**Notes for MCA-I (Semester- I)**

**Subject :- Object Oriented Software Engineering**

**(Subject Code:- IT-13)**

**Chapter: 3**] **Use-case Driven Object Oriented Analysis**

* **3.1 Introduction to OOPS Concept :-**

**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.

Grady Booch has defined object–oriented programming as *“****a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships****”*.

* **Class**

A class represents a collection of objects having same characteristic properties that exhibit common behavior. It gives the blueprint or description of the objects that can be created from it. Creation of an object as a member of a class is called instantiation. Thus, object is an instance of a class.

The constituents of a class are −

* A set of attributes for the objects that are to be instantiated from the class. Generally, different objects of a class have some difference in the values of the attributes. Attributes are often referred as class data.
* A set of operations that portray the behavior of the objects of the class. Operations are also referred as functions or methods.

Example

Let us consider a simple class, Circle, that represents the geometrical figure circle in a two–dimensional space. The attributes of this class can be identified as follows −

* x–coord, to denote x–coordinate of the center
* y–coord, to denote y–coordinate of the center
* a, to denote the radius of the circle

Some of its operations can be defined as follows −

* findArea(), method to calculate area
* findCircumference(), method to calculate circumference
* scale(), method to increase or decrease the radius

During instantiation, values are assigned for at least some of the attributes. If we create an object my\_circle, we can assign values like x-coord : 2, y-coord : 3, and a : 4 to depict its state. Now, if the operation scale() is performed on my\_circle with a scaling factor of 2, the value of the variable a will become 8. This operation brings a change in the state of my\_circle, i.e., the object has exhibited certain behavior

A class is an entity that determines how an object will behave and what the object will contain. In other words, it is a blueprint or a set of instruction to build a specific type of object. It provides initial values for member variables and member functions or methods.

* **Object**

An object is a real-world element in an object–oriented environment that may have a physical or a conceptual existence. Each object has −

* Identity that distinguishes it from other objects in the system.
* State that determines the characteristic properties of an object as well as the values of the properties that the object holds.
* Behavior that represents externally visible activities performed by an object in terms of changes in its state.

Objects can be modelled according to the needs of the application. An object may have a physical existence, like a customer, a car, etc.; or an intangible conceptual existence, like a project, a process, etc.

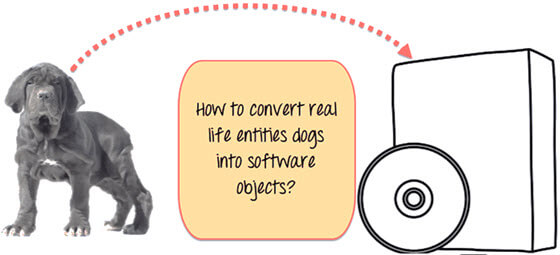
An object is nothing but a self-contained component that consists of methods and properties to make a data useful. It helps you to determine the behavior of the class.

For example, when you send a message to an object, you are asking the object to invoke or execute one of its methods.

From a programming point of view, an object can be a data structure, a variable, or a function that has a memory location allocated. The object is designed as class hierarchies.

Let's take an example of developing a pet management system, specially meant for dogs. You will need various information about the dogs like different breeds of the dogs, the age, size, etc.

You need to model real-life beings, i.e., dogs into software entities.

[](https://www.guru99.com/images/java/052016_0704_ObjectsandC1.jpg)

Moreover, the million dollar question is, how you design such software? **Here is the solution-**

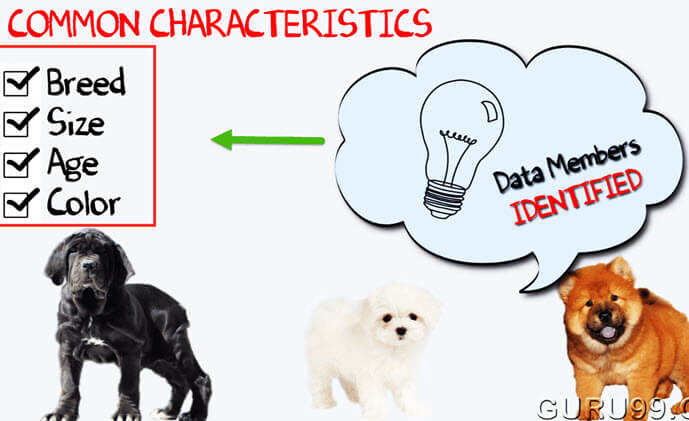
First, let's do an exercise.

You can see the picture of three different breeds of dogs below.

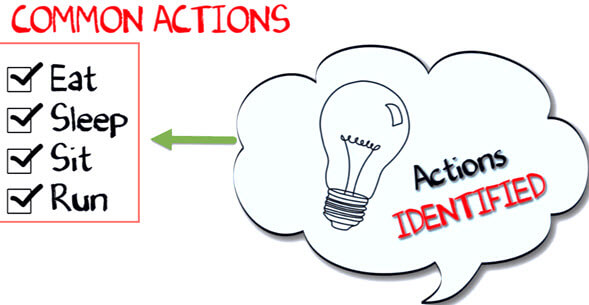
[](https://www.guru99.com/images/java/052016_0704_ObjectsandC2.jpg)

Stop here right now! List down the differences between them.

Some of the differences you might have listed out maybe breed, age, size, color, etc. If you think for a minute, these differences are also some common characteristics shared by these dogs. These characteristics (breed, age, size, color) can form a data members for your object.

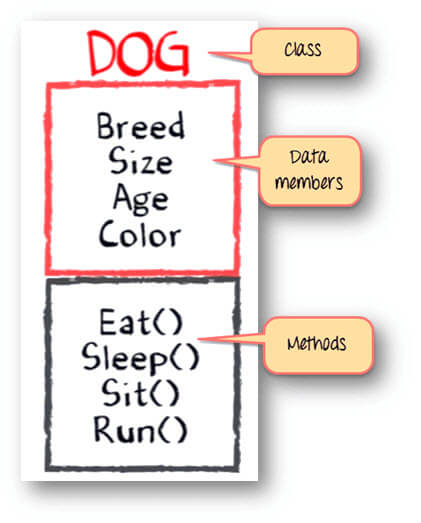
[](https://www.guru99.com/images/java/052016_0704_ObjectsandC3.jpg)

Next, list out the common behaviors of these dogs like sleep, sit, eat, etc. So these will be the actions of our software objects.

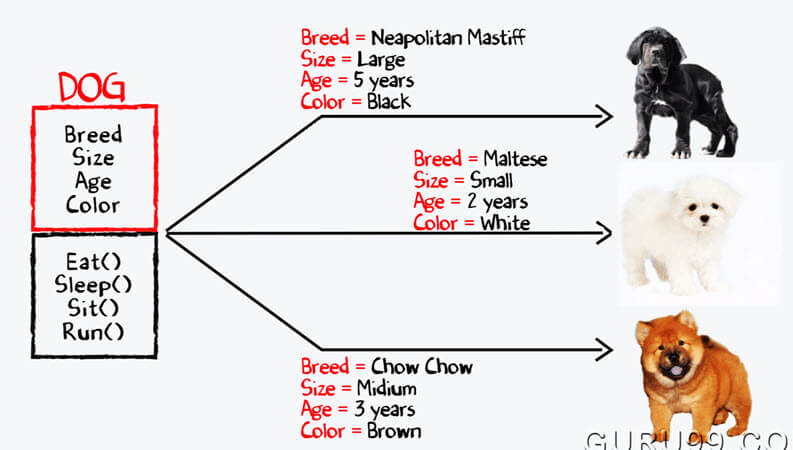
[](https://www.guru99.com/images/java/052016_0704_ObjectsandC4.jpg)

So far we have defined following things,

* **Class**: Dogs
* **Data members** or **objects**: size, age, color, breed, etc.
* **Methods**: eat, sleep, sit and run.

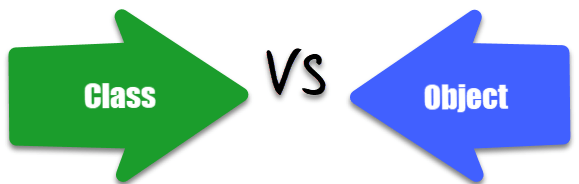
[](https://www.guru99.com/images/java/052016_0704_ObjectsandC5.jpg)

Now, for different values of data members (breed size, age, and color) in C++ or Java class, you will get different dog objects.

[](https://www.guru99.com/images/java/052016_0704_ObjectsandC6.jpg)

You can design any program using this OOPs approach.

**Class Vs. Object**

[](https://www.guru99.com/images/2/030720_0605_Differenceb7.png)

Here is the important difference between class and object:

|  |  |
| --- | --- |
| **Class** | **Object** |
| A class is a template for creating objects in program. | The object is an instance of a class. |
| A class is a logical entity | Object is a physical entity |
| A class does not allocate memory space when it is created. | Object allocates memory space whenever they are created. |
| You can declare class only once. | You can create more than one object using a class. |
| Example: Car. | Example: Jaguar, BMW, Tesla, etc. |
| Class generates objects | Objects provide life to the class. |
| Classes can't be manipulated as they are not available in memory. | They can be manipulated. |
| It doesn't have any values which are associated with the fields. | Each and every object has its own values, which are associated with the fields. |

|  |  |
| --- | --- |
| **Class** | **Object** |
| Class is a **blueprint or template** from which objects are created. | Object is an **instance** of a class. |
| Class is a **group of similar objects**. | Object is a **real world entity** such as pen, laptop, mobile, bed, keyboard, mouse, chair etc. |
| Class is a **logical** entity. | Object is a **physical** entity. |
| Class is declared **once**. | Object is created **many times** as per requirement. |
| Class **doesn't allocated memory when it is created**. | Object **allocates memory when it is created**. |

Let's see some real life example of class and object in OOPS to understand the difference well:

**Class:** Human **Object:** Man, Woman

**Class:** Fruit **Object:** Apple, Banana, Mango, Guava wtc.

**Class:** Mobile phone **Object:** iPhone, Samsung, Moto

**Class:** Fast Food **Object:** Pizza, Burger, Samosa

* **Abstraction**

Abstraction is the concept of object-oriented programming that "shows" only essential attributes and "hides" unnecessary information. The main purpose of abstraction is hiding the unnecessary details from the users. Abstraction is selecting data from a larger pool to show only relevant details of the object to the user. It helps in reducing programming complexity and efforts. It is one of the most important concepts of OOPs.

Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or the non-essentials units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.

Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

a class is designed such that its data (attributes) can be accessed only by its class methods and insulated from direct outside access. This process of insulating an object’s data is called data hiding or information hiding.

In the class Circle, data hiding can be incorporated by making attributes invisible from outside the class and adding two more methods to the class for accessing class data, namely −

* setValues(), method to assign values to x-coord, y-coord, and a
* getValues(), method to retrieve values of x-coord, y-coord, and a

Here the private data of the object my\_circle cannot be accessed directly by any method that is not encapsulated within the class Circle. It should instead be accessed through the methods setValues() and getValues().

**Example of Abstraction**:

#include <iostream>

using namespace std;

class Summation {

private:

// private variables

int myNum1, myNum2, myNum3

public:

void sum(int inNum1, int inNum2)

{

myNum1 = inNum1;

myNum2 = inNum2;

myNum3 = myNum1 + myNum2;

cout << "Sum of the two number is : " << myNum3< <endl;

}

};

int main()

{

Summation mySum;

mySum.sum(5, 4);

return 0;

}

In this case the variables myNum1, myNum2 and myNum3 are private, thereby in accessible to any code other than the class Summation. In this example the variables are set to values passed in as arguments to the sum method. This is not a very true example - often the values would NOT be set just before being used like this, but it shows the reality of the implementation.

**Output:**

Sum of the two number is: 9

* **Encapsulation**

Encapsulation is a method of making a complex system easier to handle for end users. The user need not worry about internal details and complexities of the system. Encapsulation is a process of wrapping the data and the code, that operate on the data into a single entity. You can assume it as a protective wrapper that stops random access of code defined outside that wrapper.

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Another way to think about encapsulation is, it is a protective shield that prevents the data from being accessed by the code outside this shield.

Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared.

As in encapsulation, the data in a class is hidden from other classes, so it is also known as data-hiding.

Encapsulation can be achieved by Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.

Encapsulation is the process of binding both attributes and methods together within a class. Through encapsulation, the internal details of a class can be hidden from outside. It permits the elements of the class to be accessed from outside only through the interface provided by the class.

For Example

#include <iostream>

using namespace std;

class Adder {

public:

**// constructor**

Adder(int i = 0) {

total = i;

}

**// interface to outside world**

void addNum(int number) {

total += number;

}

// interface to outside world

int getTotal() {

return total;

};

private:

**// hidden data from outside world**

int total;

};

int main() {

Adder a;

a.addNum(10);

a.addNum(20);

a.addNum(30);

cout << "Total " << a.getTotal() <<endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Total 60

Above class adds numbers together, and returns the sum. The public members **addNum** and **getTotal**are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that is hidden from the outside world, but is needed for the class to operate properly.

|  |
| --- |
| //**Another Encapsulation program**    #include<iostream>  using namespace std;    class Encapsulation  {      private:  **// data hidden from outside world**          int x;        public:  **// function to set value of**  **// variable x**          void set(int a)          {              x =a;          }    **// function to return value of**  **// variable x**          int get()          {              return x;          }  };    **// main function**  int main()  {      Encapsulation obj;        obj.set(5);        cout<<obj.get();      return 0;  } |

output:

5

In the above program the variable **x** is made private. This variable can be accessed and manipulated only using the functions get() and set() which are present inside the class. Thus we can say that here, the variable x and the functions get() and set() are binded together which is nothing but encapsulation.

Another Example :-

#include <iostream>

using namespace std;

class EncapsulationExample {

private:

// we declare a as private to hide it from outside

int number1;

public:

// set() function to set the value of a

void set(int input1)

{

number1 = input1;

}

// get() function to return the value of a

int get()

{

return number1;

}

};

// main function

int main()

{

EncapsulationExample myInstance;

myInstance.set(10);

cout << myInstance.get() << endl;

return 0;

}

In the this program, the variable number1 is made private so that this variable can be accessed and manipulated only by using the methods get() and set() that are present within the class. Therefore we can say that, the variable a and the methods set() as well as get() have bound together that is encapsulation. There is nothing special about the method names "get()" or "set()" - there may be other methods that manipulate the variable number1...all together this is called encapsulation.

**Output:**

10

The Difference Between

|  |  |
| --- | --- |
| **ABSTRACTION** | **ENCAPSULATION** |
| Abstraction is the process or method of gaining the information. | encapsulation is the process or method to contain the information. |
| In abstraction, problems are solved at the design or interface level. | While in encapsulation, problems are solved at the implementation level. |
| Abstraction is the method of hiding the unwanted information. | Whereas encapsulation is a method to hide the data in a single entity or unit along with a method to protect information from outside. |
| We can implement abstraction using abstract class and interfaces | Whereas encapsulation can be implemented using by access modifier i.e. private, protected and public. |
| In abstraction, implementation complexities are hidden using abstract classes and interfaces | While in encapsulation, the data is hidden using methods of getters and setters |
| The objects that help to perform abstraction are encapsulated. | Whereas the objects that result in encapsulation need not be abstracted. |

* **Method :-**

Methods are functions that belongs to the class.

There are two ways to define functions that belongs to a class:

* Inside class definition
* Outside class definition

 A **method** is a way to perform some task. It is a collection of instructions that performs a specific task. It provides the reusability of code. We can also easily modify code using **methods**.

The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments. It has six components that are known as **method header**, as we have shown in the following figure.



In the following example, we define a function inside the class, and we name it "myMethod".

**Note:** You access methods just like you access attributes; by creating an object of the class and using the dot syntax (.):

**Inside Class Example**

#include <iostream>

using namespace std;

class MyClass { // The class

public: // Access specifier

void myMethod() { // Method/function

cout << "Hello World!";

}

};

int main() {

MyClass myObj; // Create an object of MyClass

myObj.myMethod(); // Call the method

return 0;

}

Output :- Hello World!

To define a function outside the class definition, you have to declare it inside the class and then define it outside of the class. This is done by specifiying the name of the class, followed the scope resolution :: operator, followed by the name of the function:

**Outside Class Example**

#include <iostream>

using namespace std;

class MyClass { // The class

public: // Access specifier

void myMethod(); // Method/function declaration

};

// Method/function definition outside the class

void MyClass::myMethod() {

cout << "Hello World!";

}

int main() {

MyClass myObj; // Create an object of MyClass

myObj.myMethod(); // Call the method

return 0;

}

Output :- Hello World!

**You can also add parameters:**

#include <iostream>

using namespace std;

class Car {

public:

int speed(int maxSpeed);

};

int Car::speed(int maxSpeed) {

return maxSpeed;

}

int main() {

Car myObj; // Create an object of Car

cout << myObj.speed(200); // Call the method with an argument

return 0;

}

Output :- 200

* **Message Passing**

Any application requires a number of objects interacting in a harmonious manner. Objects in a system may communicate with each other using message passing. Suppose a system has two objects: obj1 and obj2. The object obj1 sends a message to object obj2, if obj1 wants obj2 to execute one of its methods.

The features of message passing are −

* Message passing between two objects is generally unidirectional.
* Message passing enables all interactions between objects.
* Message passing essentially involves invoking class methods.
* Objects in different processes can be involved in message passing.

A message is what you send, asking for something to be done by the object you’ve sent the message to. A method is the code that actually gets run by that receiving object, after the system has looked up the match between the receiver and the message.

[Message Passing](https://www.geeksforgeeks.org/message-passing-in-java/) **:** Message Passing in terms of computers is communication between processes. It is a form of communication used in object-oriented programming as well as parallel programming. Message passing in Java is like sending an object i.e. message from one thread to another thread. It is used when threads do not have shared memory and are unable to share monitors or semaphores or any other shared variables to communicate. The following are the main advantages of the message passing technique:

1. This model is much easier to implement than the shared memory model.
2. Implementing this model in order to build parallel hardware is much easier because it is quite tolerant of higher communication latencies.

In OOPs, there are many ways to implement the message passing technique like message passing through constructors, message passing through methods or by passing different values. The following is a simple implementation of the message passing technique by the values:

|  |
| --- |
| // Java program to demonstrate  // message passing by value    import java.io.\*;    // Implementing a message passing  // class  public class MessagePassing {        // Implementing a method to      // add two integers      void displayInt(int x, int y)      {          int z = x + y;          System.out.println(              "Int Value is : " + z);      }        // Implementing a method to multiply      // two floating point numbers      void displayFloat(float x, float y)      {          float z = x \* y;          System.out.println(              "Float Value is : " + z);      }  }  class GFG {        // Driver code      public static void main(String[] args)      {          // Creating a new object          MessagePassing mp              = new MessagePassing();            // Passing the values to compute          // the answer          mp.displayInt(1, 100);          mp.displayFloat((float)3, (float)6.9);      }  } |

**Output:**

Int Value is : 101

Float Value is : 20.7

* **Interface**: -

Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

* Interfaces specify what a class must do and not how. It is the blueprint of the class.
* An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
* If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.
* To declare an interface, use **interface** keyword. It is used to provide total abstraction. That means all the methods in an interface are declared with an empty body and are public and all fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface. To implement interface use **implements** keyword.

**Why do we use interface ?**

* It is used to achieve total abstraction.
* Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance .
* It is also used to achieve loose coupling.
* Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes?
* Interfaces allow you to specify what methods a class should implement.
* Interfaces make it easy to use a variety of different classes in the same way. When one or more classes use the same interface, it is referred to as "polymorphism".
* An interface is similar to a class except that it cannot contain code.
* An interface can define method names and arguments, but not the contents of the methods.
* Any classes implementing an interface must implement all methods defined by the interface.
* A class can implement multiple interfaces.
* An interface is declared using the "interface" keyword.
* Interfaces can't maintain Non-abstract methods.

An Interface in Java programming is defined as an abstract type used to specify the behavior of a class. A Java interface contains static constants and abstract methods. A class can implement multiple interfaces. In Java, interfaces are declared using the interface keyword. All methods in the interface are implicitly public and abstract.

