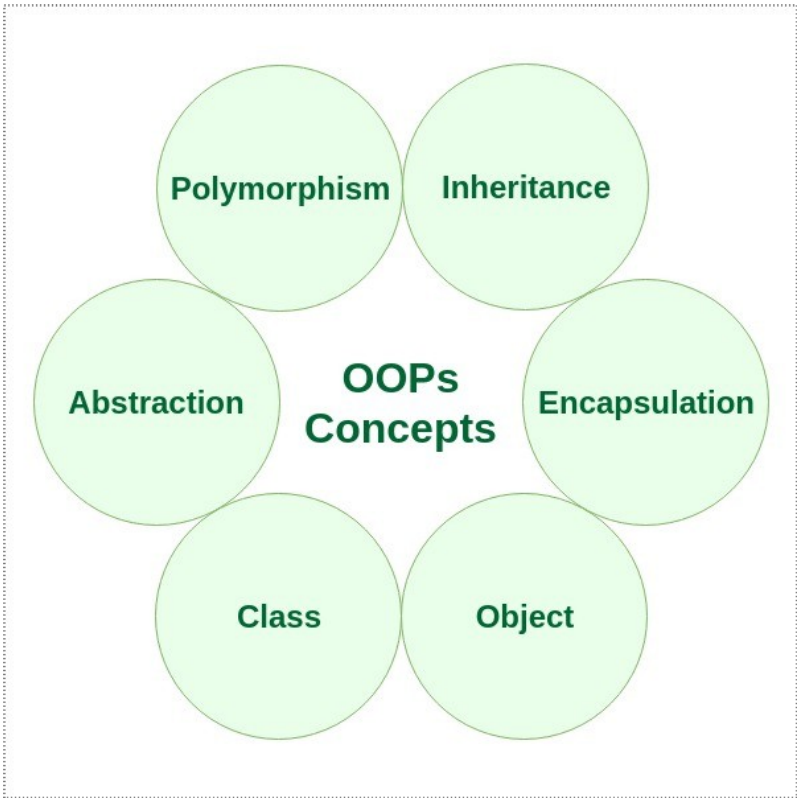
UNIT - 1

Object-oriented programming – As the name suggests uses objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

Characteristics of an Object Oriented Programming language



Class: The building block of C++ that leads to Object-Oriented programming is a Class. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

For Example: Consider the Class of Cars. There may be many cars with different names and brand but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

- A Class is a user-defined data-type which has data members and member functions.

- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions define the properties and behaviour of the objects in a Class.
- In the above example of class Car, the data member will be speed limit, mileage etc and member functions can apply brakes, increase speed etc.

We can say that a **Class in C++** is a blue-print representing a group of objects which shares some common properties and behaviours.

Object: An Object is an identifiable entity with some characteristics and behaviour. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

```
class person
{
   char name[20];
   int id;
public:
   void getdetails(){}
};

int main()
{
  person p1; // p1 is a object
}
```

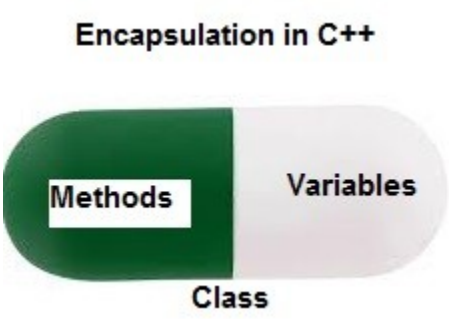
Object take up space in memory and have an associated address like a record in pascal or structure or union in C.

When a program is executed the objects interact by sending messages to one another.

Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted and type of response returned by the objects.

Encapsulation: In normal terms, Encapsulation is defined as wrapping up of data and information under a single unit. In Object-Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them.
Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him

to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name “sales section”.



Encapsulation also leads to *data abstraction or hiding*. As using encapsulation also hides the data. In the above example, the data of any of the section like sales, finance or accounts are hidden from any other section.

