, etc.

UML is not a programming language but tools can be used to generate code in various languages using UML diagrams. UML has a direct relation with object oriented analysis and design. After some standardization, UML has become an OMG standard.

## 2.1.1 Goals of UML

* *A picture is worth a thousand words*, this idiom absolutely fits describing UML. Object-oriented concepts were introduced much earlier than UML. At that point of time, there were no standard methodologies to organize and consolidate the object-oriented development. It was then that UML came into picture.
* There are a number of goals for developing UML but the most important is to define some general purpose modeling language, which all modelers can use and it also needs to be made simple to understand and use.
* UML diagrams are not only made for developers but also for business users, common people, and anybody interested to understand the system. The system can be a software or non-software system. Thus it must be clear that UML is not a development method rather it accompanies with processes to make it a successful system.

In conclusion, the goal of UML can be defined as a simple modeling mechanism to model all possible practical systems in today’s complex environment.

**2.1.2 A Conceptual Model of UML**

To understand the conceptual model of UML, first we need to clarify what is a conceptual model? and why a conceptual model is required?

* A conceptual model can be defined as a model which is made of concepts and their relationships.
* A conceptual model is the first step before drawing a UML diagram. It helps to understand the entities in the real world and how they interact with each other.

As UML describes the real-time systems, it is very important to make a conceptual model and then proceed gradually. The conceptual model of UML can be mastered by learning the following three major elements −

* UML building blocks
* Rules to connect the building blocks
* Common mechanisms of UML

**2.1.3 Why a UML for Modeling?**

* A diagram/picture = thousands words.
* Uses graphical notation to communicate more clearly than natural language and code.
* Make it easier for programmers and software architects to communicate.
* Helps acquire an overall view of a system.
* UML is not dependent on any one language or technology.
* UML moves us from fragmentation to standardization.

**2.1.4 UML history**

Time

2003: UML 2.0

2001: UML 1.4

1999: UML 1.3

1997: UML 1.0, 1.1

1996: UML 0.4 & 0.91

1995: Unified Method 0.8

Bouch 93 OMT -2

Other methods Bouch 91 OMT -1

**2.2 Notation and basic building blocks of UML**

- Graphical notations used in structural things are most widely used in UML. These are considered as the nouns of UML models. Following are the list of structural things:

1. **Class Notation:**

A class is represented by a rectangle having three sections −

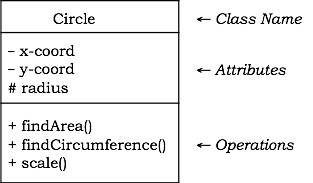
* the top section containing the name of the class
* the middle section containing class attributes
* the bottom section representing operations of the class

The visibility of the attributes and operations can be represented in the following ways −

* **Public** − A public member is visible from anywhere in the system. In class diagram, it is prefixed by the symbol ‘+’.
* **Private** − A private member is visible only from within the class. It cannot be accessed from outside the class. A private member is prefixed by the symbol ‘−’.
* **Protected** − A protected member is visible from within the class and from the subclasses inherited from this class, but not from outside. It is prefixed by the symbol ‘#’.

An abstract class has the class name written in italics.

**Example** − Let us consider the Circle class introduced earlier. The attributes of Circle are x-coord, y-coord, and radius. The operations are findArea(), findCircumference(), and scale(). Let us assume that x-coord and y-coord are private data members, radius is a protected data member, and the member functions are public. The following figure gives the diagrammatic representation of the class.

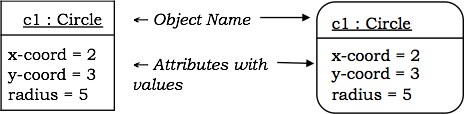


1. **Object Notation**

An object is represented as a rectangle with two sections −

* The top section contains the name of the object with the name of the class or package of which it is an instance of. The name takes the following forms −
  + **object-name** − class-name
  + **object-name** − class-name :: package-name
  + **class-name** − in case of anonymous objects
* The bottom section represents the values of the attributes. It takes the form attribute-name = value.
* Sometimes objects are represented using rounded rectangles.