Syntax for Declaring Interface

interface {

//methods

}

**Difference between Class and Interface**

|  |  |
| --- | --- |
| **Class** | **Interface** |
| In class, you can instantiate variable and create an object. | In an interface, you can't instantiate variable and create an object. |
| Class can contain concrete(with implementation) methods | The interface cannot contain concrete(with implementation) methods |
| The access specifiers used with classes are private, protected and public. | In Interface only one specifier is used- Public. |

**When to use Interface and Abstract Class?**

* Use an abstract class when a template needs to be defined for a group of subclasses
* Use an interface when a role needs to be defined for other classes, regardless of the inheritance tree of these classes

**Must know facts about Interface**

* A Java class can implement multiple Java Interfaces. It is necessary that the class must implement all the methods declared in the interfaces.
* Class should override all the abstract methods declared in the interface
* The interface allows sending a message to an object without concerning which classes it belongs.
* Class needs to provide functionality for the methods declared in the interface.
* All methods in an interface are implicitly public and abstract
* An interface cannot be instantiated
* An interface reference can point to objects of its implementing classes
* An interface can extend from one or many interfaces. Class can extend only one class but implement any number of interfaces
* An interface cannot implement another Interface. It has to extend another interface if needed.
* An interface which is declared inside another interface is referred as nested interface
* At the time of declaration, interface variable must be initialized. Otherwise, the compiler will throw an error.
* The class cannot implement two interfaces in java that have methods with same name but different return type.
* **Inheritance**

Inheritance is the mechanism that permits new classes to be created out of existing classes by extending and refining its capabilities. The existing classes are called the base classes/parent classes/super-classes, and the new classes are called the derived classes/child classes/subclasses. The subclass can inherit or derive the attributes and methods of the super-class(es) provided that the super-class allows so. Besides, the subclass may add its own attributes and methods and may modify any of the super-class methods. Inheritance defines an “is – a” relationship.

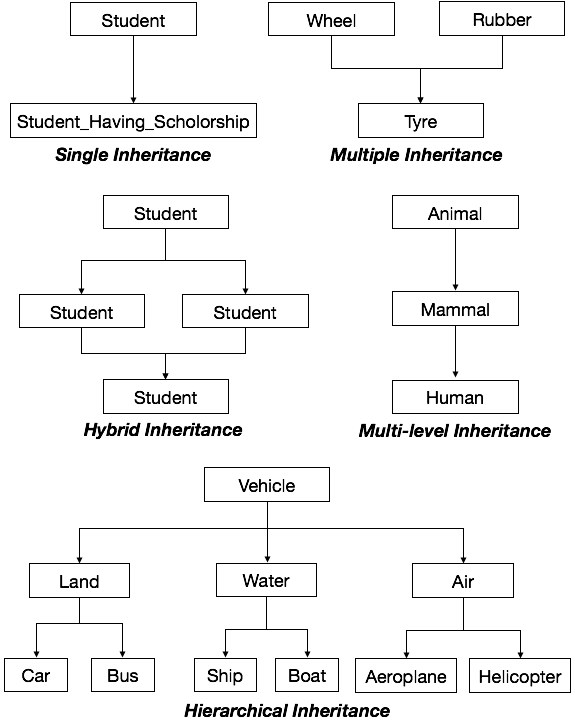
**Example**

From a class Mammal, a number of classes can be derived such as Human, Cat, Dog, Cow, etc. Humans, cats, dogs, and cows all have the distinct characteristics of mammals. In addition, each has its own particular characteristics. It can be said that a cow “is – a” mammal.

Types of Inheritance

* **Single Inheritance** − A subclass derives from a single super-class.
* **Multiple Inheritance** − A subclass derives from more than one super-classes.
* **Multilevel Inheritance** − A subclass derives from a super-class which in turn is derived from another class and so on.
* **Hierarchical Inheritance** − A class has a number of subclasses each of which may have subsequent subclasses, continuing for a number of levels, so as to form a tree structure.
* **Hybrid Inheritance** − A combination of multiple and multilevel inheritance so as to form a lattice structure.

The following figure are the examples of different types of inheritance.



* **Polymorphism**

Polymorphism as the ability of a message to be displayed in more than one form The word polymorphism means having many forms. In simple words, we can define.

Real life example of polymorphism: 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 behavior in different situations. This is called polymorphism.

Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word “poly” means many and “morphs” means forms, So it means many forms.

Polymorphism is originally a Greek word that means the ability to take multiple forms. In object-oriented paradigm, polymorphism implies using operations in different ways, depending upon the instance they are operating upon. Polymorphism allows objects with different internal structures to have a common external interface. Polymorphism is particularly effective while implementing inheritance.

**Example**

Let us consider two classes, Circle and Square, each with a method findArea(). Though the name and purpose of the methods in the classes are same, the internal implementation, i.e., the procedure of calculating area is different for each class. When an object of class Circle invokes its findArea() method, the operation finds the area of the circle without any conflict with the findArea() method of the Square class.

**Polymorphism** in Java occurs when there are one or more classes or objects related to each other by inheritance. It is the ability of an object to take many forms. Inheritance lets users inherit attributes and methods, and polymorphism uses these methods to perform different tasks. So, the goal is communication, but the approach is different.

For example, you have a Smartphone for communication. The communication mode you choose could be anything. It can be a call, a text message, a picture message, mail, etc. So, the goal is common that is communication, but their approach is different. This is called **Polymorphism.**

**Java Polymorphism in OOPs with Example**

We have one parent class, ‘Account’ with function of deposit and withdraw. Account has 2 child classes

The operation of deposit and withdraw is same for Saving and Checking accounts. So the inherited methods from Account class will work.

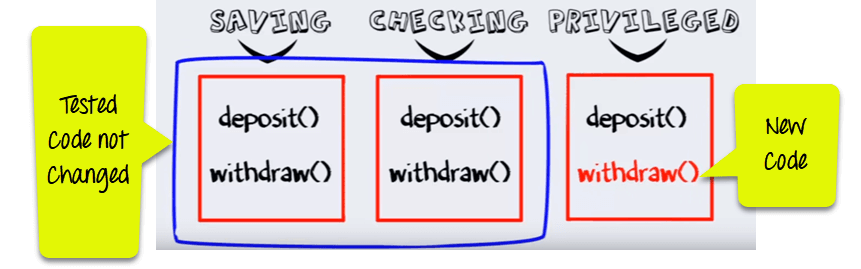
[](https://www.guru99.com/images/java/052016_0651_JavaInherit11.jpg)

**Change in Software Requirement**

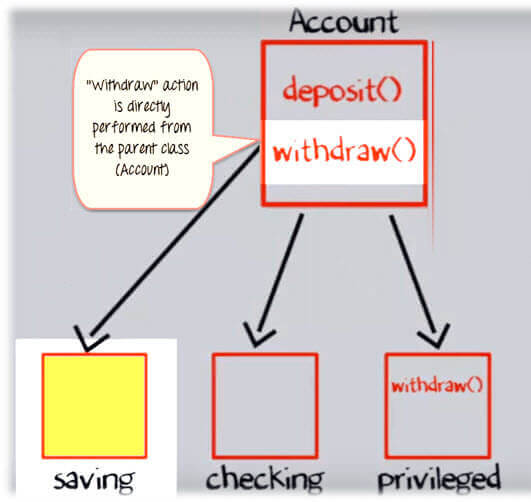
There is a change in the requirement specification, something that is so common in the software industry. You are supposed to add functionality privileged Banking Account with Overdraft Facility.

For a background, overdraft is a facility where you can withdraw an amount more than available the balance in your account.

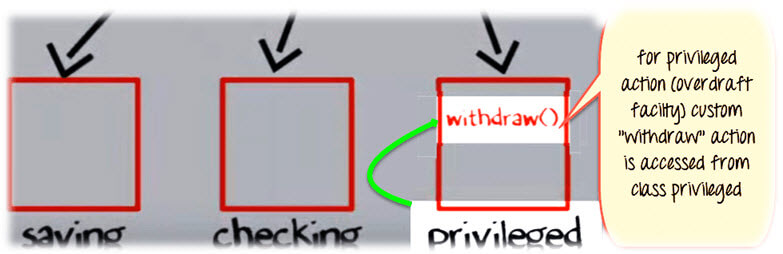
So, withdraw method for privileged needs to implemented afresh. But you do not change the tested piece of code in Savings and Checking account. This is advantage of OOPS

[](https://www.guru99.com/images/java/Poly3.png)

**Step 1)** Such that when the "withdrawn" method for saving account is called a method from parent account class is executed.

[](https://www.guru99.com/images/java/052016_0651_JavaInherit13.jpg)

**Step 2)**But when the "Withdraw" method for the privileged account (overdraft facility) is called withdraw method defined in the privileged class is executed. This is **Polymorphism in OOPs.**

[](https://www.guru99.com/images/java/052016_0651_JavaInherit14.jpg)

* **3.1.5 Structural Diagram - Class Diagram and Object diagram**
* **UML** :-

UML (Unified Modeling Language) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems. UML was created by the Object Management Group (OMG) and UML 1.0 specification draft was proposed to the OMG in January 1997. It was initially started to capture the behavior of complex software and non-software system and now it has become an OMG standard.

* UML stands for **Unified Modeling Language**.
* UML is different from the other common programming languages such as C++, Java, COBOL, etc.
* UML is a pictorial language used to make software blueprints.
* UML can be described as a general purpose visual modeling language to visualize, specify, construct, and document software system.
* Although UML is generally used to model software systems, it is not limited within this boundary. It is also used to model non-software systems as well. For example, the process flow in a manufacturing unit, etc.
* UML is not a programming language but tools can be used to generate code in various languages using UML diagrams

UML can be described as the successor of object-oriented (OO) analysis and design.

An object contains both data and methods that control the data. The data represents the state of the object. A class describes an object and they also form a hierarchy to model the real-world system. The hierarchy is represented as inheritance and the classes can also be associated in different ways as per the requirement.

Following are some fundamental concepts of the object-oriented world −

* **Objects** − Objects represent an entity and the basic building block.
* **Class** − Class is the blue print of an object.
* **Abstraction** − Abstraction represents the behavior of an real world entity.
* **Encapsulation** − Encapsulation is the mechanism of binding the data together and hiding them from the outside world.
* **Inheritance** − Inheritance is the mechanism of making new classes from existing ones.
* **Polymorphism** − It defines the mechanism to exists in different forms.

**The purpose of Object Oriented Analysis and Design (OOAD) can described as −**

* Identifying the objects of a system.
* Identifying their relationships.
* Making a design, which can be converted to executables using Object Oriented languages.

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

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

The building blocks of UML can be defined as −

* Things
* Relationships
* Diagrams

**Things**

**Things** are the most important building blocks of UML. Things can be −

* Structural
* Behavioral
* Grouping
* Annotational

**Structural Things**

**Structural things** define the static part of the model. They represent the physical and conceptual elements. Following are the brief descriptions of the structural things.

**Class −** Class represents a set of objects having similar responsibilities.

class

**Interface −** Interface defines a set of operations, which specify the responsibility of a class.

Interface

**Collaboration −**Collaboration defines an interaction between elements.

Collaboration

**Use case −**Use case represents a set of actions performed by a system for a specific goal.

Use case

**Component −**Component describes the physical part of a system.

Component

**Node −** A node can be defined as a physical element that exists at run time.



**Behavioral Things**

**A behavioral thing** consists of the dynamic parts of UML models. Following are the behavioral things −

**Interaction −** Interaction is defined as a behavior that consists of a group of messages exchanged among elements to accomplish a specific task.

Interaction

**State machine −** State machine is useful when the state of an object in its life cycle is important. It defines the sequence of states an object goes through in response to events. Events are external factors responsible for state change



**Grouping Things-**

**Grouping things** can be defined as a mechanism to group elements of a UML model together. There is only one grouping thing available −

**Package −** Package is the only one grouping thing available for gathering structural and behavioral things.



**Annotational Things-**

**Annotational things** can be defined as a mechanism to capture remarks, descriptions, and comments of UML model elements. **Note** - It is the only one Annotational thing available. A note is used to render comments, constraints, etc. of an UML element.

Note

**Relationship-**

**Relationship** is another most important building block of UML. It shows how the elements are associated with each other and this association describes the functionality of an application.

There are four kinds of relationships available.

**Dependency-**

Dependency is a relationship between two things in which change in one element also affects the other.

Dependency

**Association**-

Association is basically a set of links that connects the elements of a UML model. It also describes how many objects are taking part in that relationship.

Association

**Generalization-**

Generalization can be defined as a relationship which connects a specialized element with a generalized element. It basically describes the inheritance relationship in the world of objects.

Generalization

**Realization-**

Realization can be defined as a relationship in which two elements are connected. One element describes some responsibility, which is not implemented and the other one implements them. This relationship exists in case of interfaces.

Realization

**What is**[**UML**](https://www.geeksforgeeks.org/unified-modeling-language-uml-introduction/)**?**

It is the general-purpose modeling language used to visualize the system. It is a graphical language that is standard to the software industry for specifying, visualizing, constructing, and documenting the artifacts of the software systems, as well as for business modeling.

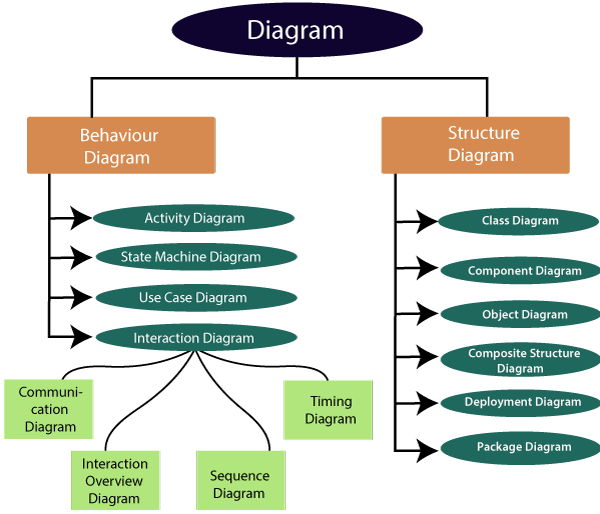
**Benefits of UML:**

* Simplifies complex software design, can also implement OOPs like a concept that is widely used.
* It reduces thousands of words of explanation in a few graphical diagrams that may reduce time consumption to understand.
* It makes communication more clear and more real.
* It helps to acquire the entire system in a view.
* It becomes very much easy for the software programmer to implement the actual demand once they have a clear picture of the problem.

UML includes the following nine diagrams:-

* Class diagram
* Object diagram
* Use case diagram
* Sequence diagram
* Collaboration diagram
* Activity diagram
* State chart diagram
* Deployment diagram
* Component diagram

The UML diagrams are categorized into **structural diagrams, behavioral diagrams,** and also interaction **overview diagrams.** The diagrams are hierarchically classified in the following figure:



**UML-Relationship**

Relationships depict a connection between several things, such as structural, behavioral, or grouping things in the unified modeling language. Since it is termed as a link, it demonstrates how things are interrelated to each other at the time of system execution. It constitutes four types of relationships, i.e., **dependency, association, generalization,** and **realization.**

* **Dependency**

Whenever there is a change in either the structure or the behavior of the class that affects the other class, such a relationship is termed as a dependency. Or, simply, we can say a class contained in other class is known as dependency. It is a unidirectional relationship.

* **Association**

Association is a structural relationship that represents how two entities are linked or connected to each other within a system. It can form several types of associations, such as **one-to-one, one-to-many, many-to-one,** and **many-to-many.** A ternary association is one that constitutes three links. It portrays the static relationship between the entities of two classes.

An association can be categorized into four types of associations, i.e., bi-directional, unidirectional, aggregation (composition aggregation), and reflexive, such that an aggregation is a special form of association and composition is a special form of aggregation. The mostly used associations are unidirectional and bi-directional.

* **Aggregation**

An aggregation is a special form of association. It portrays a part-of relationship. It forms a binary relationship, which means it cannot include more than two classes. It is also known as **Has-a relationship.** It specifies the direction of an object contained in another object. In aggregation, a child can exist independent of the parent.

* **Composition**

In a composition relationship, the child depends on the parent. It forms a two-way relationship. It is a special case of aggregation. It is known as **Part-of** relationship.

**Aggregation VS Composition relationship:-**

|  |  |  |
| --- | --- | --- |
| **Features** | **Aggregation relationship** | **Composition relationship** |
| **Dependency** | In an aggregation relationship, a child can exist independent of a parent. | In a composition relationship, the child cannot exist independent of the parent. |
| **Type of Relationship** | It constitutes a **Has-a** relationship. | It constitutes **Part-of** relationship. |
| **Type of Association** | It forms a **weak** association. | It forms a **strong** association. |
| **Examples** | A doctor has patients when the doctor gets transfer to another hospital, the patients do not accompany to a new workplace. | A hospital and its wards. If the hospital is destroyed, the wards also get destroyed. |

* **Generalization**

The generalization relationship implements the object-oriented concept called inheritance or **is-a** relationship. It exists between two objects (things or entities), such that one entity is a parent (super class or base class), and the other one is a child (subclass or derived class). These are represented in terms of inheritance. Any child can access, update, or inherit the functionality, structure, and behavior of the parent.

* **Realization**

It is a kind of relationship in which one thing specifies the behavior or a responsibility to be carried out, and the other thing carries out that behavior. It can be represented on a class diagram or component diagrams. The realization relationship is constituted between interfaces, classes, packages, and components to link a client element to the supplier element.

In UML diagrams, relationships are used to link several things. It is a connection between structural, behavioral, or grouping things. Following are the standard UML relationships enlisted below:

* Association
* Dependency
* Generalization
* Realization

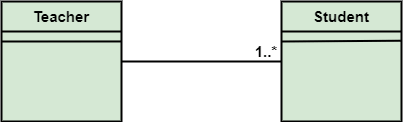
**Association**

Association relationship is a structural relationship in which different objects are linked within the system. It exhibits a binary relationship between the objects representing an activity. It depicts the relationship between objects, such as a teacher, can be associated with multiple teachers.

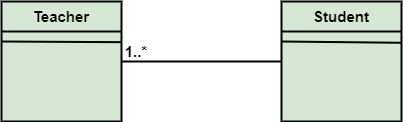
It is represented by a line between the classes followed by an arrow that navigates the direction, and when the arrow is on both sides, it is then called a bidirectional association. We can specify the multiplicity of an association by adding the adornments on the line that will denote the association.

Example:

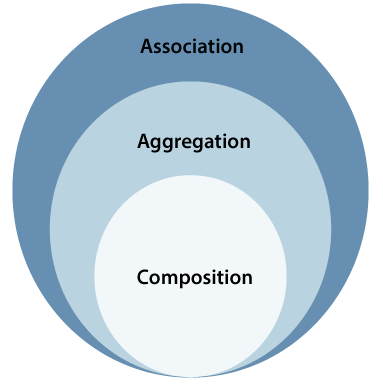
1) A single teacher has multiple students.



2) A single student can associate with many teachers.



The composition and aggregation are two subsets of association. In both of the cases, the object of one class is owned by the object of another class; the only difference is that in composition, the child does not exist independently of its parent, whereas in aggregation, the child is not dependent on its parent i.e., standalone. An aggregation is a special form of association, and composition is the special form of aggregation.

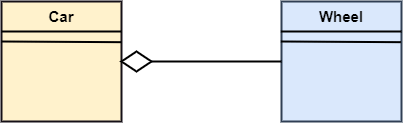


* **Aggregation**

Aggregation is a subset of association, is a collection of different things. It represents has a relationship. It is more specific than an association. It describes a part-whole or part-of relationship. It is a binary association, i.e., it only involves two classes. It is a kind of relationship in which the child is independent of its parent.

For example:

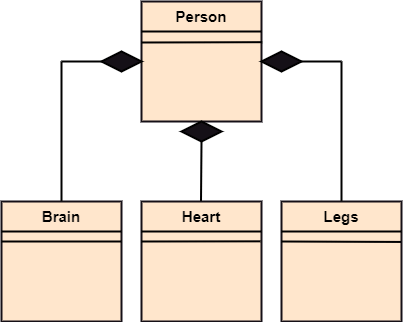
Here we are considering a car and a wheel example. A car cannot move without a wheel. But the wheel can be independently used with the bike, scooter, cycle, or any other vehicle. The wheel object can exist without the car object, which proves to be an aggregation relationship.



* **Composition**

The composition is a part of aggregation, and it portrays the whole-part relationship. It depicts dependency between a composite (parent) and its parts (children), which means that if the composite is discarded, so will its parts get deleted. It exists between similar objects.

As you can see from the example given below, the composition association relationship connects the Person class with Brain class, Heart class, and Legs class. If the person is destroyed, the brain, heart, and legs will also get discarded.