Abstraction: Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.
Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

- *Abstraction using Classes*: We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.
- *Abstraction in Header files*: One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

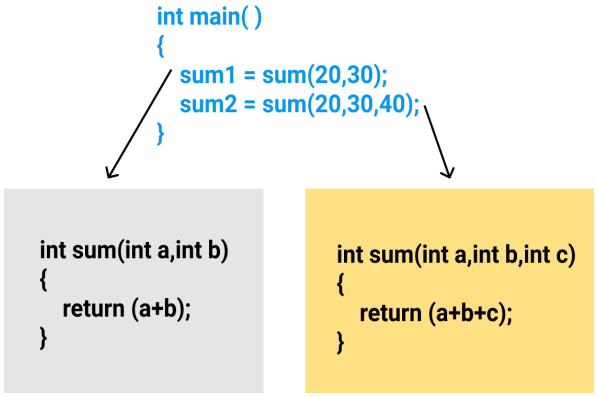
Polymorphism: The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.
A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behaviour in different situations. This is called polymorphism.

An operation may exhibit different behaviours in different instances. The behaviour depends upon the types of data used in the operation.

C++ supports operator overloading and function overloading.

- *Operator Overloading*: The process of making an operator to exhibit different behaviours in different instances is known as operator overloading.
- *Function Overloading*: Function overloading is using a single function name to perform different types of tasks.
  Polymorphism is extensively used in implementing inheritance.

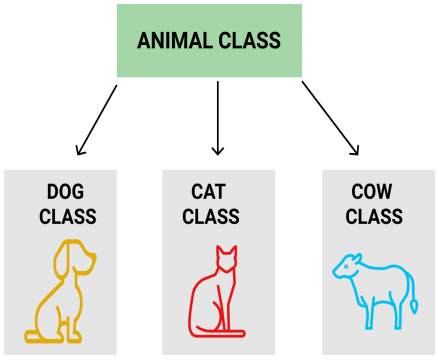
Example: Suppose we have to write a function to add some integers, some times there are 2 integers, some times there are 3 integers. We can write the Addition Method with the same name having different parameters, the concerned method will be called according to parameters.



Inheritance: The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.

- **Sub Class**: The class that inherits properties from another class is called Sub class or Derived Class.
- **Super Class**:The class whose properties are inherited by sub class is called Base Class or Super class.
- **Reusability**: Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Example: Dog, Cat, Cow can be Derived Class of Animal Base Class.



Dynamic Binding: In dynamic binding, the code to be executed in response to function call is decided at runtime. C++ has virtual functions to support this.

Message Passing: Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

Importance of Modeling

A model is a simplification of reality.

A good model includes those elements that have broad effect and omits those minor elements that are not relevant to the given level of abstraction. A model may be structural, emphasizing the organization of the system, or it may be behavioral, emphasizing the dynamics of the system.

We build models so that we can better understand the system we are developing.

Through modelling, we achieve four aims:

- Models help us to visualize a system as it is or as we want it to be.
- Models permit us to specify the structure or behavior of a system.
- Models gives us a template that guides us in constructing a system.
- Models document the decisions we have made.

The larger and more complex the system becomes, the more important modeling becomes

Principles of Modeling

First principle of modelling:

The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped.

Choose your models well. The right models will highlight the most nasty development problems. Wrong models will mislead you, causing you to focus on irrelevant issues.

Second principle of modelling:

Every model may be expressed at different levels of precision.

Sometimes, a quick and simple executable model of the user interface is exactly what you need. At other times, you have to get down to complex details such as cross-system interfaces or networking issues etc.

In any case, the best kinds of models are those that let you choose your degree of detail, depending on who is viewing it. An analyst or an end user will want to focus on issues of what and a developer will want to focus on issues of how.

Third principle of modelling:

The best models are connected to reality.

In software, the gap between the analysis model and the system's design model must be less. Failing to bridge this gap causes the system to diverge over time. In object-oriented systems, it is possible to connect all the nearly independent views of a system into one whole.

Fourth principle of modelling:

No single model is sufficient. Every system is best approached through a small set of nearly independent models.

In the case of a building, you can study electrical plans in isolation, but you can also see their mapping to the floor plan and perhaps even their interaction with the routing of pipes in the plumbing plan.

