OOAD UNIT 1:-

Course Code	Course Name	Hours Per Week			
		L	T	P	Credits
CS3CO15	Object Oriented Analysis and Design	3	1	2	5

Unit I

Structure of Complex Systems, Object Oriented Development Methods, Characteristics of Objects, Fundamental Concepts of Object orientation, UML- Overview, RUP and its Phases

Unit II

Models, Concepts in UML, Structural and Behavioral Models, Use Cases and functional Requirements, Use Case Descriptions, Classes, Relationships, Association, Generalization, Realization, Dependencies, Constraints

Unit III

State Machine View, Activity View, Interaction View, Sequence Diagram, Collaboration Diagram, Interaction Diagrams

Unit IV

Physical View, Component Diagram, Deployment Diagram, Package, Dependencies on Packages, Modelling System and Subsystems, Patterns and Types of Patterns, Applying Patterns

Unit V

Object Oriented Testing, Types of Testing, Quality Assurance Methods, Reusability, Reverse Engineering, Case Studies

Text Book:

1. Grady Booch, Object Oriented Analysis and Design with Applications, Addison Wesley
2. James Rambaugh, Ivar Jacobson, Grady Booch, The Unified Modelling Language Reference Manual, Addison Wesley

Reference Book:

1. Design Patterns - Elements of Reusable Object-Oriented Software, Gamma, et. al., Addison-Wesley.
2. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and IterativeDevelopment, by Craig Larman, Pearson Education.

Object-Oriented Analysis

Object–Oriented Analysis (OOA) is the procedure of identifying software engineering requirements and developing software specifications in terms of a software system's object model, which comprises of interacting objects.

The main difference between object-oriented analysis and other forms of analysis is that in object-oriented approach, requirements are organized around objects, which integrate both data and functions. They are modelled after real-world objects that the system interacts with. In traditional analysis methodologies, the two aspects - functions and data - are considered separately.

Grady Booch has defined OOA as, *"Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain"*.

The primary tasks in object-oriented analysis (OOA) are −

- Identifying objects
- Organizing the objects by creating object model diagram
- Defining the internals of the objects, or object attributes
- Defining the behavior of the objects, i.e., object actions
- Describing how the objects interact

The common models used in OOA are use cases and object models.

Object-Oriented Design

Object–Oriented Design (OOD) involves implementation of the conceptual model produced during object-oriented analysis. In OOD, concepts in the analysis model, which are technology−independent, are mapped onto implementing classes, constraints are identified and interfaces are designed, resulting in a model for the solution domain, i.e., a detailed description of how the system is to be built on concrete technologies.

The implementation details generally include −

- Restructuring the class data (if necessary),
- Implementation of methods, i.e., internal data structures and algorithms,
- Implementation of control, and
- Implementation of associations.

Grady Booch has defined object-oriented design as *"a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design"*.

Object-Oriented Programming

Object-oriented programming (OOP) is a programming paradigm based upon objects (having both data and methods) that aims to incorporate the advantages of modularity and reusability. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.

The important features of object–oriented programming are −

- Bottom–up approach in program design
- Programs organized around objects, grouped in classes
- Focus on data with methods to operate upon object's data
- Interaction between objects through functions
- Reusability of design through creation of new classes by adding features to existing classes

Some examples of object-oriented programming languages are C++, Java, Smalltalk, Delphi, C#, Perl, Python, Ruby, and PHP.

Firstly, we have to discuss to difference between ooad and oops.

OOP

- A programming approach based on Objects
- Language-based
- Use in system implementation phase (Coding) in SDLC
- The output is a working program

OOAD

- A designing approach for OOP
- Visualizing things
- Use in analyzing and design phase in SDLC
- The output is a set of diagrams

OOAD

In the object-oriented approach, the focus is on capturing the structure and behavior of information systems into small modules that combines both data and process. The main aim of Object Oriented Design (OOD) is to improve the quality and productivity of system analysis and design by making it more usable.

In analysis phase, OO models are used to fill the gap between problem and solution. It performs well in situation where systems are undergoing continuous design, adaption, and maintenance. It identifies the objects in problem domain, classifying them in terms of data and behavior.

The OO model is beneficial in the following ways −

- It facilitates changes in the system at low cost.
- It promotes the reuse of components.
- It simplifies the problem of integrating components to configure large system.
- It simplifies the design of distributed systems.

Object-Oriented Program Design

The object-oriented program design involves:

- Identifying the components of the system or application that you want to build
- Analyzing and identifying patterns to determine what components are used repeatedly or share characteristics
- Classifying components based on similarities and differences

After performing this analysis, you define classes that describe the objects your application uses.

When Should You Create Object-Oriented Design

You can implement simple programming tasks as simple functions. However, as the magnitude and complexity of your tasks increase, functions become more complex and difficult to manage.

1)Understand a Problem in Terms of Its Objects.

2) Identify Differences.

3) Add Only What Is Necessary:-These institutions might engage in activities that are not of interest to your application. During the design phase, determine what operations and data an object must contain based on your problem definition.

4) Reducing Redundancy

As the complexity of your program increases, the benefits of an object-oriented design become more apparent. For example, suppose that you implement the following procedure as part of your application:

1. Check inputs

2. Perform computation on the first input argument
3. Transform the result of step 2 based on the second input argument
4. Check validity of outputs and return values

5)Defining Consistent Interfaces

The use of a class as the basis for similar, but more specialized classes is a useful technique in object-oriented programming. This class defines a common interface. Incorporating this kind of class into your program design enables you to:

- Identify the requirements of a particular objective
- Encode requirements into your program as an interface class

6)Reducing Complexity

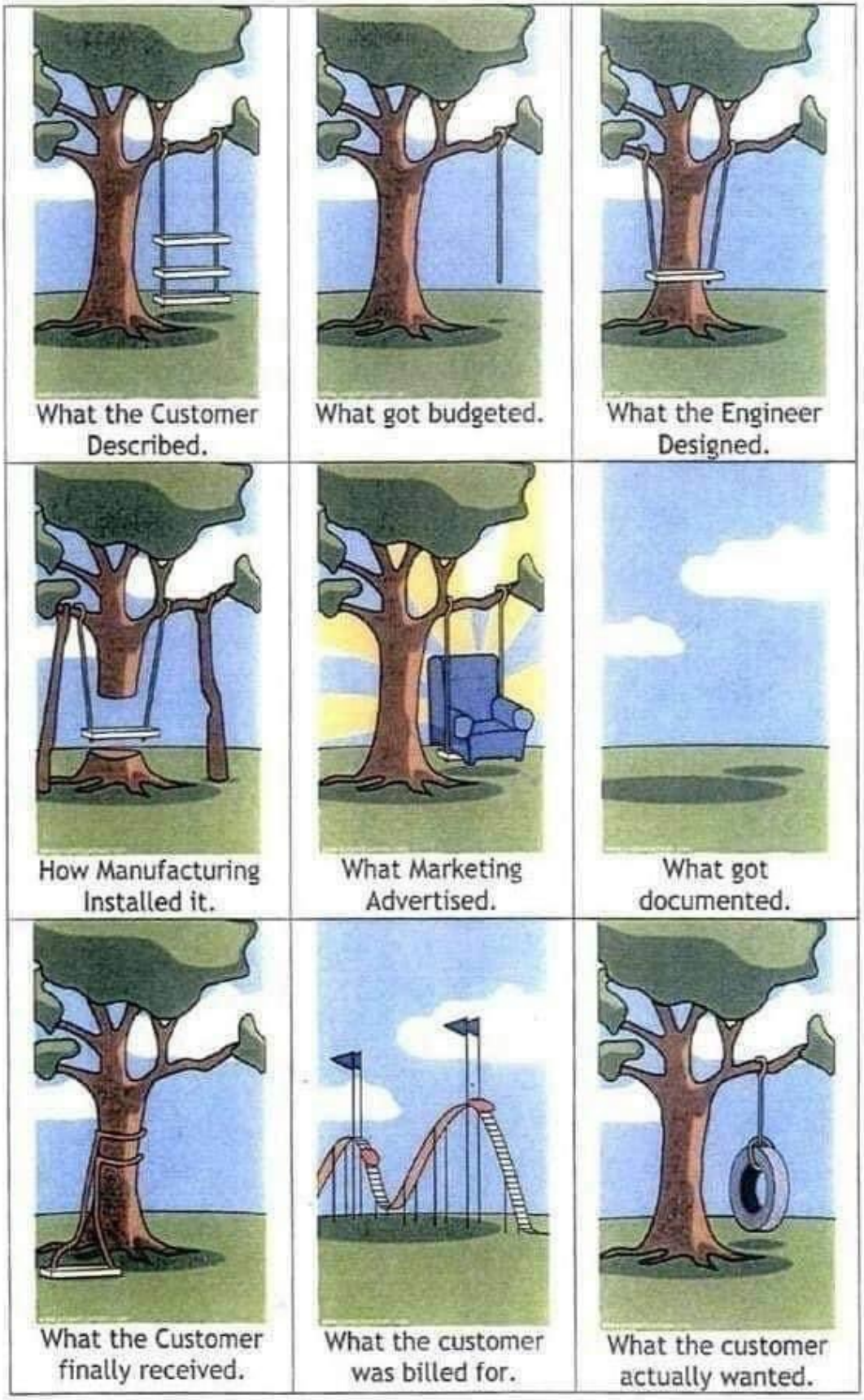
Objects reduce complexity by reducing what you must know to use a component or system:

- Objects provide an interface that hides implementation details.
- Objects enforce rules that control how objects interact.

7) Fostering Modularity

As you decompose a system into objects (car –> engine –> fuel system –> oxygen sensor), you form modules around natural boundaries. Classes provide three levels of control over code modularity:

- Public — Any code can access this particular property or call this method.
- Protected — Only this object's methods and methods of objects derived from this object's class can access this property or call this method.
- Private — Only the object's own methods can access this property or call this method.


What the Customer Described.
What got budgeted.
What the Engineer Designed.
How Manufacturing Installed it.
What Marketing Advertised.
What got documented.
What the Customer finally received.
What the customer was billed for.
What the customer actually wanted.

Structure Of Complex System:-

A **complex system** is a system composed of many components which may interact with each other. Examples of complex systems are Earth's global climate, organisms, the human brain, infrastructure such as power grid, transportation or communication systems, complex software and electronic systems, social and economic organizations (like cities), an ecosystem, a living cell, and ultimately the entire universe.

Complex systems are systems whose behavior is intrinsically difficult to model due to the dependencies, competitions, relationships, or other types of interactions between their parts or between a given system and its environment. Systems that are "complex" have distinct properties that arise from these relationships, such as nonlinearity, emergence, spontaneous order, adaptation, and feedback loops, among others. Because such systems appear in a wide variety of fields, the commonalities among them have become the topic of their independent area of research. In many cases, it is useful to represent such a system as a network where the nodes represent the components and links to their interactions.