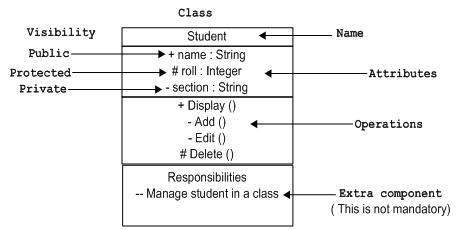


|  |  |  |
| --- | --- | --- |
| **Association** | **Aggregation** | **Composition** |
| Association relationship is represented using an arrow. | Aggregation relationship is represented by a straight line with an empty diamond at one end. | The composition relationship is represented by a straight line with a black diamond at one end. |
| In UML, it can exist between two or more classes. | It is a part of the association relationship. | It is a part of the aggregation relationship. |
| It incorporates one-to-one, one-to-many, many-to-one, and many-to-many association between the classes. | It exhibits a kind of weak relationship. | It exhibits a strong type of relationship. |
| It can associate one more objects together. | In an aggregation relationship, the associated objects exist independently within the scope of the system. | In a composition relationship, the associated objects cannot exist independently within the scope of the system. |
| In this, objects are linked together. | In this, the linked objects are independent of each other. | Here the linked objects are dependent on each other. |
| It may or may not affect the other associated element if one element is deleted. | Deleting one element in the aggregation relationship does not affect other associated elements. | It affects the other element if one of its associated element is deleted. |
| Example: A tutor can associate with multiple students, or one student can associate with multiple teachers. | Example: A car needs a wheel for its proper functioning, but it may not require the same wheel. It may function with another wheel as well. | Example: If a file is placed in a folder and that is folder is deleted. The file residing inside that folder will also get deleted at the time of folder deletion. |

* **Class Notation**

UML *class* is represented by the following figure. The diagram is divided into four parts.

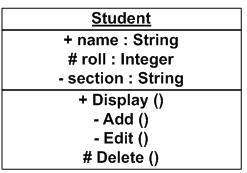
* The top section is used to name the class.
* The second one is used to show the attributes of the class.
* The third section is used to describe the operations performed by the class.
* The fourth section is optional to show any additional components.



Classes are used to represent objects. Objects can be anything having properties and responsibility.

* **Object Notation**

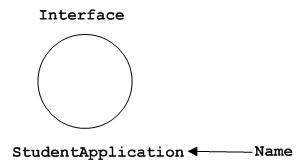
The *object* is represented in the same way as the class. The only difference is the *name* which is underlined as shown in the following figure.



As the object is an actual implementation of a class, which is known as the instance of a class. Hence, it has the same usage as the class.

* **Interface Notation**

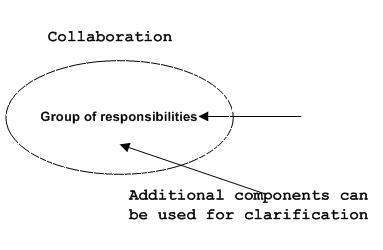
Interface is represented by a circle as shown in the following figure. It has a name which is generally written below the circle.



Interface is used to describe the functionality without implementation. Interface is just like a template where you define different functions, not the implementation. When a class implements the interface, it also implements the functionality as per requirement.

* **Collaboration Notation**

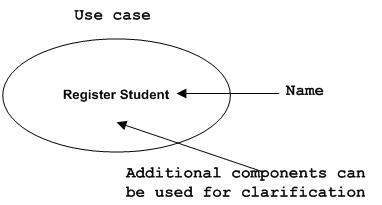
Collaboration is represented by a dotted eclipse as shown in the following figure. It has a name written inside the eclipse.



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

* **Use Case Notation**

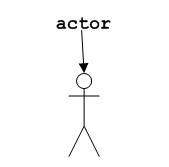
Use case is represented as an eclipse with a name inside it. It may contain additional responsibilities.



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

* **Actor Notation**

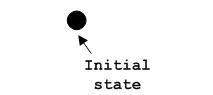
An actor can be defined as some internal or external entity that interacts with the system.



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

* **Initial State Notation**

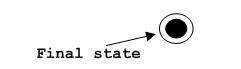
Initial state is defined to show the start of a process. This notation is used in almost all diagrams.



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

* **Final State Notation**

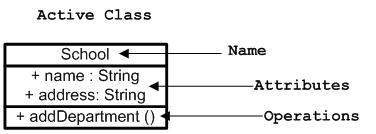
Final state is used to show the end of a process. This notation is also used in almost all diagrams to describe the end.



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

* **Active Class Notation**

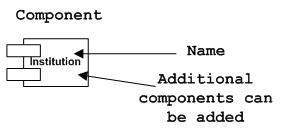
Active class looks similar to a class with a solid border. Active class is generally used to describe the concurrent behavior of a system.



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

* **Component Notation**

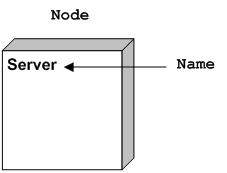
A component in UML is shown in the following figure with a name inside. Additional elements can be added wherever required.



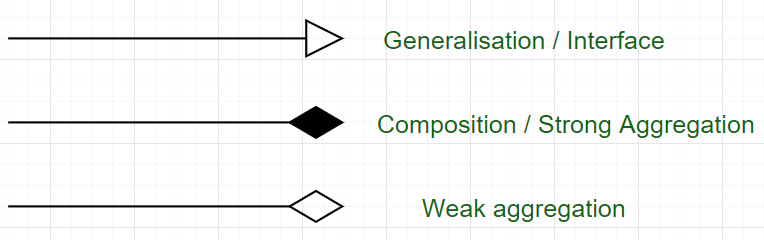
Component is used to represent any part of a system for which UML diagrams are made.

* **Node Notation**

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



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

The exact meaning of the arrows :   


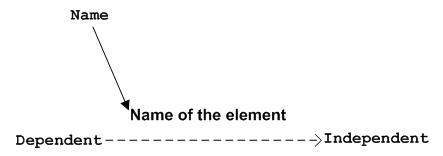
* Relationships

A model is not complete unless the relationships between elements are described properly. The *Relationship* gives a proper meaning to a UML model. Following are the different types of relationships available in UML.

* Dependency
* Association
* Generalization
* Extensibility
* **Dependency Notation**

Dependency is an important aspect in UML elements. It describes the dependent elements and the direction of dependency.

Dependency is represented by a dotted arrow as shown in the following figure. The arrow head represents the independent element and the other end represents the dependent element.

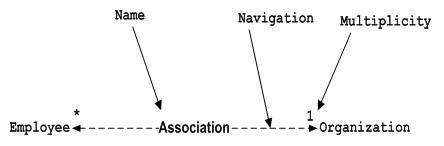


Dependency is used to represent the dependency between two elements of a system

* **Association Notation**

Association describes how the elements in a UML diagram are associated. In simple words, it describes how many elements are taking part in an interaction.

Association is represented by a dotted line with (without) arrows on both sides. The two ends represent two associated elements as shown in the following figure. The multiplicity is also mentioned at the ends (1, \*, etc.) to show how many objects are associated.



Association is used to represent the relationship between two elements of a system.

* **Generalization Notation**

Generalization describes the inheritance relationship of the object-oriented world. It is a parent and child relationship.

Generalization is represented by an arrow with a hollow arrow head as shown in the following figure. One end represents the parent element and the other end represents the child element.

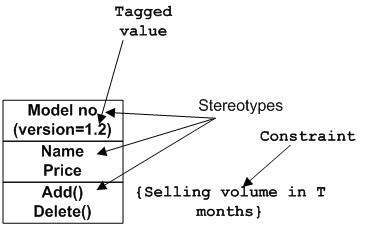
Generalization Notation

Generalization is used to describe parent-child relationship of two elements of a system.

* **Extensibility Notation**

All the languages (programming or modeling) have some mechanism to extend its capabilities such as syntax, semantics, etc. UML also has the following mechanisms to provide extensibility features.

* Stereotypes (Represents new elements)
* Tagged values (Represents new attributes)
* Constraints (Represents the boundaries)



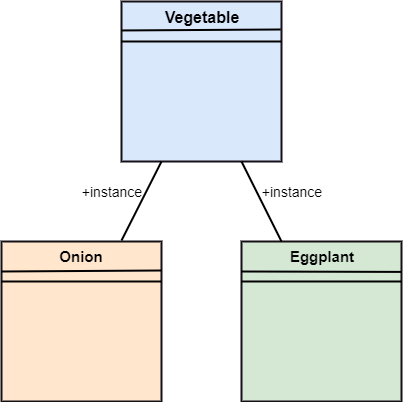
Extensibility notations are used to enhance the power of the language. It is basically additional elements used to represent some extra behavior of the system. These extra behaviors are not covered by the standard available notations.

**Association** is the semantic relationship between classes that shows how one instance is connected or merged with others in a system. The objects are combined either logically or physically. Since it connects the object of one class to the object of another class, it is categorized as a structural relationship

* **Reflexive Association**

In the reflexive associations, the links are between the objects of the same classes. In other words, it can be said that the reflexive association consists of the same class at both ends. An object can also be termed as an instance.

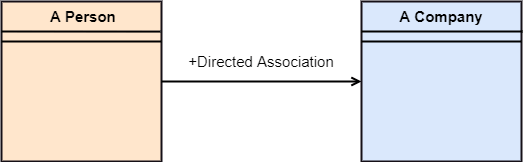
Let's have a look at its example of a class vegetable. The vegetable class has two objects, i.e., onion and eggplant. According to the reflexive association's definition, the link between the onion and eggplant (Brinjal) exist, as they belong to the same class, i.e., vegetable.



* **Directed Association**

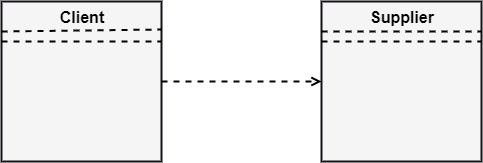
The directed association is concerned with the direction of flow inside association classes. The flow of association can be shown by employing a directed association. The directed association between two classes is represented by a line with an arrowhead, which indicates the navigation direction. The flow of association from one class to another is always in one direction.

It can be said that there is an association between a person and the company. The person works for the company. Here the person works for the company, and not the company works for a person.



* **Dependency :-**

Dependency depicts how various things within a system are dependent on each other. In UML, a dependency relationship is the kind of relationship in which a client (one element) is dependent on the supplier (another element). It is used in class diagrams, component diagrams, deployment diagrams, and use-case diagrams, which indicates that a change to the supplier necessitates a change to the client. An example is given below:



* **Generalization**

In UML modeling, a generalization relationship is a relationship that implements the concept of object orientation called inheritance. The generalization relationship occurs between two entities or objects, such that one entity is the parent, and the other one is the child. The child inherits the functionality of its parent and can access as well as update it.

Generalization relationship is utilized in class, component, deployment, and use case diagrams to specify that the child inherits actions, characteristics, and relationships from its parent.

The generalization relationship is incorporated to record attributes, operations, and relationships in a parent model element so that it can be inherited in one or more child model elements.

The parent model element can have as many children, and also, the child can have one or more parents. But most commonly, it can be seen that there is one parent model element and multiple child model elements. The generalization relationship does not consist of names. The generalization relationship is represented by a solid line with a hollow arrowhead pointing towards the parent model element from the child model element.



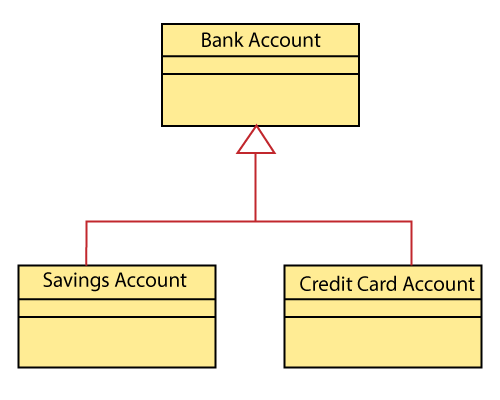
* **Stereotypes and their constraints**

**<<implementation>> -** It is used to show that the child is implemented by its parent, such that the child object inherits the structure and behavior of its parent object without disobeying the rules. The **implementation** of stereotype is mostly used in single inheritance.

In the generalization stereotype, there are two types of constraints that are **complete** and **incomplete** to check if all the child objects are involved or not in the relationship.

Example:

As we know, the bank account can be of two types; Savings Account and Credit Card Account. Both the savings and the credit card account inherits the generalized properties from the Bank Account, which is Account Number, Account Balance, etc.



**Realization**

In UML modeling, the realization is a relationship between two objects, where the client (one model element) implements the responsibility specified by the supplier (another model element). The realization relationship can be employed in class diagrams and components diagrams.

The realization relationship does not have names. It is mostly found in the interfaces. It is represented by a dashed line with a hollow arrowhead at one end that points from the client to the server.

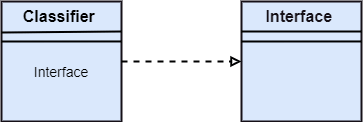
**Interface Realization**

Interface realization is a kind of specialized relation between the classifier and the interface. In interface realization relationship, realizing classifiers conforms to the contract defined by the interface.

A classifier implementing an interface identifies the objects that conform to the interface and any of its ancestors. A classifier can execute one or more interfaces. The set of interfaces that are implemented by the classifier are its **given interfaces.** The given interfaces are the set of services offered by the classifiers to its clients.

The interface realization relationship does not contain names, and if you name it, then the name will appear beside the connector in the diagram.

The interface realization relationship is represented by a dashed line with a hollow arrowhead, which points from the classifier to the given interface.



* **Class Diagram**

Class diagrams are the most common diagrams used in UML. Class diagram consists of classes, interfaces, associations, and collaboration. Class diagrams basically represent the object-oriented view of a system, which is static in nature.

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

Class diagram represents the object orientation of a system. Hence, it is generally used for development purpose. This is the most widely used diagram at the time of system construction.

Class diagram is a static diagram. It represents the static view of an application. Class diagram is not only used for visualizing, describing, and documenting different aspects of a system but also for constructing executable code of the software application.

Class diagram describes the attributes and operations of a class and also the constraints imposed on the system. The class diagrams are widely used in the modeling of object oriented systems because they are the only UML diagrams, which can be mapped directly with object-oriented languages.

Class diagram shows a collection of classes, interfaces, associations, collaborations, and constraints. It is also known as a structural diagram.

**Purpose of Class Diagrams**

The purpose of class diagram is to model the static view of an application. Class diagrams are the only diagrams which can be directly mapped with object-oriented languages and thus widely used at the time of construction.

UML diagrams like activity diagram, sequence diagram can only give the sequence flow of the application, however class diagram is a bit different. It is the most popular UML diagram in the coder community.

The purpose of the class diagram can be summarized as −

* Analysis and design of the static view of an application.
* Describe responsibilities of a system.
* Base for component and deployment diagrams.
* Forward and reverse engineering.

**Benefits of Class Diagrams**

1. It can represent the object model for complex systems.
2. It reduces the maintenance time by providing an overview of how an application is structured before coding.
3. It provides a general schematic of an application for better understanding.
4. It represents a detailed chart by highlighting the desired code, which is to be programmed.
5. It is helpful for the stakeholders and the developers.

**Vital components of a Class Diagram**

The class diagram is made up of three sections:

* **Upper Section:** The upper section encompasses the name of the class. A class is a representation of similar objects that shares the same relationships, attributes, operations, and semantics. Some of the following rules that should be taken into account while representing a class are given below:
* Capitalize the initial letter of the class name.
* Place the class name in the center of the upper section.
* A class name must be written in bold format.
* The name of the abstract class should be written in italics format.
* **Middle Section:** The middle section constitutes the attributes, which describe the quality of the class. The attributes have the following characteristics:

The attributes are written along with its visibility factors, which are public (+), private (-), protected (#), and package (~).

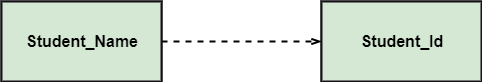
* The accessibility of an attribute class is illustrated by the visibility factors.
* A meaningful name should be assigned to the attribute, which will explain its usage inside the class.
* **Lower Section:** The lower section contain methods or operations. The methods are represented in the form of a list, where each method is written in a single line. It demonstrates how a class interacts with data.



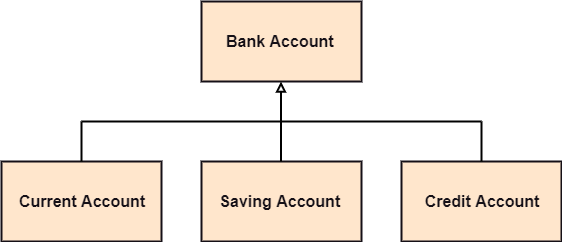
**Relationships**

In UML, relationships are of three types:

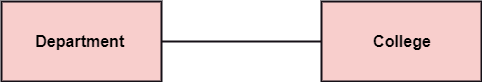
* **Dependency:** A dependency is a semantic relationship between two or more classes where a change in one class cause changes in another class. It forms a weaker relationship.  
  In the following example, Student\_Name is dependent on the Student\_Id.



* **Generalization:** A generalization is a relationship between a parent class (superclass) and a child class (subclass). In this, the child class is inherited from the parent class.  
  For example, The Current Account, Saving Account, and Credit Account are the generalized form of Bank Account.

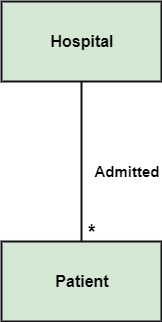


* **Association:** It describes a static or physical connection between two or more objects. It depicts how many objects are there in the relationship.  
  For example, a department is associated with the college.



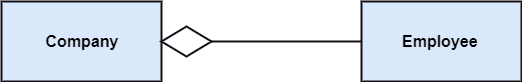
**Multiplicity:** It defines a specific range of allowable instances of attributes. In case if a range is not specified, one is considered as a default multiplicity.

For example, multiple patients are admitted to one hospital.



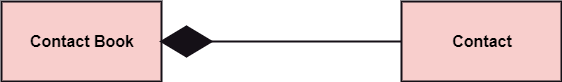
**Aggregation:** An aggregation is a subset of association, which represents has a relationship. It is more specific then association. It defines a part-whole or part-of relationship. In this kind of relationship, the child class can exist independently of its parent class.

The company encompasses a number of employees, and even if one employee resigns, the company still exists.



**Composition:** The composition is a subset of aggregation. It portrays the dependency between the parent and its child, which means if one part is deleted, then the other part also gets discarded. It represents a whole-part relationship.

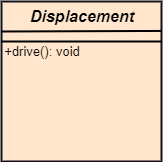
A contact book consists of multiple contacts, and if you delete the contact book, all the contacts will be lost.



**Abstract Classes**

In the abstract class, no objects can be a direct entity of the abstract class. The abstract class can neither be declared nor be instantiated. It is used to find the functionalities across the classes. The notation of the abstract class is similar to that of class; the only difference is that the name of the class is written in italics. Since it does not involve any implementation for a given function, it is best to use the abstract class with multiple objects.

Let us assume that we have an abstract class named **displacement** with a method declared inside it, and that method will be called as a **drive ()**. Now, this abstract class method can be implemented by any object, for example, car, bike, scooter, cycle, etc.



**How to draw a Class Diagram?**

Class diagrams are the most popular UML diagrams used for construction of software applications. It is very important to learn the drawing procedure of class diagram.

Class diagrams have a lot of properties to consider while drawing but here the diagram will be considered from a top level view.

Class diagram is basically a graphical representation of the static view of the system and represents different aspects of the application. A collection of class diagrams represent the whole system.

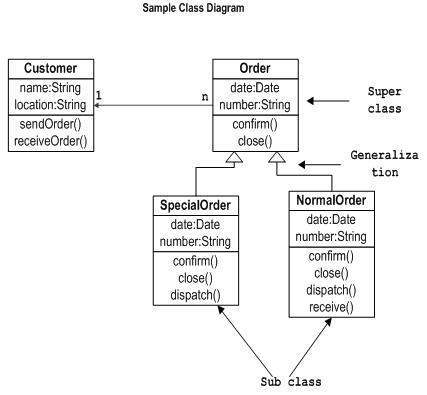
The following points should be remembered while drawing a class diagram −

* The name of the class diagram should be meaningful to describe the aspect of the system.
* Each element and their relationships should be identified in advance.
* Responsibility (attributes and methods) of each class should be clearly identified
* For each class, minimum number of properties should be specified, as unnecessary properties will make the diagram complicated.
* Use notes whenever required to describe some aspect of the diagram. At the end of the drawing it should be understandable to the developer/coder.
* Finally, before making the final version, the diagram should be drawn on plain paper and reworked as many times as possible to make it correct.