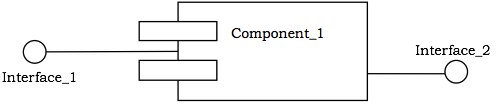
**Example** − Let us consider an object of the class Circle named c1. We assume that the center of c1 is at (2, 3) and the radius of c1 is 5. The following figure depicts the object.



1. **Interface Notations**

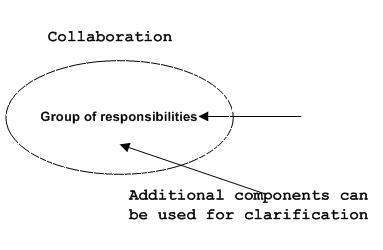
* Interface is a collection of methods of a class or component. It specifies the set of services that may be provided by the class or component.

**Notation** − Generally, an interface is drawn as a circle together with its name. An interface is almost always attached to the class or component that realizes it. The following figure gives the notation of an interface.



1. **Collaboration Notation**

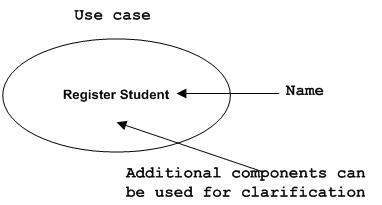
Collaboration is represented by a dotted eclipse as shown in the following figure. It has a name written inside the eclipse.



Collaboration represents responsibilities. Generally, responsibilities are in a group.

1. **Use Case Notation**

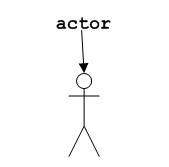
Use case is represented as an eclipse with a name inside it. It may contain additional responsibilities.



Use case is used to capture high level functionalities of a system.

1. **Actor Notation**

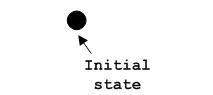
An actor can be defined as some internal or external entity that interacts with the system.



An actor is used in a use case diagram to describe the internal or external entities.

1. **Initial State Notation:**

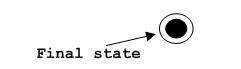
Initial state is defined to show the start of a process. This notation is used in almost all diagrams.



The usage of Initial State Notation is to show the starting point of a process.

1. **Final State Notation**

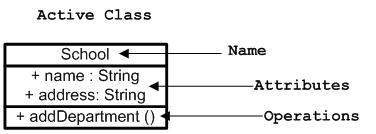
Final state is used to show the end of a process. This notation is also used in almost all diagrams to describe the end.



The usage of Final State Notation is to show the termination point of a process.

1. **Active Class Notation**

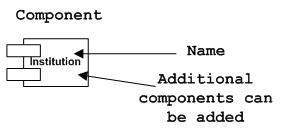
Active class looks similar to a class with a solid border. Active class is generally used to describe the concurrent behavior of a system.



Active class is used to represent the concurrency in a system.

1. **Component Notation**

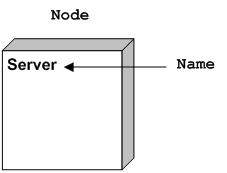
A component in UML is shown in the following figure with a name inside. Additional elements can be added wherever required.



Component is used to represent any part of a system for which UML diagrams are made.

1. **Node Notation**

A node in UML is represented by a square box as shown in the following figure with a name. A node represents the physical component of the system.



Node is used to represent the physical part of a system such as the server, network, etc.

**2.3 UML Diagrams**

* UML diagrams are the ultimate output of the entire discussions. All the elements, relationships are used to make a complete UML diagram and the diagram represents a system.
* The visual effect of the UML diagram is the most important part of the entire process. All the other elements are used to make it complete.
* UML includes the following diagrams:
  + Use case diagram
  + Sequence diagram
  + Collaboration diagram
  + State diagram
  + Component diagram
  + Deployment diagram

**2.3.1 Use Case Diagram**

Use case diagrams present an outside view of the manner the elements in a system behave and how they can be used in the context.