Object Oriented Development Methods: -

What is Object Oriented Methodology?

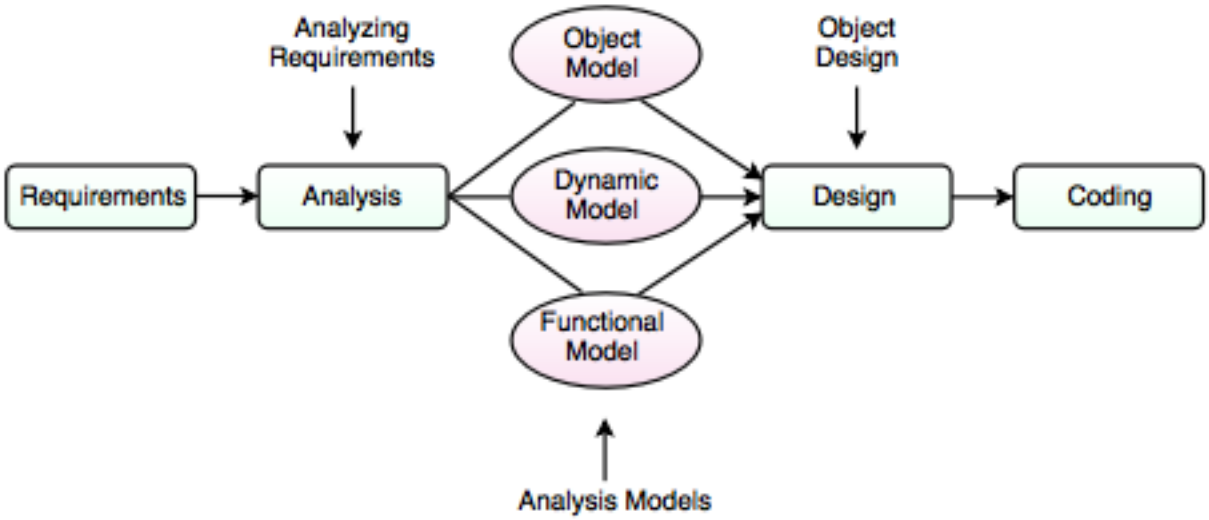
- It is a new system development approach, encouraging and facilitating re-use of software components.
- It employs international standard Unified Modeling Language (UML) from the Object Management Group (OMG).
- Using this methodology, a system can be developed on a component basis, which enables the effective re-use of existing components, it facilitates the sharing of its other system components.
- Object Oriented Methodology asks the analyst to determine what the objects of the system are?, What responsibilities and relationships an object has to do with the other objects? and How they behave over time?

There are three types of Object Oriented Methodologies

1. Object Modeling Techniques (OMT)
2. Object Process Methodology (OPM)
3. Rational Unified Process (RUP)

1. Object Modeling Techniques (OMT)

- It was one of the first object oriented methodologies and was introduced by Rumbaugh in 1991.
- OMT uses three different models that are combined in a way that is analogous to the older structured methodologies.



a.Analysis

- The main goal of the analysis is to build models of the world.
- The requirements of the users, developers and managers provide the information needed to develop the initial problem statement.

b. OMT Models

I. Object Model

- It depicts the object classes and their relationships as a class diagram, which represents the static structure of the system.
- It observes all the objects as static and does not pay any attention to their dynamic nature.

II. Dynamic Model

- It captures the behavior of the system over time and the flow control and events in the Event-Trace Diagrams and State Transition Diagrams.
- It portrays the changes occurring in the states of various objects with the events that might occur in the system.

III. Functional Model

- It describes the data transformations of the system.
- It describes the flow of data and the changes that occur to the data throughout the system.

c. Design

- It specifies all of the details needed to describe how the system will be implemented.
- In this phase, the details of the system analysis and system design are implemented.
- The objects identified in the system design phase are designed.

2. Object Process Methodology (OPM)

- It is also called as second generation methodology.
- It was first introduced in 1995.
- It has only one diagram that is the Object Process Diagram (OPD) which is used for modeling the structure, function and behavior of the system.
- It has a strong emphasis on modeling but has a weaker emphasis on process.
- It consists of three main processes:

I. Initiating: It determines high level requirements, the scope of the system and the resources that will be required.

II. Developing: It involves the detailed analysis, design and implementation of the system.

III. Deploying: It introduces the system to the user and subsequent maintenance of the system.

3. Rational Unified Process (RUP)

- It was developed in Rational Corporation in 1998.
- It consists of four phases which can be broken down into iterations.
  I. Inception
  II. Elaboration
  III. Construction
  IV. Transition
- Each iteration consists of nine work areas called disciplines.
- A discipline depends on the phase in which the iteration is taking place.
- For each discipline, RUP defines a set of artefacts (work products), activities (work undertaken on the artefacts) and roles (the responsibilities of the members of the development team).

Objectives of Object Oriented Methodologies

- To encourage greater re-use.
- To produce a more detailed specification of system constraints.
- To have fewer problems with validation (Are we building the right product?).

Benefits of Object Oriented Methodologies

1. It represents the problem domain, because it is easier to produce and understand designs.

2. It allows changes more easily.

3. It provides nice structures for thinking, abstracting and leads to modular design.

4. Simplicity:

- The software object's model complexity is reduced and the program structure is very clear.

5. Reusability:

- It is a desired goal of all development process.
- It contains both data and functions which act on data.
- It makes easy to reuse the code in a new system.
- Messages provide a predefined interface to an object's data and functionality.

6. Increased Quality:

- This feature increases in quality is largely a by-product of this program reuse.

7. Maintainable:

- The OOP method makes code more maintainable.
- The objects can be maintained separately, making locating and fixing problems easier.

8. Scalable:

- The object oriented applications are more scalable than structured approach.
- It makes easy to replace the old and aging code with faster algorithms and newer technology.

9. Modularity:

- The OOD systems are easier to modify.
- It can be altered in fundamental ways without ever breaking up since changes are neatly encapsulated.

10. Modifiability:

- It is easy to make minor changes in the data representation or the procedures in an object oriented program.

11. Client/Server Architecture:

- It involves the transmission of messages back and forth over a network.

Characteristics of an object: -

sFor example, you can think of your bank account as an object, but it is not made of material. (Although you and the bank may use paper and other material in keeping track of your account, your account exists independently of this material.) Although it is not material, your account has properties (a balance, an interest rate, an owner) and you can do things to it (deposit money, cancel it) and it can do things (charge for transactions, accumulate interest).

The last three items on the list seem clear enough. In fact, they have names:

- An object has **identity** (each object is a distinct individual).
- An object has **state** (it has various properties, which might change).
- An object has **behavior** (it can do things and can have things done to it).

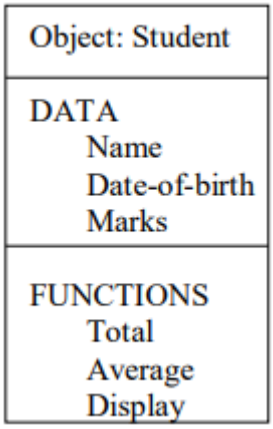
Fundamentals Concept of Object Orientation: -

1. Objects

Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program must handle. The fundamental idea behind object-oriented approach is to combine both data and function into a single unit and these units are called objects.

The term objects mean a combination of data and program that represent some real word entity.

When a program executed, the object interacts by sending messages to one another. Each object contain data, and code to manipulate the data.



2. Class

A group of objects that share common properties for data part and some program part are collectively called as class. In C ++ a class is a new data type that contains member variables and member functions that operate on the variables.

The entire set of data and code of an object can be made a user-defined data type with the help of a class. Objects are variable of the type class. Once a class has been defined, we can create any number of objects belonging to that class.

When class is defined, objects are created as:

<classname><objectname>;

If **employee** has been defined as a class, then the statement

employee manager;

will create an object **manager** belonging to the class **employee**.

3. Encapsulation

Wrapping of data and functions together as a single unit is known as encapsulation. By default, data is not accessible to outside world and they are only accessible through the functions which are wrapped in a class. Prevention of data from direct access by the program is called data hiding or information hiding.

4. Data Abstraction

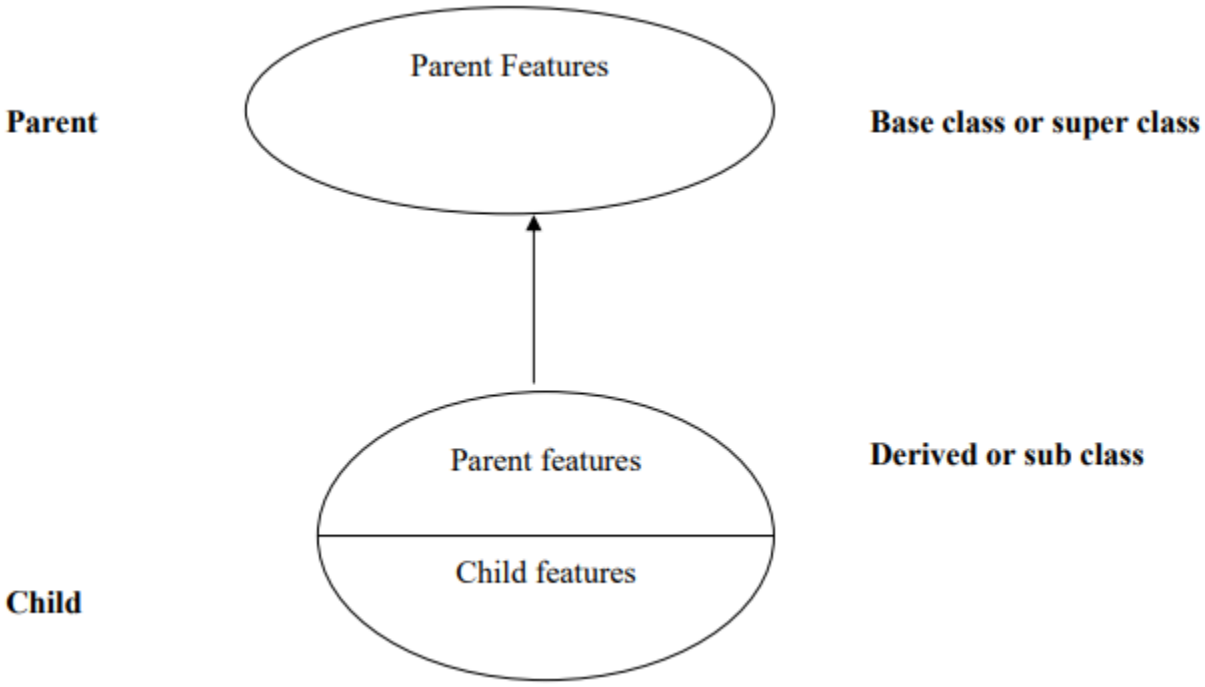
Abstraction refers to the act of representing essential features without including the back ground details or explanation. Classes use the concept of abstraction and are defined as a list of attributes such as size, weight, cost and functions to operate on these attributes. They encapsulate all essential properties of the object that are to be created. The attributes are called as data members as they hold data and the functions which operate on these data are called as member functions.