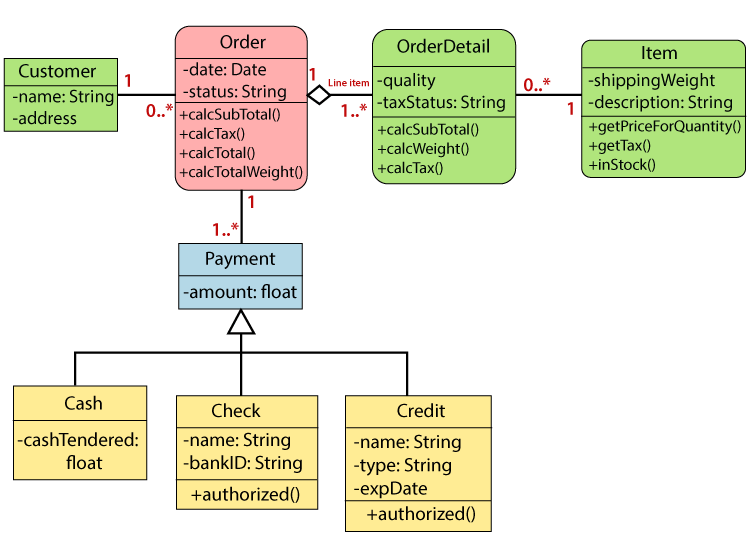
The following diagram is an **example of an Order System of an application**. It describes a particular aspect of the entire application.

* First of all, Order and Customer are identified as the two elements of the system. They have a one-to-many relationship because a customer can have multiple orders.
* Order class is an abstract class and it has two concrete classes (inheritance relationship) SpecialOrder and NormalOrder.
* The two inherited classes have all the properties as the Order class. In addition, they have additional functions like dispatch () and receive ().

The following class diagram has been drawn considering all the points mentioned above.



A class diagram describing the **sales order system** is given below.



**Where to Use Class Diagrams?**

Class diagram is a static diagram and it is used to model the static view of a system. The static view describes the vocabulary of the system.

Class diagram is also considered as the foundation for component and deployment diagrams. Class diagrams are not only used to visualize the static view of the system but they are also used to construct the executable code for forward and reverse engineering of any system.

Generally, UML diagrams are not directly mapped with any object-oriented programming languages but the class diagram is an exception.

Class diagram clearly shows the mapping with object-oriented languages such as Java, C++, etc. From practical experience, class diagram is generally used for construction purpose.

**Class diagrams are used for −**

* Describing the static view of the system.
* Showing the collaboration among the elements of the static view.
* Describing the functionalities performed by the system.
* Construction of software applications using object oriented languages.

### Object Diagram

Object diagrams can be described as an instance of class diagram. Thus, these diagrams are more close to real-life scenarios where we implement a system.

Object diagrams are a set of objects and their relationship is just like class diagrams. They also represent the static view of the system.

The usage of object diagrams is similar to class diagrams but they are used to build prototype of a system from a practical perspective.

Object diagrams are derived from class diagrams so object diagrams are dependent upon class diagrams.

Object diagrams represent an instance of a class diagram. The basic concepts are similar for class diagrams and object diagrams. Object diagrams also represent the static view of a system but this static view is a snapshot of the system at a particular moment.

Object diagrams are used to render a set of objects and their relationships as an instance.

**Purpose of Object Diagrams**

The purpose of a diagram should be understood clearly to implement it practically. The purposes of object diagrams are similar to class diagrams.

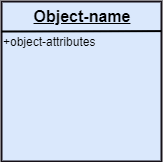
The difference is that a class diagram represents an abstract model consisting of classes and their relationships. However, an object diagram represents an instance at a particular moment, which is concrete in nature.

It means the object diagram is closer to the actual system behavior. The purpose is to capture the static view of a system at a particular moment.

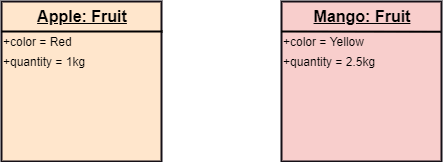
The purpose of the object diagram can be summarized as −

* Forward and reverse engineering.
* Object relationships of a system
* Static view of an interaction.
* Understand object behavior and their relationship from practical perspective
* It is used to represent an instance of a system.

Notation of an Object Diagram



Example of Object Diagram



**How to Draw an Object Diagram?**

We have already discussed that an object diagram is an instance of a class diagram. It implies that an object diagram consists of instances of things used in a class diagram.

So both diagrams are made of same basic elements but in different form. In class diagram elements are in abstract form to represent the blue print and in object diagram the elements are in concrete form to represent the real world object.

To capture a particular system, numbers of class diagrams are limited. However, if we consider object diagrams then we can have unlimited number of instances, which are unique in nature. Only those instances are considered, which have an impact on the system.

From the above discussion, it is clear that a single object diagram cannot capture all the necessary instances or rather cannot specify all the objects of a system. Hence, the solution is −

* First, analyze the system and decide which instances have important data and association.
* Second, consider only those instances, which will cover the functionality.
* Third, make some optimization as the number of instances are unlimited.

Before drawing an object diagram, the following things should be remembered and understood clearly −

* Object diagrams consist of objects.
* The link in object diagram is used to connect objects.
* Objects and links are the two elements used to construct an object diagram.

After this, the following things are to be decided before starting the construction of the diagram −

* The object diagram should have a meaningful name to indicate its purpose.
* The most important elements are to be identified.
* The association among objects should be clarified.
* Values of different elements need to be captured to include in the object diagram.
* Add proper notes at points where more clarity is required.

The following diagram is an example of an object diagram. It represents the Order management system which we have discussed in the chapter Class Diagram. The following diagram is an instance of the system at a particular time of purchase. It has the following objects.

* Customer
* Order
* SpecialOrder
* NormalOrder

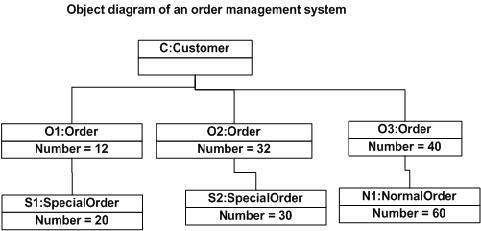
Now the customer object (C) is associated with three order objects (O1, O2, and O3). These order objects are associated with special order and normal order objects (S1, S2, and N1). The customer has the following three orders with different numbers (12, 32 and 40) for the particular time considered.

The customer can increase the number of orders in future and in that scenario the object diagram will reflect that. If order, special order, and normal order objects are observed then you will find that they have some values.

For orders, the values are 12, 32, and 40 which implies that the objects have these values for a particular moment (here the particular time when the purchase is made is considered as the moment) when the instance is captured

The same is true for special order and normal order objects which have number of orders as 20, 30, and 60. If a different time of purchase is considered, then these values will change accordingly.

The following object diagram has been drawn considering all the points mentioned above



**Where to Use Object Diagrams?**

Object diagrams can be imagined as the snapshot of a running system at a particular moment. Let us consider an example of a running train

Now, if you take a snap of the running train then you will find a static picture of it having the following −

* A particular state which is running.
* A particular number of passengers. which will change if the snap is taken in a different time

Here, we can imagine the snap of the running train is an object having the above values. And this is true for any real-life simple or complex system.

**Object Diagrams are used for −**

* Making the prototype of a system.
* Reverse engineering.
* Modeling complex data structures.
* Understanding the system from practical perspective.

Class vs. Object diagram

|  |  |  |
| --- | --- | --- |
| **Serial No.** | **Class Diagram** | **Object Diagram** |
| 1. | It depicts the static view of a system. | It portrays the real-time behavior of a system. |
| 2. | Dynamic changes are not included in the class diagram. | Dynamic changes are captured in the object diagram. |
| 3. | The data values and attributes of an instance are not involved here. | It incorporates data values and attributes of an entity. |
| 4. | The object behavior is manipulated in the class diagram. | Objects are the instances of a class. |

* **Component Diagram**

Component diagrams represent a set of components and their relationships. These components consist of classes, interfaces, or collaborations. Component diagrams represent the implementation view of a system.

During the design phase, software artifacts (classes, interfaces, etc.) of a system are arranged in different groups depending upon their relationship. Now, these groups are known as components.

Finally, it can be said component diagrams are used to visualize the implementation.

Component diagrams are different in terms of nature and behavior. Component diagrams are used to model the physical aspects of a system. Now the question is, what are these physical aspects? Physical aspects are the elements such as executables, libraries, files, documents, etc. which reside in a node.

Component diagrams are used to visualize the organization and relationships among components in a system. These diagrams are also used to make executable systems.

**Purpose of Component Diagrams**

Component diagram is a special kind of diagram in UML. The purpose is also different from all other diagrams discussed so far. It does not describe the functionality of the system but it describes the components used to make those functionalities.

Thus from that point of view, component diagrams are used to visualize the physical components in a system. These components are libraries, packages, files, etc.

Component diagrams can also be described as a static implementation view of a system. Static implementation represents the organization of the components at a particular moment.

A single component diagram cannot represent the entire system but a collection of diagrams is used to represent the whole.

The purpose of the component diagram can be summarized as −

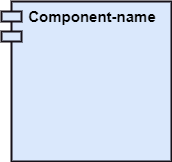
* Visualize the components of a system.
* Construct executables by using forward and reverse engineering.
* Describe the organization and relationships of the components.

A component diagram is used to break down a large object-oriented system into the smaller components, so as to make them more manageable. It models the physical view of a system such as executables, files, libraries, etc. that resides within the node.

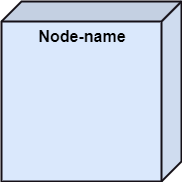
It visualizes the relationships as well as the organization between the components present in the system. It helps in forming an executable system. A component is a single unit of the system, which is replaceable and executable. The implementation details of a component are hidden, and it necessitates an interface to execute a function. It is like a black box whose behavior is explained by the provided and required interfaces.

Notation of a Component Diagram

* a) A component



* b) A node



**Why use Component Diagram?**

The component diagrams have remarkable importance. It is used to depict the functionality and behavior of all the components present in the system, unlike other diagrams that are used to represent the architecture of the system, working of a system, or simply the system itself.

In UML, the component diagram portrays the behavior and organization of components at any instant of time. The system cannot be visualized by any individual component, but it can be by the collection of components.

Following are some reasons for the requirement of the component diagram:

1. It portrays the components of a system at the runtime.
2. It is helpful in testing a system.
3. It envisions the links between several connections.

**When to use a Component Diagram?**

It represents various physical components of a system at runtime. It is helpful in visualizing the structure and the organization of a system. It describes how individual components can together form a single system. Following are some reasons, which tells when to use component diagram:

1. To divide a single system into multiple components according to the functionality.
2. To represent the component organization of the system.

**How to Draw a Component Diagram?**

The component diagram is helpful in representing the physical aspects of a system, which are files, executables, libraries, etc. The main purpose of a component diagram is different from that of other diagrams. It is utilized in the implementation phase of any application.

Once the system is designed employing different UML diagrams, and the artifacts are prepared, the component diagram is used to get an idea of implementation. It plays an essential role in implementing applications efficiently.

Following are some artifacts that are needed to be identified before drawing a component diagram:

1. What files are used inside the system?
2. What is the application of relevant libraries and artifacts?
3. What is the relationship between the artifacts?

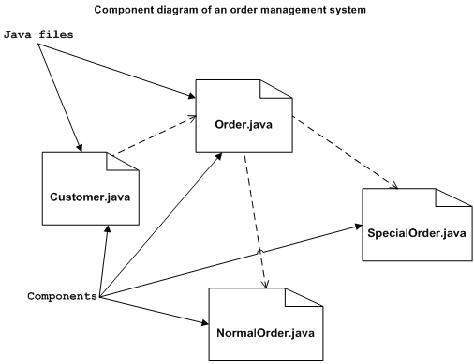
Following are some points that are needed to be kept in mind after the artifacts are identified:

1. Using a meaningful name to ascertain the component for which the diagram is about to be drawn.
2. Before producing the required tools, a mental layout is to be made.
3. To clarify the important points, notes can be incorporated.

Following is a component diagram for order management system. Here, the artifacts are files. The diagram shows the files in the application and their relationships. In actual, the component diagram also contains dlls, libraries, folders, etc.

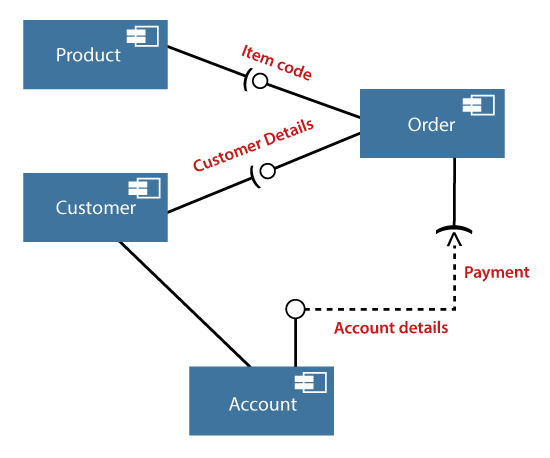
In the following diagram, four files are identified and their relationships are produced. Component diagram cannot be matched directly with other UML diagrams discussed so far as it is drawn for completely different purpose.

The following component diagram has been drawn considering all the points mentioned above.



Example of a Component Diagram

A component diagram for an online shopping system is given below:



**Where to Use Component Diagrams?**

We have already described that component diagrams are used to visualize the static implementation view of a system. Component diagrams are special type of UML diagrams used for different purposes.

These diagrams show the physical components of a system. To clarify it, we can say that component diagrams describe the organization of the components in a system.

Organization can be further described as the location of the components in a system. These components are organized in a special way to meet the system requirements.

As we have already discussed, those components are libraries, files, executables, etc. Before implementing the application, these components are to be organized. This component organization is also designed separately as a part of project execution.

Component diagrams are very important from implementation perspective. Thus, the implementation team of an application should have a proper knowledge of the component details

Component diagrams can be used to −

* Model the components of a system.
* Model the database schema.
* Model the executables of an application.
* Model the system's source code.
* **Deployment Diagram**

Deployment diagrams are a set of nodes and their relationships. These nodes are physical entities where the components are deployed.

Deployment diagrams are used for visualizing the deployment view of a system. This is generally used by the deployment team.

Deployment diagrams are used to visualize the topology of the physical components of a system, where the software components are deployed.

Deployment diagrams are used to describe the static deployment view of a system. Deployment diagrams consist of nodes and their relationships.

**Purpose of Deployment Diagrams**

The term Deployment itself describes the purpose of the diagram. Deployment diagrams are used for describing the hardware components, where software components are deployed. Component diagrams and deployment diagrams are closely related.

Component diagrams are used to describe the components and deployment diagrams shows how they are deployed in hardware.

UML is mainly designed to focus on the software artifacts of a system. However, these two diagrams are special diagrams used to focus on software and hardware components.

Most of the UML diagrams are used to handle logical components but deployment diagrams are made to focus on the hardware topology of a system. Deployment diagrams are used by the system engineers.

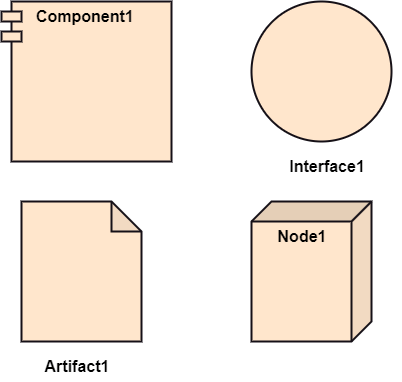
The purpose of deployment diagrams can be described as −

* Visualize the hardware topology of a system.
* Describe the hardware components used to deploy software components.
* Describe the runtime processing nodes.

**Symbol and notation of Deployment diagram**

The deployment diagram consist of the following notations:

1. A component
2. An artifact
3. An interface
4. A node



**How to Draw a Deployment Diagram?**

Deployment diagram represents the deployment view of a system. It is related to the component diagram because the components are deployed using the deployment diagrams. A deployment diagram consists of nodes. Nodes are nothing but physical hardware used to deploy the application.

Deployment diagrams are useful for system engineers. An efficient deployment diagram is very important as it controls the following parameters −

* Performance
* Scalability
* Maintainability
* Portability

Before drawing a deployment diagram, the following artifacts should be identified −

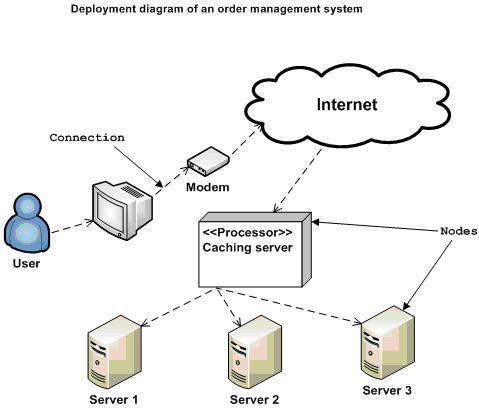
* Nodes
* Relationships among nodes

Following is a sample deployment diagram to provide an idea of the deployment view of order management system. Here, we have shown nodes as −

* Monitor
* Modem
* Caching server
* Server

The application is assumed to be a web-based application, which is deployed in a clustered environment using server 1, server 2, and server 3. The user connects to the application using the Internet. The control flows from the caching server to the clustered environment.

The following deployment diagram has been drawn considering all the points mentioned above.

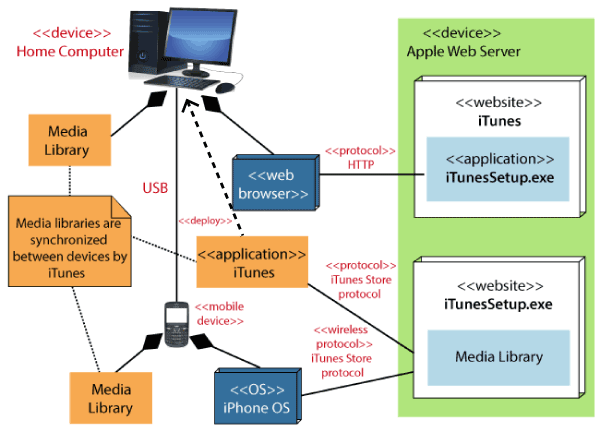


**Example of a Deployment diagram**

A deployment diagram for the Apple iTunes application is given below.

The iTunes setup can be downloaded from the iTunes website, and also it can be installed on the home computer. Once the installation and the registration are done, iTunes application can easily interconnect with the Apple iTunes store. Users can purchase and download music, video, TV serials, etc. and cache it in the media library.

Devices like Apple iPod Touch and Apple iPhone can update its own media library from the computer with iTunes with the help of USB or simply by downloading media directly from the Apple iTunes store using wireless protocols, for example; Wi-Fi, 3G, or EDGE.



**Where to Use Deployment Diagrams?**

Deployment diagrams are mainly used by system engineers. These diagrams are used to describe the physical components (hardware), their distribution, and association.

Deployment diagrams can be visualized as the hardware components/nodes on which the software components reside.

Software applications are developed to model complex business processes. Efficient software applications are not sufficient to meet the business requirements. Business requirements can be described as the need to support the increasing number of users, quick response time, etc.

To meet these types of requirements, hardware components should be designed efficiently and in a cost-effective way.

Now-a-days software applications are very complex in nature. Software applications can be standalone, web-based, distributed, mainframe-based and many more. Hence, it is very important to design the hardware components efficiently.

Deployment diagrams can be used −

* To model the hardware topology of a system.
* To model the embedded system.
* To model the hardware details for a client/server system.
* To model the hardware details of a distributed application.
* For Forward and Reverse engineering.
* For modeling the embedded system.

**Note** − If the above descriptions and usages are observed carefully then it is very clear that all the diagrams have some relationship with one another. Component diagrams are dependent upon the classes, interfaces, etc. which are part of class/object diagram. Again, the deployment diagram is dependent upon the components, which are used to make component diagrams.

* **Use Case Diagram** :-

To model a system, the most important aspect is to capture the dynamic behavior. Dynamic behavior means the behavior of the system when it is running /operating.

A Use Case diagram is used to represent the dynamic behavior of a system. It encapsulates the system's functionality by incorporating use cases, actors, and their relationships. It models the tasks, services, and functions required by a system/subsystem of an application. It depicts the high-level functionality of a system and also tells how the user handles a system.

**What is the Use Case Diagram?**

**Use Case Diagram** captures the system's functionality and requirements by using actors and use cases. Use Cases model the services, tasks, function that a system needs to perform. Use cases represent high-level functionalities and how a user will handle the system. Use-cases are the core concepts of Unified Modelling language modeling.

