Institut Universitaire et Stratégique de l'Estuaire

Estuary Academic and Strategic Institute (IUEs/Insam)

Sous la tutelleacadémique des Universités de Dschang et de Buéa.

LECTURE NOTES on Modeling In Information System Modeling

SOFTWARE ENGINNERING LEVEL II

Academic Year: 2024/2025

General Instructional Objectives

At the end of this course the student will be able to:

- Explain the term MODELING in software engineering is.
- Explain the various modeling techniques involved in software development
- Explain how to model a system using OOM tools

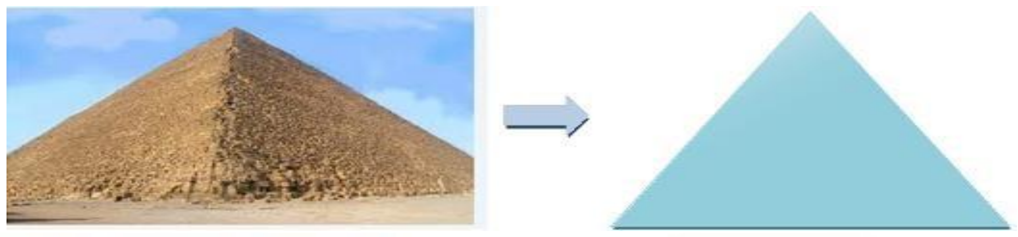
GENERAL INTRODUCTION

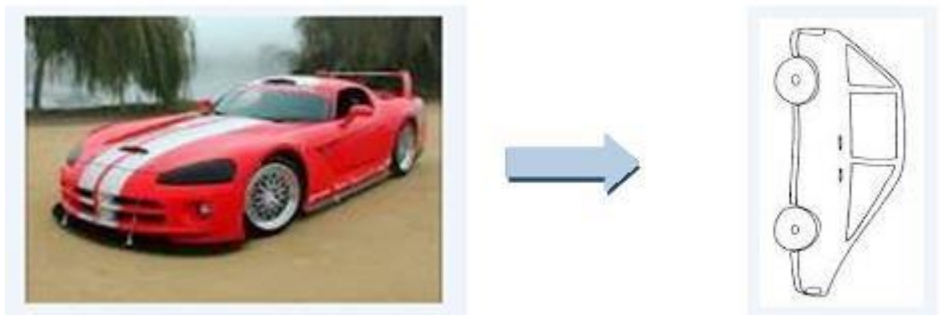
Intention of this course OOM (object oriented modeling) is to learn how to apply object -oriented concepts to all the stages of the software development life cycle.

Software design is the most crucial step in the software development process that is why it must be given a good care. Software designers must go through many modeling steps to end up with a good design that will allow for a smooth development process later. For this, designers usually have to choose between two main modeling methodologies: Merise and OMT. Both methodologies are widely used; however, each one has its own advantages and disadvantages.

System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system. It is about representing a system using some kind of graphical notation. Models help the analyst to understand the functionality of the system; they are used to communicate with customers.

A model is an abstract representation of something for understanding it before building it.





A model is an abstraction of something for understanding it before building it. Because, real systems that we want to study are generally very complex. In order to understand the real system, we have to simplify the system. So a model is an abstraction that hides the non-essential characteristics of a system and highlights those characteristics, which are pertinent to understand it. Efraim Turban describes a model as a simplified representation of reality. A model provides a means for conceptualization and communication of ideas in a precise and unambiguous form. The characteristics of simplification and representation are difficult to achieve in the real world, since they frequently contradict each other. Thus, modeling enables us to cope with the complexity of a system.

Why do we model?

Before constructing anything, a designer first build a model. The main reasons for constructing models include:

- To test a physical entity before actually building it.
- To set the stage for communication between customers and developers.
- For visualization i.e. for finding alternative representations.
- For reduction of complexity in order to understand it.

Models can explain the system from **different perspectives**:

- An **external** perspective, where you model the context or environment of the system.
- An **interaction** perspective, where you model the interactions between a system and its environment, or between the components of a system.

- A **structural** perspective, where you model the organization of a system or the structure of the data that is processed by the system.
- A **behavioral** perspective, where you model the dynamic behavior of the system and how it responds to events.

I. Types of modeling techniques

There are three common types of system modeling techniques namely: **structured modeling, systemic modeling and object-oriented modeling.**

Each modeling technique uses a well-suited modeling language. A **modeling language** is a set of concepts and rules for constructing models describing information systems.

structured modeling technique

The first generation information system design methods were based on this modeling technique
It should be noted that to model here, we use a descending "top-down" approach, from top to bottom, which starts from the general, goes towards the particular and implements Descartes's principle. The most common example of **structured modeling technique is SADT (Structured Analysis and Design Technique)** which offers variety of boxes and arrows to represent a system

a) Systematic modeling technique

Second generation information systems design methods are entirely focused on data modeling. The most popular systemic method is MERISE. It shows that the information system is related on the one hand to the operational system of the organization and on the other hand to its management system. The MERISE method is inspired by systems theory and therefore starts from the general to the particular: that is to say, begins by understanding the organization, then delimits the domain, then designs the projects to finally realize the application

b) **Object-oriented modeling Technique**

Object-oriented modeling and design is a way of analyzing and designing software application using models organized around real world concepts such as object, which combines both data structure and behavior.

OOM means that we organize software as a collection of discrete objects (that incorporate both data structure and behavior).
Common examples of OOM modeling languages are Object-Modeling Technique (OMT), Object-Oriented Software Engineering (OOSE)

II. OBJECT ORIENTED MODELING

Object oriented modeling is entirely a new way of thinking about problems. This methodology is all about visualizing the things by using models organized around real world concepts. Object oriented models help in understanding problems, communicating with experts from a distance, modeling enterprises, and designing programs and database. We all can agree that developing a model for a software system, prior to its development or transformation, is as essential as having a blueprint for large building essential for its construction. Object diagrams represent oriented models. A good model always helps communication among project teams, and to assure architectural soundness. It is important to note that with the increasing complexity of systems, importance of modeling techniques increases. Because of its characteristics, Object Oriented Modeling is a suitable modeling technique for handling a complex system. OOM is building a model of an application, which includes implementation details of the system, during design of the system.

Object Oriented Methodologies

We live in a world of objects. These objects exist in nature, in man-made entities, in business, and in the products that we use. They can be categorized, described, organized, combined, manipulated and created. Therefore, an object-oriented view has come into picture for creation of computer software. An object-oriented approach to the development of software was proposed in late 1960s.

Object Oriented Methodology (OOM) is a new system development approach encouraging and facilitating reuse of software components. With this methodology, a computer system can be developed on a component basis, which enables the effective reuse of existing components and facilitates the sharing of its components by other systems. By using OOM, higher productivity, lower maintenance cost and better quality can be achieved.

OOM requires that object-oriented techniques be used during the analysis, design and implementation of the system. This methodology makes the analyst to determine what the objects of the system are, how they behave over time or in response to events, and what responsibilities and relationships an object has to other objects. Object-oriented analysis has the analyst look at all the objects in a system, their commonalties, difference, and how the system needs to manipulate the objects.

During design, overall architecture of the system is described. During implementation phase, the class objects and the interrelationships of these classes are translated and actually coded using the programming language.

The databases are created and the complete system is made operational.

1.3.1 Object Oriented Process

The OOM for building systems takes the objects as the basis. For this, first the system to be developed is observed and analyzed and the requirements are defined. Once this is done, the objects in the required system are identified. For example, in case of a Banking System, a customer is an object, a ledger is an object, passbook is an object and even an account is an object.

OOM is somewhat similar to the traditional approach of system designing, in that it also follows a sequential process of system designing but with a different approach. The basic steps of system designing using OOM may be listed as:

- System Analysis
- System Design
- Object Design
- Implementation

1.3.1.1 System Analysis

As in any other system development model, system analysis is the first phase of OOM too. In this phase, the developer interacts with the user of the system to find out the user requirements and analyses the system to understand the functioning of it.

Based on this system study, the analyst prepares a model of the desired system. This model is purely based on what the system is required to do. At this stage the implementation details are not taken care of. Only the model of the system is prepared based on the idea that the system is made up of a set of interacting objects. The important elements of the system are emphasized.

1.3.1.2 System Design

System Design is the next development stage in OOM where the overall architecture of the desired system is decided. The system is organized as a set of sub systems interacting with each other. While designing the system as a set of interacting subsystems, the analyst takes care of specifications as observed in system analysis as well as what is required out of the new system by the end user.

The system analysis is to perceive the system as a set of interacting objects. A bigger system may also be seen as a set of interacting smaller subsystems that in turn are composed of a set of interacting objects. While designing the system, the stress lies on the objects comprising the system and not on the processes being carried out in the system.

1.3.1.3 Object Design

In this phase, the details of the system analysis and system design are implemented. The Objects identified in the system design phase are designed. Here the implementation of these objects is decided in the form of data structures required and the interrelationships between the objects.

For example, we can define a data type called customer and then create and use several objects of this data type. This concept is known as creating a class.

In this phase of the development process, the designer also decides about the classes in the system based on these concepts. He decides on whether the classes need to be created from scratch or any existing classes can be used as it is or new classes can be inherited from them.

1.3.1.4 Implementation

During this phase, the class objects and the interrelationships of these classes are translated and actually coded by using an object-oriented programming language. The required databases are created and the complete system is transformed into operational one.

1.3.2 Advantages of Object Oriented Methodology

- As compared to the conventional system development techniques, OOM provides many benefits.

- The systems designed using OOM are closer to the real world as the real world functioning of the system is directly mapped into the system designed using OOM. Because of this, it becomes easier to produce and understand designs.
- The objects in the system are immune to requirement changes because of data hiding and encapsulation features of objectorientation. Here, encapsulation we mean a technique that allows the programmer to hide the internal functioning of the objects from the users of the objects. Encapsulation separates the internal functioning of the object from the external functioning thus providing the user flexibility to change the external behavior of the object making the programmer code safe against the changes made by the user.
- OOM designs encourage more reusability. The classes once defined can easily be used by other applications. This is achieved by defining classes and putting them into a library of classes where all the classes are maintained for future use. Whenever a new class is needed the programmer first looks into the library of classes and if it is available, it can be used as it is or with some modification. This reduces the development cost & time and increases quality.
- Another way of reusability is provided by the inheritance feature of the object-orientation. The concept of inheritance allows the programmer to use the existing classes in new applications i.e. by making small additions to the existing classes can quickly create new classes. This provides all the benefits of reusability discussed in the previous point.
- As the programmer has to spend less time and effort so he can utilize saved time (due to the reusability feature of the OOM) in concentrating on other aspects of the system.
- OOM approach is more natural as it deals with the real world objects. So, it provides nice structures for thinking and abstracting and leads to modular design.

Check Your Progress 1 1) What is OOM? …………………………………………………………………………..
……………………………………………………………………………. ……………………………………………………………..
2) List different steps involved in OOM process.
……………………………………………………………………………. ……………………………………………………………
3) Differentiate OO development from structured development.
……
….. ………
Structured approach of problem solving is based on the concept of decomposition of system in to subsystem. In this approach of system development readjustment of some new changes in the

system is very difficult. On the other hand, in object oriented approach holistic view of application domain is considered and related object are identified. Further classification of objects are done. Object oriented approach give space for further enhancement of the system without too much increase in systems complexity.

There are four aspects (characteristics) required by an OO approach
Identity.
Classification.
Inheritance.
Polymorphism.

Identity: Identity means that data is contained into discrete, distinguishable entities called objects. Objects can be concrete or conceptual thing or even a specific person of the system Classification means that objects with the same data structure (attribute) and behavior (operations) are grouped into a class. E.g. paragraph, monitor, chess piece. Each object is said to be an instance of its class
A class is a collection of similar objects (collection of object with the same attributes and behavior). It is a template where certain basic characteristics of a set of objects are defined. The class defines the basic attributes and the operations of the objects of that type. Defining a class does not define any object, but it only creates a template. For objects to be actually created instances of the class are created as per the requirement of the case.