Class use the concept of data abstraction so they are called abstract data type (ADT).

5. Inheritance

Inheritance is the mechanism by which one class can inherit the properties of another. It allows a hierarchy of classes to be build, moving from the most general to the most specific. When one class is inherited by another, the class that is inherited is called the base class. The inheriting class is called the derived class. In general, the process of inheritance begins with the definition of a base class. The base class defines all qualities that will be common to any derived class. . In OOPs, the concept of inheritance provides the idea of reusability. In essence, the base class represent the most general description of a set of traits.

The derived class inherits those general traits and adds properties that are specific to that class.



6. Polymorphism

Polymorphism comes from the Greek words “poly” and “morphism”. “poly” means many and “morphism” means form i.e.. many forms. Polymorphism means the ability to take more than one form. For example, an operation have different behavior in different instances. The behavior depends upon the type of the data used in the operation.

Different ways to achieving polymorphism in C++ program:

1) Function overloading

2) Operator overloading

It is able to express the operation of addition by a single operater say '+'. When this is possible you use the expression x + y to denote the sum of x and y, for many different types of x and y; integers , float and complex no. You can even define the + operation for two strings to mean the concatenation of the strings.

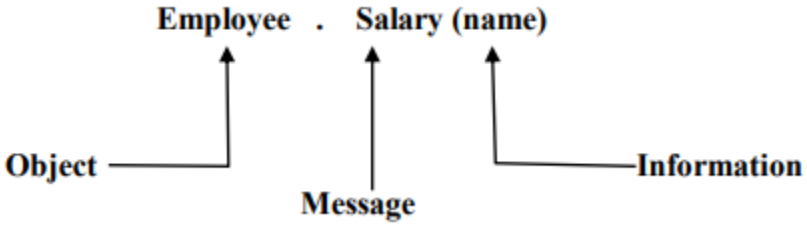
7. Dynamic Binding

Binding refers to the linking of a procedure call to the code to the executed in response to the call. Dynamic binding means the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with a polymorphic reference depends upon the dynamic type of that reference.

8. Message Passing

An object oriented program consists of a set of objects that communicate with each other.

A message for an object is a request for execution of a procedure and therefore will invoke a function (procedure) in the receiving object that generates the desired result. Message passing involves specifying the name of the object, the name of the function (message) and information to be sent.

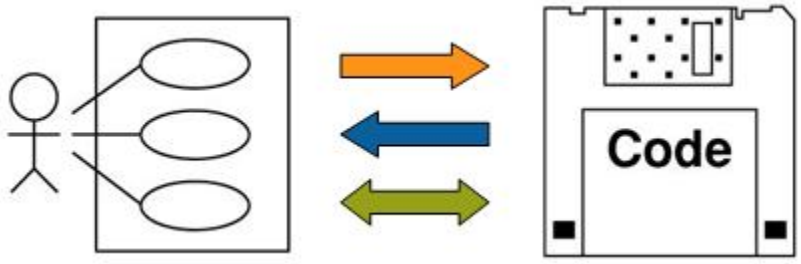


UML- Overview

Introduction to UML

UML is a standardized graphical display format for the *visualization*, *specification*, *design* and *documentation* of (software) systems. It offers a set of standardized diagram types with which complex data, processes and systems can easily be arranged in a clear, intuitive manner.

UML is neither a procedure nor a process; rather, it provides a "dictionary" of symbols - each of which has a specific meaning. It offers diagram types for object-oriented analysis, design and programming, thereby ensuring a seamless transition from requirements placed on a system to final implementation. Structure and system behavior are likewise shown, thereby offering clear reference points for solution optimization.



Documentation

One major aspect of UML is the ability to use diagrams as a part of project documentation. These can be utilized in various ways in the most diverse kinds of documents; for example, Use Case Diagrams used in describing functional requirements can be specified in the requirements definition. Classes or component diagrams can be used as software architecture in a design document. As a matter of principle, UML diagrams can be used in practically any technical documentation (e.g. test plans) while also serving as part of the user handbook.

Historical Development of UML

Despite the fact that the idea of object orientation is more than 30 years old, and the development of object-oriented programming languages spans almost the same length of time, the first books on object-oriented analysis and design methods didn't appear until the early 1990's. The godfathers of this idea were Grady Booch,

Ivar Jacobson and James Rumbaugh. Each of these three "veterans" had developed his own method, each one specialized in and limited to its own area of application.

In 1995 Booch and Rumbaugh began to merge their methods into a common Unified Method (UM) notation. The Unified Method was soon renamed as Unified Modeling Language (UML), a more adequate term since it is mostly concerned with the unification of the graphical presentation and semantics of modeling elements, and does not describe an actual method. Indeed, "modeling language" is basically just another term for notation.

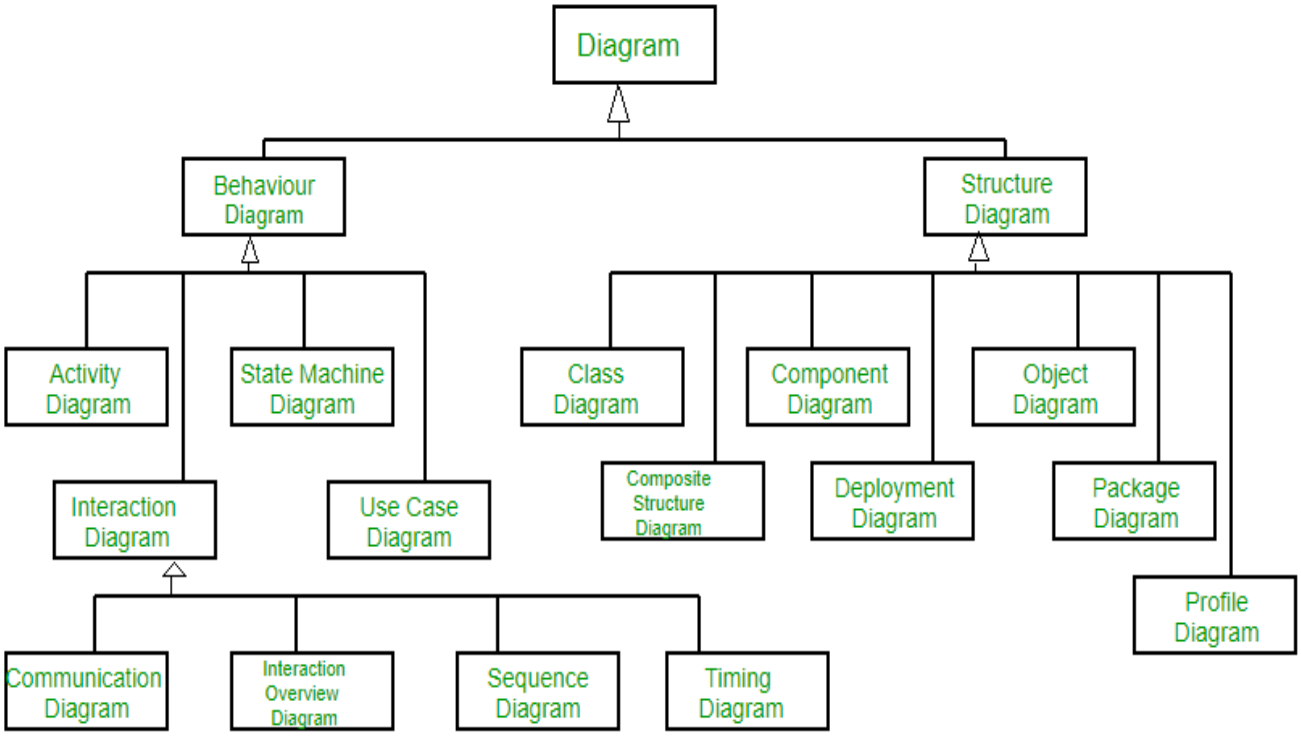
Short time later Ivar Jacobson joined in the foray and his Use Cases were soon integrated. From that point on, these three pioneers called themselves the "Amigos".

Advantages of UML

The use of UML as a "common language" leads to an improvement in cooperation between technical and non-technical competencies like project leaders, business analysts, software/hardware architects, designers and developers. It helps in the better understanding of systems, in revealing simplification and/or recoverability options, and in the easier recognition of possible risks. Through early detection of errors in the analysis and design phase of a project, costs can be reduced during the implementation phase. The advantages associated with Round-Trip Engineering offer developers the ability to save a great deal of time.

UML is linked with **object-oriented** design and analysis. UML makes the use of elements and forms associations between them to form diagrams. Diagrams in UML can be broadly classified as:

1. **Structural Diagrams –** Capture static aspects or structure of a system. Structural Diagrams include: Component Diagrams, Object Diagrams, Class Diagrams and Deployment Diagrams.
2. **Behavior Diagrams –** Capture dynamic aspects or behavior of the system. Behavior diagrams include: Use Case Diagrams, State Diagrams, Activity Diagrams and Interaction Diagrams.



Structural UML Diagrams –

1. **Class Diagram –** The most widely use UML diagram is the class diagram. It is the building block of all object oriented software systems. We use class diagrams to depict the static structure of a system by showing system's classes,their methods and attributes. Class diagrams also help us identify relationship between different classes or objects.
2. **Composite Structure Diagram –** We use composite structure diagrams to represent the internal structure of a class and its interaction points with other parts of the system. A composite structure diagram represents relationship between parts and their configuration which determine how the classifier (class, a component, or a deployment node) behaves. They represent internal structure of a structured classifier making the use of parts, ports, and connectors. We can also model collaborations using composite structure diagrams. They are similar to class diagrams except they represent individual parts in detail as compared to the entire class.

3. **Object Diagram –** An Object Diagram can be referred to as a screenshot of the instances in a system and the relationship that exists between them. Since object diagrams depict behavior when objects have been instantiated, we are able to study the behavior of the system at a particular instant. An object diagram is similar to a class diagram except it shows the instances of classes in the system. We depict actual classifiers and their relationships making the use of class diagrams. On the other hand, an Object Diagram represents specific instances of classes and relationships between them at a point of time.

4.Component Diagram – Component diagrams are used to represent the how the physical components in a system have been organized. We use them for modelling implementation details. Component Diagrams depict the structural relationship between software system elements and help us in understanding if functional requirements have been covered by planned development. Component Diagrams become essential to use when we design and build complex systems. Interfaces are used by components of the system to communicate with each other.