**Why Use-Case diagram?**

A Use Case consists of use cases, persons, or various things that are invoking the features called as actors and the elements that are responsible for implementing the use cases. Use case diagrams capture the dynamic behavior of a live system. It models how an external entity interacts with the system to make it work. Use case diagrams are responsible for visualizing the external things that interact with the part of the system.

**Purpose of Use Case Diagrams:-**

The purpose of use case diagram is to capture the dynamic aspect of a system. However, this definition is too generic to describe the purpose, as other four diagrams (activity, sequence, collaboration, and Statechart) also have the same purpose. We will look into some specific purpose, which will distinguish it from other four diagrams.

Use case diagrams are used to gather the requirements of a system including internal and external influences. These requirements are mostly design requirements. Hence, when a system is analyzed to gather its functionalities, use cases are prepared and actors are identified.

When the initial task is complete, use case diagrams are modelled to present the outside view.

In brief, the purposes of use case diagrams can be said to be as follows −

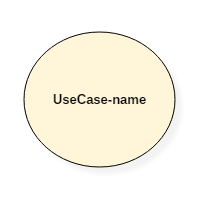
* Used to gather the requirements of a system.
* Used to get an outside view of a system.
* Identify the external and internal factors influencing the system.
* Show the interaction among the requirements are actors.

**Use-case diagram notations**

Following are the common notations used in a use case diagram

**Use-case:**

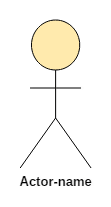
Use cases are used to represent high-level functionalities and how the user will handle the system. A use case represents a distinct functionality of a system, a component, a package, or a class. It is denoted by an oval shape with the name of a use case written inside the oval shape. The notation of a use case in UML is given below:

[](https://www.guru99.com/images/1/052919_0831_UMLUseCaseD1.png)

UML UseCase Notation

**Actor:**

It is used inside use case diagrams. The actor is an entity that interacts with the system. A user is the best example of an actor. An actor is an entity that initiates the use case from outside the scope of a use case. It can be any element that can trigger an interaction with the use case. One actor can be associated with multiple use cases in the system. The actor notation in UML is given below.

[](https://www.guru99.com/images/1/052919_0831_UMLUseCaseD2.png)

UML Actor Notation

**How to Draw a Use Case Diagram?**

Use case diagrams are considered for high level requirement analysis of a system. When the requirements of a system are analyzed, the functionalities are captured in use cases.

We can say that use cases are nothing but the system functionalities written in an organized manner. The second thing which is relevant to use cases are the actors. Actors can be defined as something that interacts with the system.

*Actors can be a human user, some internal applications, or may be some external applications.* When we are planning to draw a use case diagram, we should have the following items identified.

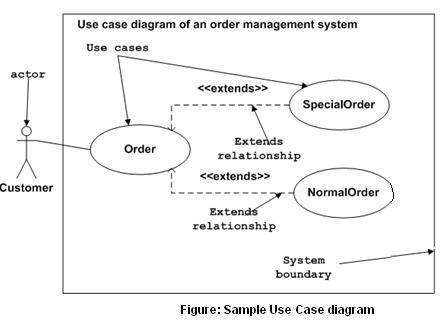
* Functionalities to be represented as use case
* Actors
* Relationships among the use cases and actors.

Use case diagrams are drawn to capture the functional requirements of a system. After identifying the above items, we have to use the following guidelines to draw an efficient use case diagram

* The name of a use case is very important. The name should be chosen in such a way so that it can identify the functionalities performed.
* Give a suitable name for actors.
* Show relationships and dependencies clearly in the diagram.
* Do not try to include all types of relationships, as the main purpose of the diagram is to identify the requirements.
* Use notes whenever required to clarify some important points.
* The most significant interactions should be represented among the multiple no of interactions between the use case and actors.

Following is a sample use case diagram representing the order management system. Hence, if we look into the diagram then we will find three use cases **(Order, SpecialOrder, and NormalOrder)** and one actor which is the customer.

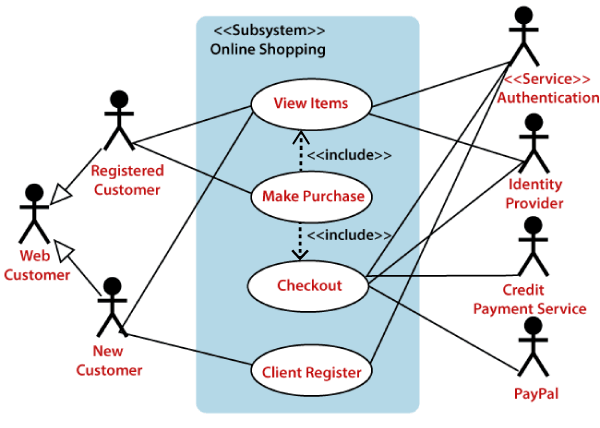
The SpecialOrder and NormalOrder use cases are extended from *Order* use case. Hence, they have extended relationship. Another important point is to identify the system boundary, which is shown in the picture. The actor Customer lies outside the system as it is an external user of the system.



**Another Example of a Use Case Diagram**

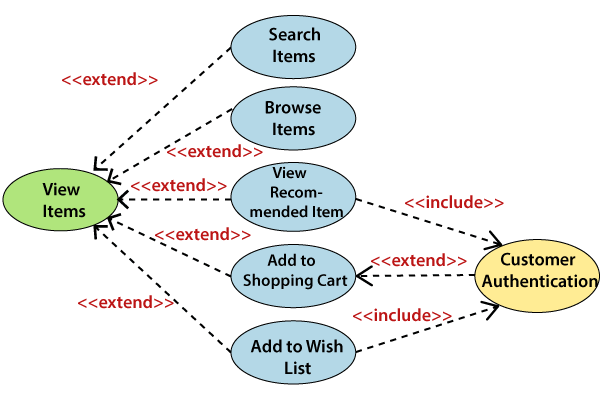
A use case diagram depicting the Online Shopping website is given below.

Here the Web Customer actor makes use of any online shopping website to purchase online. The top-level uses are as follows; View Items, Make Purchase, Checkout, Client Register. The **View Items** use case is utilized by the customer who searches and view products. The **Client Register** use case allows the customer to register itself with the website for availing gift vouchers, coupons, or getting a private sale invitation. It is to be noted that the **Checkout** is an included use case, which is part of **Making Purchase,** and it is not available by itself.



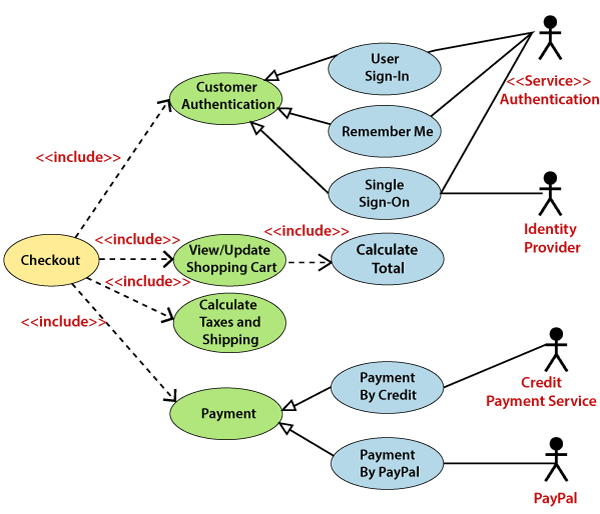
The **View Items** is further extended by several use cases such as; Search Items, Browse Items, View Recommended Items, Add to Shopping Cart, Add to Wish list. All of these extended use cases provide some functions to customers, which allows them to search for an item. The View Items is further extended by several use cases such as; Search Items, Browse Items, View Recommended Items, Add to Shopping Cart, Add to Wish list. All of these extended use cases provide some functions to customers, which allows them to search for an item.

Both **View Recommended Item** and **Add to Wish List** include the Customer Authentication use case, as they necessitate authenticated customers, and simultaneously item can be added to the shopping cart without any user authentication.



Similarly, the **Checkout** use case also includes the following use cases, as shown below. It requires an authenticated Web Customer, which can be done by login page, user authentication cookie ("Remember me"), or Single Sign-On (SSO). SSO needs an external identity provider's participation, while Web site authentication service is utilized in all these use cases.

The Checkout use case involves Payment use case that can be done either by the credit card and external credit payment services or with PayPal.



Important tips for drawing a Use Case diagram

Following are some important tips that are to be kept in mind while drawing a use case diagram:

1. A simple and complete use case diagram should be articulated.
2. A use case diagram should represent the most significant interaction among the multiple interactions.
3. At least one module of a system should be represented by the use case diagram.
4. If the use case diagram is large and more complex, then it should be drawn more generalized.

**Where to Use a Use Case Diagram?**

To understand the dynamics of a system, we need to use different types of diagrams. Use case diagram is one of them and its specific purpose is to gather system requirements and actors.

Use case diagrams specify the events of a system and their flows. But use case diagram never describes how they are implemented. Use case diagram can be imagined as a black box where only the input, output, and the function of the black box is known.

These diagrams are used at a very high level of design. This high level design is refined again and again to get a complete and practical picture of the system. A well-structured use case also describes the pre-condition, post condition, and exceptions. These extra elements are used to make test cases when performing the testing.

Although use case is not a good candidate for forward and reverse engineering, still they are used in a slightly different way to make forward and reverse engineering. The same is true for reverse engineering. Use case diagram is used differently to make it suitable for reverse engineering.

In forward engineering, use case diagrams are used to make test cases and in reverse engineering use cases are used to prepare the requirement details from the existing application.

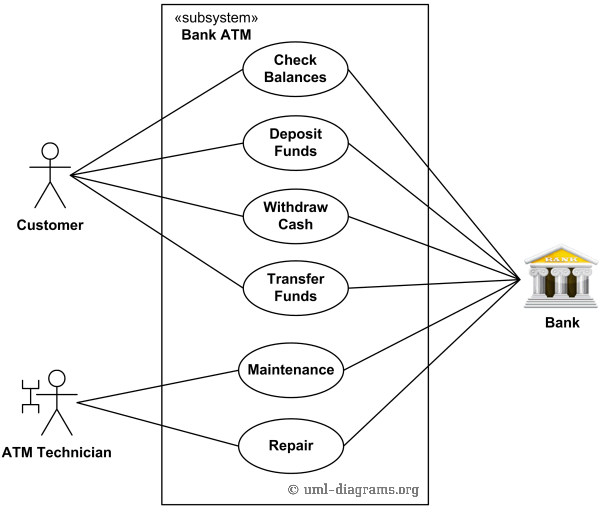
**Use case diagrams can be used for −**

* Requirement analysis and high level design.
* Model the context of a system.
* Reverse engineering.
* Forward engineering.

**Bank ATM *UML Use Case Diagram Examples***

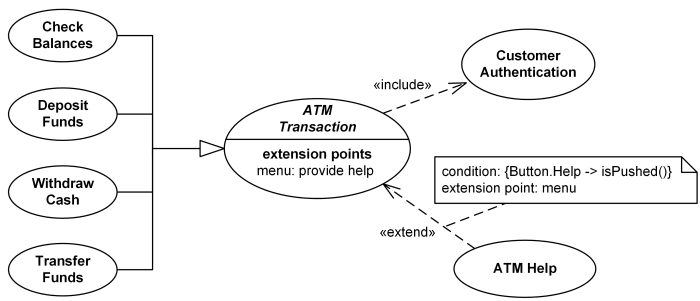
An automated teller machine (**ATM**) or the automatic banking machine (**ABM**) is a banking subsystem ([**subject**](https://www.uml-diagrams.org/use-case-subject.html)) that provides bank customers with access to financial transactions in a public space without the need for a cashier, clerk, or bank teller.

*Customer* ([**actor**](https://www.uml-diagrams.org/use-case-actor.html)) uses bank ATM to *Check Balances* of his/her bank accounts, *Deposit Funds*, *Withdraw Cash* and/or *Transfer Funds* ([**use cases**](https://www.uml-diagrams.org/use-case.html)). *ATM Technician* provides *Maintenance* and *Repairs*. All these use cases also involve *Bank* actor whether it is related to customer transactions or to the ATM servicing.



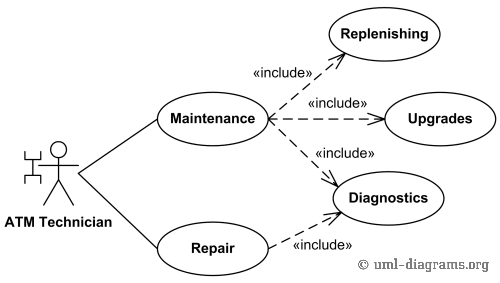
*An example of use case diagram for Bank ATM subsystem - top level use cases.*

On most bank ATMs, the customer is authenticated by inserting a plastic ATM card and entering a personal identification number (PIN). *Customer Authentication* use case is required for every ATM transaction so we show it as <<[**include**](https://www.uml-diagrams.org/use-case-include.html)>> relationship. Including this use case as well as transaction [**generalizations**](https://www.uml-diagrams.org/use-case.html#generalization) make the *ATM Transaction* an [**abstract use case**](https://www.uml-diagrams.org/use-case.html#abstract-use-case).



*Bank ATM Transactions and Customer Authentication Use Cases Example.*

Customer may need some help from the ATM. *ATM Transaction* use case is <<[**extended**](https://www.uml-diagrams.org/use-case-extend.html)>> via [**extension point**](https://www.uml-diagrams.org/use-case-extend.html#extension-point) called *menu* by the *ATM Help* use case whenever *ATM Transaction* is at the location specified by the *menu* and the bank customer requests help, e.g. by selecting Help menu item.

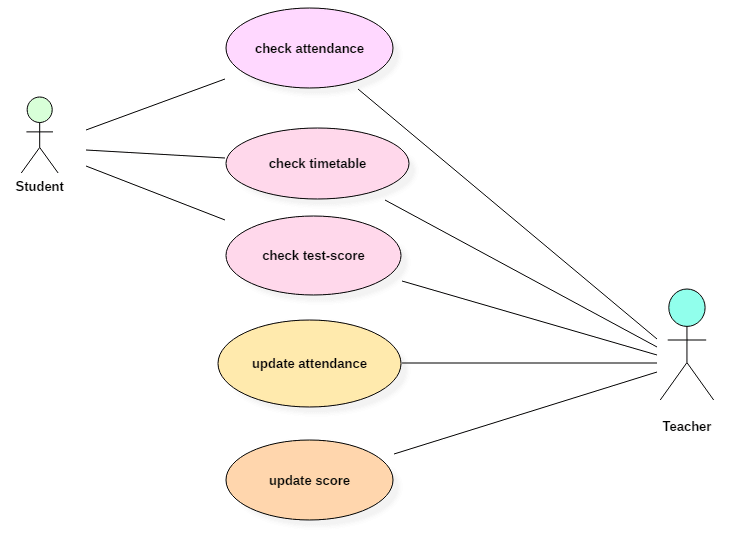


*Bank ATM Maintenance, Repair, Diagnostics Use Cases Example.*

*ATM Technician* maintains or repairs Bank ATM. *Maintenance* use case includes *Replenishing* ATM with cash, ink or printer paper, *Upgrades* of hardware, firmware or software, and remote or on-site *Diagnostics*.

**Another example of a use-case diagram**

Following use case diagram represents the working of the student management system:

[](https://www.guru99.com/images/1/052919_0831_UMLUseCaseD3.png)

UML Use Case Diagram

In the above use case diagram, there are two actors named student and a teacher. There are a total of five use cases that represent the specific functionality of a student management system. Each actor interacts with a particular use case. A student actor can check attendance, timetable as well as test marks on the application or a system. This actor can perform only these interactions with the system even though other use cases are remaining in the system.

It is not necessary that each actor should interact with all the use cases, but it can happen.

The second actor named teacher can interact with all the functionalities or use cases of the system. This actor can also update the attendance of a student and marks of the student. These interactions of both student and teacher actor together sums up the entire student management application.

**When to use a use-case diagram?**

A use case is a unique functionality of a system which is accomplished by a user. A purpose of use case diagram is to capture core functionalities of a system and visualize the interactions of various things called as actors with the use case. This is the general use of a use case diagram.

The use case diagrams represent the core parts of a system and the workflow between them. In use case, implementation details are hidden from the external use only the event flow is represented.

With the help of use case diagrams, we can find out pre and post conditions after the interaction with the actor. These conditions can be determined using various test cases.

In general use case diagrams are used for:

1. Analyzing the requirements of a system
2. High-level visual software designing
3. Capturing the functionalities of a system
4. Modeling the basic idea behind the system
5. Forward and reverse engineering of a system using various test cases.

Use cases are intended to convey desired functionality so the exact scope of a use case may vary according to the system and the purpose of creating UML model.

**Summary**

* Use case diagrams are a way to capture the system's functionality and requirements in UML diagrams.
* It captures the dynamic behavior of a live system.
* A use case diagram consists of a use case and an actor.
* A use case represents a distinct functionality of a system, a component, a package, or a class.
* An actor is an entity that initiates the use case from outside the scope of a use case.
* The name of an actor or a use case must be meaningful and relevant to the system.
* A purpose of use case diagram is to capture the core functionalities of a system.
* **Activity Diagrams :-**

Activity diagram is another important diagram in UML to describe the dynamic aspects of the system.

Activity diagram is basically a flowchart to represent the flow from one activity to another activity. The activity can be described as an operation of the system.

The control flow is drawn from one operation to another. This flow can be sequential, branched, or concurrent. Activity diagrams deal with all type of flow control by using different elements such as fork, join, etc

**Purpose of Activity Diagrams**

The basic purposes of activity diagrams is similar to other four diagrams. It captures the dynamic behavior of the system. Other four diagrams are used to show the message flow from one object to another but activity diagram is used to show message flow from one activity to another.

Activity is a particular operation of the system. Activity diagrams are not only used for visualizing the dynamic nature of a system, but they are also used to construct the executable system by using forward and reverse engineering techniques. The only missing thing in the activity diagram is the message part.

It does not show any message flow from one activity to another. Activity diagram is sometimes considered as the flowchart. Although the diagrams look like a flowchart, they are not. It shows different flows such as parallel, branched, concurrent, and single.

The purpose of an activity diagram can be described as −

* Draw the activity flow of a system.
* Describe the sequence from one activity to another.
* Describe the parallel, branched and concurrent flow of the system.

**Why use Activity Diagram?**

* An event is created as an activity diagram encompassing a group of nodes associated with edges. To model the behavior of activities, they can be attached to any modeling element. It can model use cases, classes, interfaces, components, and collaborations.
* It mainly models processes and workflows. It envisions the dynamic behavior of the system as well as constructs a runnable system that incorporates forward and reverse engineering. It does not include the message part, which means message flow is not represented in an activity diagram.
* It is the same as that of a flowchart but not exactly a flowchart itself. It is used to depict the flow between several activities.

**How to Draw an Activity Diagram?**

Activity diagrams are mainly used as a flowchart that consists of activities performed by the system. Activity diagrams are not exactly flowcharts as they have some additional capabilities. These additional capabilities include branching, parallel flow, swimlane, etc.

Before drawing an activity diagram, we must have a clear understanding about the elements used in activity diagram. The main element of an activity diagram is the activity itself. An activity is a function performed by the system. After identifying the activities, we need to understand how they are associated with constraints and conditions.

Before drawing an activity diagram, we should identify the following elements −

* Activities
* Association
* Conditions
* Constraints

Once the above-mentioned parameters are identified, we need to make a mental layout of the entire flow. This mental layout is then transformed into an activity diagram.

Following are the rules that are to be followed for drawing an activity diagram:

1. A meaningful name should be given to each and every activity.
2. Identify all of the constraints.
3. Acknowledge the activity associations.

**Components of an Activity Diagram**

Following are the component of an activity diagram:

**Activities**

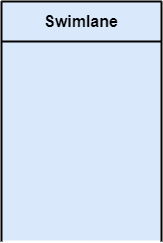
The categorization of behavior into one or more actions is termed as an activity. In other words, it can be said that an activity is a network of nodes that are connected by edges. The edges depict the flow of execution. It may contain action nodes, control nodes, or object nodes.

The control flow of activity is represented by control nodes and object nodes that illustrates the objects used within an activity. The activities are initiated at the initial node and are terminated at the final node.