Inheritance: It is the sharing of attributes and operations (features) among classes based on a hierarchical relationship. A super class has general information that sub classes refine and elaborate.
E.g. Scrolling window and fixed window are sub classes of window.

Polymorphism: Polymorphism means that the same operation may behave differently for different classes. For E.g. The addition (+) operation behaves differently for string.
Note: An operation is a procedure/transformation that an object performs or is subjected to. An implementation of an operation by a specific class is called a method.

Abstraction is one of the very important concepts of object-oriented systems Abstraction focuses on the essential, inherent aspects of an object of the system. It does not represent the irrelevant properties of the system. In system development, abstraction helps to focus on what an object is supposed to do, before deciding how it should be implemented. The use of abstraction protects the freedom to make decisions for as long as possible, by avoiding intermediate commitments in problem solving. Most of the modern languages provide data abstraction. With the abstraction, ability to use inheritance and ability to apply polymorphism provides additional freedom and capability for system development.

Encapsulation

Encapsulation, or information hiding, is the feature of separating the external aspects of an object, from the internal implementation details of that object. It helps in hiding the actual implementation of characteristics of objects. You can say that encapsulation is hiding part of implementation that do internal things, and these hidden parts are not concerned to outside world. Encapsulation enables you to combine data structure and behavior in a single entity. Encapsulation helps in system enhancement. If there is a need to change the implementation of object without affecting its external nature, encapsulation is of great help.

BENEFITS OF OBJECT ORIENTED MODELING

There are several advantages and benefits of object oriented modeling. Reuse and emphasis on quality are the major highlights of OOM. OOM provides resistance to change, encapsulation and abstraction, etc. Due to its very nature, all these features add to the systems development:

• Faster development
• Increased Quality
• Easier maintenance
• Reuse of software and designs, frameworks
• Reduced development risks for complex systems integration. The conceptual structure of object orientation helps in providing abstraction mechanisms for modeling, which includes • Classes • Objects • Inheritance • Association etc

. How are data and functions organized in an object-oriented program?

6. What are the unique advantages of an object-oriented programming paradigm?
7. Distinguish between the following terms:
   (a) Object and classes
   (b) Data abstraction and data encapsulation
   (c) Inheritance and polymorphism (d) Dynamic binding and message passing
8. Describe inheritance as applied to OOP.
9. What do you mean by dynamic binding? How it is useful in OOP?

Unified Modeling Language (UML) Introduction

Way back in the late twentieth century — 1997 to be exact — the Object Management Group (OMG) released the Unified Modeling Language (UML). One of the purposes of UML was to provide the development community with a stable and common design language that could be used to develop and build computer applications. UML brought forth a unified standard modeling notation that IT professionals had been wanting for years. Using UML, IT professionals could now read and disseminate system structure and design plans — just as construction workers have been doing for years with blueprints of buildings.

It is now the twenty-first century — 2003 to be precise — and UML has gained traction in our profession. On 75 percent of the resumes I see, there is a bullet point claiming knowledge of UML. However, after speaking with a majority of these job candidates, it becomes clear that they do not truly know UML. Typically, they are either using it as a buzz word, or they have had a sliver of exposure to UML. This lack of understanding inspired me to write this quick introduction to UML, focused on the basic diagrams used in visual modeling. When you are finished reading you will not have enough knowledge to put UML on your resume, but you will have a starting point for digging more deeply into the language.

A little background

As I mentioned, UML was meant to be a unifying language enabling IT professionals to model computer applications. The primary authors were Jim Rumbaugh, Ivar Jacobson, and Grady Booch, who originally had their own competing methods (OMT, OOSE, and Booch). Eventually, they joined forces and brought about an open standard. (Sound familiar?

Unified Modeling Language (UML) is a general purpose modelling language. The main aim of UML is to define a standard way to **visualize** the way a system has been designed. It is quite similar to blueprints used in other fields of engineering.

UML is **not a programming language**; it is rather a visual language. We use UML diagrams to portray the **behavior and structure** of a system. UML helps software engineers, businesspersons and system architects with modelling, design and analysis. The Object Management Group (OMG) adopted Unified Modelling Language as a standard in 1997. Its been managed by OMG ever since. International Organization for Standardization (ISO) published UML as an approved standard in 2005. UML has been revised over the years and is reviewed periodically.

Do we really need UML?

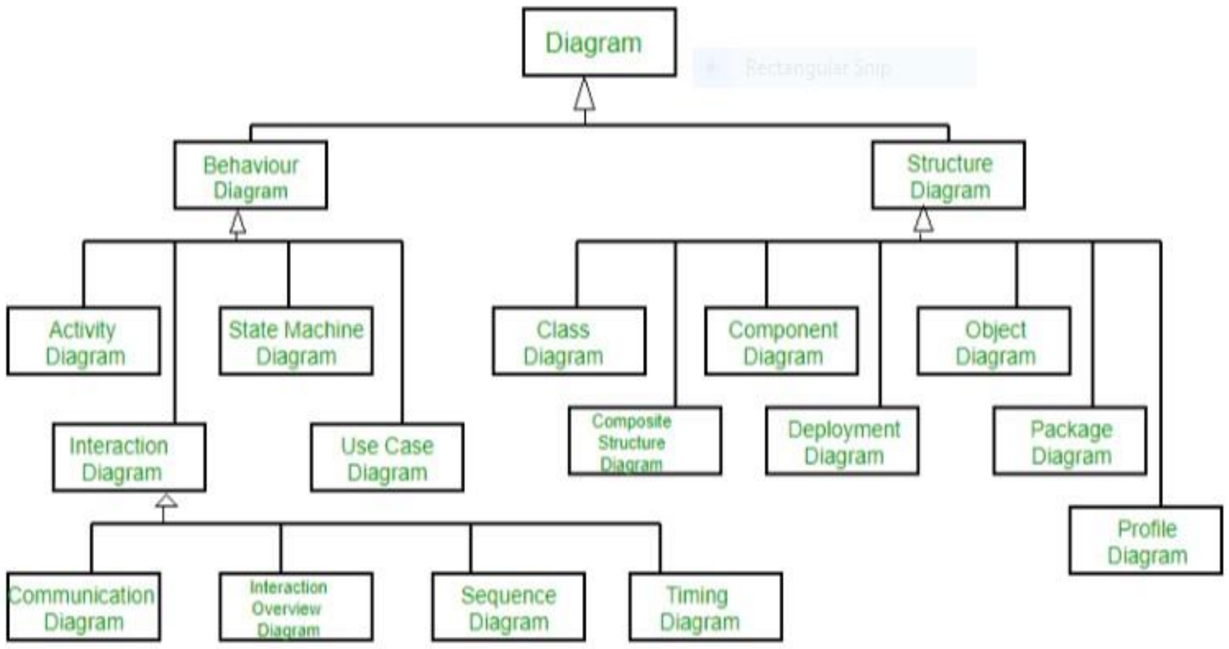
- Complex applications need collaboration and planning from multiple teams and hence require a clear and concise way to communicate amongst them.

- Businessmen do not understand code. So UML becomes essential to communicate with non programmers essential requirements, functionalities and processes of the system.
- A lot of time is saved down the line when teams are able to visualize processes, user interactions and static structure of the system.

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

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

The image below shows the hierarchy of diagrams according to UML 2.2



Object Oriented Concepts Used in UML –

1. **Class –** A class defines the blue print i.e. structure and functions of an object.
2. **Objects –** Objects help us to decompose large systems and help us to modularize our system. Modularity helps to divide our system into understandable components so

that we can build our system piece by piece. An object is the fundamental unit (building block) of a system which is used to depict an entity.

3. **Inheritance** – Inheritance is a mechanism by which child classes inherit the properties of their parent classes.
4. **Abstraction –** Mechanism by which implementation details are hidden from user.
5. **Encapsulation –** Binding data together and protecting it from the outer world is referred to as encapsulation.
6. **Polymorphism** – Mechanism by which functions or entities are able to exist in different forms.

Additions in UML 2.0 –

- Software development methodologies like agile have been incorporated and scope of original UML specification has been broadened.
- Originally UML specified 9 diagrams. UML 2.x has increased the number of diagrams from 9 to 13. The four diagrams that were added are : timing diagram, communication diagram, interaction overview diagram and composite structure diagram. UML 2.x renamed statechart diagrams to state machine diagrams.

UML 2.x added the ability to decompose software system into components and sub-components.

UML - Modeling Types

It is very important to distinguish between the UML model. Different diagrams are used for different types of UML modeling. There are three important types of UML modeling.

Structural Modeling

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

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

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

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

Behavioral Modeling

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

- Activity diagrams
- Interaction diagrams
- Use case diagrams

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

Architectural Modeling

Architectural model represents the overall framework of the system. It contains both structural and behavioral elements of the system. Architectural model can be defined as the blueprint of the entire system. Package diagram comes under architectural modeling.

UML - Basic Notations

UML is popular for its diagrammatic notations. We all know that UML is for visualizing, specifying, constructing and documenting the components of software and non-software systems. Hence, visualization is the most important part which needs to be understood and remembered.

UML notations are the most important elements in modeling. Efficient and appropriate use of notations is very important for making a complete and meaningful model. The model is useless, unless its purpose is depicted properly.

Hence, learning notations should be emphasized from the very beginning. Different notations are available for things and relationships. UML diagrams are made using the notations of things and relationships. Extensibility is another important feature which makes UML more powerful and flexible.

The chapter describes basic UML notations in detail. This is just an extension to the UML building block section discussed in Chapter Two.

UML - Standard Diagrams

In the previous chapters, we have discussed about the building blocks and other necessary elements of UML. Now we need to understand where to use those elements.

The elements are like components which can be associated in different ways to make a complete UML picture, which is known as diagram. Thus, it is very important to understand the different diagrams to implement the knowledge in real-life systems.

Any complex system is best understood by making some kind of diagrams or pictures. These diagrams have a better impact on our understanding. If we look around, we will realize that the diagrams are not a new concept but it is used widely in different forms in different industries.

We prepare UML diagrams to understand the system in a better and simple way. A single diagram is not enough to cover all the aspects of the system. UML defines various kinds of diagrams to cover most of the aspects of a system.

You can also create your own set of diagrams to meet your requirements. Diagrams are generally made in an incremental and iterative way.

There are two broad categories of diagrams and they are again divided into subcategories −

- Structural Diagrams
- Behavioral Diagrams

Structural Diagrams

The structural diagrams represent the static aspect of the system. These static aspects represent those parts of a diagram, which forms the main structure and are therefore stable.

These static parts are represented by classes, interfaces, objects, components, and nodes. The four structural diagrams are −

- Class diagram
- Object diagram
- Component diagram Deployment diagram

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.

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.

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.

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.

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.

Behavioral Diagrams

Any system can have two aspects, static and dynamic. So, a model is considered as complete when both the aspects are fully covered.

Behavioral diagrams basically capture the dynamic aspect of a system. Dynamic aspect can be further described as the changing/moving parts of a system.

UML has the following five types of behavioral diagrams −

- Use case diagram
- Sequence diagram
- Collaboration diagram
- Statechart diagram
- Activity diagram

Use Case Diagram

Use case diagrams are a set of use cases, actors, and their relationships. They represent the use case view of a system.

A use case represents a particular functionality of a system. Hence, use case diagram is used to describe the relationships among the functionalities and their internal/external controllers. These controllers are known as **actors**.

Sequence Diagram

A sequence diagram is an interaction diagram. From the name, it is clear that the diagram deals with some sequences, which are the sequence of messages flowing from one object to another.

Interaction among the components of a system is very important from implementation and execution perspective. Sequence diagram is used to visualize the sequence of calls in a system to perform a specific functionality.

Collaboration Diagram

Collaboration diagram is another form of interaction diagram. It represents the structural organization of a system and the messages sent/received. Structural organization consists of objects and links.

The purpose of collaboration diagram is similar to sequence diagram. However, the specific purpose of collaboration diagram is to visualize the organization of objects and their interaction.