Object Oriented Modeling

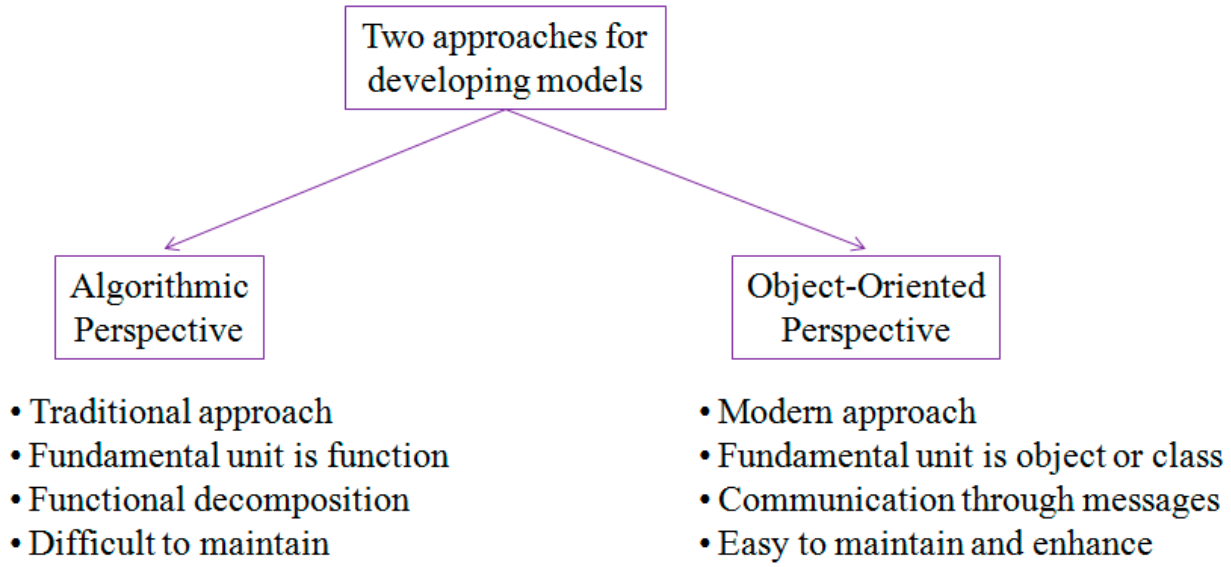
In software field, there are several ways to approach for building a model. The two most common ways are: from an **algorithmic perspective** and from an **object oriented perspective**.

The traditional view of software development takes an **algorithmic perspective**. In this approach, the **main building block** of all software **is the procedure or function**. This view leads the developers to focus on issues of control and the decomposition of larger algorithms into smaller ones.

As requirements change, and the system grows, systems built with an algorithmic focus turn out to be very hard to maintain.

The contemporary (another) view of software development takes an **object oriented perspective**. In object oriented modeling, the **main building block** of all software systems **is the object or class**.

Simply put, an object is a thing, generally drawn from the elements of the problem space or the solution space. A class is a description of a set of common objects. Every object has state, identity and behavior.



For example, consider simple three-tier architecture for a billing system, involving a user interface, middleware, and a database. In the user interface, you will find concrete objects such as buttons, menus and dialog boxes.

In the database you will find concrete objects such as tables representing entities from the problem domain, including customers, products and orders. In the middle layer, you will find objects such as transactions and business rules.

The object oriented approach to software development is a part of the mainstream development simply because it has proven to be of value in building systems in all sorts of problem domains and cover all degrees of size and complexity.

What is UML

The UML stands for Unified modeling language, is a standardized general-purpose visual modeling language in the field of Software Engineering. It is used for specifying, visualizing, constructing, and documenting the primary artifacts of the software system. It helps in designing and characterizing, especially those software systems that incorporate the

concept of Object orientation. It describes the working of both the software and hardware systems.