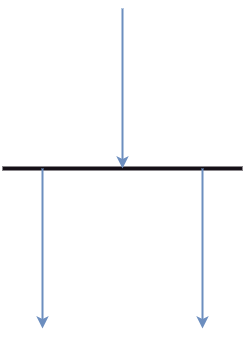
**Activity partition /swimlane**

The swimlane is used to cluster all the related activities in one column or one row. It can be either vertical or horizontal. It used to add modularity to the activity diagram. It is not necessary to incorporate swimlane in the activity diagram. But it is used to add more transparency to the activity diagram.



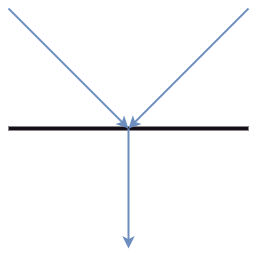
**Forks**

Forks and join nodes generate the concurrent flow inside the activity. A fork node consists of one inward edge and several outward edges. It is the same as that of various decision parameters. Whenever a data is received at an inward edge, it gets copied and split crossways various outward edges. It split a single inward flow into multiple parallel flows.



**Join Nodes**

Join nodes are the opposite of fork nodes. A Logical AND operation is performed on all of the inward edges as it synchronizes the flow of input across one single output (outward) edge.



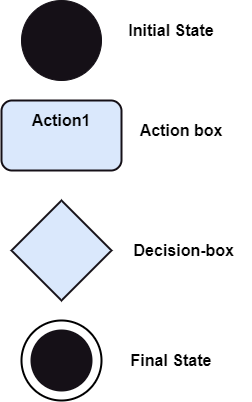
**Activity diagram constitutes following notations:**

**Initial State**: It depicts the initial stage or beginning of the set of actions.

**Final State**: It is the stage where all the control flows and object flows end.

**Decision Box**: It makes sure that the control flow or object flow will follow only one path.

**Action Box**: It represents the set of actions that are to be performed.



**Where to Use Activity Diagrams?**

The basic usage of activity diagram is similar to other four UML diagrams. The specific usage is to model the control flow from one activity to another. This control flow does not include messages.

Activity diagram is suitable for modeling the activity flow of the system. An application can have multiple systems. Activity diagram also captures these systems and describes the flow from one system to another. This specific usage is not available in other diagrams. These systems can be database, external queues, or any other system.

We will now look into the practical applications of the activity diagram. From the above discussion, it is clear that an activity diagram is drawn from a very high level. So it gives high level view of a system. This high level view is mainly for business users or any other person who is not a technical person.

This diagram is used to model the activities which are nothing but business requirements. The diagram has more impact on business understanding rather than on implementation details.

Activity diagram can be used for −

* Modeling work flow by using activities.
* Modeling business requirements.
* High level understanding of the system's functionalities.
* Investigating business requirements at a later stage.

**When to use an Activity Diagram?**

An activity diagram can be used to portray business processes and workflows. Also, it used for modeling business as well as the software. An activity diagram is utilized for the followings:

1. To graphically model the workflow in an easier and understandable way.
2. To model the execution flow among several activities.
3. To model comprehensive information of a function or an algorithm employed within the system.
4. To model the business process and its workflow.
5. To envision the dynamic aspect of a system.
6. To generate the top-level flowcharts for representing the workflow of an application.
7. To represent a high-level view of a distributed or an object-oriented system.

8. Model business processes and their workflows.

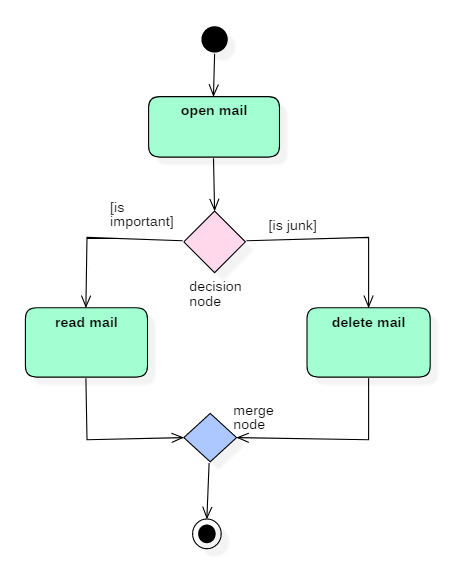
9. Capture the dynamic behavior of a system.

10. Generate high-level flowcharts to represent the workflow of any application.

11.Model high-level view of an object-oriented or a distributed system.

**Example of Activity Diagram**

Let us consider mail processing activity as a sample for Activity Diagram. Following diagram represents activity for processing e-mails.

[](https://www.guru99.com/images/1/052919_1151_UMLActivity2.png)

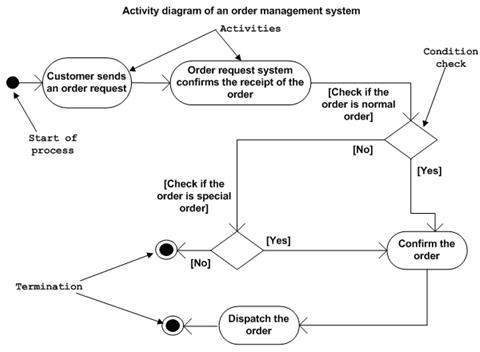
In the above activity diagram, three activities are specified. When the mail checking process begins user checks if mail is important or junk. Two guard conditions [is essential] and [is junk] decides the flow of execution of a process. After performing the activity, finally, the process is terminated at termination node.

**Following is an example** of an activity diagram for order management system. In the diagram, four activities are identified which are associated with conditions. One important point should be clearly understood that an activity diagram cannot be exactly matched with the code. The activity diagram is made to understand the flow of activities and is mainly used by the business users

Following diagram is drawn with the four main activities −

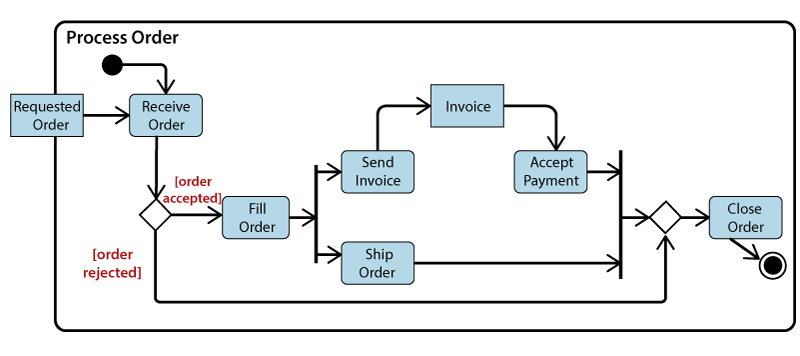
* Send order by the customer
* Receipt of the order
* Confirm the order
* Dispatch the order

After receiving the order request, condition checks are performed to check if it is normal or special order. After the type of order is identified, dispatch activity is performed and that is marked as the termination of the process.



An example of an activity diagram showing the business flow activity of order processing is given below.

Here the input parameter is the Requested order, and once the order is accepted, all of the required information is then filled, payment is also accepted, and then the order is shipped. It permits order shipment before an invoice is sent or payment is completed.



* **3.2.3 Sequence Diagram :-**

The sequence diagram represents the flow of messages in the system and is also termed as an event diagram. It helps in envisioning several dynamic scenarios. It portrays the communication between any two lifelines as a time-ordered sequence of events, such that these lifelines took part at the run time. In UML, the lifeline is represented by a vertical bar, whereas the message flow is represented by a vertical dotted line that extends across the bottom of the page. It incorporates the iterations as well as branching.

A **SEQUENCE DIAGRAM** simply depicts interaction between objects in a sequential order. The purpose of a sequence diagram in UML is to visualize the sequence of a message flow in the system. The sequence diagram shows the interaction between two lifelines as a time-ordered sequence of events.

* A sequence diagram shows an implementation of a scenario in the system. Lifelines in the system take part during the execution of a system.
* In a sequence diagram, a lifeline is represented by a vertical bar.
* A message flow between two or more objects is represented using a vertical dotted line which extends across the bottom of the page.
* In a sequence diagram, different types of messages and operators are used which are described above.
* In a sequence diagram, iteration and branching are also used.

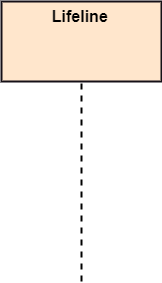
**Purpose of a Sequence Diagram:-**

1. To model high-level interaction among active objects within a system.
2. To model interaction among objects inside a collaboration realizing a use case.
3. It either models generic interactions or some certain instances of interaction.

**Notations of a Sequence Diagram**

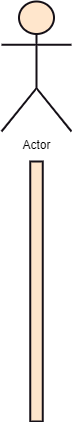
**Lifeline**

An individual participant in the sequence diagram is represented by a lifeline. It is positioned at the top of the diagram.



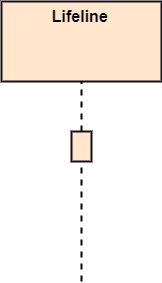
**Actor**

A role played by an entity that interacts with the subject is called as an actor. It is out of the scope of the system. It represents the role, which involves human users and external hardware or subjects. An actor may or may not represent a physical entity, but it purely depicts the role of an entity. Several distinct roles can be played by an actor or vice versa.



**Activation**

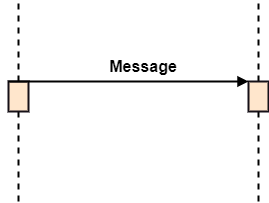
It is represented by a thin rectangle on the lifeline. It describes that time period in which an operation is performed by an element, such that the top and the bottom of the rectangle is associated with the initiation and the completion time, each respectively.

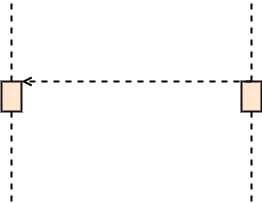


**Messages**

The messages depict the interaction between the objects and are represented by arrows. They are in the sequential order on the lifeline. The core of the sequence diagram is formed by messages and lifelines.

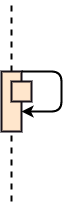
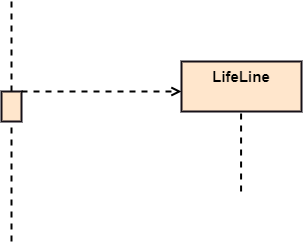
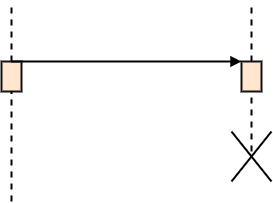
Following are types of messages enlisted below:

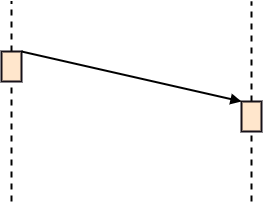
* **Call Message:** It defines a particular communication between the lifelines of an interaction, which represents that the target lifeline has invoked an operation.  
  
* **Return Message:** It defines a particular communication between the lifelines of interaction that represent the flow of information from the receiver of the corresponding caller message.



* **Self Message:** It describes a communication, particularly between the lifelines of an interaction that represents a message of the same lifeline, has been invoked.

Sequence Diagram

* **Recursive Message:** A self message sent for recursive purpose is called a recursive message. In other words, it can be said that the recursive message is a special case of the self message as it represents the recursive calls.  
  
* **Create Message:** It describes a communication, particularly between the lifelines of an interaction describing that the target (lifeline) has been instantiated.  
  
* **Destroy Message:** It describes a communication, particularly between the lifelines of an interaction that depicts a request to destroy the lifecycle of the target.  
  
* **Duration Message:** It describes a communication particularly between the lifelines of an interaction, which portrays the time passage of the message while modeling a system.



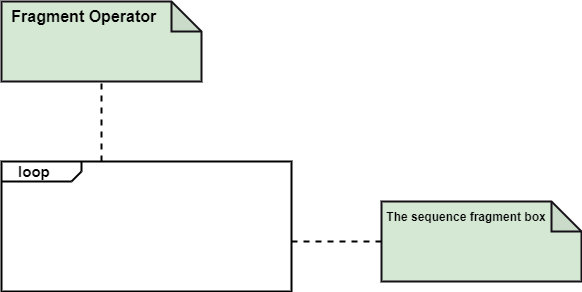
**Note**

A note is the capability of attaching several remarks to the element. It basically carries useful information for the modelers.



**Sequence Fragments**

1. Sequence fragments have been introduced by UML 2.0, which makes it quite easy for the creation and maintenance of an accurate sequence diagram.
2. It is represented by a box called a combined fragment, encloses a part of interaction inside a sequence diagram.
3. The type of fragment is shown by a fragment operator.

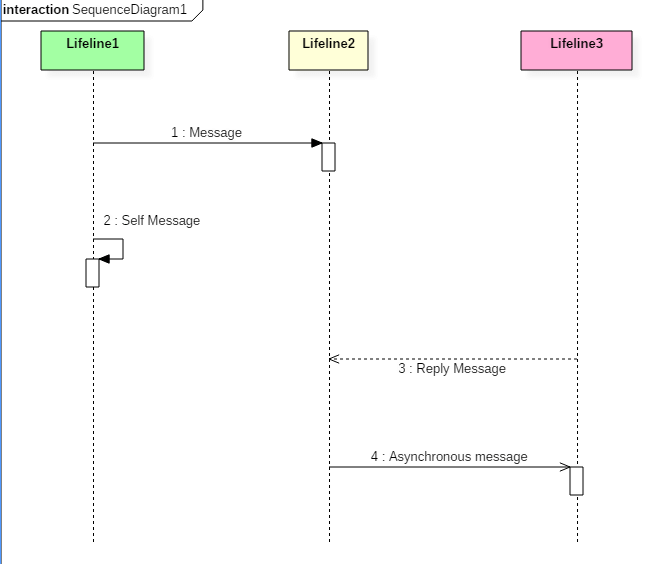


### Types of fragments

Following are the types of fragments enlisted below;

|  |  |
| --- | --- |
| **Operator** | **Fragment Type** |
| alt | Alternative multiple fragments: The only fragment for which the condition is true, will execute. |
| opt | Optional: If the supplied condition is true, only then the fragments will execute. It is similar to alt with only one trace. |
| par | Parallel: Parallel executes fragments. |
| loop | Loop: Fragments are run multiple times, and the basis of interaction is shown by the guard. |
| region | Critical region: Only one thread can execute a fragment at once. |
| neg | Negative: A worthless communication is shown by the fragment. |
| ref | Reference: An interaction portrayed in another diagram. In this, a frame is drawn so as to cover the lifelines involved in the communication. The parameter and return value can be explained. |
| sd | Sequence Diagram: It is used to surround the whole sequence diagram. |

**Example :-**

[](https://www.guru99.com/images/1/062819_0838_Interaction2.png)

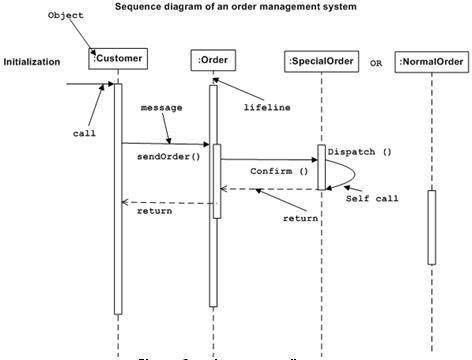
The above sequence diagram contains lifeline notations and notation of various messages used in a sequence diagram such as a create, reply, asynchronous message, etc.

**Example 1:-**

The sequence diagram has four objects (Customer, Order, SpecialOrder and NormalOrder).

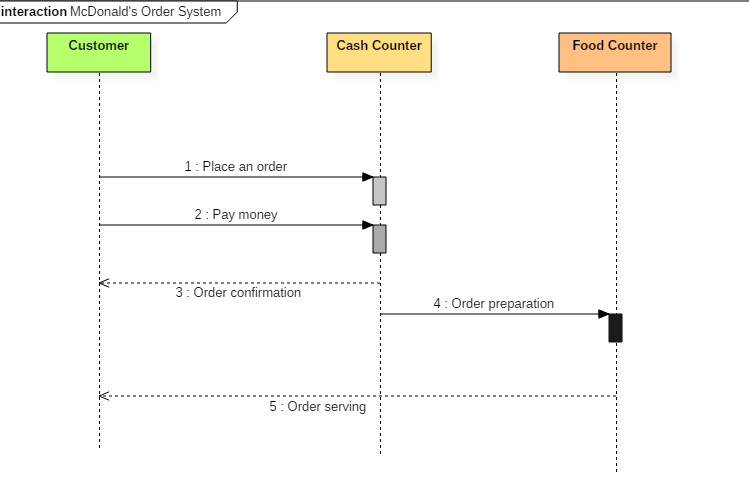
The following diagram shows the message sequence for *SpecialOrder* object and the same can be used in case of *NormalOrder* object. It is important to understand the time sequence of message flows. The message flow is nothing but a method call of an object.

The first call is *sendOrder ()* which is a method of *Order object*. The next call is *confirm ()* which is a method of *SpecialOrder* object and the last call is *Dispatch ()* which is a method of *SpecialOrder* object. The following diagram mainly describes the method calls from one object to another, and this is also the actual scenario when the system is running.



**Sequence diagram Example 2:-**

The following sequence diagram example represents McDonald's ordering system:

[](https://www.guru99.com/images/1/062819_0838_Interaction3.png)

Sequence diagram of McDonald's ordering system

The ordered sequence of events in a given sequence diagram is as follows:

1. Place an order.
2. Pay money to the cash counter.
3. Order Confirmation.
4. Order preparation.
5. Order serving.

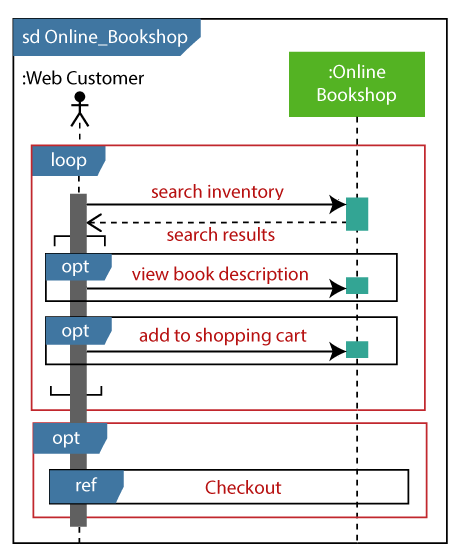
If one changes the order of the operations, then it may result in crashing the program. It can also lead to generating incorrect results. Each sequence in the above-given sequence diagram is denoted using a different type of message. One cannot use the same type of message to denote all the interactions in the diagram because it creates complications in the system.

You must be careful while selecting the notation of a message for any particular interaction. The notation must match with the particular sequence inside the diagram.

Sequence Diagram Example 3:-

An example of a high-level sequence diagram for online bookshop is given below.

Any online customer can search for a book catalog, view a description of a particular book, add a book to its shopping cart, and do checkout.



**Benefits of a Sequence Diagram**

* Sequence diagrams are used to explore any real application or a system.
* Sequence diagrams are used to represent message flow from one object to another object.
* Sequence diagrams are easier to maintain.
* Sequence diagrams are easier to generate.
* Sequence diagrams can be easily updated according to the changes within a system.
* Sequence diagram allows reverse as well as forward engineering.

**Drawbacks of a sequence diagram**

* Sequence diagrams can become complex when too many lifelines are involved in the system.
* If the order of message sequence is changed, then incorrect results are produced.
* Each sequence needs to be represented using different message notation, which can be a little complex.
* The type of message decides the type of sequence inside the diagram.

# 3.2.4 Collaboration Diagram

The collaboration diagram is used to show the relationship between the objects in a system. Both the sequence and the collaboration diagrams represent the same information but differently. Instead of showing the flow of messages, it depicts the architecture of the object residing in the system as it is based on object-oriented programming. An object consists of several features. Multiple objects present in the system are connected to each other. The collaboration diagram, which is also known as a communication diagram, is used to portray the object's architecture in the system.

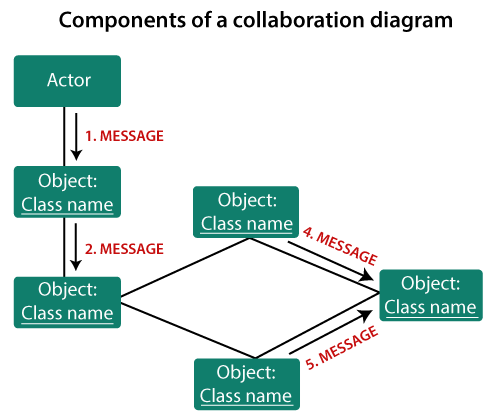
**COLLABORATION DIAGRAM** depicts the relationships and interactions among software objects. They are used to understand the object architecture within a system rather than the flow of a message as in a sequence diagram. They are also known as “**Communication Diagrams.**”

As per Object-Oriented Programming (OOPs), an object entity has various attributes associated with it. Usually, there are multiple objects present inside an object-oriented system where each object can be associated with any other object inside the system. Collaboration Diagrams are used to explore the architecture of objects inside the system. The message flow between the objects can be represented using a collaboration diagram.