Statechart Diagram

Any real-time system is expected to be reacted by some kind of internal/external events. These events are responsible for state change of the system.

Statechart diagram is used to represent the event driven state change of a system. It basically describes the state change of a class, interface, etc.

State chart diagram is used to visualize the reaction of a system by internal/external factors.

Activity Diagram

Activity diagram describes the flow of control in a system. It consists of activities and links. The flow can be sequential, concurrent, or branched.

Activities are nothing but the functions of a system. Numbers of activity diagrams are prepared to capture the entire flow in a system.

Activity diagrams are used to visualize the flow of controls in a system. This is prepared to have an idea of how the system will work when executed.

Note − Dynamic nature of a system is very difficult to capture. UML has provided features to capture the dynamics of a system from different angles. Sequence diagrams and collaboration diagrams are isomorphic, hence they can be converted from one another without losing any information. This is also true for Statechart and activity diagram.

UML - Class Diagram

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 objectoriented systems because they are the only UML diagrams, which can be mapped directly with objectoriented 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.

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 −

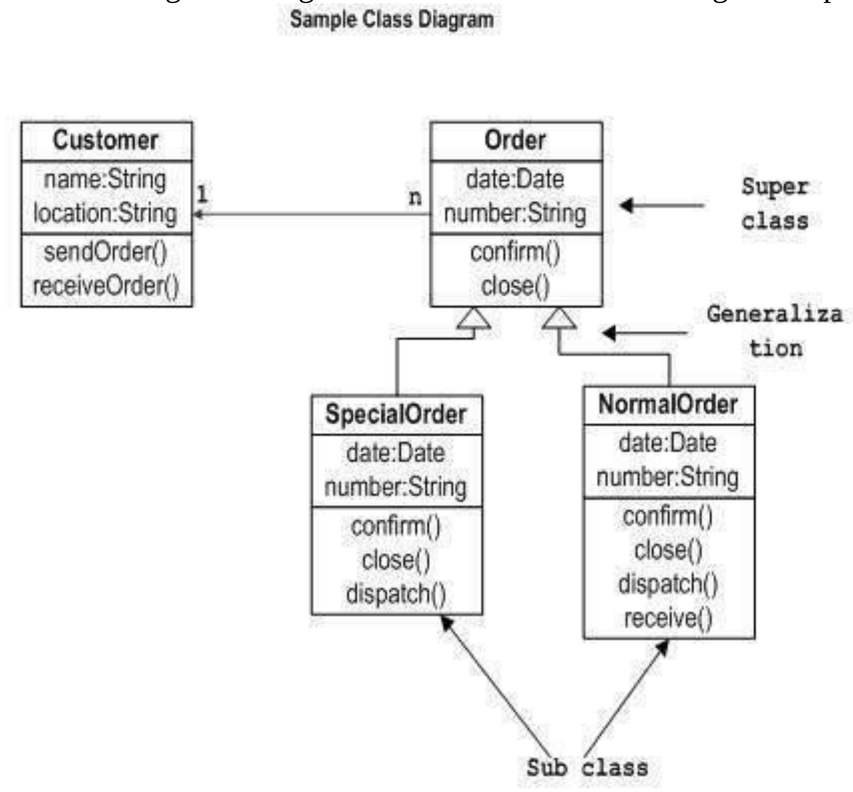
- The name of the class diagram should be meaningful to describe the aspect of the system.
- Each element and their relationships should be identified in advance.
- Responsibility (attributes and methods) of each class should be clearly identified

- 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.



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.

In a nutshell it can be said, 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.

UML - Object Diagrams

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 behaviour and their relationship from practical perspective

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.

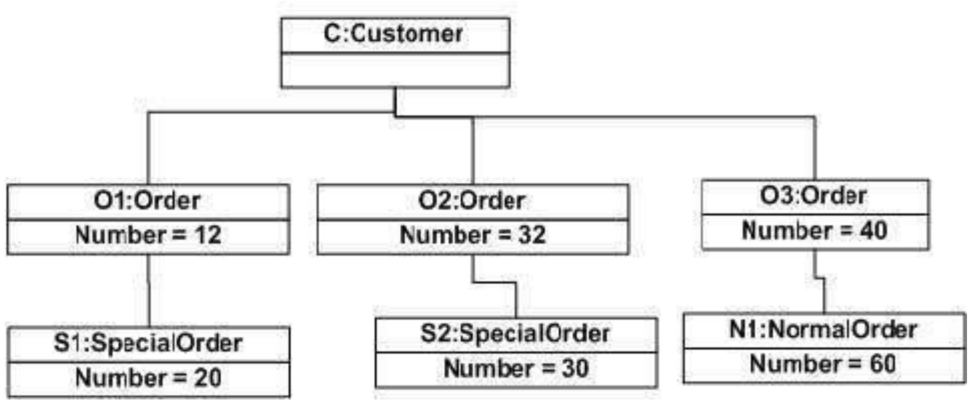
The customer can increase the number of orders in future and in that scenario the object diagram will reflect that. If order, special order, and normal order objects are observed then you will find that they have some values.

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

Object diagram of an order management system



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.

In a nutshell, it can be said that object diagrams are used for −

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

UML - Component Diagrams

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.

How to Draw a Component Diagram?

Component diagrams are used to describe the physical artifacts of a system. This artifact includes files, executables, libraries, etc

The purpose of this diagram is different. Component diagrams are used during the implementation phase of an application. However, it is prepared well in advance to visualize the implementation details.

Initially, the system is designed using different UML diagrams and then when the artifacts are ready, component diagrams are used to get an idea of the implementation.

This diagram is very important as without it the application cannot be implemented efficiently. A well-prepared component diagram is also important for other aspects such as application performance, maintenance, etc.

Before drawing a component diagram, the following artifacts are to be identified clearly −

- Files used in the system.
- Libraries and other artifacts relevant to the application.

  Relationships among the artifacts.

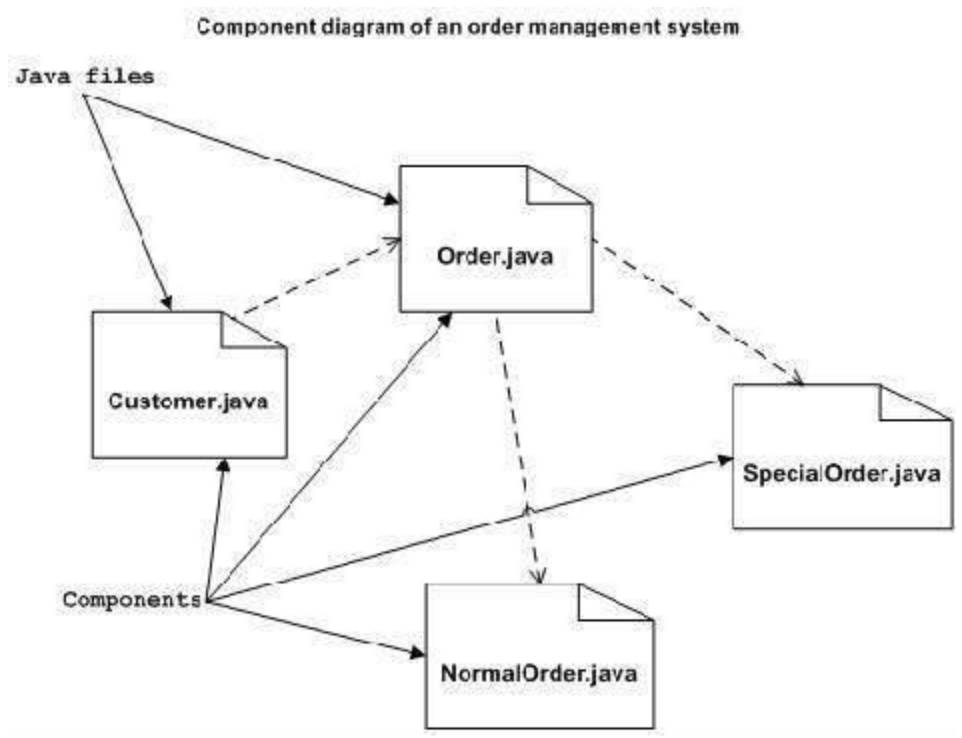
After identifying the artifacts, the following points need to be kept in mind.

- Use a meaningful name to identify the component for which the diagram is to be drawn.
- Prepare a mental layout before producing the using tools. Use notes for clarifying important points.

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.



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.

UML - Deployment Diagrams

eployment 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.

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

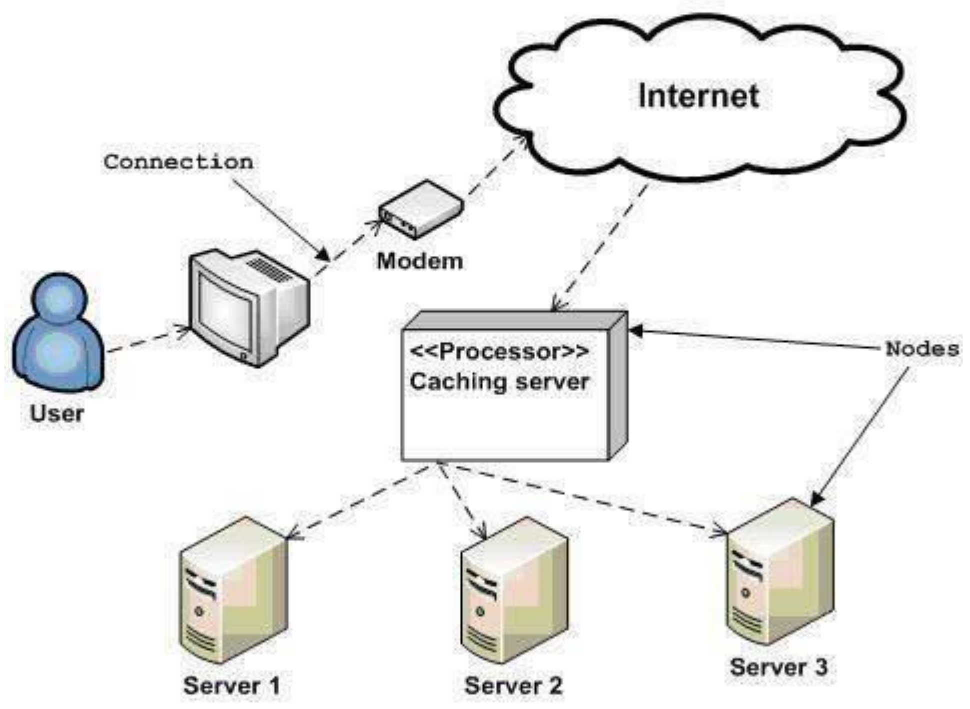
Following is a sample deployment diagram to provide an idea of the deployment view of order management system. Here, we have shown nodes as −

- 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.

Deployment diagram of an order management system



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.

UML - Use Case Diagrams

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.

Only static behavior is not sufficient to model a system rather dynamic behavior is more important than static behavior. In UML, there are five diagrams available to model the dynamic nature and use case diagram is one of them. Now as we have to discuss that the use case diagram is dynamic in nature, there should be some internal or external factors for making the interaction.

These internal and external agents are known as actors. Use case diagrams consists of actors, use cases and their relationships. The diagram is used to model the system/subsystem of an application. A single use case diagram captures a particular functionality of a system.

Hence to model the entire system, a number of use case diagrams are used.

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.

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.

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.