Use case diagrams comprise of −

* Use cases
* Actors
* Relationships like dependency, generalization, and association

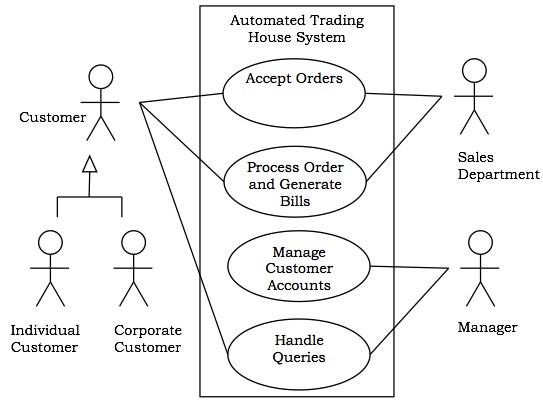
Use case diagrams are used −

* To model the context of a system by enclosing all the activities of a system within a rectangle and focusing on the actors outside the system by interacting with it.
* To model the requirements of a system from the outside point of view.

**Example**

Let us consider an Automated Trading House System. We assume the following features of the system −

* The trading house has transactions with two types of customers, individual customers and corporate customers.
* Once the customer places an order, it is processed by the sales department and the customer is given the bill.
* The system allows the manager to manage customer accounts and answer any queries posted by the customer.



**1. Use Case Diagram (Core Components)**

1. **Actors**
   * A role that a user plays with respect to the system including human user and other system.
2. **Use Case:**
   * A set of scenarios that describing an interaction between user and a system.
   * A use case is a single unit of meaningful work. E.g. Login, register, place an order, etc.
   * Each use case has a description which describes the functionality that will be built in the proposed system.
3. **System boundary:**
   * A rectangle diagram representing the boundary between the actors and the system.

**2. Use Case diagram (Core\_Relationship)**

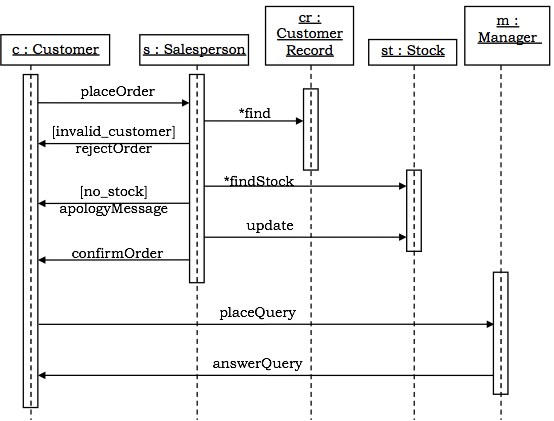
1. **Association:** 
   * Communication between an actor and a use case; represented by a solid line.
2. **Generalization:**
   * Relationship between one general use case and one specific use case.
   * Represented by a line with a triangular arrow head toward the parent use case, the more general modeling element.
3. **Include:**
   * A dotted line labeled <<include>> beginning at base use case and ending with an arrows pointing to the include use case.
   * An “include” relationship is used to indicate that a particular use case must include another use case to perform its function.
4. **Extend:**
   * A dotted line labeled <<extend>> with an arrow toward the base case.
   * The extending use case may add behavior to the base use case.

**2.3.2 Sequence Diagram:**

Sequence diagrams are interaction diagrams that illustrate the ordering of messages according to time.

**Notations** − These diagrams are in the form of two-dimensional charts. The objects that initiate the interaction are placed on the x–axis. The messages that these objects send and receive are placed along the y–axis, in the order of increasing time from top to bottom.

**Example** − A sequence diagram for the Automated Trading House System is shown in the following figure.

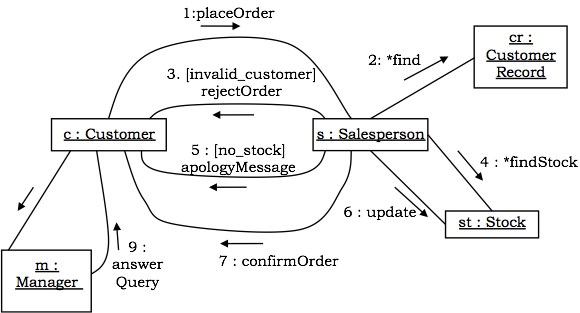


**2.3.3 Collaboration Diagram**

Collaboration diagrams are interaction diagrams that illustrate the structure of the objects that send and receive messages.

**Notations** − In these diagrams, the objects that participate in the interaction are shown using vertices. The links that connect the objects are used to send and receive messages. The message is shown as a labeled arrow.