**Notations of a Collaboration Diagram**

Following are the components of a component diagram that are enlisted below:

1. **Objects:** The representation of an object is done by an object symbol with its name and class underlined, separated by a colon.  
   In the collaboration diagram, objects are utilized in the following ways:
   * The object is represented by specifying their name and class.
   * It is not mandatory for every class to appear.
   * A class may constitute more than one object.
   * In the collaboration diagram, firstly, the object is created, and then its class is specified.
   * To differentiate one object from another object, it is necessary to name them.
2. **Actors:** In the collaboration diagram, the actor plays the main role as it invokes the interaction. Each actor has its respective role and name. In this, one actor initiates the use case.
3. **Links:** The link is an instance of association, which associates the objects and actors. It portrays a relationship between the objects through which the messages are sent. It is represented by a solid line. The link helps an object to connect with or navigate to another object, such that the message flows are attached to links.
4. **Messages:** It is a communication between objects which carries information and includes a sequence number, so that the activity may take place. It is represented by a labeled arrow, which is placed near a link. The messages are sent from the sender to the receiver, and the direction must be navigable in that particular direction. The receiver must understand the message.



**When to use a Collaboration Diagram?**

The collaborations are used when it is essential to depict the relationship between the object. Both the sequence and collaboration diagrams represent the same information, but the way of portraying it quite different. The collaboration diagrams are best suited for analyzing use cases.

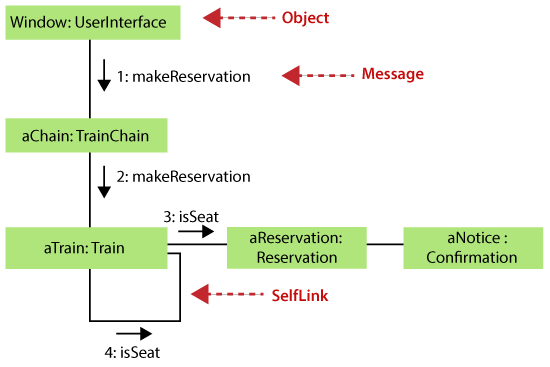
Following are some of the use cases enlisted below for which the collaboration diagram is implemented:

1. To model collaboration among the objects or roles that carry the functionalities of use cases and operations.
2. To model the mechanism inside the architectural design of the system.
3. To capture the interactions that represent the flow of messages between the objects and the roles inside the collaboration.
4. To model different scenarios within the use case or operation, involving a collaboration of several objects and interactions.
5. To support the identification of objects participating in the use case.
6. In the collaboration diagram, each message constitutes a sequence number, such that the top-level message is marked as one and so on. The messages sent during the same call are denoted with the same decimal prefix, but with different suffixes of 1, 2, etc. as per their occurrence.

**Steps for creating a Collaboration Diagram**

1. Determine the behavior for which the realization and implementation are specified.
2. Discover the structural elements that are class roles, objects, and subsystems for performing the functionality of collaboration.
   * Choose the context of an interaction: system, subsystem, use case, and operation.
3. Think through alternative situations that may be involved.
   * Implementation of a collaboration diagram at an instance level, if needed.
   * A specification level diagram may be made in the instance level sequence diagram for summarizing alternative situations.

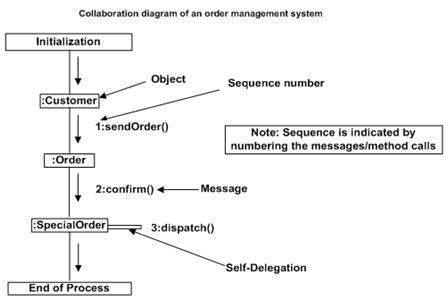
**Example of a Collaboration Diagram**



It shows the object organization as seen in the following diagram. In the collaboration diagram, the method call sequence is indicated by some numbering technique. The number indicates how the methods are called one after another. We have taken the same order management system to describe the collaboration diagram.

Method calls are similar to that of a sequence diagram. However, difference being the sequence diagram does not describe the object organization, whereas the collaboration diagram shows the object organization.

To choose between these two diagrams, emphasis is placed on the type of requirement. If the time sequence is important, then the sequence diagram is used. If organization is required, then collaboration diagram is used.



Following diagram represents the sequencing over student management system:

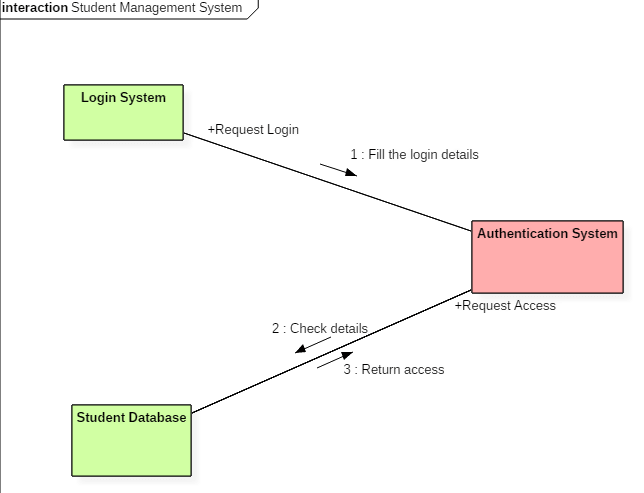
[](https://www.guru99.com/images/1/062819_0838_Interaction5.png)

Fig:- Collaboration diagram for student management system

The above collaboration diagram represents a student information management system. The flow of communication in the above diagram is given by,

1. A student requests a login through the login system.
2. An authentication mechanism of software checks the request.
3. If a student entry exists in the database, then the access is allowed; otherwise, an error is returned.

**Benefits of a Collaboration Diagram :-**

1. The collaboration diagram is also known as Communication Diagram.
2. It mainly puts emphasis on the structural aspect of an interaction diagram, i.e., how lifelines are connected.
3. The syntax of a collaboration diagram is similar to the sequence diagram; just the difference is that the lifeline does not consist of tails.
4. The messages transmitted over sequencing is represented by numbering each individual message.
5. The collaboration diagram is semantically weak in comparison to the sequence diagram.
6. The special case of a collaboration diagram is the object diagram.
7. It focuses on the elements and not the message flow, like sequence diagrams.
8. Since the collaboration diagrams are not that expensive, the sequence diagram can be directly converted to the collaboration diagram.
9. There may be a chance of losing some amount of information while implementing a collaboration diagram with respect to the sequence diagram.

**The drawback of a Collaboration Diagram:-**

1. Multiple objects residing in the system can make a complex collaboration diagram, as it becomes quite hard to explore the objects.
2. It is a time-consuming diagram.
3. After the program terminates, the object is destroyed.
4. As the object state changes momentarily, it becomes difficult to keep an eye on every single that has occurred inside the object of a system.

* **3.2.5 State Transition Diagram :-**

The name of the diagram itself clarifies the purpose of the diagram and other details. **It describes different states of a component in a system. The states are specific to a component/object of a system**.

**A Statechart diagram describes a state machine. State machine can be defined as a machine which defines different states of an object and these states are controlled by external or internal events.**

**State Diagram** are used to capture the behavior of a software system. UML State machine diagrams can be used to model the behavior of a class, a subsystem, a package, or even an entire system. **It is also called a Statechart or State Transition diagram.**

**Statechart diagrams provide us an efficient way to model the interactions or communications that occur within the external entities and a system.** These diagrams are used to model the event-based system. A state of an object is controlled with the help of an event.

Statechart diagrams are used to describe various states of an entity within the application system.

The state machine diagram is also called the Statechart or State Transition diagram, which shows the order of states underwent by an object within the system. **It captures the software system's behavior. It models the behavior of a class, a subsystem, a package, and a complete system.**

It tends out to be an efficient way of modeling the interactions and collaborations in the external entities and the system. It models event-based systems to handle the state of an object. It also defines several distinct states of a component within the system. Each object/component has a specific state.

**Purpose of Statechart Diagrams :-**

Statechart diagram is one of the five UML diagrams **used to model the dynamic nature of a system.** They define different states of an object during its lifetime and these states are changed by events. **Statechart diagrams are useful to model the reactive systems. Reactive systems can be defined as a system that responds to external or internal events.**

Statechart diagram describes the flow of control from one state to another state. States are defined as a condition in which an object exists and it changes when some event is triggered. **The most important purpose of Statechart diagram is to model lifetime of an object from creation to termination.**

Statechart diagrams are also used for forward and reverse engineering of a system. However, the main purpose is to model the reactive system.

Following are the main purposes of using Statechart diagrams −

* To model the dynamic aspect of a system.
* To model the life time of a reactive system.
* To describe different states of an object during its life time.
* Define a state machine to model the states of an object.

**Following are the types of a state machine diagram that are given below:-**

1. **Behavioral state machine**The behavioral state machine diagram records the behavior of an object within the system. It depicts an implementation of a particular entity. It models the behavior of the system.
2. **Protocol state machine**It captures the behavior of the protocol. The protocol state machine depicts the change in the state of the protocol and parallel changes within the system. But it does not portray the implementation of a particular component. (Protocol = Procedure/etiquette / code of behavior /set of Rules)

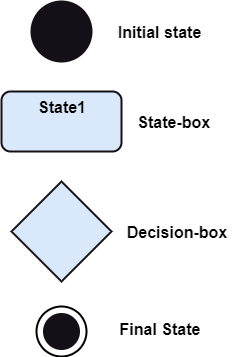
**Why State Machine Diagram?**

Since it records the dynamic view of a system, it portrays the behavior of a software application. During a lifespan, an object underwent several states, such that the lifespan exist until the program is executing. Each state depicts some useful information about the object.

It blueprints an interactive system that response back to either the internal events or the external ones. The execution flow from one state to another is represented by a state machine diagram. It visualizes an object state from its creation to its termination.

The main purpose is to depict each state of an individual object. It represents an interactive system and the entities inside the system. It records the dynamic behavior of the system.

**Following are the Notations of a state machine diagram enlisted below:**



**Initial state:** It defines the initial state (beginning) of a system, and it is represented by a black filled circle.

**Final state:** It represents the final state (end) of a system. It is denoted by a filled circle present within a circle.

**Decision box:** It is of diamond shape that represents the decisions to be made on the basis of an evaluated guard.

**Transition:** A change of control from one state to another due to the occurrence of some event is termed as a transition. It is represented by an arrow labeled with an event due to which the change has ensued.

**State box:** It depicts the conditions or circumstances of a particular object of a class at a specific point of time. A rectangle with round corners is used to represent the state box.

**Types of State**

The UML consist of three states:

1. **Simple state:** It does not constitute any substructure.
2. **Composite state:** It consists of nested states (sub-states), such that it does not contain more than one initial state and one final state. It can be nested to any level.
3. **Submachine state:** The submachine state is semantically identical to the composite state, but it can be reused.

**When to use a State Machine Diagram?**

The state machine diagram implements the real-world models as well as the object-oriented systems. It records the dynamic behavior of the system, which is used to differentiate between the dynamic and static behavior of a system.

It portrays the changes underwent by an object from the start to the end. It basically envisions how triggering an event can cause a change within the system.

State machine diagram is used for:

1. For modeling the object states of a system.
2. For modeling the reactive system as it consists of reactive objects.
3. For pinpointing the events responsible for state transitions.
4. For implementing forward and reverse engineering.

**How to Draw a Statechart Diagram?**

Statechart diagram is used to describe the states of different objects in its life cycle. Emphasis is placed on the state changes upon some internal or external events. These states of objects are important to analyze and implement them accurately.

Statechart diagrams are very important for describing the states. States can be identified as the condition of objects when a particular event occurs.

Before drawing a Statechart diagram we should clarify the following points −

* Identify the important objects to be analyzed.
* Identify the states.
* Identify the events.

The state machine diagram is used to portray various states underwent by an object. The change in one state to another is due to the occurrence of some event. All of the possible states of a particular component must be identified before drawing a state machine diagram.

The primary focus of the state machine diagram is to depict the states of a system. These states are essential while drawing a state transition diagram. The objects, states, and events due to which the state transition occurs must be acknowledged before the implementation of a state machine diagram.

Following are the steps that are to be incorporated while drawing a state machine diagram:

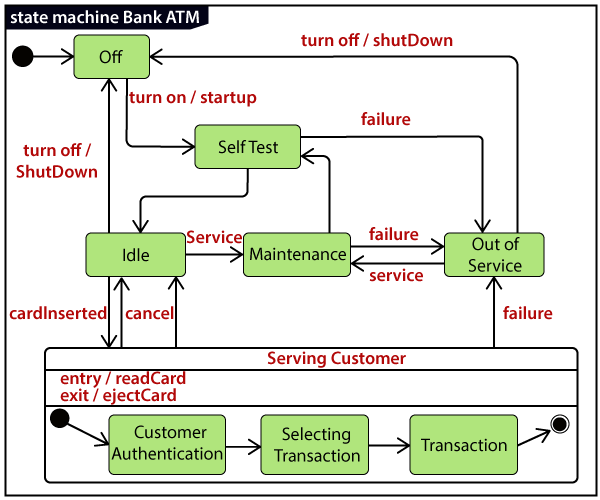
1. A unique and understandable name should be assigned to the state transition that describes the behavior of the system.
2. Out of multiple objects, only the essential objects are implemented.
3. A proper name should be given to the events and the transitions.

**Example of a State Machine Diagram**

An example of a top-level state machine diagram showing Bank Automated Teller Machine (ATM) is given below.

Initially, the ATM is turned off. After the power supply is turned on, the ATM starts performing the startup action and enters into the **Self Test**state. If the test fails, the ATM will enter into the **Out Of** **Service** state, or it will undergo **a triggerless transition** to the **Idle** state. This is the state where the customer waits for the interaction.

Whenever the customer inserts the bank or Credit/Debit card in the ATM's card reader, the ATM state changes from **Idle** to **Serving Customer**, the entry action **readCard** is performed after entering into **Serving Customer** state. Since the customer can cancel the transaction at any instant, so the transition from **Serving Customer** state back to the **Idle** state could be triggered by **cancel** event.



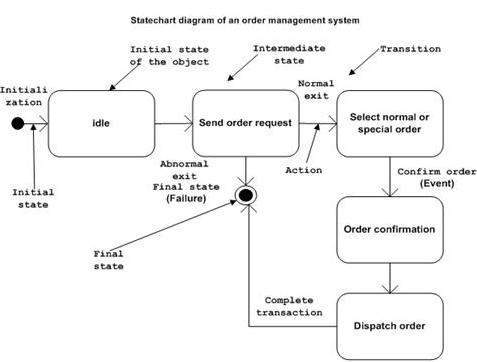
Here the **Serving Customer** is a composite state with sequential substates that are **Customer Authentication, Selecting Transaction,**and **Transaction.**

**Customer Authentication** and **Transaction** are the composite states itself is displayed by a hidden decomposition indication icon. After the transaction is finished, the **Serving Customer** encompasses a triggerless transition back to the **Idle** state. On leaving the state, it undergoes the exit action **ejectCard** that discharges the customer card.

**Following is an example of a Statechart diagram where the state of Order object is analyzed:-**

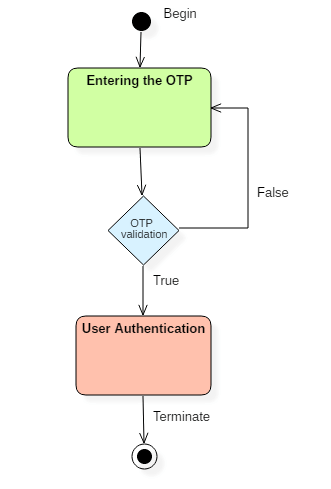
The first state is an idle state from where the process starts. The next states are arrived for events like send request, confirm request, and dispatch order. These events are responsible for the state changes of order object.

During the life cycle of an object (here order object) it goes through the following states and there may be some abnormal exits. This abnormal exit may occur due to some problem in the system. When the entire life cycle is complete, it is considered as a complete transaction as shown in the following figure. The initial and final state of an object is also shown in the following figure.



**Example of State Machine**

Following state diagram example chart represents the user authentication process.

[](https://www.guru99.com/images/1/052919_0832_StateMachin2.png)

UML state diagram

There are a total of two states, and the first state indicates that the OTP has to be entered first. After that, OTP is checked in the decision box, if it is correct, then only state transition will occur, and the user will be validated. If OTP is incorrect, then the transition will not take place, and it will again go back to the beginning state until the user enters the correct OTP as shown in the above state machine diagram example.

State Machine vs. Flowchart

|  |  |
| --- | --- |
| **State Machine** | **Flowchart** |
| It portrays several states of a system. | It demonstrates the execution flow of a program. |
| It encompasses the concept of WAIT, i.e., wait for an event or an action. | It does not constitute the concept of WAIT. |
| It is for real-world modeling systems. | It envisions the branching sequence of a system. |
| It is a modeling diagram. | It is a data flow diagram (DFD) |
| It is concerned with several states of a system. | It focuses on control flow and path. |

**Case Study1**

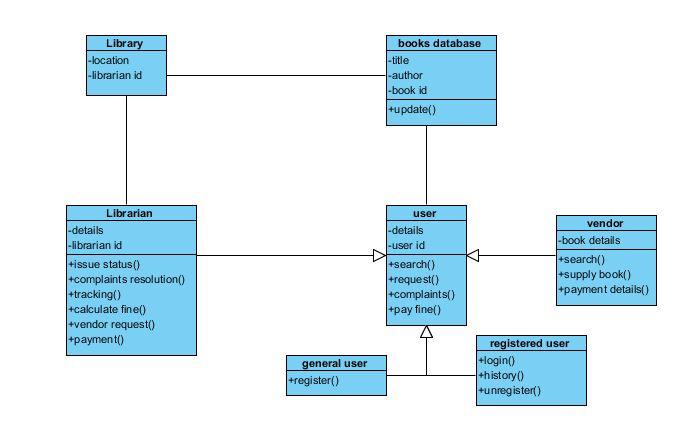
Library Management System

Problem Statement:

The case study titled Library Management System is library management software for the purpose of monitoring and controlling the transactions in a library. This case study on the library management system gives us the complete information about the library and the daily transactions done in a Library. We need to maintain the record of new s and retrieve the details of books available in the library which mainly focuses on basic operations in a library like adding new member, new books, and up new information, searching books and members and facility to borrow and return books. It features a familiar and well thought-out, an attractive user interface, combined with strong searching, insertion and reporting capabilities. The report generation facility of library system helps to get a good idea of which are ths borrowed by the members, makes users possible to generate hard copy.  
  
The following are the brief description on the functions achieved through this case study:  
  
End-Users:  
•Librarian: To maintain and update the records and also to cater the needs of the users.  
•Reader: Need books to read and also places various requests to the librarian.  
•Vendor: To provide and meet the requirement of the prescribed books.  
  