5.Deployment Diagram – Deployment Diagrams are used to represent system hardware and its software.It tells us what hardware components exist and what software components run on them.We illustrate system architecture as distribution of software artifacts over distributed targets. An artifact is the information that is generated by system software. They are primarily used when a software is being used, distributed or deployed over multiple machines with different configurations.
6.Package Diagram – We use Package Diagrams to depict how packages and their elements have been organized. A package diagram simply shows us the dependencies between different packages and internal composition of packages. Packages help us to organize UML diagrams into meaningful groups and make the diagram easy to understand. They are primarily used to organise class and use case diagrams.

Behavior Diagrams –

1. **State Machine Diagrams –** A state diagram is used to represent the condition of the system or part of the system at finite instances of time. It’s a behavioral diagram and it represents the behavior using finite

state transitions. State diagrams are also referred to as **State machines** and **State-chart Diagrams.** These terms are often used interchangeably. So simply, a state diagram is used to model the dynamic behavior of a class in response to time and changing external stimuli.

2. **Activity Diagrams –** We use Activity Diagrams to illustrate the flow of control in a system. We can also use an activity diagram to refer to the steps involved in the execution of a use case. We model sequential and concurrent activities using activity diagrams. So, we basically depict workflows visually using an activity diagram. An activity diagram focuses on condition of flow and the sequence in which it happens. We describe or depict what causes a particular event using an activity diagram.
3. **Use Case Diagrams –** Use Case Diagrams are used to depict the functionality of a system or a part of a system. They are widely used to illustrate the functional requirements of the system and its interaction with external agents(actors). A use case is basically a diagram representing different scenarios where the system can be used. A use case diagram gives us a high-level view of what the system or a part of the system does without going into implementation details.
4. **Sequence Diagram –** A sequence diagram simply depicts interaction between objects in a sequential order i.e. the order in which these interactions take place. 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.
5. **Communication Diagram –** A Communication Diagram (known as Collaboration Diagram in UML 1.x) is used to show sequenced messages exchanged between objects. A communication diagram focuses primarily on objects and their relationships. We can represent similar information using Sequence diagrams; however, communication diagrams represent objects and links in a free form.

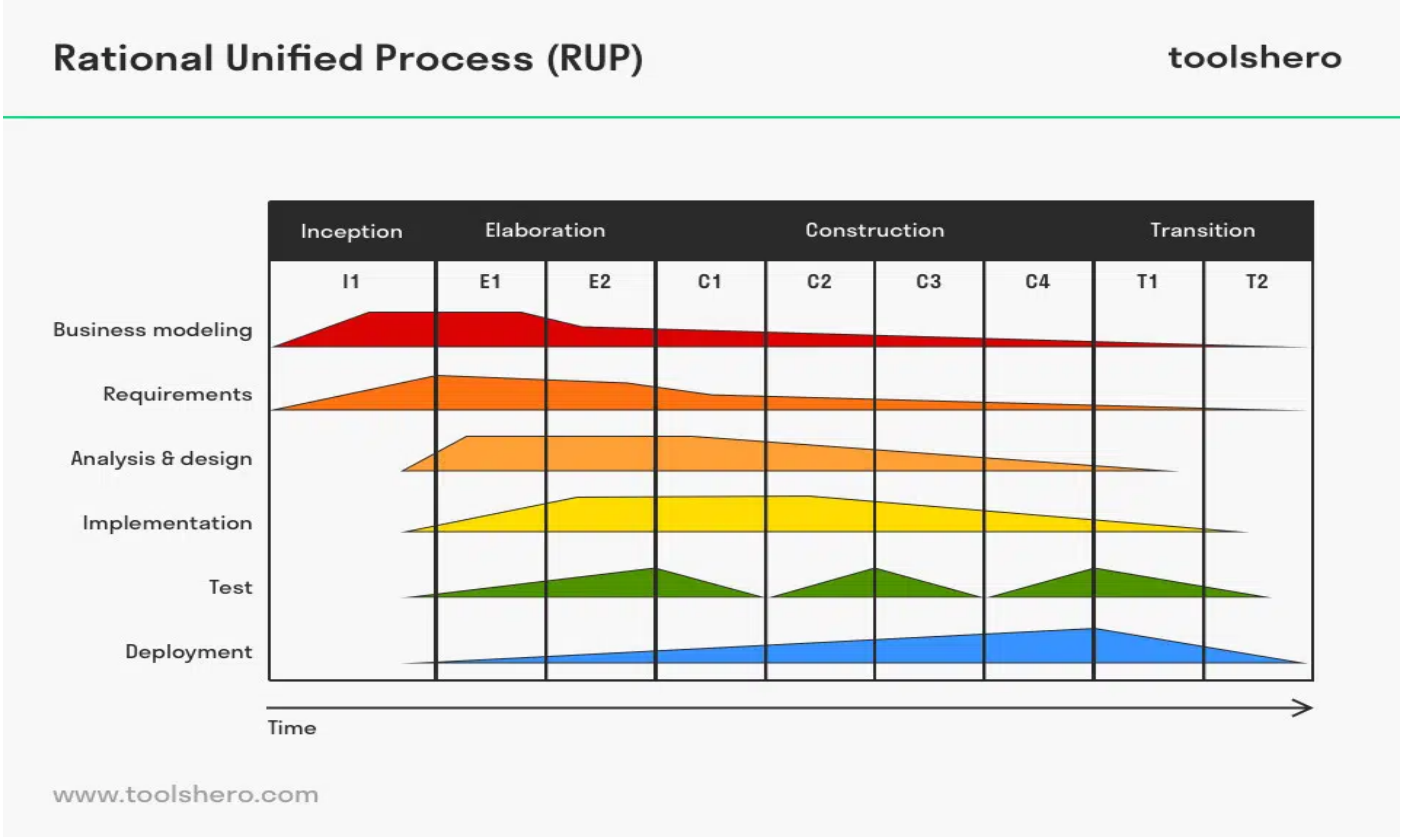
6. **Timing Diagram –** Timing Diagram are a special form of Sequence diagrams which are used to depict the behavior of objects over a time frame. We use them to show time and duration constraints which govern changes in states and behavior of objects.
7. **Interaction Overview Diagram –** An Interaction Overview Diagram models a sequence of actions and helps us simplify complex interactions into simpler occurrences. It is a mixture of activity and sequence diagrams.

RUP & ITS PHASES

What is a Rational Unified Process (RUP)?

Rational Unified Process (RUP) is an agile software development method, in which the life cycle of a project, or the development of software, is divided into four phases. Various activities take place during these phases: modelling, analysis and design, implementation, testing and application.

The Rational Unified Process (RUP) is iterative, meaning repeating; and agile. Iterative because all of the process's core activities repeat throughout the project. The process is agile because various components can be adjusted, and phases of the cycle can be repeated until the software meets requirements and objectives.



Rational Unified Process: time dimension

The time dimension means the dynamic organisation from the process over time. The software's life cycle is itself divided further into cycles. Each cycle corresponds to, for example, a period in which a new generation of a product is being worked on. The Rational Unified Process (RUP) divides development into the four consecutive phases:

- Inception phase
- Elaboration phase
- Construction phase
- Transition phase

Each phase is finalised with a milestone. A milestone is a point in time where decisions of critical importance must to be made. In order to be able to make those decisions, the objectives must have been accomplished.

For example, a milestone from the first two phases is the progress of the use case. A use case is a description of a system's behaviour and describes who can do what using a system. This is an important component in the development of software.

As can also be seen in the RUP visualisation, testing already starts in the first phase. Normally, a product will already have to be completed by then. That is because this involves prototypes and test models.

Phase 1: Inception

During the first phase, the basic idea and structure of the project are determined. In this phase, the team meets regularly to determine the project's necessity, but also its viability and suitability. Viability and suitability also include the expected costs and the means needed to complete the project after the green light has been given.

Depending on the project, the result of the first phase could be:

- A vision statement
- First use case (20% completed)
- Market research results
- Financial prognosis

- Risk assessment
- Project plan
- Corporate or business model
- Prototypes

The results should then be assessed according to several criteria:

- Were all interested parties included and do they all agree?
- Are the requirements of the development reliable?
- Are the costs credible? What are the priorities and risks?

Phase 2: elaboration

During the elaboration phase, the system's requirements and its required architecture are assessed and analysed. This is where the project begins to take shape. The objective of the elaboration phase is to analyse products and to lay a foundation for the future architecture. Results of the elaboration phase include:

- Use case (80% completed)
- Description of the feasible architecture
- Project development plan
- Prototypes for tackling risks
- User manual

Criteria for the results:

- Is the architecture stable?
- Are important risks being tackled?
- Is the development plan sufficiently detailed and accurate?
- Do all interested parties agree on the current design?
- Are the expenditures acceptable?

Phase 3: construction

In the construction phase of the Rational Unified Process (RUP), the software system is constructed in its entirety. The emphasis is on the development of components and other features of the system. The majority of coding also takes place in this phase. In this production process, the emphasis is on

managing costs and means, as well as ensuring quality. Results from the production phase include:

- Fully completed software system
- User manual

To be assessed according to:

- Is the product stable and complete enough for use?
- Are all interested parties/users ready for the transition into the product's usage?
- Are all the expenditures and means still in good order?

Phase 4: transition

The objective of the transition phase is to transfer the product to its new user. As soon as the user starts using the system, problems almost always arise that require changes to be made to the system. The goal, however, is to ensure a positive and smooth transition to the user. Results and activities in the last phase:

- Beta testing
- Conversion of existing user databases
- Training new users
- Rolling out of the project to marketing and distribution

Input from the new users should guide the assessment here.

Rational Unified Process: process dimension

The various phases related to developing software systems are now clear. As in any other process, the RUP describes who does what, where, and when. The 'who' in this process is the employee who is actively engaged in building the system. 'What' refers to something concrete, a piece of information. These 'artefacts' may take many forms, for example that of a user case or prototype.

The various phases already indicate the various activities involved in the development of a system. Here follows a more detailed explanation of the core activities.

1. Corporate modelling

One of the problems in the use of technical systems is that of the system and the user not being able to communicate properly. This leads to inefficiency in multiple areas. For example, the input the developer receives from the user is not properly used for the development of the generation of systems. Rational Unified Process (RUP) partly solves this problem by creating an universal language and offering processes.

2. Requirements

The objective of requirements is to describe what the system should do and how it should function. Both the user and the developer should agree on the requirements as described in the first phase. Everything is included in a vision document. After that, a use case is developed.

3. Analysis and design

The objective of analysis and design is to show how the system is realised in the implementation phase. It should meet all requirements, be robust, and execute all its tasks as described in the use case. This model design functions as a blueprint for the rest of the process.