Goals of UML

- Since it is a general-purpose modeling language, it can be utilized by all the modelers.
- UML came into existence after the introduction of object-oriented concepts to systemize and consolidate the object-oriented development, due to the absence of standard methods at that time.
- The UML diagrams are made for business users, developers, ordinary people, or anyone who is looking forward to understand the system, such that the system can be software or non-software.
- Thus it can be concluded that the UML is a simple modeling approach that is used to model all the practical systems.

Characteristics of UML

The UML has the following features:

- It is a generalized modeling language.
- It is distinct from other programming languages like C++, Python, etc.
- It is interrelated to object-oriented analysis and design.
- It is used to visualize the workflow of the system.
- It is a pictorial language, used to generate powerful modeling artifacts.

Conceptual Modeling

Before moving ahead with the concept of UML, we should first understand the basics of the conceptual model.

A conceptual model is composed of several interrelated concepts. It makes it easy to understand the objects and how they interact with each other. This is the first step before drawing UML diagrams.

Following are some object-oriented concepts that are needed to begin with UML:

- **Object:** An object is a real world entity. There are many objects present within a single system. It is a fundamental building block of UML.
- **Class:** A class is a software blueprint for objects, which means that it defines the variables and methods common to all the objects of a particular type.
- **Abstraction:** Abstraction is the process of portraying the essential characteristics of an object to the users while hiding the irrelevant information. Basically, it is used to envision the functioning of an object.
- **Inheritance:** Inheritance is the process of deriving a new class from the existing ones.
- **Polymorphism:** It is a mechanism of representing objects having multiple forms used for different purposes.
- **Encapsulation:** It binds the data and the object together as a single unit, enabling tight coupling between them.

UML-Building Blocks

UML is composed of three main building blocks, i.e., things, relationships, and diagrams. Building blocks generate one complete UML model diagram by rotating around several different blocks. It plays an essential role in developing UML diagrams. The basic UML building blocks are enlisted below:

1. Things
2. Relationships
3. Diagrams

Things

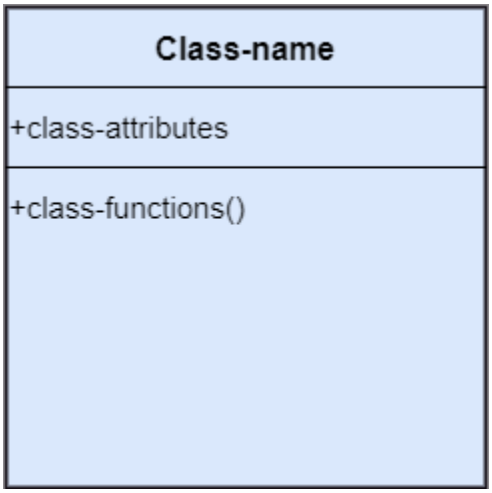
Anything that is a real world entity or object is termed as things. It can be divided into several different categories:

- Structural things
- Behavioral things
- Grouping things
- Annotational things

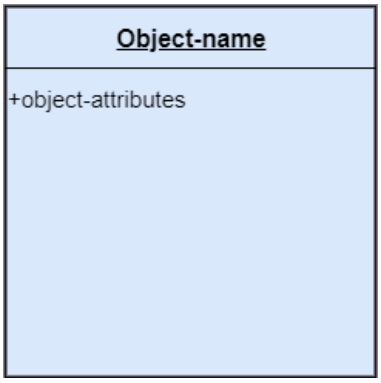
Structural things

Nouns that depicts the static behavior of a model is termed as structural things. They display the physical and conceptual components. They include class, object, interface, node, collaboration, component, and a use case.

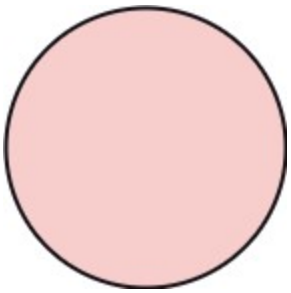
Class: A Class is a set of identical things that outlines the functionality and properties of an object. It also represents the abstract class whose functionalities are not defined. Its notation is as follows;



Object:: An individual that describes the behavior and the functions of a system. The notation of the object is similar to that of the class; the only difference is that the object name is always underlined and its notation is given below;

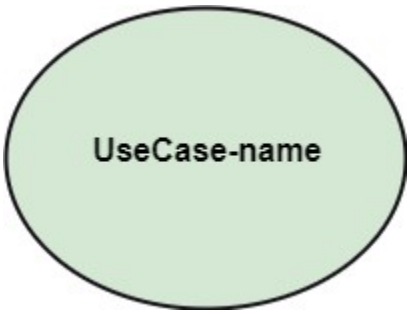


Interface: A set of operations that describes the functionality of a class, which is implemented whenever an interface is implemented.