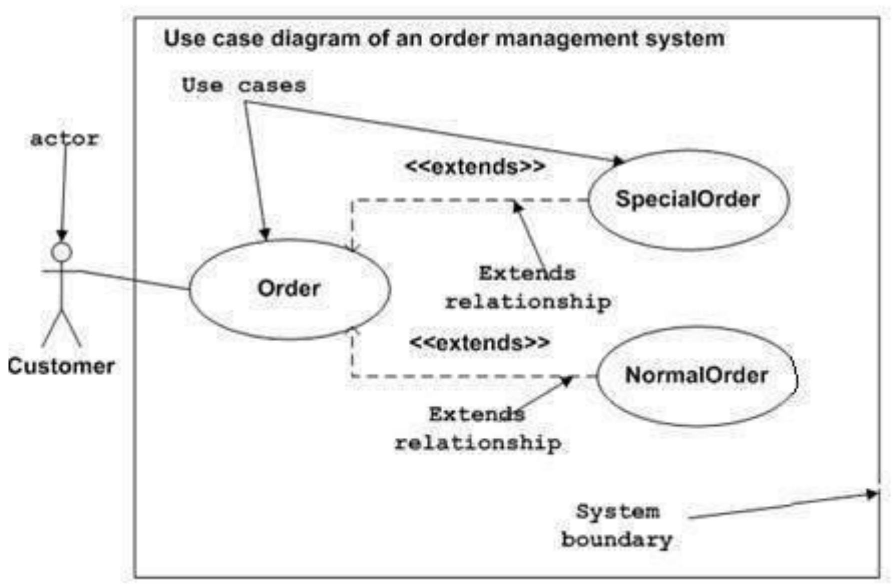


Figure: Sample Use Case diagram

Where to Use a Use Case Diagram?

As we have already discussed there are five diagrams in UML to model the dynamic view of a system. Now each and every model has some specific purpose to use. Actually these specific purposes are different angles of a running system.

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.

UML - Interaction Diagrams

From the term Interaction, it is clear that the diagram is used to describe some type of interactions among the different elements in the model. This interaction is a part of dynamic behavior of the system.

This interactive behavior is represented in UML by two diagrams known as **Sequence diagram** and **Collaboration diagram**. The basic purpose of both the diagrams are similar.

Sequence diagram emphasizes on time sequence of messages and collaboration diagram emphasizes on the structural organization of the objects that send and receive messages.

Purpose of Interaction Diagrams

The purpose of interaction diagrams is to visualize the interactive behavior of the system. Visualizing the interaction is a difficult task. Hence, the solution is to use different types of models to capture the different aspects of the interaction.

Sequence and collaboration diagrams are used to capture the dynamic nature but from a different angle.

The purpose of interaction diagram is −

- To capture the dynamic behaviour of a system.
- To describe the message flow in the system.
- To describe the structural organization of the objects.
- To describe the interaction among objects.

How to Draw an Interaction Diagram?

As we have already discussed, the purpose of interaction diagrams is to capture the dynamic aspect of a system. So to capture the dynamic aspect, we need to understand what a dynamic aspect is and how it is visualized. Dynamic aspect can be defined as the snapshot of the running system at a particular moment

We have two types of interaction diagrams in UML. One is the sequence diagram and the other is the collaboration diagram. The sequence diagram captures the time sequence of the message flow from one object to another and the collaboration diagram describes the organization of objects in a system taking part in the message flow.

Following things are to be identified clearly before drawing the interaction diagram

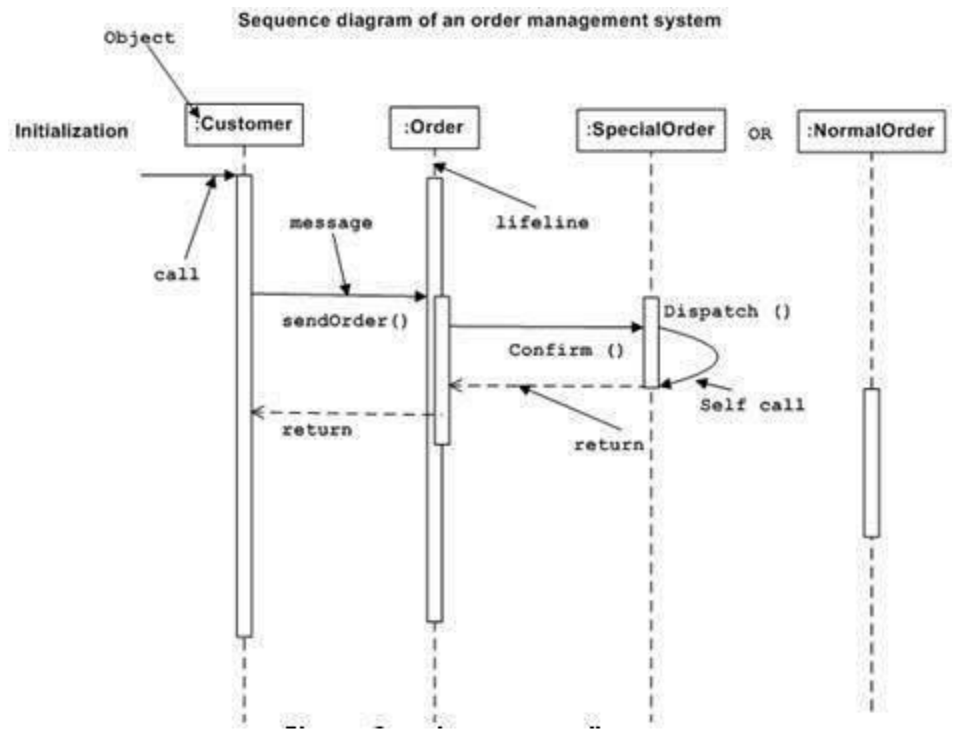
- Objects taking part in the interaction.
- Message flows among the objects.
- The sequence in which the messages are flowing. Object organization.

Following are two interaction diagrams modeling the order management system. The first diagram is a sequence diagram and the second is a collaboration diagram

The Sequence Diagram

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.

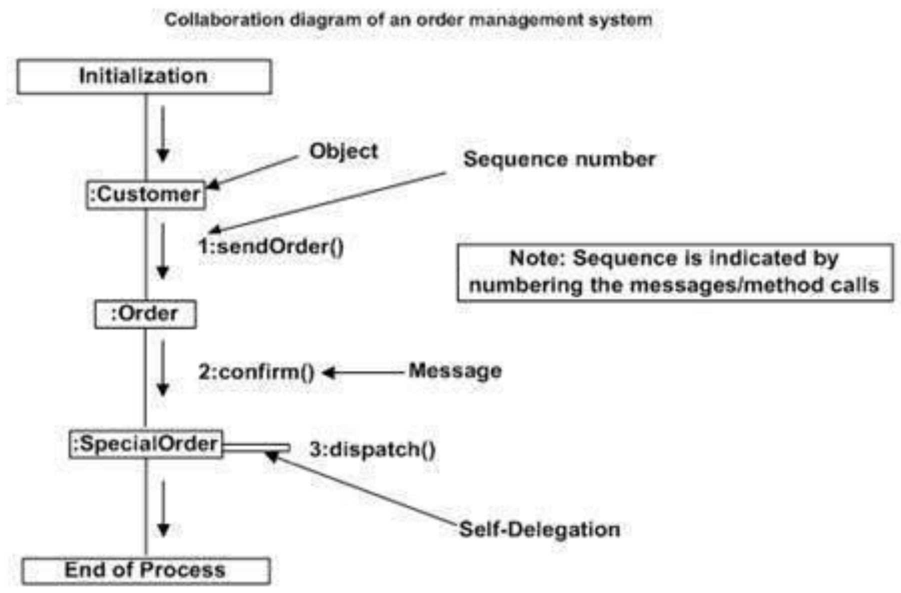


The Collaboration Diagram

The second interaction diagram is the 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.



Where to Use Interaction Diagrams?

We have already discussed that interaction diagrams are used to describe the dynamic nature of a system. Now, we will look into the practical scenarios where these diagrams are used. To understand the practical application, we need to understand the basic nature of sequence and collaboration diagram.

The main purpose of both the diagrams are similar as they are used to capture the dynamic behavior of a system. However, the specific purpose is more important to clarify and understand.

Sequence diagrams are used to capture the order of messages flowing from one object to another. Collaboration diagrams are used to describe the structural organization of the objects taking part in the interaction. A single diagram is not sufficient to describe the dynamic aspect of an entire system, so a set of diagrams are used to capture it as a whole.

Interaction diagrams are used when we want to understand the message flow and the structural organization. Message flow means the sequence of control flow from one object to another. Structural organization means the visual organization of the elements in a system.

Interaction diagrams can be used −

- To model the flow of control by time sequence.
- To model the flow of control by structural organizations.
- For forward engineering.
- For reverse engineering.

UML - Statechart Diagrams

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.

Activity diagram explained in the next chapter, is a special kind of a Statechart diagram. As Statechart diagram defines the states, it is used to model the lifetime of an object.

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.

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.

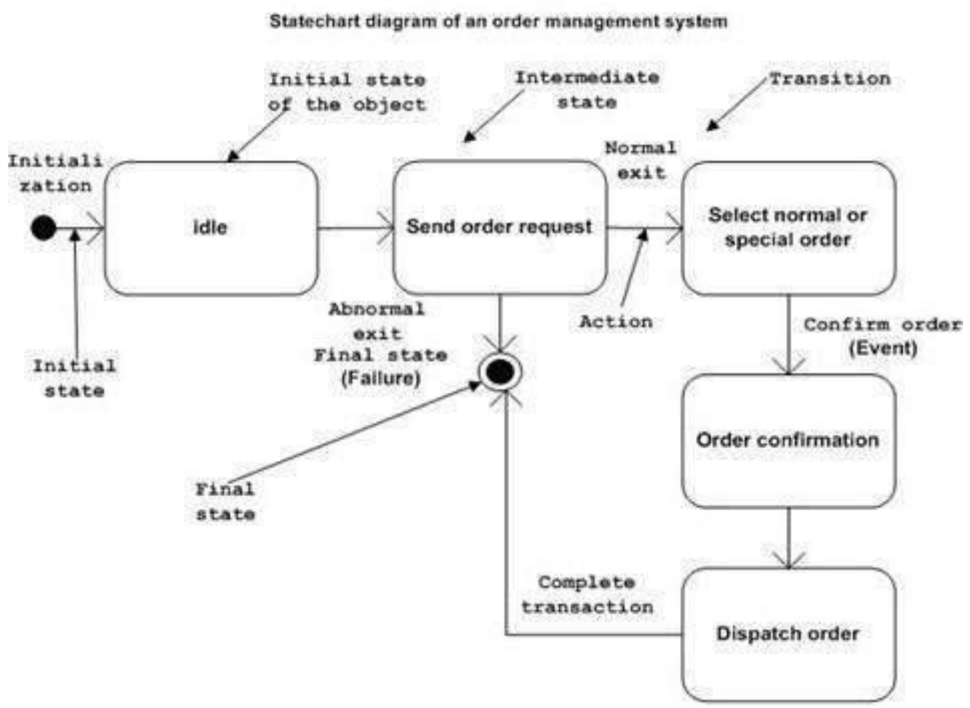
Before drawing a Statechart diagram we should clarify the following points −

- Identify the important objects to be analyzed.

- Identify the states.
- Identify the events.

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.



Where to Use Statechart Diagrams?

From the above discussion, we can define the practical applications of a Statechart diagram. Statechart diagrams are used to model the dynamic aspect of a system like other four diagrams discussed in this tutorial. However, it has some distinguishing characteristics for modeling the dynamic nature.

Statechart diagram defines the states of a component and these state changes are dynamic in nature. Its specific purpose is to define the state changes triggered by events. Events are internal or external factors influencing the system.

Statechart diagrams are used to model the states and also the events operating on the system. When implementing a system, it is very important to clarify different states of an object during its life time and Statechart diagrams are used for this purpose. When these states and events are identified, they are used to model it and these models are used during the implementation of the system.

If we look into the practical implementation of Statechart diagram, then it is mainly used to analyze the object states influenced by events. This analysis is helpful to understand the system behavior during its execution.

The main usage can be described as −

- To model the object states of a system.
- To model the reactive system. Reactive system consists of reactive objects.
- To identify the events responsible for state changes.
- Forward and reverse engineering.

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.