**Example** − Collaboration diagram for the Automated Trading House System is illustrated in the figure below.



**2.3.4 State-chart Diagrams**

A state–chart diagram shows a state machine that depicts the control flow of an object from one state to another. A state machine portrays the sequences of states which an object undergoes due to events and their responses to events.

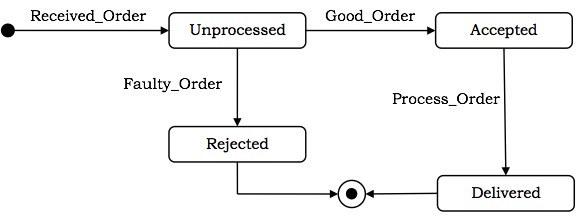
State–Chart Diagrams comprise of −

* States: Simple or Composite
* Transitions between states
* Events causing transitions
* Actions due to the events

State-chart diagrams are used for modeling objects which are reactive in nature.

**Example**

In the Automated Trading House System, let us model Order as an object and trace its sequence. The following figure shows the corresponding state–chart diagram.



**2.3.5 Activity Diagram**

An activity diagram depicts the flow of activities which are ongoing non-atomic operations in a state machine. Activities result in actions which are atomic operations.

Activity diagrams comprise of −

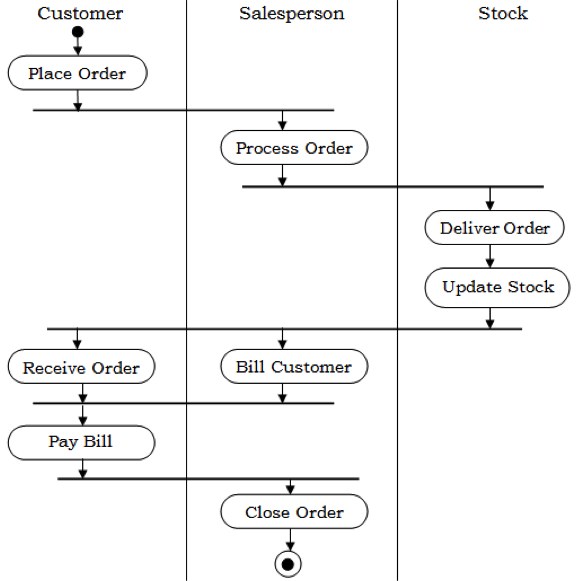
* Activity states and action states
* Transitions
* Objects

Activity diagrams are used for modeling −

* workflows as viewed by actors, interacting with the system.
* details of operations or computations using flowcharts.

**Example**

The following figure shows an activity diagram of a portion of the Automated Trading House System.



**2.4 Object Oriented Development Life Cycle**

* The essence of the software development process that consist of analysis, design, implementation, testing and refinement is to transform user’s need into a software solution that satisfies those needs.
* Some people view software development process as interesting but feel it has a little importance in developing software.
* In general, dynamic of software development provides little room for shortcuts, and bypass have been less than successful.



**2.4.1 Iterative and Incremental Life Cycle**

**a) Iterative Life Cycle**

- Iterative is a design methodology based on cyclic process or prototyping, testing, analyzing and refining a product or a process.

- Based on the results of testing the most recent iteration of a design, changes and refinements are made.

- The process is intended to ultimately improve the quality and functionality of design.

- In iterative design, interaction with the designed system is used as a form of research for informing and evolving a project, as successive versions, or iteration of a design are implemented.



- Requirements of the complete system and clearly defined and understood.

- Major requirements must be defined, however, some functionalities or requested enhancements may evolve with time.

- There is a time to the market constraint.

- A new technology is being used and is being learnt by the development team while working on the project.

- Resources with needed skill set are not available and are planned to be used on contract basis for specific iterations.

- There are some high risk features and goals which may change in its future.

**b) Incremental Life Cycle**

- The incremental approach is a method of software development where the model is designed, implemented and tested incrementally until the product is finished.  
- It involves both development and maintenance. The product is defined as finished when it satisfies all of its requirements.

**When to Use the incremental model:**

* + This model can be used when the requirements of the complete system are clearly defined and understood.
  + Major requirements must be defined; however, some details can evolve with time.
  + There is a need to get a product to the market early.
  + A new technology is being used.
  + Resources with needed skill set are not available.
  + There are some high risk features and goals.