4. Implementation

Implementation is found throughout the Rational Unified Process (RUP), as is every other activity, but it is also one of the model's engineering disciplines. The objective of implementation is to construct the full system. This is where components are tested and released.

5. Testing

The objective of testing is to verify the proper integration of all the components and the software. The testing phase is also where defects are identified and resolved. Testing does not only happen in the testing phase. The Rational Unified Process (RUP) is iterative, so testing happens throughout the project.

Tests are carried out along three dimensions:

- Reliability
- Functionality
- Application management and system performance

6. Application

The objective of applying a system is, naturally, successfully releasing a software system and enabling the user to work with the new system. It includes many activities described in the transitional phase 4, including:

- Packaging
- Distribution
- Installation
- Help and assistance
- Beta tests
- Data migration
- Acceptance

Unit II

Models, Concepts in UML, Structural and Behavioral Models, Use Cases and functional Requirements, Use Case Descriptions, Classes, Relationships, Association, Generalization, Realization, Dependencies, Constraints.

Models: -

Intention of object-oriented modeling and design is to learn how to apply object -oriented concepts to all the stages of the software development life cycle. Object-oriented modeling and design are a way of thinking about problems using models organized around real-world concepts. The fundamental construct is the object, which combines both data structure and behavior.

Purpose of Models:

1. Testing a physical entity before building it
2. Communication with customers
3. Visualization
4. Reduction of complexity

Types of Models:

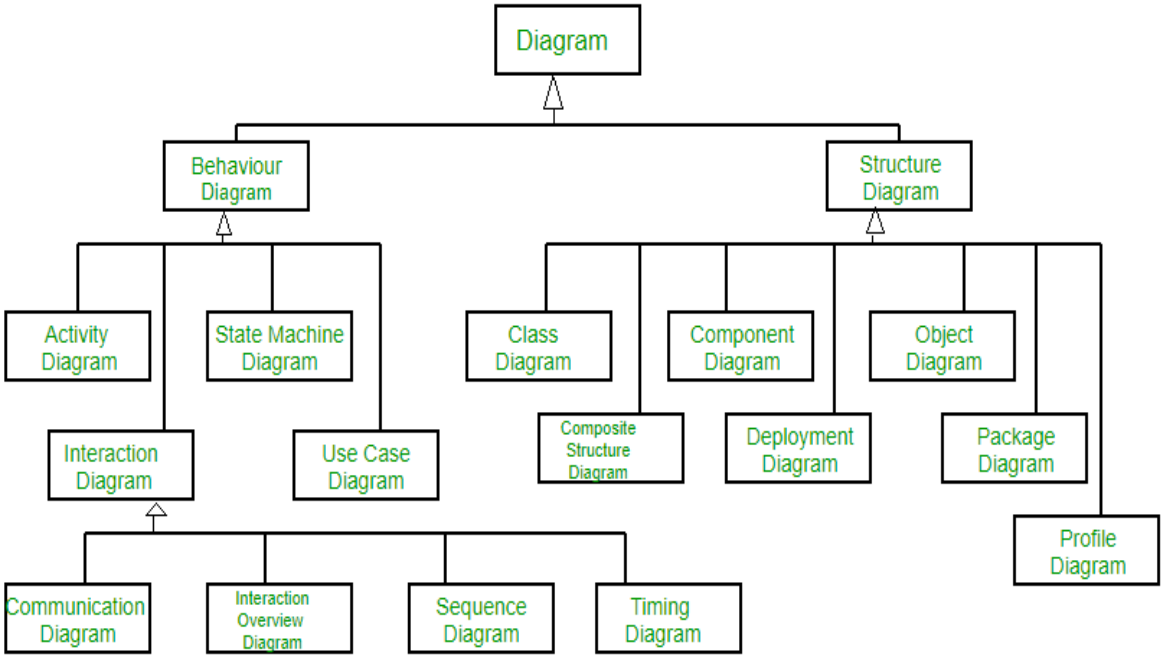
There are 3 types of models in the object-oriented modeling and design are: Class Model, State Model, and Interaction Model. These are explained as following below.

1. **Class Model:**
   The class model shows all the classes present in the system. The class model shows the attributes and the behavior associated with the objects.
   The class diagram is used to show the class model. The class diagram shows the class name followed by the attributes followed by the functions or the methods that are associated with the object of the class. Goal in constructing class model is to capture those concepts from the real world that are important to an application.

2. **State Model:**
   State model describes those aspects of objects concerned with time and the sequencing of operations – events that mark changes, states that define the context for events, and the organization of events and states. Actions and events in a state diagram become operations on objects in the class model. State diagram describes the state model.

3. **Interaction Model:**
   Interaction model is used to show the various interactions between objects, how the objects collaborate to achieve the behavior of the system as a whole.
   The following diagrams are used to show the interaction model:
   - Use Case Diagram
   - Sequence Diagram
   - Activity Diagram

Structural and Behavioral Models: -



Structural Modeling

Structural modeling captures the static features of a system. They consist of the following −

- Classes diagrams
- Objects diagrams
- Deployment diagrams
- Package diagrams
- Composite structure diagram
- Component diagram

Structural model represents the framework for the system and this framework is the place where all other components exist. Hence, the class diagram, component diagram and deployment diagrams are part of structural modeling. They all represent the elements and the mechanism to assemble them.

The structural model never describes the dynamic behavior of the system. Class diagram is the most widely used structural diagram.

Behavioral Modeling

Behavioral model describes the interaction in the system. It represents the interaction among the structural diagrams. Behavioral modeling shows the dynamic nature of the system. They consist of the following −

- Activity diagrams
- Interaction diagrams
- Use case diagrams

All the above show the dynamic sequence of flow in a system.

Use-Case Model

The Use-case model is defined as a model which is used to show how users interact with the system in order to solve a problem. As such, the use case model defines the user's objective, the interactions between the system and the user, and the system's behavior required to meet these objectives.

Various model elements are contained in use-case model, such as actors, use cases, and the association between them.

We use a use-case diagram to graphically portray a subset of the model in order to make the communication simpler. There will regularly be a numerous use-case diagram which is related to the given model, each demonstrating a subset of the model components related to a specific purpose. A similar model component might be appearing on a few use-case diagrams; however, each use-case should be consistent. If, in order to handle the use-case model, tools are used then this consistency restriction is automated so that any variations to the component of the model (changing the name, for instance) will be reflected automatically on each use-case diagram, which shows that component.

Origin of Use-Case

Nowadays, use case modeling is frequently connected with UML, in spite of the fact that it has been presented before UML existed. Its short history is:

- Ivar Jacobson, in the year 1986, originally formulated textual and visual modeling methods to specify use cases.
- And in the year 1992, his co-authored book named Object-Oriented Software Engineering - A Use Case Driven Approach, assisted with promoting the strategy for catching functional requirements, particularly in software development.

Components of Basic Model

There are various components of the basic model:

1. Actor
2. Use Case
3. Associations

Actor

Usually, actors are people involved with the system defined on the basis of their roles. An actor can be anything such as human or another external system.

Use Case

The use case defines how actors use a system to accomplish a specific objective. The use cases are generally introduced by the user to meet the objectives of the activities and variants involved in the achievement of the goal.

Associations

Associations are another component of the basic model. It is used to define the associations among actors and use cases they contribute in. This association is called communicates-association.

Advanced Model Components

There are various components of the advanced model:

1. Subject
2. Use-Case Package
3. Generalizations
4. Dependencies

Tips for Drawing a Use-Case Diagram

There are various tips for drawing a use-case diagram:

- It must be complete.
- It must be simple.
- The use-case diagram must show each and every interaction with the use case.
- It is must that the use-case should be generalized if it is large.

- At least one system module must be defined in the use case diagram.
- When there are number of actors or use-cases in the use-case diagram, only the significant use-cases must be represented.
- The use-case diagrams must be clear and easy so that anyone can understand them easily.

Importance of Use-Case Diagram

Use Cases have been broadly used over the last few years. There are various benefits of the use-case diagram:

- Use-case diagram provides an outline related to all components in the system. Use-case diagram helps to define the role of administrators, users, etc.
- The use-Case diagram helps to provide solutions and answers to various questions that may pop up if you begin a project unplanned.
- It helps us to define the needs of the users extensively and explore how it will work.

Basic Use-Case Diagram Symbols and Notations

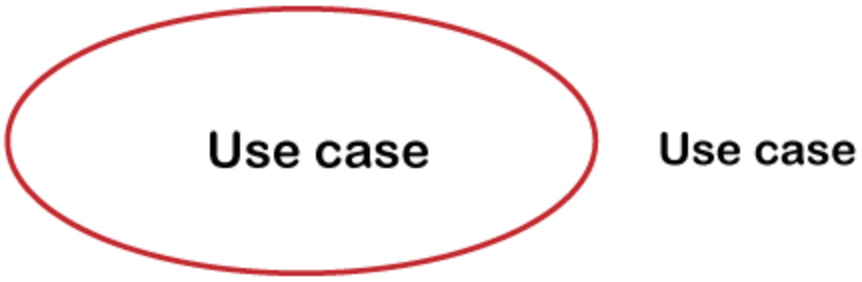
There are following use-case diagram symbols and notations:

System

With the help of the rectangle, we can draw the boundaries of the system, which includes use-cases. We need to put the actors outside the system's boundaries.

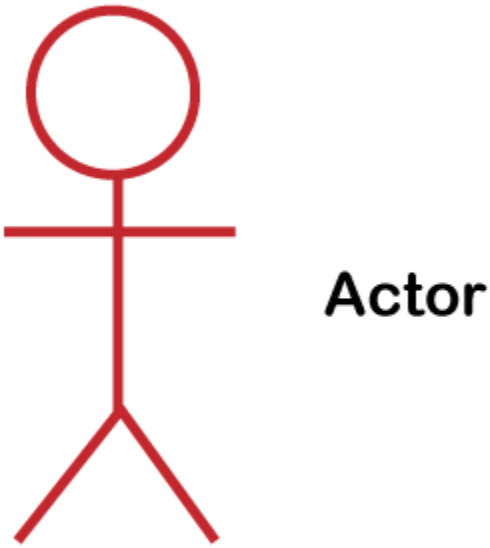
Use-Case

With the help of the Ovals, we can draw the use-cases. With the verb we have to label the ovals in order to represent the functions of the system.