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

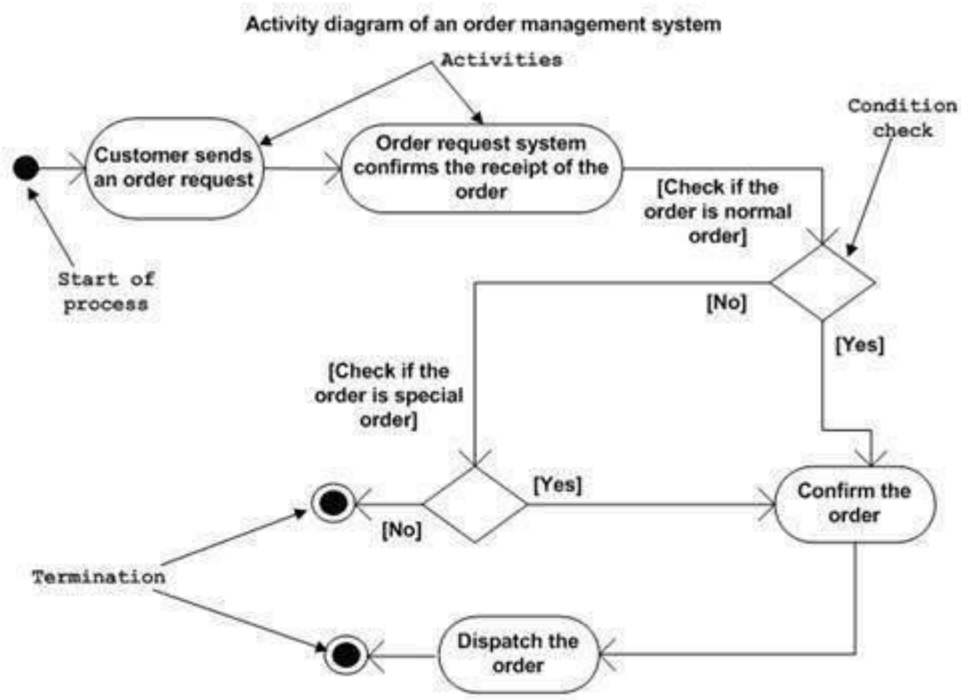
Once the above-mentioned parameters are identified, we need to make a mental layout of the entire flow. This mental layout is then transformed into an activity diagram.

Following 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.


Activity diagram of an order management system
Activities
Condition check
Customer sends an order request
Order request system confirms the receipt of the order
[Check if the order is normal order]
Start of process
[No]
[Yes]
[Check if the order is special order]
[Yes]
Confirm the order
[No]
Termination
Dispatch the order

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.

What are the three types of modeling in UML?

The three types of modeling in UML are as follows:

1. Structural modeling:

- It captures the static features of a system.
- It consists of the following diagrams:
1. Classes diagrams
2. Objects diagrams
3. Deployment diagrams
4. Package diagrams
5. Composite structure diagram
6. Component diagram

- This model represents the framework for the system and all the components exist here.
- It represents the elements and the mechanism to assemble them.
- It never describes the dynamic behavior of the system.

2. Behavioral modeling:

- It describes the interaction within the system.
- The interaction among the structural diagrams is represented here.
- It shows the dynamic nature of the system.
- It consists of the following diagrams:
1. Activity diagrams
2. Interaction diagrams
3. Use case diagrams
- These diagrams show the dynamic sequence of flow in the system.

3. Architectural modeling:

- It represents the overall framework of the system.
- The structural and the behaviour elements of the system are there in this system.
- It is defined as the blue print for the entire system. - Package diagram is used.

What are the different views in UML?

- Use Case view - Presents the requirements of a system.

- Design View - Capturing the vocabulary.
- Process View - Modeling the systems processes and threads. - Implementation view - Addressing the physical implementation of the system.
- Deployment view - Model the components required for deploying the system.

Use case view

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

Design View

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

Implementation View

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

Process View It is the view that demonstrates the concurrency of

the system.

- It incorporates the threads and processes that make concurrent system and synchronized mechanisms.
- It primarily addresses the system's scalability, throughput, and performance.

- Its static and dynamic aspects are expressed the same way as the design view but focus more on the active classes that represent these threads and processes.

Deployment View

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

UML CONCEPTUAL MODEL

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

- UML building block
- Rules to connect the building blocks
- Common mechanisms of UML

CHAPTER TWO

UML USE CASE DIAGRAM

Specific Instructional Objectives

At the end of this lesson the student will be able to:

- Identify different use cases of a system.
- Identify the purpose of use cases.
- Represent use cases for a particular system.
- Explain the utility of the use case diagram.
- Factorize use cases into different component use cases.
- Explain the organization of use cases.

Use Case Model

The use case model for any system consists of a set of “use cases”. Intuitively, use cases represent the different ways in which a system can be used by the users.

A simple way to find all the use cases of a system is to ask the question: “What the users can do using the system?” Thus for the Library Information System (LIS), the use cases could be**:**

- issue-book
- query-book
- return-book
- create-member
- add-book, etc

Use cases correspond to the high-level functional requirements. The use cases partition the system behavior into transactions, such that each transaction performs some useful action from the user’s point of view. To complete each transaction may involve either a single message or multiple message exchanges between the user and the system to complete.

Purpose of use cases

The purpose of a use case is to define a piece of coherent behavior without revealing the internal structure of the system. The use cases do not mention any specific algorithm to be used or the internal data representation, internal structure of the software, etc. A use case typically represents a sequence of interactions between the user and the system. These interactions consist of one mainline sequence. The mainline sequence represents the normal interaction between a user and the system. The mainline sequence is the most occurring sequence of interaction. For example, the mainline sequence of the withdraw cash use case supported by a bank ATM drawn, complete the transaction, and get the amount. Several variations to the main line sequence may also exist. Typically, a variation from the mainline sequence occurs when some specific conditions hold. For the bank ATM example, variations or alternate scenarios may occur, if the password is invalid or the amount to be withdrawn exceeds the amount balance. The variations are also called alternative paths. A use case can be viewed as a set of related scenarios tied together by a common goal. The mainline sequence and each of the variations are called scenarios or instances of the use case. Each scenario is a single path of user events and system activity through the use case.

Representation of use cases

Use cases can be represented by drawing a use case diagram and writing an accompanying text elaborating the drawing. In the use case diagram, each use case is represented by an ellipse with the name of the use case written inside the ellipse. All the ellipses (i.e. use cases) of a system are enclosed within a rectangle which represents the system boundary. The name of the system being modeled (such as Library Information System) appears inside the rectangle.

The different users of the system are represented by using the stick person icon. Each stick person icon is normally referred to as an actor. An actor is a role played by a user with respect to the system use. It is possible that the same user may play the role of multiple actors. Each actor can participate in one or more use cases. The line connecting the actor and the use case is called the communication relationship. It indicates that the actor makes use of the functionality provided by the use case. Both the human users and the external systems can be represented by stick person icons. When a stick person icon represents an external system, it is annotated by the stereotype <<external system>>.

7.4.3 How to Identify the Use Cases of a System?

Identification of the use cases involves brain storming and reviewing the SRS document. Typically, the high-level requirements specified in the SRS document correspond to the use cases. In the absence of a wellformulated SRS document, a popular method of identifying the use cases is actor-based. This involves first identifying the different types of actors and their usage of the system. Subsequently, for each actor the

Example 1:

The use case model for the Tic-tac-toe problem is shown in fig. 7.2. This software has only one use case “play move”. Note that the use case “get-user-move” is not used here. The name “get-user-move” would be inappropriate because the use cases should be named from the users’ perspective.

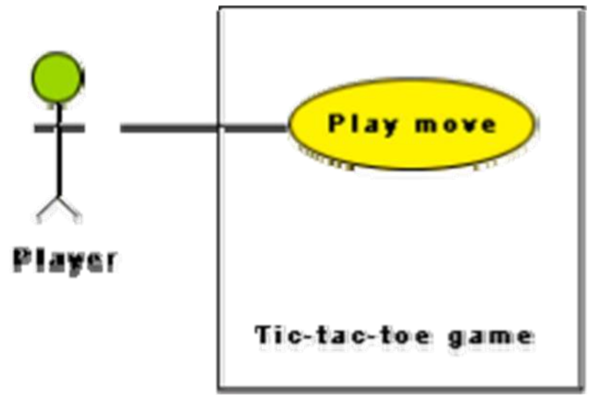


Fig. 7.2: Use case model for tic-tac-toe game

Text Description

Each ellipse on the use case diagram should be accompanied by a text description. The text description should define the details of the interaction between the user and the computer and other aspects of the use case. It should include all the behavior associated with the use case in terms of the mainline sequence, different variations to the normal behavior, the system responses associated with the use case, the exceptional conditions that may occur in the behavior, etc. The behavior description is often written in a conversational style describing the interactions between the actor and the system. The text description may be informal, but some structuring is recommended. The following are some of the information which may be included in a use case text description in addition to the mainline sequence, and the alternative scenarios.

Contact persons: This section lists the personnel of the client organization with whom the use case was discussed, date and time of the meeting,

etc.

Actors: In addition to identifying the actors, some information about actors using this use case which may help the implementation of the use case may be recorded.

Pre-condition: The preconditions would describe the state of the system before the use case execution starts.

Post-condition: This captures the state of the system after the use case has successfully completed.

Non-functional requirements: This could contain the important constraints for the design and implementation, such as platform and

environment conditions, qualitative statements, response time requirements, etc.

Exceptions, error situations: This contains only the domain-related errors such as lack of user's access rights, invalid entry in the input fields, etc. Obviously, errors that are not domain related, such as software errors, need not be discussed here.

Sample dialogs: These serve as examples illustrating the use case.

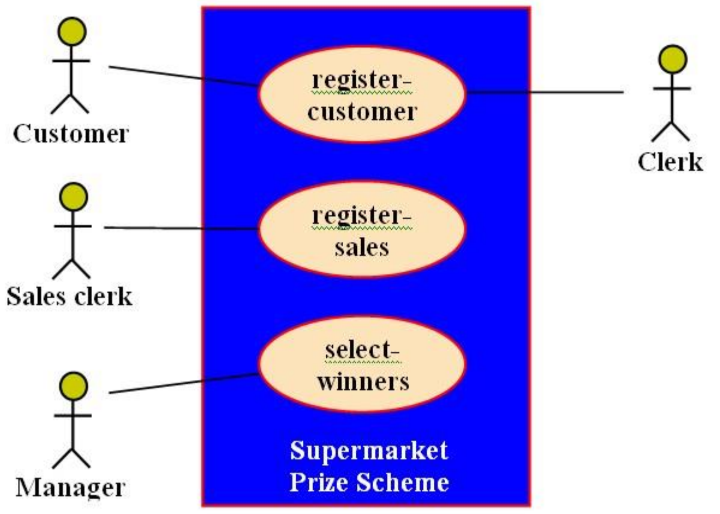
Specific user interface requirements: These contain specific requirements for the user interface of the use case. For example, it may contain forms to be used, screen shots, interaction style, etc.

Document references: This part contains references to specific domain-related documents which may be useful to understand the system operation.

Example 2:

The use case model for the Supermarket Prize Scheme described in Lesson 5.2 is shown in fig. 7.3. As discussed earlier, the use cases correspond to the high-level functional requirements. From the problem description and the context diagram in fig. 5.5, we can identify three use cases: “register-customer”, “register-sales”, and “select-winners”. As a sample, the text description for the use case “register-customer” is shown.

Supermarket



Prize Scheme

Fig. 7.3 Use case model for Supermarket Prize Scheme

Text description

U1: register-customer: Using this use case, the customer can register himself by providing the necessary details.

Scenario 1: Mainline sequence

1. **Customer: select register customer option.**
2. **System: display prompt to enter name, address, and telephone number.**
3. **Customer: enter the necessary values.**
4. **System: display the generated id and the message that the customer has been successfully registered.**

Scenario 2: at step 4 of mainline sequence

1. **System: displays the message that the customer has already registered.**

Scenario 2: at step 4 of mainline sequence

1. **System: displays the message that some input information has not been entered. The system display a prompt to enter the missing value.**

The description for other use cases is written in a similar fashion.