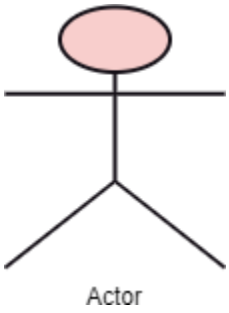


Collaboration: It represents the interaction between things that is done to meet the goal. It is symbolized as a dotted ellipse with its name written inside it.

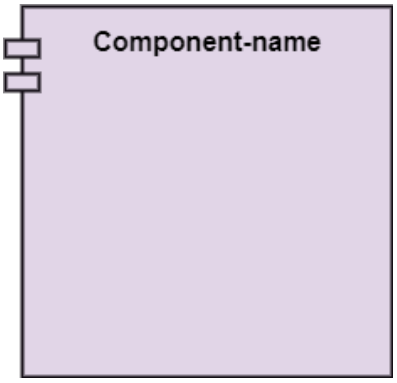
Use case: Use case is the core concept of object-oriented modeling. It portrays a set of actions executed by a system to achieve the goal.



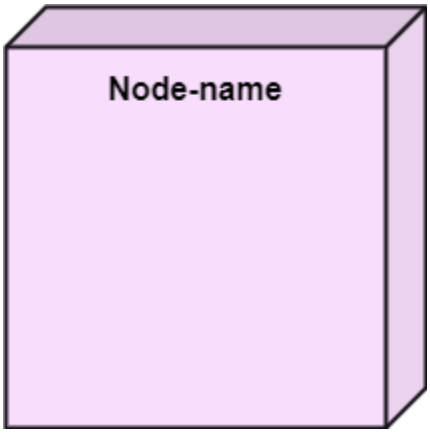
Actor: It comes under the use case diagrams. It is an object that interacts with the system, for example, a user.



Component: It represents the physical part of the system.



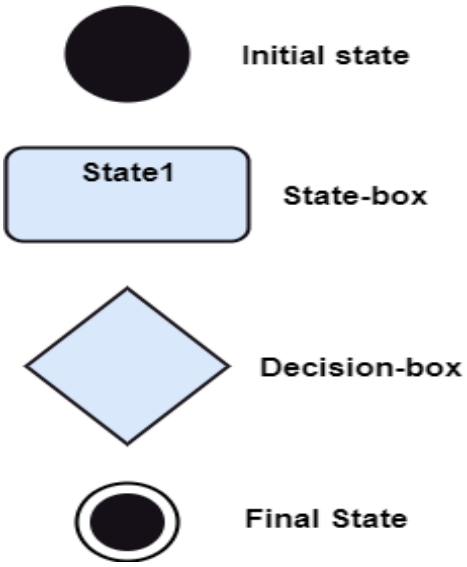
Node: A physical element that exists at run time.



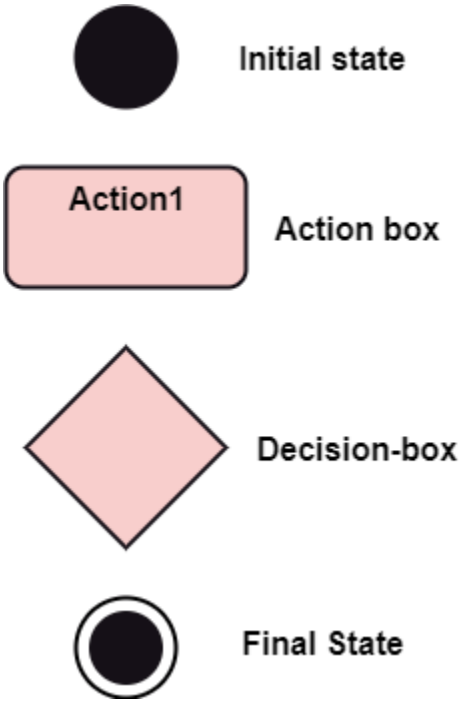
Behavioral Things

They are the verbs that encompass the dynamic parts of a model. It depicts the behavior of a system. They involve state machine, activity diagram, interaction diagram, grouping things, annotation things