  
Class Diagram

Classes identified:

Library  
Librarian  
Books Database

User  
Vendor  
  
  
  
Use-case Diagram

Actors vs Use Cases:

Librarian:-   
•Issue a book

•Update and maintain records

•Request the vendor for a book

•Track complaints

User:-  
•Register  
•Login  
•Search a book

•Request for issue

•View history

•Request to the Librarian

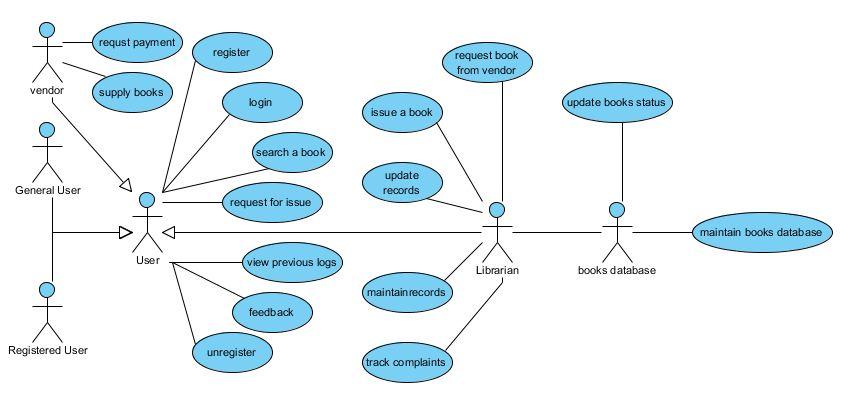
•Unregister  
  
Books Database :-

•Update records

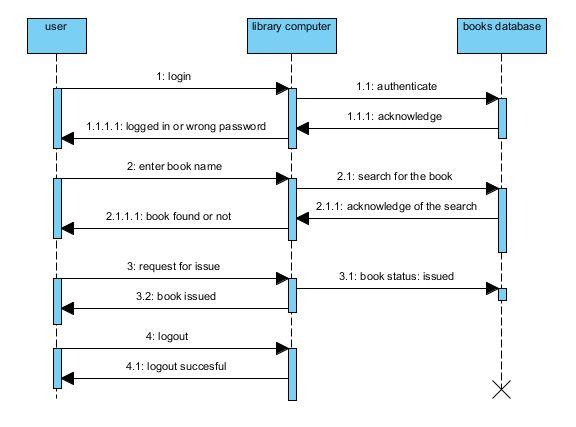
•Show books status

Vendors  
•Provide books to the library

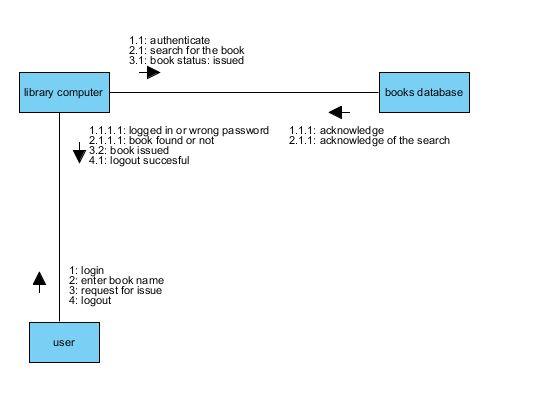
•Payment acknowledgement

  
  
Sequence Diagram

Sequence diagram for searching a book and issuing it as per the request by the user from the librarian:

  
  
Collaboration Diagram

Collaboration Diagram for searching a book and issuing it as per the request by the user from the librarian:

  
  
Activity Diagram

Activities:  
  
User Login and Authentication

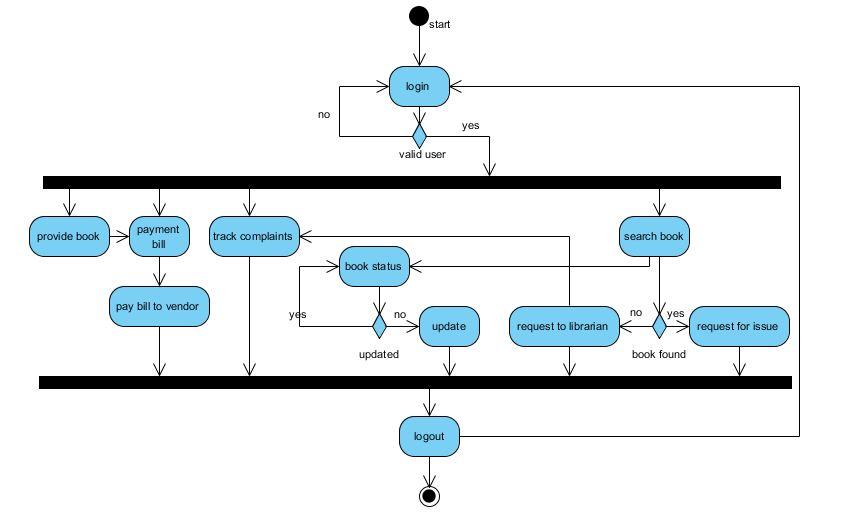
Search book operation for Reader

Acknowledge and Issue books to the users by the Librarian

Provide books requested by the Librarian from the Vendor

Bill payment from the Librarian to the Vendor

Status of the books updated in the Books Database

  
  
State Chart Diagram

States:  
  
Authentication  
Successfully logged on or re-login

Search for a book (user) / request the vendor (librarian) / provide the requested

book (vendor)

Receive acknowledgement

Logged off / re-search / new function

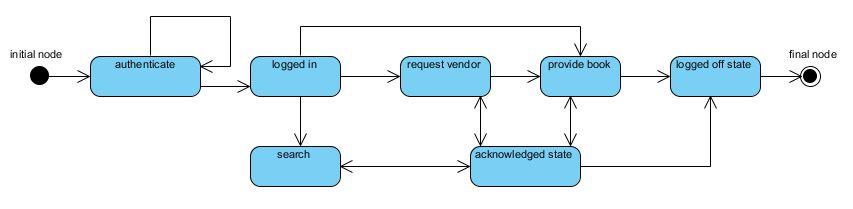
Transitions:  
  
Authenticate ---> Logged in

Logged in ---> Search <---> Acknowledgement

Logged in ---> Request Vendor <---> Provide Book <---> Acknowledgement

Logged in ---> Provide Book <---> Acknowledgement

Acknowledgement ---> Logged off

  
  
Component Diagram

Components:  
  
Register Page (visitor / vendor)

Login Page (user / librarian / vendor)

Search Page (user / librarian / vendor)

Request Vendor Page (librarian)

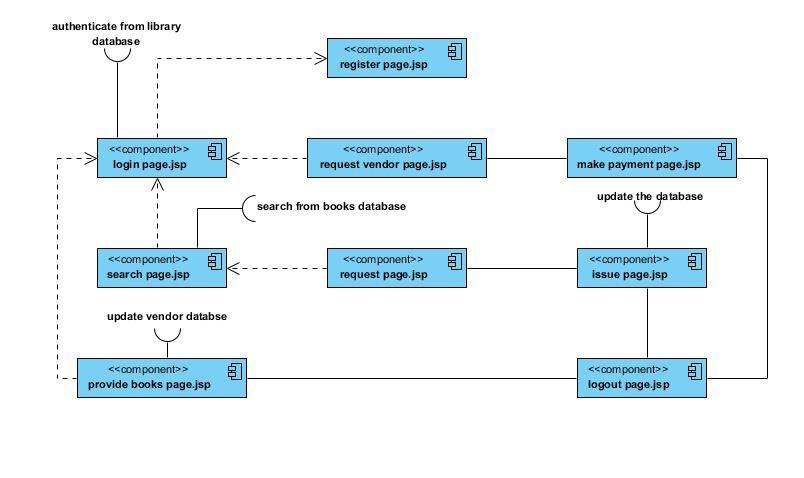
Request Book Issue Page (user / vendor)

Issue Status Page (librarian)

Make Payment Page (librarian / vendor)

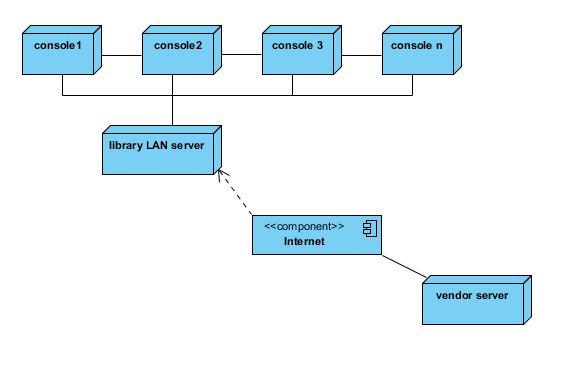
Provide Books Page (librarian)

Logout Page (user / librarian / vendor)

  
  
Deployment Diagram

Systems Used:

Local Consoles / Computers for login and search purposes by users, librarian and vendors.  
Library LAN Server interconnecting all the systems to the Database.  
Internet to provide access to Vendors to supply the requested books by the Librarian.  
Vendor Server to maintain the records of the requests made by the librarian and books provided to the library.



**Case Study2**

Online Mobile Recharge

Problem Statement:

The case study 'Online Mobile Recharge' gives us the information about all the mobile service providers. This application provides us the complete information regarding any mobile service provider in terms of their plans, options, benefits, etc. Suppose, any Airtel customer wants to have the information of all the schemes and services provided by the company, he/she can have the information and according to his convenience he can recharge the mobile from the same application. The major advantage of this proposed system is to have the recharging facility of any service provider under same roof.

End users:

Service Provider:

Service Provider is the one who is nothing but the mobile service provider like all the companies who are giving the mobile connections come under this module. Functionality of this module is to make the mobile recharging of their company basing on the availability of balance in the admin account. Request comes from the user and it is going to be verified at the admin for the availability of balance and then the request is forwarded to the service provided to make the mobile recharge.  
  
Third party System Administrator:

Administrator is the one who monitors all users and user transactions. Admin also monitors all the Service Providers, all the user accounts, and amounts paid by the user and amounts paid to Service providers. When the request given by the user admin checks the available balance in the user account then request is forwarded to the Service Provider from there user request gets processed. Admin haves the complete information related to user and all the information related to the schemes and other information of different recharge coupons provided by the Service Providers. All the data is maintained at the Admin level. Admin is having the rights to restrict any user.

User:  
There are 2 categories in the user Module:

•Registered User and

•Visitor  
  
Any person who wants to utilize the services of Online Mobile Recharge at any time from any where they should get registered in this application. After getting registered user can recharge the mobile at any time and from any where. Visitor is the one who visits the Online Mobile Recharge application and have the complete information related to the Service Providers and can make the mobile recharge by entering the bank details or by giving the credit card details.  
  
Class Diagram

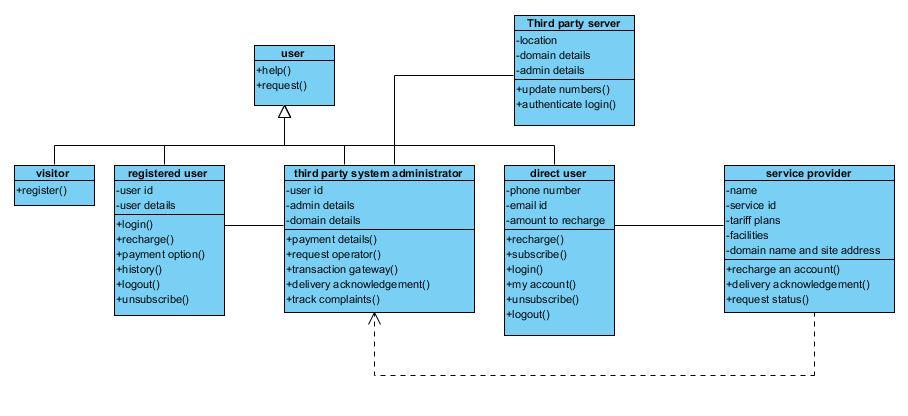
Classes Identified:

User: Registered , Visitor

Third Party System Administrator

Third Party Server/ Database

Service Provider

Direct or Non- Third Party User (Direct access through Service Provider Site)  
  
  
  
Use-Case Diagram

Actors vs Use Cases:

User:-  
•Register.  
•Recharge.  
•Select Payment Gateway.

•Select service Provider.

•Make payment.

Third Party Administrator:-

•Forward User request to Service Provider.

•Track Complaints.

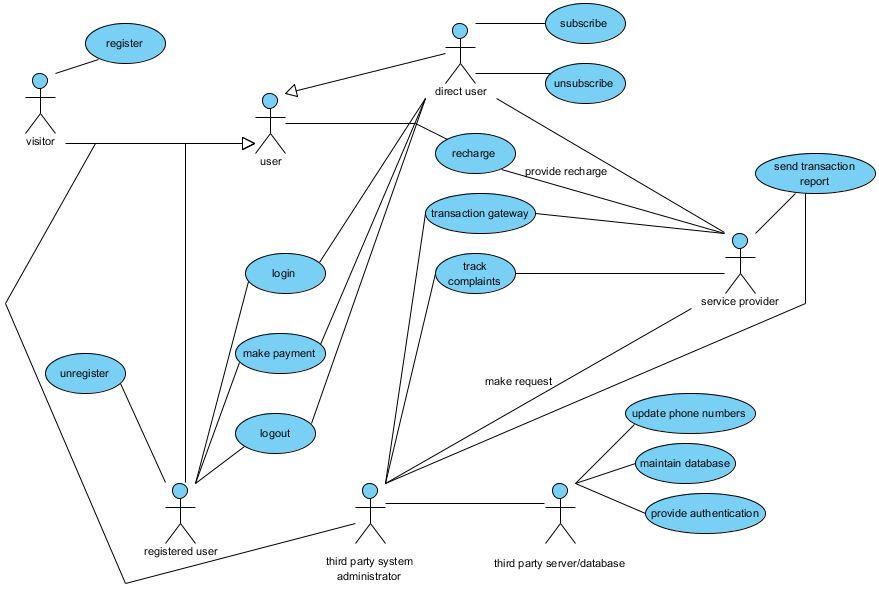
Third Party Server/ Database:-

•Authenticate the Registered users.

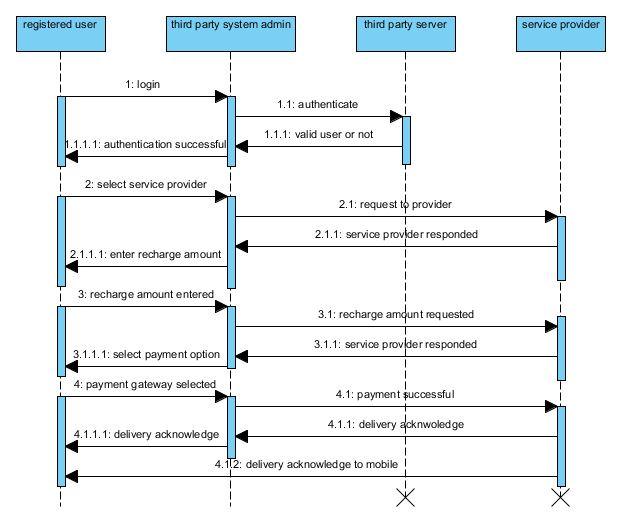
•Maintain the Log.

Service Provider:-

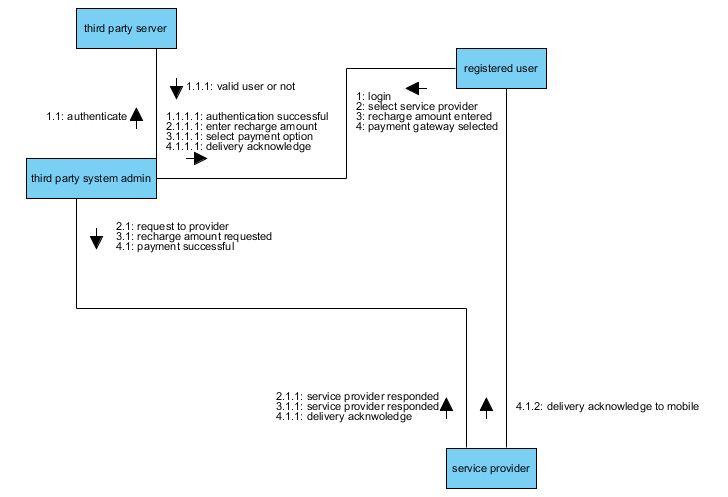
•Recharge the user requested either directly or through the third party system.  
•Provide various plans to the user.



Sequence Diagram

Sequence Diagram for a user to recharge his account through third party site:  
  


Collaboration Diagram

Collaboration diagram for a user to recharge his account through third arty site:  
  
  
  
Activity Diagram

Activities:  
  
User login and authentication for Registered user.

Forward the request to service provider if logged in as a Administrator.  
Enter service provider site for a direct user.

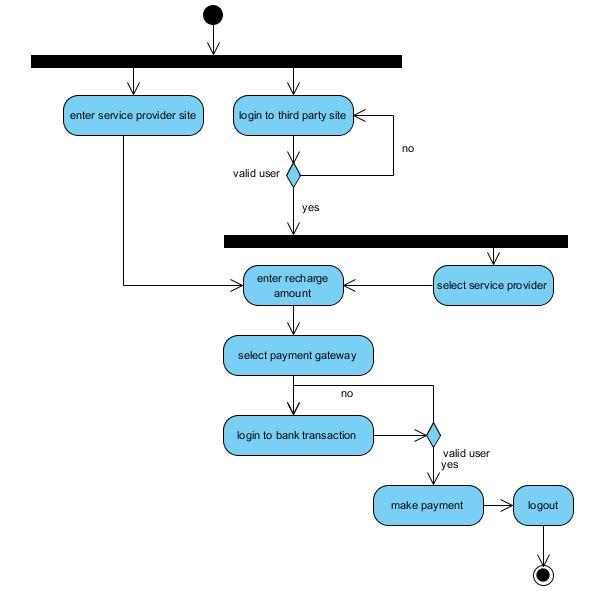
Enter recharge amount.

Select Payment Gateway.

Login and authenticate Bank Account.

Make payment.

Check for the recharge processed successfully or not.



State Chart Diagram

States:  
  
Authentication for registered users / Registration for unregistered users  
Successfully logged on or re-login

Operator Selection

Show the tariff plans available and applicable

Request recharge

Go through Payment Gateway Transaction process

•Authentication to enter the gateway site

•Successfully logged on or re-login

•Payment made

Logged off.

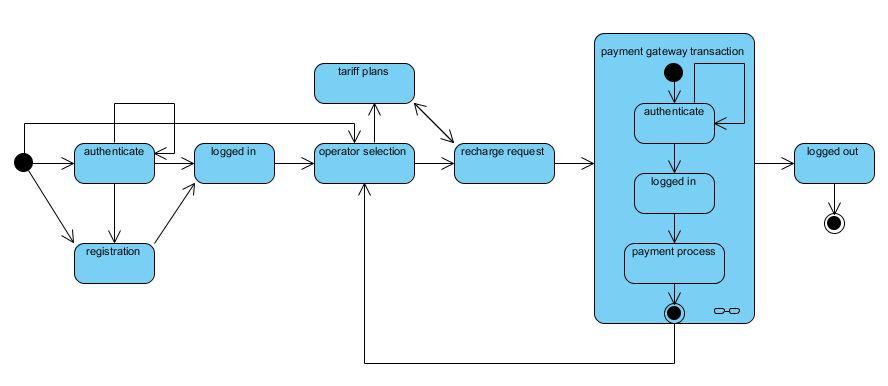
Transitions:  
  
Registration ---> Authenticate ---> Logged in

Logged in ---> Operator Selection ---> Tariff Plans <---> Request Recharge

Operator Selection ---> Request Recharge ---> Payment Gateway Transaction

Payment Gateway Transaction ---> Operator Selection

Payment Gateway Transaction ---> Logged off

  
  
Component Diagram

Components:  
  
Third Party Home Page (visitor / registered user / admin / service provider)

Third Party Register Page (visitor)

Third Party Login Page (registered user)

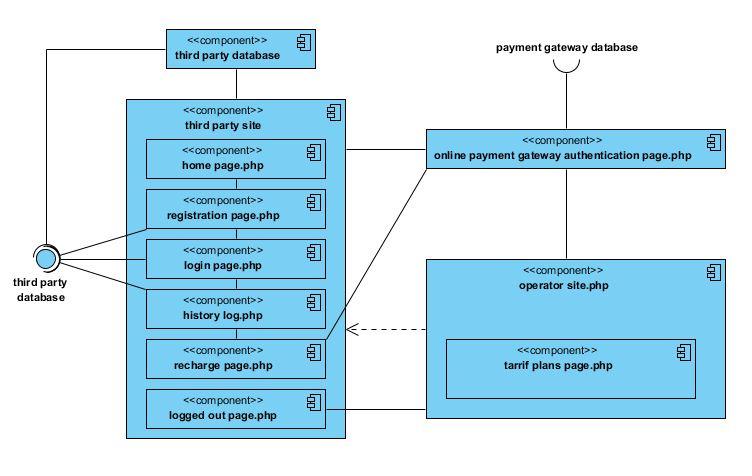
Third Party User History Page (registered user)

Request Recharge Page (registered user)

Third Party Logout Page (registered user)

Online Payment Transaction Gateway Page (direct user / registered user)

Service Provider Home Page (visitor / registered user / admin / service provider)

Tariff Plans Page (visitor / registered user / admin / service provider)  
  
  
  
Deployment Diagram

Systems Used:

1)Consoles / Computers for registration, login purposes by third party users and for quick recharge by direct users.

2)Third Party Server to receive and respond to all the requests from various users.  
3)Internet to provide access to users to recharge their accounts through payment gateways by placing requests through Third Party Sites and Service Providers sites.  
4)Payment Gateway Server like Bank's server to provide online payment through their personal accounts to meet the requirements of the users.  
5)Service Provider Server to maintain the records of the requests made by the users.  
  