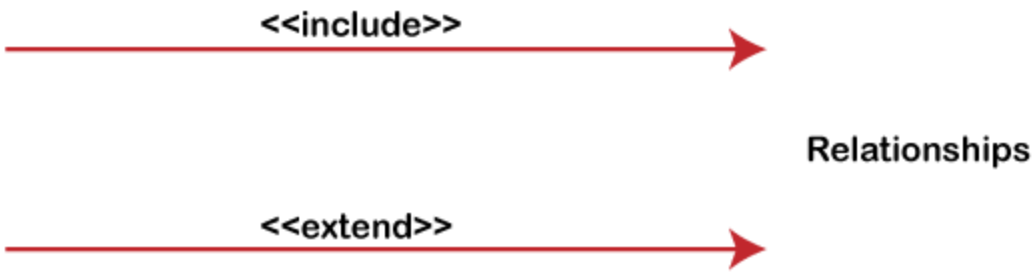
Actors

Actors mean the system's users. If one system is the actor of the other system, then with the actor stereotype, we have to tag the actor system.



Relationships

With the simple line we can represent relationships between an actor and use cases. For relationships between use-case, we use arrows which are labeled either "extends" or "uses". The "extends" relationship shows the alternative options under the specific use case. The "uses" relationship shows that single use-case is required to accomplish a job.



Guidelines for Better Use-Cases

With regards to examine the system's requirements, use-case diagrams are another one to one. Use-cases are simple to understand and visual. The following are some guidelines that help you to make better use cases that are appreciated by your customers and peers the same.

Generally, the use-case diagram contains use-cases, relationships, and actors. Systems and boundaries may be included in the complex larger diagrams. We'll talk about the guidelines of the use-case diagram on the basis of the objects.

Do not forget that these are the use case diagram's guidelines, not rules of the use-case diagram.

Actors

- **The actor's name should be meaningful and relevant to the business: -**
  If the use-case interacting with the outside organization, then we have to give the actor's name with the function instead of the organization name, such as Airline company is better than the PanAir).
- **Place inheriting actors below the parent actor** :-We have to place the inheriting actors below the parent actor because it makes the actors more readable and easily highlights the use-cases, which are exact for that actor.
- **External Systems are actors** :-If send-email is our use-case and when the use-case interrelates with the email management software, then in this case, the software is an actor to that specific user-case.

Use-Cases

- **The name of the use-case begins with a verb**
  The use-case models action, so the name of the use-case must start with a verb.
- **The name of the use-case must be descriptive**

The use-case is created to provide more information to others who are looking at a diagram, such as instead of "Print," "print Invoice is good.

- **Put the use-cases to the right of the included use-cases.**
  In order to add clarity and enhance readability, we have to place the included use-cases to the right of the invoking use-cases.
- **Place inheriting use-case below the parent use-case**
  In order to enhance the diagram's readability, we have to place the inheriting use-case below the parent use-case.

Systems/Packages

- Give descriptive and meaningful names to these objects.
- Use them carefully and only if needed.

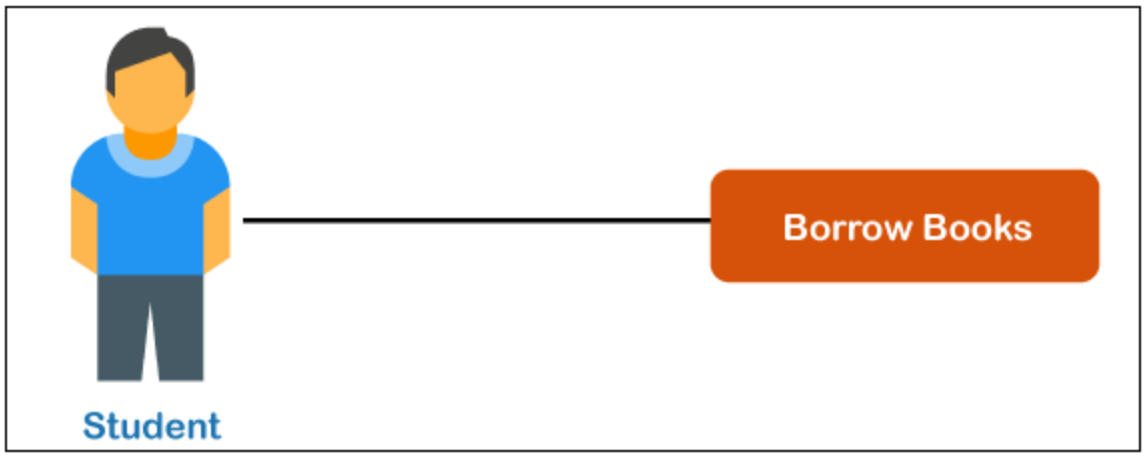
Relationships

- When we are using <<extend>> arrow, points to the base use-case.
- When we are using <<include>> then arrow points to the comprised use-case.
- Actor and use-case relationship do not display arrows.
- <<extend>> may have an optional extension condition.
- <<include>> and <<extend>> both are shown as dashed arrows.

Use-Case Examples

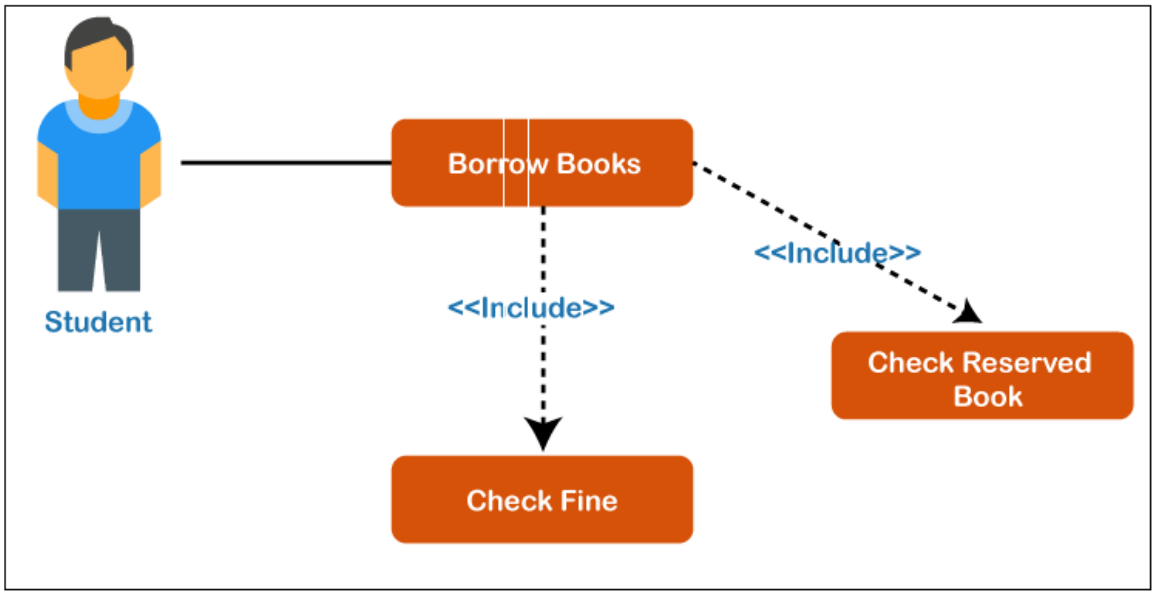
Use-Case Example-Association Link

In this use-case diagram, we show a group of use cases for a system which means the relationship among the use-cases and the actor.



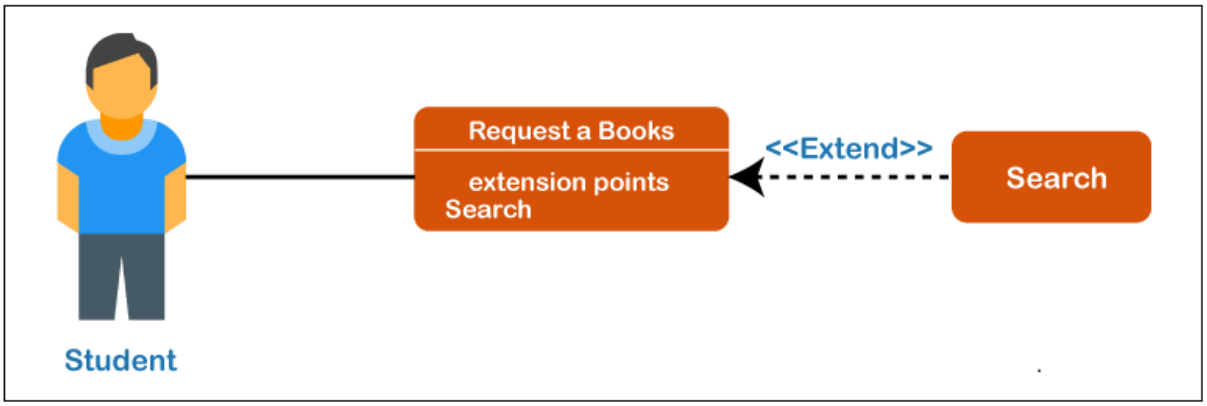
Use-Case Example-Include Relationship

In order to add additional functionality that is not specified in the base use-case, we use to include relationship. We use it to comprise the general behavior from an included use case into a base case so that we can support the reuse general behavior.



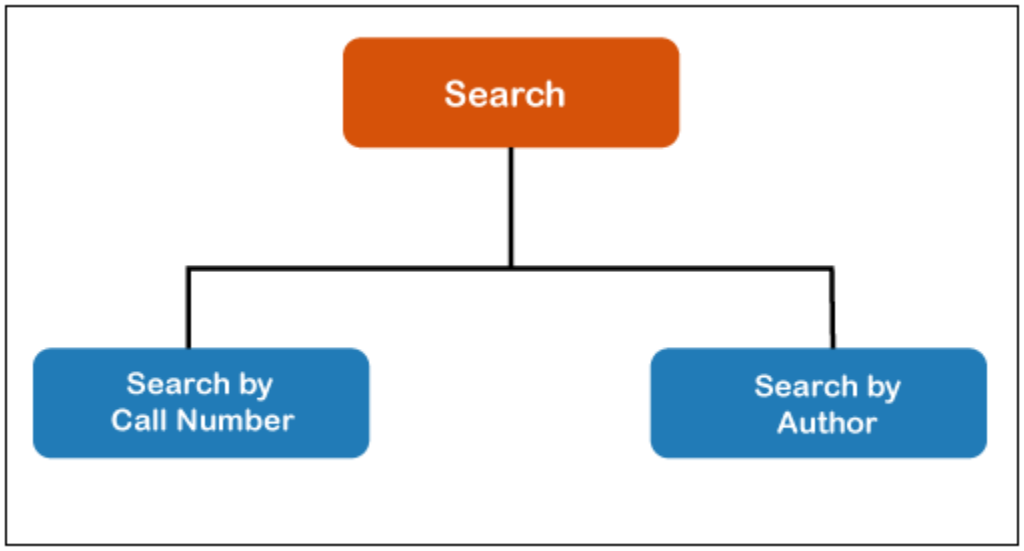
Use-Case Example-Extend Relationship

With the help of the extend relationship, we can show the system behavior or optional functionality. We use <<extend>> relationship in order to comprise optional behavior from an extending use-case in an extended use-case. For example, the below diagram of the use-case displays an extend connector and an extension point "Search".