Utility of use case diagrams

From use case diagram, it is obvious that the utility of the use cases are represented by ellipses. They along with the accompanying text description serve as a type of requirements specification of the system and form the core model to which all other models must conform. But, what about the actors (stick person icons)? One possible use of identifying the different types of users (actors) is in identifying and implementing a security mechanism through a login system, so that each actor can involve only those functionalities to which he is entitled to. Another possible use is in preparing the documentation (e.g. users' manual) targeted at each category of user. Further, actors help in identifying the use cases and understanding the exact functioning of the system.

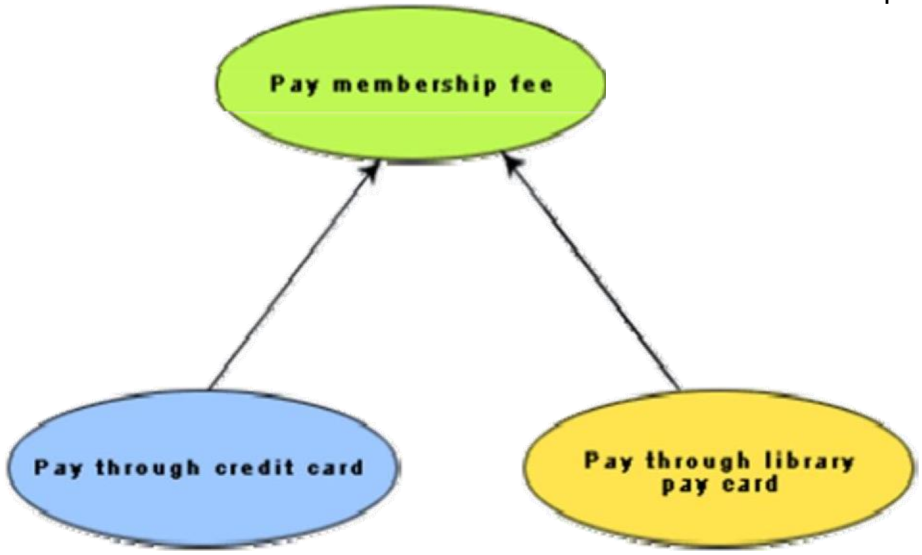
Factoring of use cases

It is often desirable to factor use cases into component use cases. Actually, factoring of use cases are required under two situations. First, complex use cases need to be factored into simpler use cases. This would not only make the behavior associated with the use case much more comprehensible, but also make the corresponding interaction diagrams more tractable. Without decomposition, the interaction diagrams for complex use cases may become too large to be accommodated on a single sized (A4) paper. Secondly, use cases need to be factored whenever there is common behavior across different use cases. Factoring would make it possible to define such behavior only once and reuse it whenever required. It is desirable to factor out common usage such as error handling from a set of use cases. This makes analysis of the class design much simpler and elegant. However, a word of caution here. Factoring of use cases should not be done except for achieving the above two objectives. From the design point of view, it is not advantageous to break up a use case into many smaller parts just for the shake of it.

UML offers three mechanisms for factoring of use cases as follows:

Generalization

Use case generalization can be used when one use case that is similar to another, but does something slightly differently or something more. Generalization works the same way with use cases as it does with classes. The child use case inherits the behavior and meaning of the parent use case. The notation is the same too (as shown in fig. 7.4). It is important to remember that the base and the derived use cases are separate use cases and should have



separate text descriptions.

Fig. 7.4: Representation of use case generalization

Includes

The includes relationship in the older versions of UML (prior to UML 1.1) was known as the uses relationship. The includes relationship involves one use case including the behavior of another use case in its sequence of events and actions. The includes relationship occurs when a chunk of behavior that is similar across a number of use cases. The factoring of such behavior will help in not repeating the specification and implementation across different use cases. Thus, the includes relationship explores the issue of reuse by factoring out the commonality across use cases. It can also be gainfully employed to decompose a large and complex use cases into more manageable parts. As shown in fig. 7.5, the includes relationship is represented using a predefined stereotype <<include>>. In the includes relationship, a base use case compulsorily and automatically includes the behavior of the common use cases. As shown in example fig. 7.6, issue-book and renew-book both include checkreservation use case. The base use case may include several use cases. In such cases, it may interleave their associated common use cases together. The common use case becomes a separate use case and the independent text description should be provided for it.

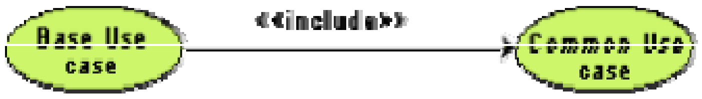


Fig. 7.5: Representation of use case inclusion

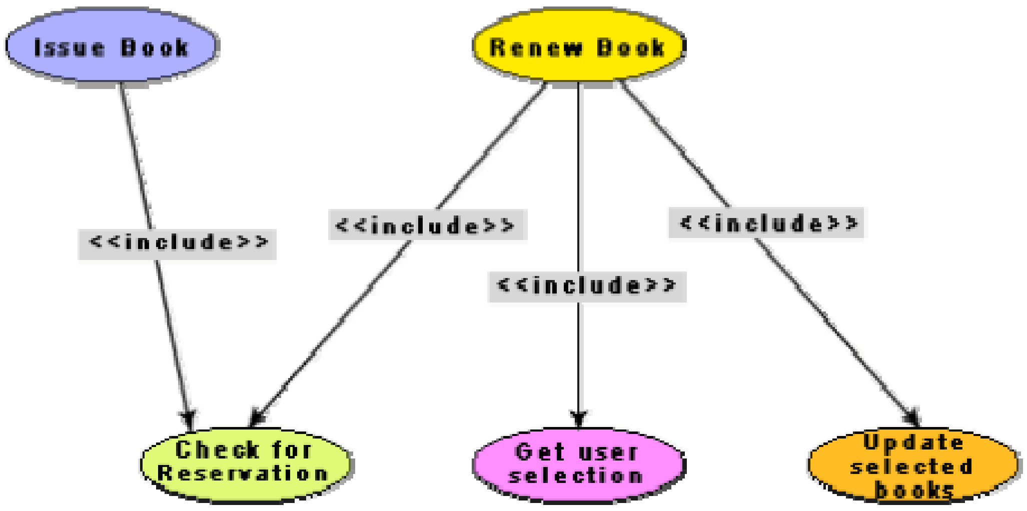


Fig. 7.6: Example use case inclusion

Extends

The main idea behind the extends relationship among the use cases is that it allows you to show optional system behavior. An optional system behavior is extended only under certain conditions. This relationship among use cases is also predefined as a stereotype as shown in fig. 7.7. The extends relationship is similar to generalization. But unlike generalization, the extending use case can add additional behavior only at an extension point only when certain conditions are satisfied. The extension points are points within the use case where variation to the mainline (normal) action sequence may occur. The extends relationship is normally used to capture alternate paths or scenarios.

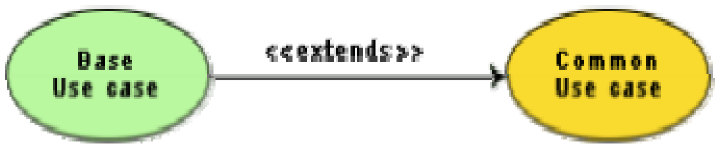


Fig. 7.7: Example use case extension

Organization of use cases

When the use cases are factored, they are organized hierarchically. The high- level use cases are refined into a set of smaller and more refined use cases as shown in fig. 7.8. Top-level use cases are super-ordinate to the refined use cases. The refined use cases are sub-ordinate to the top-level use cases. Note that only the complex use cases should be decomposed and organized in a hierarchy. It is not necessary to decompose simple use cases. The functionality of the superordinate use cases is traceable to their sub-ordinate use cases. Thus, the functionality provided by the super-ordinate use cases is composite of the functionality of the sub-ordinate use cases. In the highest level of the use case model, only the fundamental use cases are shown. The focus is on the application context. Therefore, this level is also referred to as the context diagram. In the context diagram, the system limits are emphasized. In the top- level diagram, only those use cases with which external users of the system. The subsystemlevel use cases specify the services offered by the subsystems. Any number of levels involving the subsystems may be utilized. In the lowest level of the use case hierarchy, the class-level use cases specify the functional fragments or operations offered by the classes.

Use Case Diagram Relationships Explained with Examples

Updated on: 2 July 2019

When it comes to drawing use case diagrams one area many struggles with is showing various relationships in use case diagrams. In fact many tend to confuse <<extend>>, <<include>> and generalization. This article will look into various **use case diagram relationships** in detail and explain them using examples. To get a deeper understanding of use cases, check out our use case diagram tutorial. If you want to draw them while learning you can use our tool to create use case diagrams.

There can be 5 relationship types in a use case diagram.

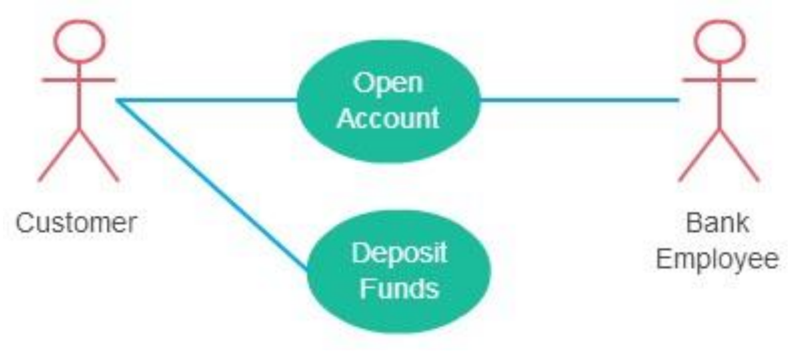
- Association between actor and use case
- Generalization of an actor
- Extend between two use cases
- Include between two use cases
- Generalization of a use case

Let's take a look at these relationships in detail.

Association Between Actor and Use Case

This one is straightforward and present in every use case diagram. Few things to note.

- An actor must be associated with at least one use case.
- An actor can be associated with multiple use cases.
- Multiple actors can be associated with a single use case.

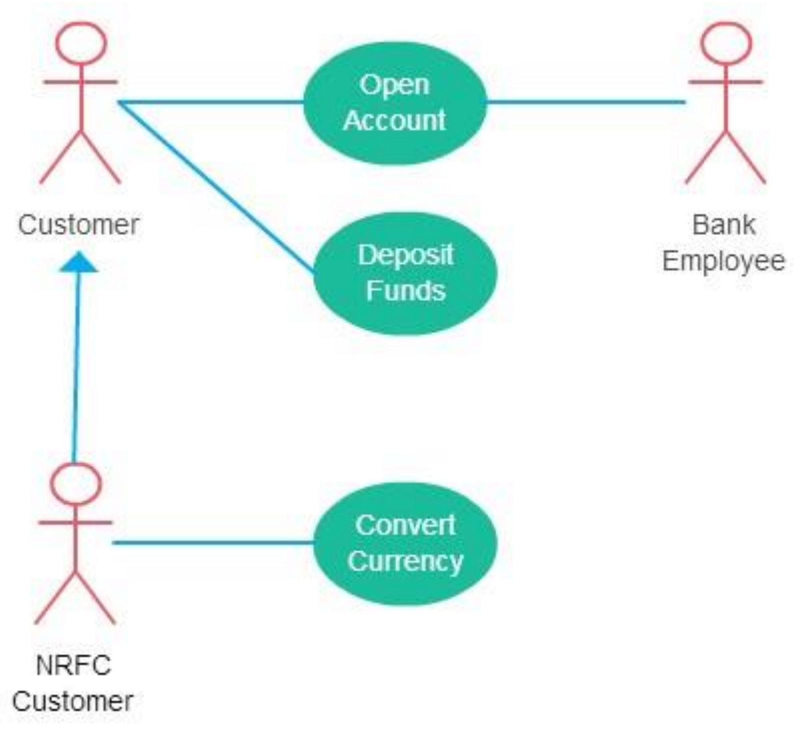


Different ways association relationship appears in use case diagrams

Check out the use case diagram guidelines for other things to consider when adding an actor.