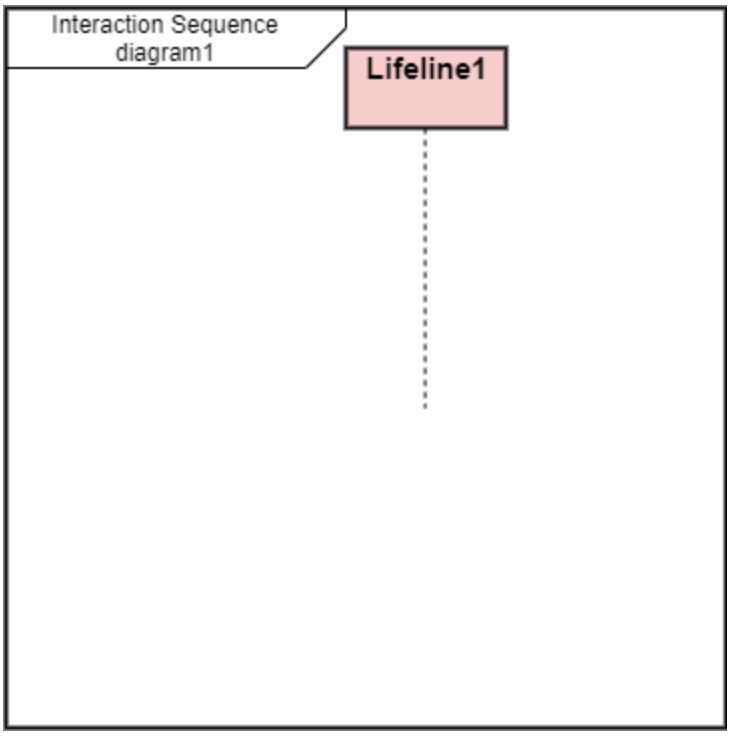
State Machine: It defines a sequence of states that an entity goes through in the software development lifecycle. It keeps a record of several distinct states of a system component.



Activity Diagram: It portrays all the activities accomplished by different entities of a system. It is represented the same as that of a state machine diagram. It consists of an initial state, final state, a decision box, and an action notation.



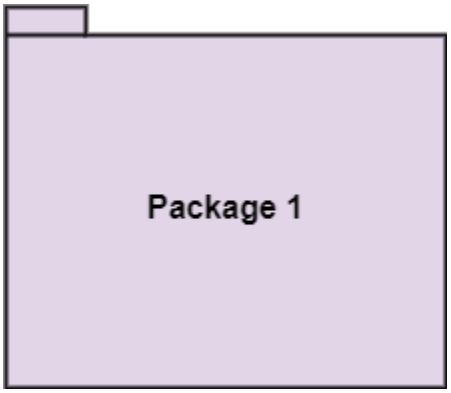
Interaction Diagram: It is used to envision the flow of messages between several components in a system.



Grouping Things

It is a method that together binds the elements of the UML model. In UML, the package is the only thing, which is used for grouping.

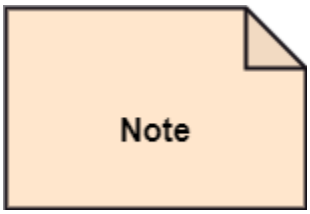
Package: Package is the only thing that is available for grouping behavioral and structural things.



Annotation Things

It is a mechanism that captures the remarks, descriptions, and comments of UML model elements. In UML, a note is the only Annotational thing.

Note: It is used to attach the constraints, comments, and rules to the elements of the model. It is a kind of yellow sticky note.