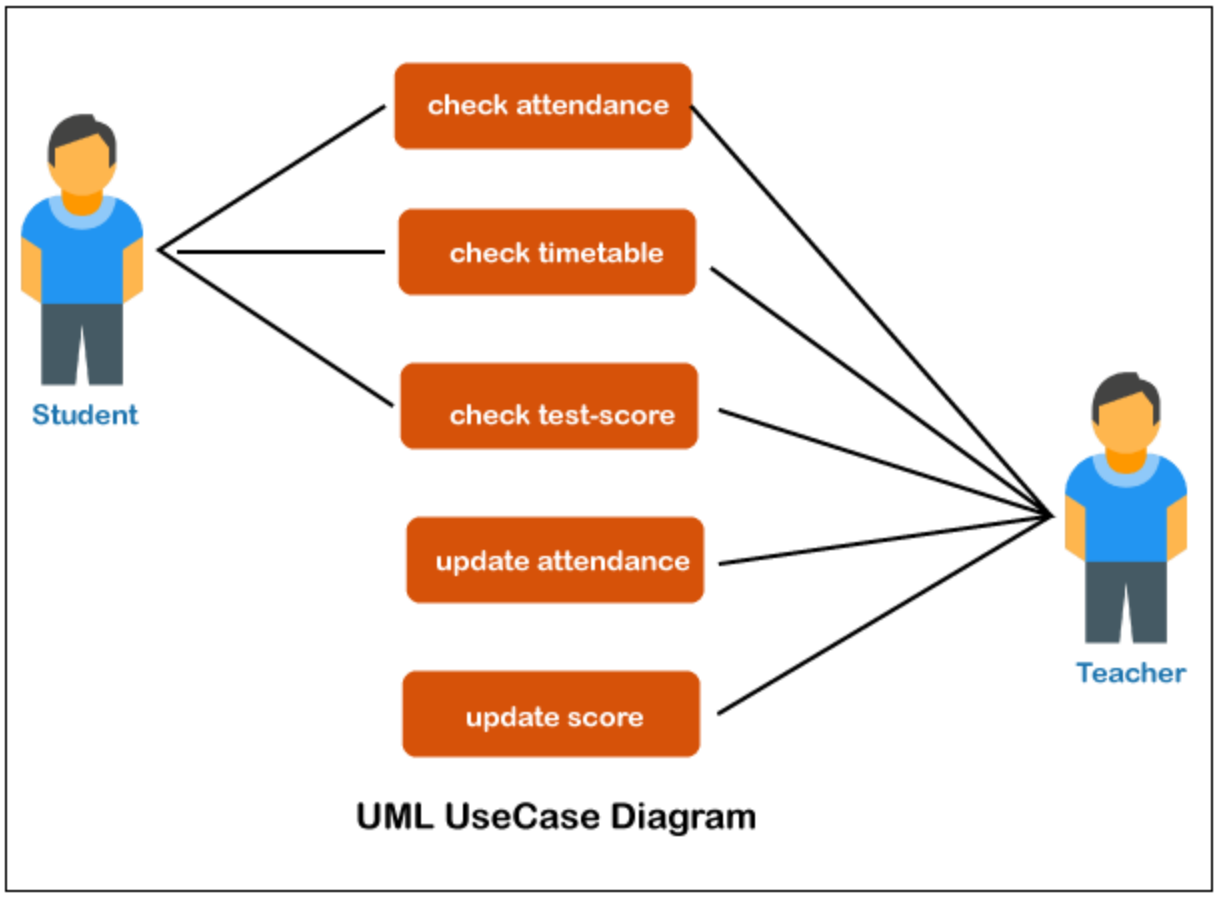
Use-Case Example-Generalization Relationship

In the generalization relationship, the child use-case can inherit the meaning and behavior of the parent use-case. The child use-case is able to override and adds the parent's behavior. The below diagram is an example of generalization relationship in which two generalization connector is connected among three use-cases.

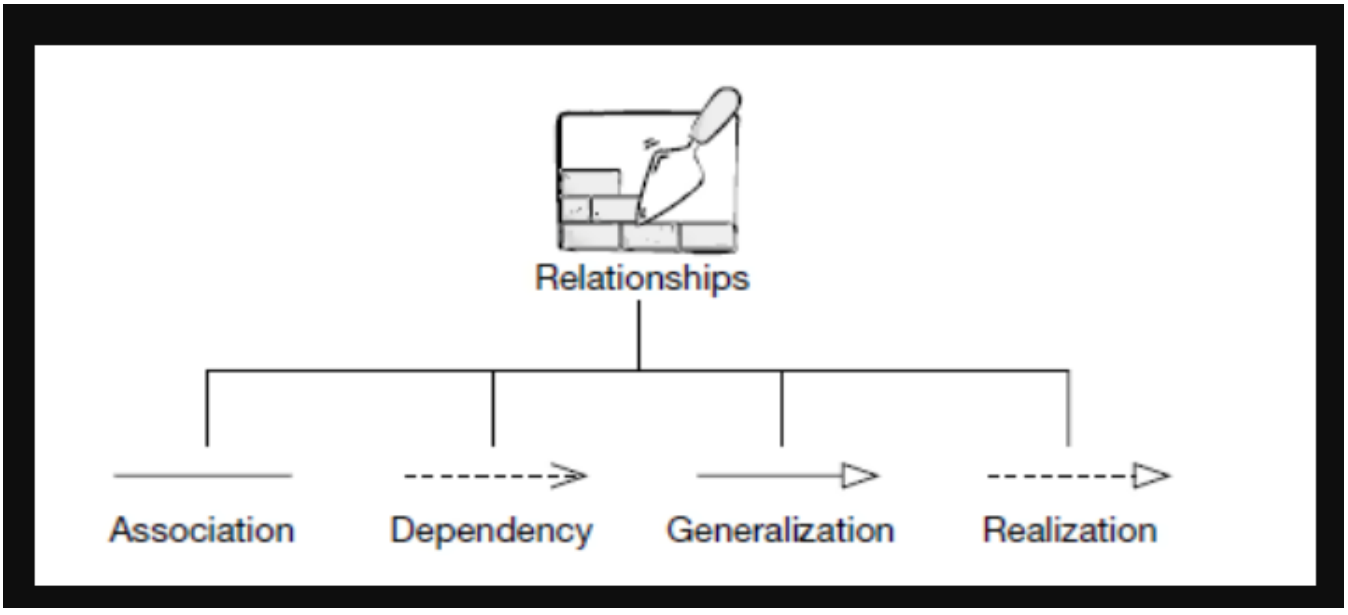


Use-Case Diagram-Student Management System

The below figure shows the working of the student management system:



Types of UML Class Diagram Relationships



Association

It is a set of links that connects elements of the UML model. It also defines how many objects are taking part in that relation.

Dependency

In a dependency relationship, as the name suggests, two or more elements are dependent on each other. In this kind of a relationship, if we make a change to a particular element, then it is likely possible that all the other elements will also get affected by the change.

Generalization

It is also called a parent-child relationship. In generalization, one element is a specialization of another general component. It may be substituted for it. It is mostly used to represent inheritance.

Realization

In a realization relationship of UML, one entity denotes some responsibility which is not implemented by itself and the other entity that implements them. This relationship is mostly found in the case of **interfaces.**

Association

It is a structural relationship that represents objects can be connected or associated with another object inside the system. Following constraints can be applied to the association relationship.

- **{implicit}** – Implicit constraints specify that the relationship is not manifest; it is based upon a concept.
- **{ordered}** – Ordered constraints specify that the set of objects at one end of an association are in a specific way.
- **{changeable}** – Changeable constraint specifies that the connection between various objects in the system can be added, removed, and modified as per the requirement.
- **{addOnly}** – It specifies that the new connections can be added from an object which is situated at the other end an association.

- **{frozen}** – It specifies that when a link is added between two objects, then it cannot be modified while the frozen constraint is active on the given link or a connection.

We can also create a class that has association properties; it is called as an association class.

Reflexive association

The reflexive association is a subtype of association relationship in UML. In a reflexive association, the instances of the same class can be related to each other. An instance of a class is also said to be an object.

Reflexive association states that a link or a connection can be present within the objects of the same class.

Let us consider an example of a class fruit. The fruit class has two instances, such as mango and apple. Reflexive association states that a link between mango and apple can be present as they are instances of the same class, such as fruit.

Directed association

As the name suggests, the directed association is related to the direction of flow within association classes.

In a directed association, the flow is directed. The association from one class to another class flows in a single direction only.

It is denoted using a solid line with an arrowhead.

Example:

ou can say that there is a directed association relationship between a server and a client. A server can process the requests of a client. This flow is unidirectional, that flows from server to client only. Hence a directed association relationship can be present within servers and clients of a system.

Dependency

Using a dependency relationship in UML, one can relate how various things inside a particular system are dependent on each other. Dependency is used to describe the relationship between various elements in UML that are dependent upon each other.

Stereotypes

- **«bind»** – Bind is a constraint which specifies that the source can initialize the template at a target location, using provided parameters or values.
- **«derive»** – It represents that the location of a source object can be calculated from the target object.
- **«friend»** – It specifies that the source has unique visibility in the target object.
- **«instanceOf»** – It specifies that the instance of a target classifier is the source object.
- **«instantiate»** – It specifies that the source object is capable of creating instances of a target object.
- **«refine»** – It specifies that the source object has exceptional abstraction than that of the target object.
- **«use»** – It is used when packages are created in UML. The use stereotype describes that the elements of a source package can be present inside the target package as well. It describes that the source package makes use of some elements of a target package.
- **«substitute»** – specifies that the client may be substituted for the supplier at runtime.
- **«access»** – It specifies that the source package access the elements of the target package **which is also called as a private merging.**
- **«import»** – It specifies that the target can import the element of a source package like they are defined inside the **target which is also called as a public merging.**
- **«permit»** – specifies that source element has access to the supplier element whatever the declared visibility of the supplier.
- **«extend»** – Helps you to specifies that the target can extend the behavior of the source element.
- **«include»** – Allows you to specifies the source element which can be included the behavior of another element at a specified location. (same as a function call in c/c++)

- **«become»** – It specifies that the target is similar to the source with different values and roles.
- **«call»** – It specifies that the source can invoke a target object method.
- **«copy»** – It specifies that the target object is independent, copy of a source object.
- **«parameter» – the** supplier is a parameter of the client operations**.**
- **«send» –** the client is an operation that sends the supplier some unspecified target.

Stereotypes among state machine

- **«send»** – Specifies that the source operation sends the target event.

Generalization

It is a relationship between a general entity and a unique entity which is present inside the system.