Generalization of an Actor

Generalization of an actor means that one actor can inherit the role of the other actor. The descendant inherits all the use cases of the ancestor. The descendant has one or more use cases that are specific to that role. Let's expand the previous use case diagram to show the generalization of an actor.



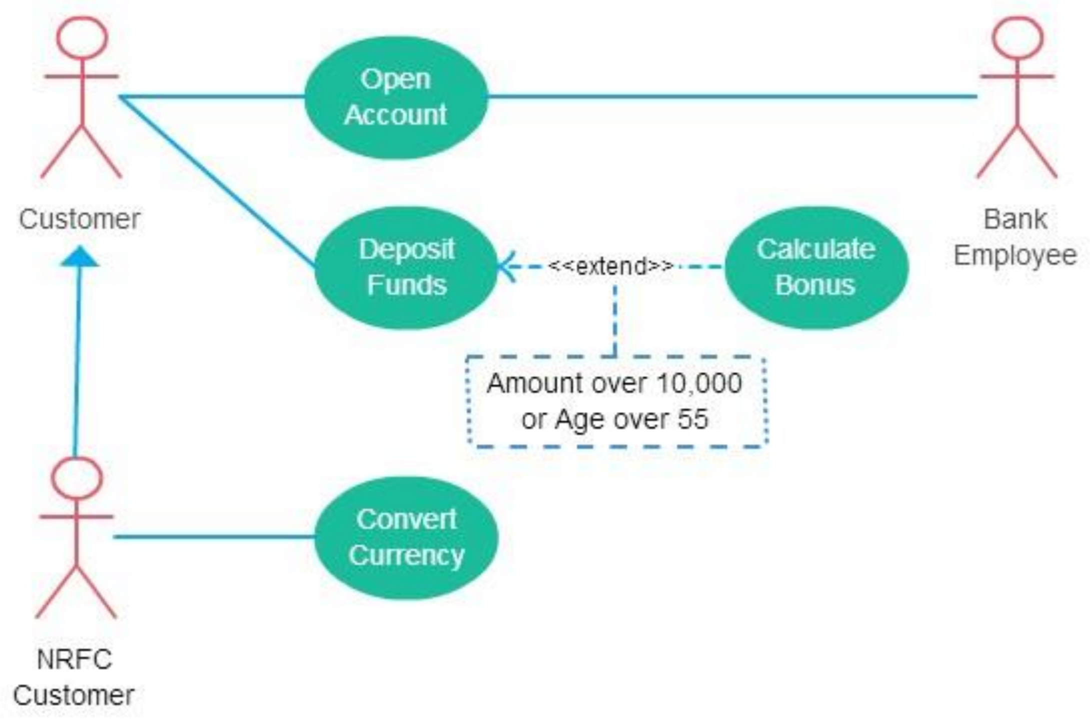
A generalized actor in an use case diagram

Extend Relationship Between Two Use Cases

Many people confuse the extend relationship in use cases. As the name implies it extends the base use case and adds more functionality to the system. Here are a few things to consider when using the <<**extend**>> relationship.

- **The extending use case is dependent on the extended (base) use case**. In the below diagram the "Calculate Bonus" use case doesn't make much sense without the "Deposit Funds" use case.
- **The extending use case is usually optional** and can be triggered conditionally. In the diagram, you can see that the extending use case is triggered only for deposits over 10,000 or when the age is over 55.
- **The extended (base) use case must be meaningful on its own**. This means it should be independent and must not rely on the behavior of the extending use case.

Lets expand our current example to show the <<extend>> relationship.



Extend relationship in use case diagrams

Although extending use case is optional most of the time it is not a must. An extending use case can have non-optional behavior as well. This mostly happens when your modeling complex behaviors.

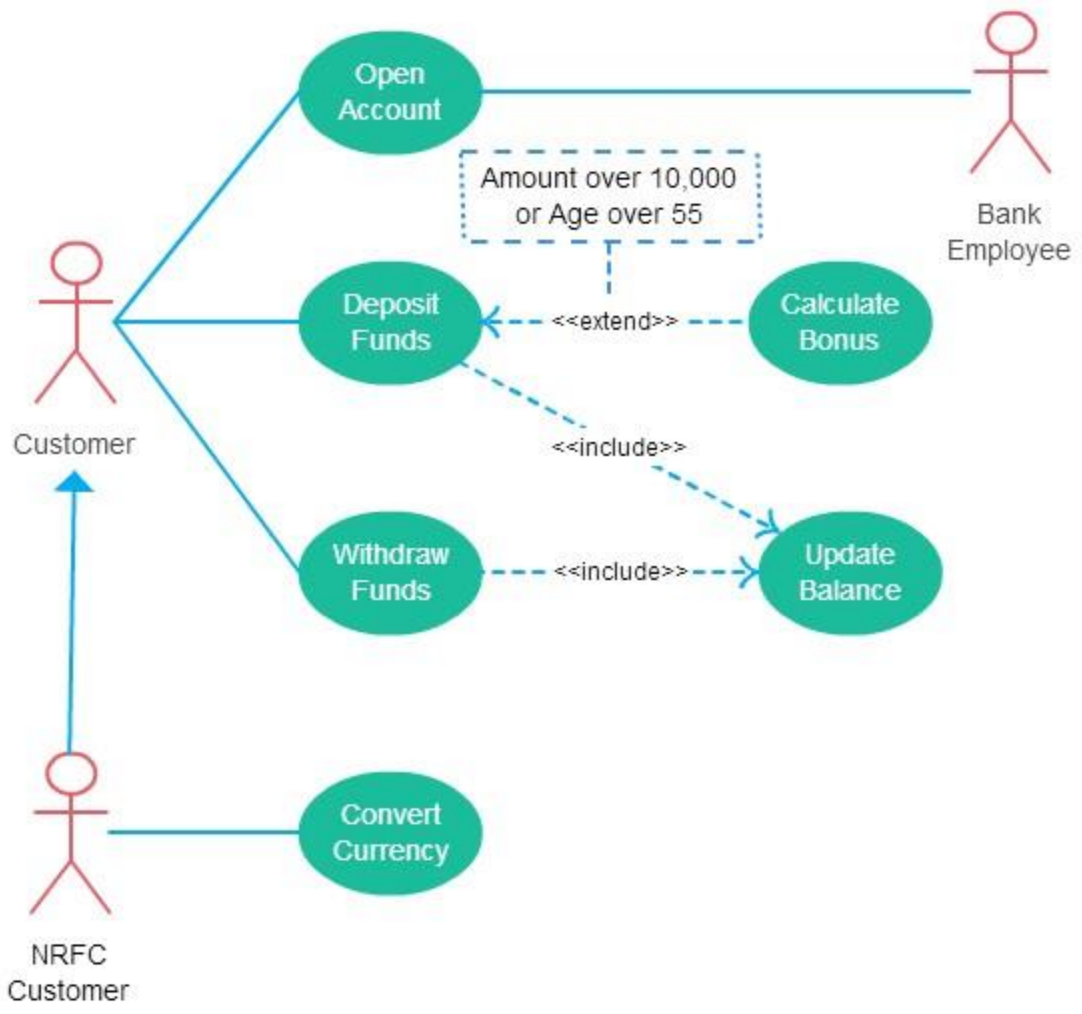
For example, in an accounting system, one use case might be "Add Account Ledger Entry". This might have extending use cases "Add Tax Ledger Entry" and "Add Payment Ledger Entry". These are not optional but depend on the account ledger entry. Also, they have their own specific behavior to be modeled as a separate use case.

Include Relationship Between Two Use Cases

Include relationship show that the behavior of the included use case is part of the including (base) use case. The main reason for this is to reuse common actions across multiple use cases. In some situations, this is done to simplify complex behaviors. Few things to consider when using the <<include>> relationship.

- The base use case is incomplete without the included use case.
- The included use case is mandatory and not optional.

Lest expand our banking system use case diagram to show include relationships as well.



Includes is usually used to model common behavior

For some further reading regarding the difference between extend and include relationships in use case diagrams check this StackOverflow link.

Generalization of a Use Case

This is similar to the generalization of an actor. The behavior of the ancestor is inherited by the descendant. This is used when there is common behavior between two use cases and also specialized behavior specific to each use case.

For example, in the previous banking example, there might be a use case called "Pay Bills". This can be generalized to "Pay by Credit Card", "Pay by Bank Balance" etc.

I hope you found this article about **use case relationships** helpful and useful. You can use our diagramming tool to easily create use case diagrams online. As always if you have any questions don't hesitate to ask them in the comments section.

Exercise 1

1. Define the following terms as used OOM (**5 marks**)
   - i. Model
   - ii. Modeling
   - iii. UML iv. Class
   - v. Object
   - vi. Encapsulation
   - vii. Use case
   - viii. Actor ix. Abstraction
   - x. object

2. Differentiate between the following pairs giving examples (**3 marks**)
   - A. software and program
   - B. use case and scenario
   - C. systematic and Objected modeling **Exercise 2**

1. Explain the following relationship as used with use case diagram , giving an example in each case (8 marks)
   - a. Generalization of an actor
   - b. Extend between two use cases
   - c. Include between two use cases
   - d. Generalization of a use case
2. Explain the following UML relationships , giving an example in each case (10 marks)
   - a. Inheritance (or Generalization
   - b. Association
   - c. Aggregation
   - d. Composition
   - e. cardinality

Exercise 3 : Hospital Management System (10 marks)

Hospital Management System is a large system including several subsystems or modules providing variety of functions.

Hospital Reception subsystem or module supports some of the many job duties of hospital receptionist. Receptionist schedules patient's appointments and admission to the hospital, perform patient registration and patient hospital admission, collects information from patient upon patient's arrival and/or by phone. For the patient that will stay in the hospital ("inpatient") she or he should have a bed allotted in a ward. Receptionists might also receive patient's payments, record them in a database and provide receipts, file insurance claims and medical reports.

QUESTION : Draw the use case diagram for the **Hospital Reception** subsystem **Exercise 4**

1. What is model? Why do we model?
2. What is object oriented methodology? What are the advantages of object oriented methodology?
3. What is object oriented process? Discuss the steps of object oriented process.
4. What is Object Modeling Technique (OMT)? What are the phases of OMT? Discuss each in brief.
5. What are the three models involved in OMT? Define each one of them.
6. What is object? Discuss the main characteristics of the object with examples from the real world.
7. List 20 objects from the real world around you. Write their attributes and operations.
8. What is class? What is OMT notation for a class discuss the relationships between class and object.
9. List 10 classes from the real world and define them.
10. Define link and give five different examples of links and represent them using OMT notations.

11. Define association. Discuss the characteristics and OMT notations of the association.
12. Give five examples of each unary, binary and ternary associations from the real world.
13. What do you mean by multiplicity of the association? Discuss different types of multiplicity by giving suitable examples.