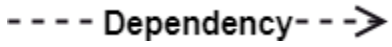


Relationships

It illustrates the meaningful connections between things. It shows the association between the entities and defines the functionality of an application. There are four types of relationships given below:

Dependency: Dependency is a kind of relationship in which a change in target element affects the source element, or simply we can say the source element is dependent on the target element. It is one of the most important notations in UML. It depicts the dependency from one entity to another.

It is denoted by a dotted line followed by an arrow at one side as shown below,



Association: A set of links that associates the entities to the UML model. It tells how many elements are actually taking part in forming that relationship.

Generalization: It portrays the relationship between a general thing (a parent class or superclass) and a specific kind of that thing (a child class or subclass). It is used to describe the concept of inheritance.

It is denoted by a straight line followed by an empty arrowhead at one side.

Realization: It is a semantic kind of relationship between two things, where one defines the behavior to be carried out, and the other one implements the mentioned behavior. It exists in interfaces.

It is denoted by a dotted line with an empty arrowhead at one side.

Diagrams

The diagrams are the graphical implementation of the models that incorporate symbols and text. Each symbol has a different meaning in the context of the UML diagram. There are thirteen different types of UML diagrams that are available in UML 2.0, such that each diagram has its own set of a symbol. And each diagram manifests a different dimension, perspective, and view of the system.

UML diagrams are classified into three categories that are given below:

1. Structural Diagram
2. Behavioral Diagram
3. Interaction Diagram

Structural Diagram: It represents the static view of a system by portraying the structure of a system. It shows several objects residing in the system. Following are the structural diagrams given below:

- Class diagram
- Object diagram
- Package diagram
- Component diagram
- Deployment diagram

Behavioral Diagram: It depicts the behavioral features of a system. It deals with dynamic parts of the system. It encompasses the following diagrams:

- Activity diagram
- State machine diagram
- Use case diagram

Interaction diagram: It is a subset of behavioral diagrams. It depicts the interaction between two objects and the data flow between them. Following are the several interaction diagrams in UML:

- Timing diagram
- Sequence diagram

- Collaboration diagram

UML- Architecture

Software architecture is all about how a software system is built at its highest level. It is needed to think big from multiple perspectives with quality and design in mind. The software team is tied to many practical concerns, such as:

- The structure of the development team.
- The needs of the business.
- Development cycle.
- The intent of the structure itself.

Software architecture provides a basic design of a complete software system. It defines the elements included in the system, the functions each element has, and how each element relates to one another. In short, it is a big picture or overall structure of the whole system, how everything works together.

To form an architecture, the software architect will take several factors into consideration:

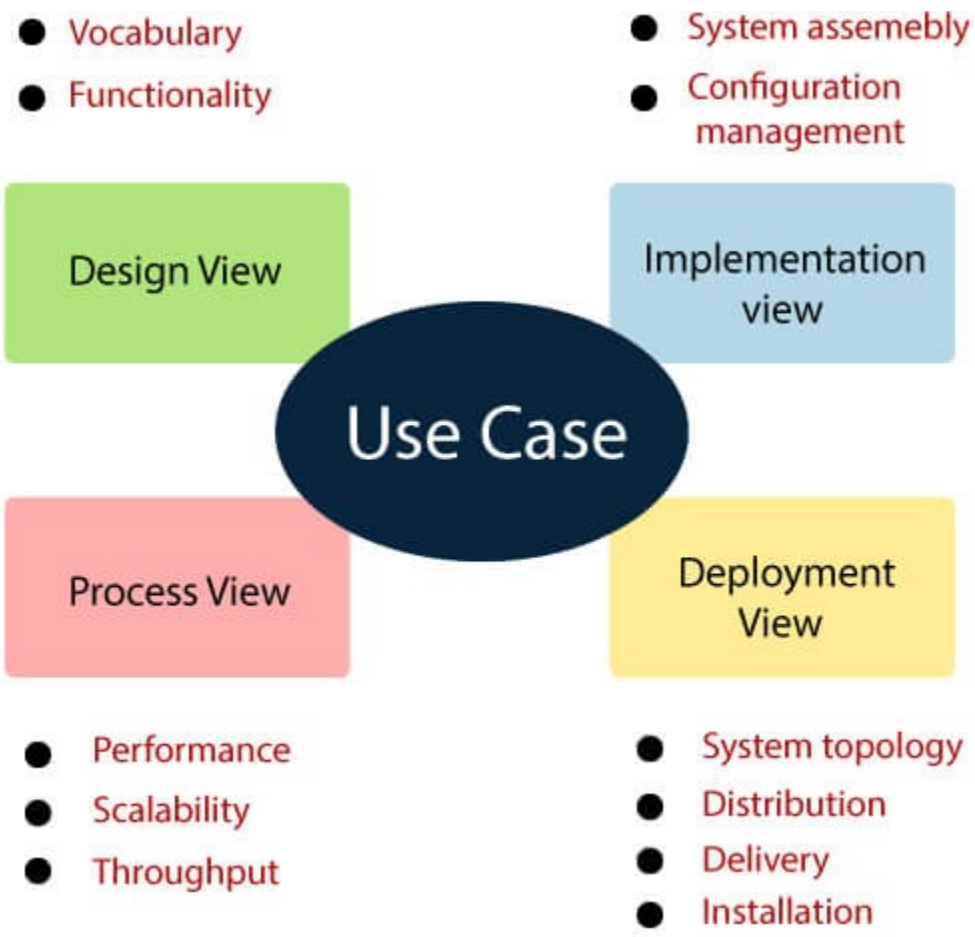
- What will the system be used for?
- Who will be using the system?
- What quality matters to them?
- Where will the system run?

The architect plans the structure of the system to meet the needs like these. It is essential to have proper software architecture, mainly for a large software system. Having a clear design of a complete system as a starting point provides a solid basis for developers to follow.

Each developer will know what needs to be implemented and how things relate to meet the desired needs efficiently. One of the main advantages of software architecture is that it provides high productivity to the software team. The software development becomes more effective as it comes up with an explained structure in place to coordinate work, implement individual features, or ground discussions on potential issues. With a lucid architecture, it is easier to know where the key responsibilities are residing in the system and where to make changes to add new requirements or simply fixing the failures.

In addition, a clear architecture will help to achieve quality in the software with a well-designed structure using principles like separation of concerns; the system becomes easier to maintain, reuse, and adapt. The software architecture is useful to people such as software developers, the project manager, the client, and the end-user. Each one will have different perspectives to view the system and will bring different agendas to a project. Also, it provides a collection of several views. It can be best understood as a collection of five views:

1. Use case view
2. Design view
3. Implementation view
4. Process view
5. Development view



Use case view

- It is a view that shows the functionality of the system as perceived by external actors.
- It reveals the requirements of the system.
- With UML, it is easy to capture the static aspects of this view in the use case diagrams, whereas it?s dynamic aspects are captured in interaction diagrams, state chart diagrams, and activity diagrams.

Design View

- It is a view that shows how the functionality is designed inside the system in terms of static structure and dynamic behavior.
- It captures the vocabulary of the problem space and solution space.
- With UML, it represents the static aspects of this view in class and object diagrams, whereas its dynamic aspects are captured in interaction diagrams, state chart diagrams, and activity diagrams.

Implementation View

- It is the view that represents the organization of the core components and files.
- It primarily addresses the configuration management of the system?s releases.
- With UML, its static aspects are expressed in component diagrams, and the dynamic aspects are captured in interaction diagrams, state chart diagrams, and activity diagrams.

Process View

- It is the view that demonstrates the concurrency of the system.
- It incorporates the threads and processes that make concurrent system and synchronized mechanisms.
- It primarily addresses the system's scalability, throughput, and performance.
- Its static and dynamic aspects are expressed the same way as the design view but focus more on the active classes that represent these threads and processes.

Deployment View

- It is the view that shows the deployment of the system in terms of physical architecture.
- It includes the nodes, which form the system hardware topology where the system will be executed.
- It primarily addresses the distribution, delivery, and installation of the parts that build the physical system.