In a generalization relationship, the object-oriented concept called **inheritance** can be implemented. A generalization relationship exists between two objects, also called as entities or things. In a generalization relationship, one entity is a parent, and another is said to be as a child. These entities can be represented using inheritance.

In inheritance, a child of any parent can access, update, or inherit the functionality as specified inside the parent object. A child object can add its functionality to itself as well as inherit the structure and behavior of a parent object.

This type of relationship collectively known as a generalization relationship.

Stereotypes and their constraints

- **«implementation»** – This stereotype is used to represent that the child entity is being implemented by the parent entity by inheriting the structure and behavior of a parent object without violating the rules.**Note** This stereotype if widely used in a singl**e inheritance**.

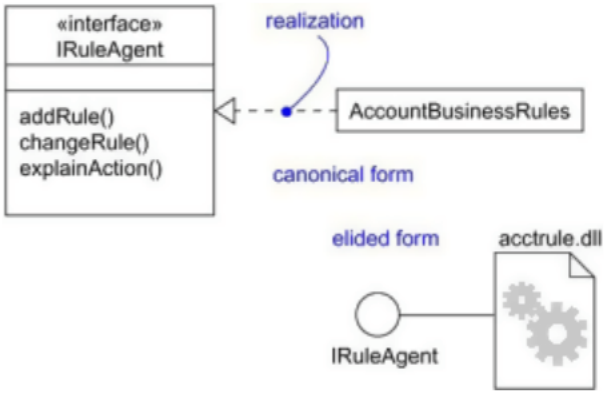
Generalization relationship contains constraints such as complete, incomplete to check whether all the child entities are being included in the relationship or not.

Realization

In a realization relationship of UML, one entity denotes some responsibility which is not implemented by itself and the other entity that implements them. This relationship is mostly found in the case of **interfaces.**

Realization can be represented in two ways:

- Using a **canonical form**
- Using an **elided form**



Realization in UML

In the above diagram, account business rules realize the interface IRuleAgent.

Types of realization:

1. Canonical formIn a realization relationship of UML, the canonical form is used to realize interfaces across the system. It uses an interface stereotype to create an interface and realization relationship is used to realize the particular interface.In a canonical form, the realization relationship is denoted using the dashed directed line with a sizeable open arrowhead.In the above diagram, interface Iruleagent is realized using an object called as Account Business Rules.
2. Elided formRealization in the UML class diagram can also be shown using an elided form. In an elided form, the interface is denoted using a circle which is also called as a lollipop notation.This interface, when realized using anything present inside the system, creates an elided structure.In the above

diagram, the interface Iruleagent is denoted using an elided form which is being realized by acctrule.dll.

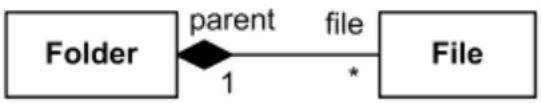
Composition

It is not a standard UML relationship, but it is still used in various applications.

Composite aggregation is a subtype of aggregation relation with characteristics as:

- it is a two-way association between the objects.
- It is a whole/part relationship.
- If a composite is deleted, all other parts associated with it are deleted.

Composite aggregation is described as a binary association decorated with a filled black diamond at the aggregate (whole) end.



Composition in UML

A folder is a structure which holds n number of files in it. A folder is used to store the files inside it. Each folder can be associated with any number of files. In a computer system, every single file is a part of at least one folder inside the file organization system. The same file can also be a part of another folder, but it is not mandatory. Whenever a file is removed from the folder, the folder stays un-affected whereas the data related to that particular file is destroyed. If a delete operation is executed on the folder, then it also affects all the files which are present inside the folder. All the files associated with the folder are automatically destroyed once the folder is removed from the system.

This type of relationship in UML is known by composite aggregation relationship.

Aggregation

An aggregation is a subtype of an association relationship in UML. Aggregation and composition are both the types of association relationship in UML. An aggregation

relationship can be described in simple words as ” an object of one class can own or access the objects of another class.”

In an aggregation relationship, the dependent object remains in the scope of a relationship even when the source object is destroyed.

Let us consider an example of a car and a wheel. A car needs a wheel to function correctly, but a wheel doesn’t always need a car. It can also be used with the bike, bicycle, or any other vehicles but not a particular car. Here, the wheel object is meaningful even without the car object. Such type of relationship is called an aggregation relation.

Summary

- Relationship in UML allows one thing to relate with other things inside the system.
- An association, dependency, generalization, and realization relationships are defined by UML.
- Composition relationship can also be used to represent that object can be a part of only one composite at a time.
- Association is used to describe that one object can be associated with another object.
- Dependency denotes that objects can be dependent on each other.
- A realization is a meaningful relationship between classifiers.
- Generalization is also called as a parent-child relationship.

Constraints: -

A constraint is **a Boolean condition that must evaluate to true in order for** an object diagram to instantiate a class diagram. Constraints can be written in English or they can be Boolean expressions in some language like Java .

Constraints **specify the conditions or restrictions that need to be satisfied over time**. They allow adding new rules or modifying existing ones. Constraints can appear in all the three models of object-oriented analysis. In Object Modelling, the constraints define the relationship between objects.

Constraints These simple and advanced properties of associations are sufficient for most of the structural relationships you'll encounter. However, if you want to specify a shade of meaning, the UML defines five constraints that may be applied to association relationships. First, you can specify whether the objects at one end of an association (with a multiplicity greater than one) are ordered or unordered.

1. ordered: - Specifies that the set of objects at one end of an association are in an explicit order.

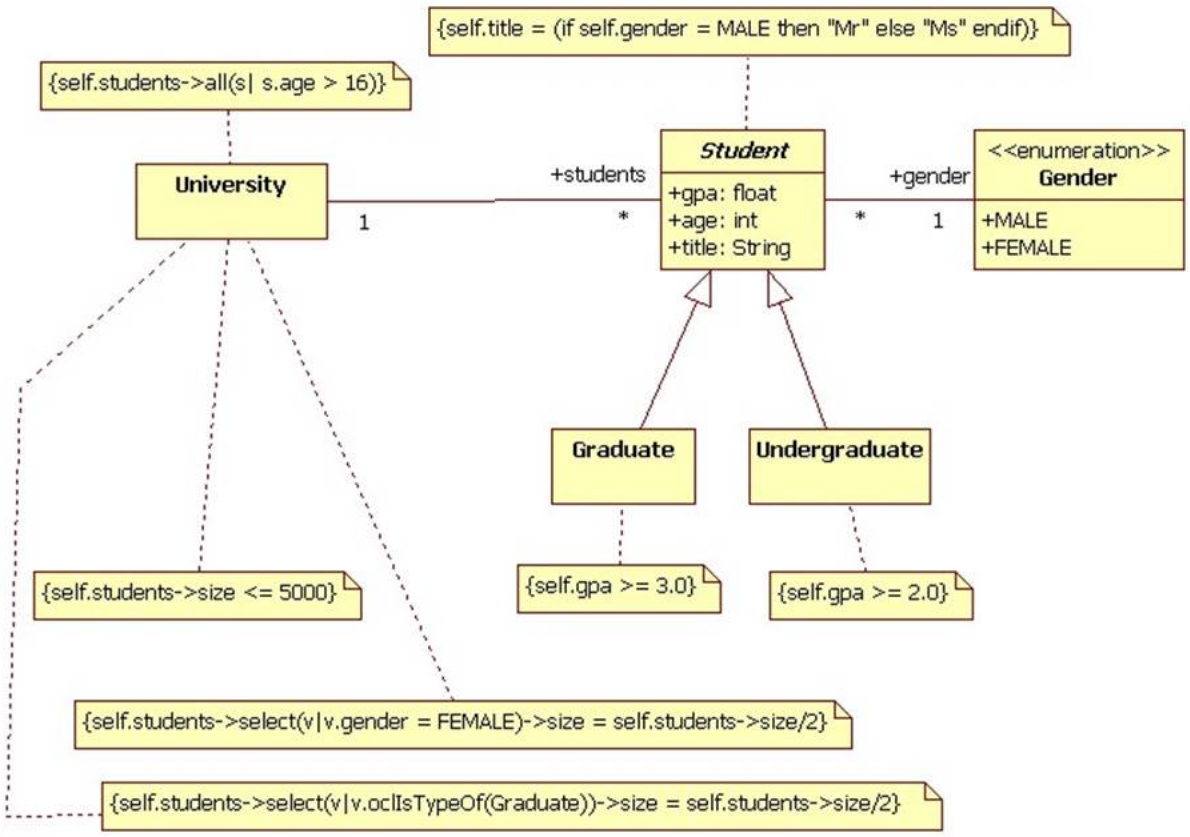
2. set: -The objects are unique with no duplicates.

3. bag: -The objects are non-unique, may be duplicates.

4. ordered: -set the objects are unique but ordered.

5. list or sequence the objects are ordered, may be duplicates.

6. Read-only: -A link, once added from an object on the opposite end of the association, may not be modified or deleted. The default in the absence of this constraint is unlimited changeability.



Functional Requirement: -

Functional Requirements

Functional requirements define a function that a system or system element must be qualified to perform and must be documented in different forms. The functional requirements describe the behavior of the system as it correlates to the system's functionality.

Functional requirements should be written in a simple language, so that it is easily understandable. The examples of functional requirements are authentication, business rules, audit tracking, certification requirements, transaction corrections, etc.

Non-functional requirements

Non-functional requirements are not related to the software's functional aspect. They can be the necessities that specify the criteria that can be used to decide the operation instead of specific behaviors of the system. Basic non-functional requirements are - usability, reliability, security, storage, cost, flexibility, configuration, performance, legal or regulatory requirements, etc.

They are divided into two main categories:

Execution qualities like security and usability, which are observable at run time.

Evolution qualities like testability, maintainability, extensibility, and scalability that embodied in the static structure of the software system.

Difference Between Functional & Non-Functional Requirement:-

Functional Requirements	Non-functional requirements
Functional requirements help to understand the functions of the system.	They help to understand the system's performance.
Functional requirements are mandatory.	While non-functional requirements are not mandatory.
They are easy to define.	They are hard to define.
They describe what the product does.	They describe the working of product.
It concentrates on the user's requirement.	It concentrates on the expectation and experience of the user.
It helps us to verify the software's functionality.	It helps us to verify the software's performance.
These requirements are specified by the user.	These requirements are specified by the software developers, architects, and technical persons.
There is functional testing such as API testing, system, integration, etc.	There is non-functional testing such as usability, performance, stress, security, etc.
Examples of the functional requirements are - Authentication of a user on trying to log in to the system.	Examples of the non-functional requirements are - The background color of the screens should be light blue.
These requirements are important to system operation.	These are not always the important requirements, they may be desirable.
Completion of Functional requirements allows the system to perform, irrespective of meeting the non-functional requirements.	While system will not work only with non-functional requirements.