OBJECT ORIENTED MODELING TUTORIALS

1- What is modelling? And what is the purpose of modelling?
2- What is the difference between a scenario and a use case? When do you use each construct? **(4 Mark)**
3- Can the system under consideration be represented as an actor? Justify your answer. **(3 marks)** 4- Consider an ATM system. Identify at least three different actors that interact with this system. **(3marks)**
5- An actor is any entity (user or system) that interacts with the system of interest. For an ATM, this includes:
6- Draw a class diagram representing a book defined by the following statement: "A book is composed of a number of parts, which in turn are composed of a number of chapters. Chapters are composed of sections." Focus only on classes and relationships. **(7 marks)**
7- Extend the class diagram of the question above to include the following attributes: **(8 marks)**
   A. abook includes a publisher, publication date, and an ISBN
   B. apart includes a title and a number
   C. achapter includes a title, a number, and an abstract
   D. asection includes a title and a number

Question 1

1) Define UML? **3marks**
2) Differentiate between: **5x4marks**
   i. Structural diagrams and Behavioral diagrams
   ii. Encapsulation and Inheritance
   iii. Class and object
   iv. Single and multiple inheritance
3) **OML** diagrams are divided into two main categories.
   i) List these categories and give three diagrams of each category **15marks**
4) In a school management system (SMS), a student can check attendance, timetable as well as test marks on the system and the teacher can interact with all the functionalities of the student including updating attendance of a student and marks of the student
   i) Draw a use case diagram of the SMS system. **12marks**

<u>Part I:</u> object modeling (25marks)

1. What is a model? (1mk)
2. Why is it important to always model computer system? (2mks)
3. What is object Oriented Modeling? (2mkq)
4. According to you what is UML Language? (2mks)

5. To understand UML diagrams and to learn UML, the user must have knowledge about the conceptual model of UML. The conceptual model consists of three parts. Name them (3mks)
6. UML Architecture can be best represented as a collection five views. Name the five (05) (3mks)
7. What is association? Illustrate with an example the concept of association. (2mks)
8. What is multiplicity in association? Give example to explain multiplicity. (2mks)
9. Create the use case diagram according to the following elevator control system functional requirements: (2x4=8mks)
   a) The elevator control system shall allow the passenger to call the elevator and to select the destination floor,
   b) When the passenger pushes the external button (to call the elevator), or the internal button (to select the destination floor), the central control system switches the button light on,
   c) When the passenger calls the elevator or selects the destination floor, the central control system opens/closes the elevator door,
   d) When the passenger calls the elevator or selects the destination floor, the central control system moves/stops the elevator to/at the passenger destination floor.

When the passenger leaves the elevator, the central control system switches the button light off.

3) An airline operates flights. Each airline has an ID. Each flight has an ID a departure airport and an arrival airport: an airport as a unique identifier. Each flight has a pilot and a copilot, and it uses an aircraft of a certain type; a flight has also a departure time and an arrival time.

   An airline owns a set of aircrafts of different types.

   An aircraft can be in a working state or it can be under repair.

   In a particular moment an aircraft can be landed or airborne.

   A company has a set of pilots: each pilot has an experience level:1 is minimum, 3 is maximum.

   A type of aeroplane may need a particular number of pilots, with a different role (e.g.: captain, co-pilot, navigator): there must be at least one captain and one co-pilot, and a captain must have a level 3.

QUESTION: Design the Class diagram of the system

Section A: information system and data base

I- Object modeling: 25mrks

1. What is dynamic model? How is it represented?
2. Define the following concepts with examples and their notations: state, event, transition, action and activity
3. What is scenario? Write a scenario for making a call on a telephone 4. Distinguish between the following:
   a. State and event
   b. Simple and compound states
   c. Action and activity
   d. Initial and final states
   e. Entry and exit point
5. What is object? Discuss the min characteristics of the object with example from the real world
6. List 5 objects from the real world around you. Write their attributes and operations

7. What is class? what is OMT notation for a class discuss the relationship between class and object
8. List 4 classes from the real world and define them
9. Define link and give five examples of links and represent them using OMT notation
10. Read the following descriptions. “Customers of the garage can buy cars. Customers with a bad credit should pay an extra down payment”. Which of the following diagrams represent this description?

CLASS DIAGRAM

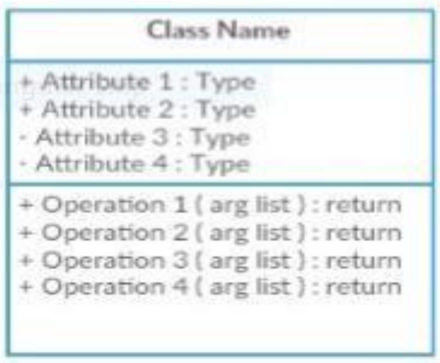
UML specification defines two major kinds of UML diagram: structure diagrams and behavior diagrams.

Structure diagrams show the **static structure** of the system and its parts on different abstraction and implementation **levels** and how they are related to each other. Structure diagrams is very important in software development because its elements represent the meaningful concepts of a system, and may include abstract, real world and implementation concepts. Examples of structure diagrams include class diagram, object diagram etc.

Behavior diagrams show the **dynamic behavior** of the objects in a system, which can be described as a series of changes to the system over **time**. Example of behavior diagram include use case diagram, activity diagram etc.

Class diagram is the main building block in object-oriented modeling. It is used to show the different objects in a system, their attributes, their operations and the relationships among them. **Class diagram** is **UML** structure diagram which shows structure of the designed system at the level of classes and interfaces, shows their features, constraints and relationships - associations, generalizations, dependencies, etc

The following figure is an example of a simple class:


Class Name
+ Attribute 1 : Type
+ Attribute 2 : Type
- Attribute 3 : Type
- Attribute 4 : Type
+ Operation 1 (arg list) : return
+ Operation 2 (arg list) : return
+ Operation 3 (arg list) : return
+ Operation 4 (arg list) : return

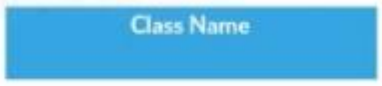
In the example, a class called “loan account” is depicted. Classes in class diagrams are represented by boxes that are partitioned into three:

1. The top partition contains the name of the class.

2. The middle part contains the class's attributes.
3. The bottom partition shows the possible operations that are associated with the class. The example shows how a class can encapsulate all the relevant data of a particular object in a very systematic and clear way. A class diagram is a collection of classes similar to the one above.

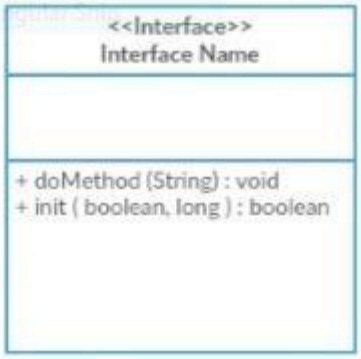
The first one shows the class's name, while the middle one shows the class's attributes which are the characteristics of the objects. The bottom one lists the class's operations, which represents the behavior of the class.

The class notation without the last two compartments is called a simple class and it only contains the name of the class.



Simple Class

Interface


<<Interface>>
Interface Name
+ doMethod (String) : void
+ init (boolean, long) : boolean

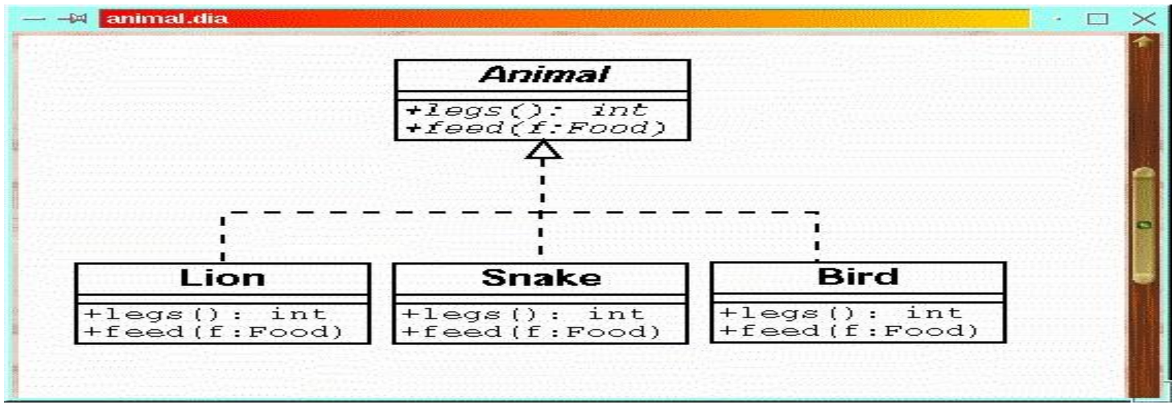
An interface is a class whereby only the various operations are described with the attributes.
An interface is like a class, but has no attributes and none of its operations has implementations.

Abstract Classes

Abstraction is the process of omitting details. We need this to understand complex objects. Modern programming and modelling languages help us by providing the ideas of an *abstract class* and an *abstract operation*.

An *abstract class* is a class that has one or more *abstract* operations. An *abstract operation* is not implemented in its class but in classes that are derived from it. A class with no abstract operations is called a *concrete* class. An abstract class defines an incomplete framework into which you can plug in specific implementations.

For example, the class of Animals may be abstract class and Lion may be derived from it. The Abstract Animal has an operation that returns the number of legs. Lions implement this *legs()* operation as returning the number 4. Similarly, a Snake has 0 legs, and a Bird 2 leg. Here is the UML diagram of the abstract class Animal with abstract operations. The *italic* names indicate



abstract classes and operations in the UML. The diagram also shows that Lion, Snake, and Bird implement the abstract class. The dashed generalization arrow indicates this.

An object that is declared to be in a concrete class (a Snake, Lion, or Bird above) has defined behavior. If the concrete class implements or *relizes* an abstract class then the object must satisfy the abstract operations. So Animal above is realized or implemented by Lion, Snake, and Bird.

•How to Draw a Class Diagram

Class diagrams go hand in hand with object-oriented design. So knowing its basics is a key part of being able to draw good class diagrams.

When required to describe the static view of a system or its functionalities, you would be required to draw a class diagram. Here are the steps you need to follow to create a class diagram.

Step 1: Identify the class names

The first step is to identify the primary objects of the system.

Step 2: Distinguish relationships

Next step is to determine how each of the classes or objects are related to one another. Look out for commonalities and abstractions among them; this will help you when grouping them when drawing the class diagram.

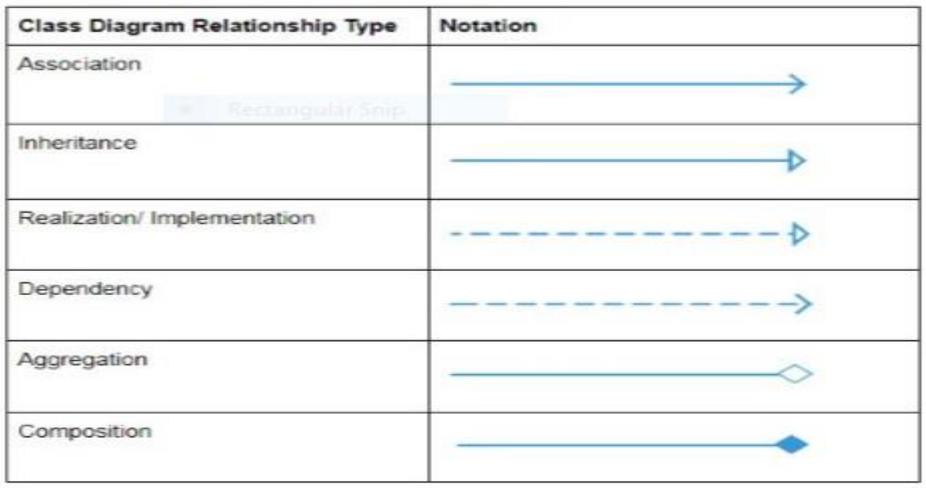
Step 3: Create the Structure

First, add the class names and link them with the appropriate connectors. You can add attributes and functions/ methods/ operations later.

Relationships in Class Diagrams

Relationships in UML are used to represent a connection between structural, behavioral, or grouping things. It is also called a link that describes how two or more things can relate to each other during the execution of a system. Type of UML Relationship are Association, Dependency , Generalization , and Realization.

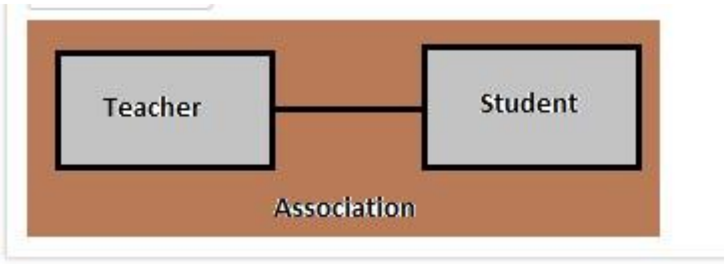
We distinguish six types of relationships with class diagram as follows



Class Diagram Relationship Type	Notation
Association	———→
Inheritance	———▷
Realization/ Implementation	- - - - ▷
Dependency	- - - - →
Aggregation	———◇
Composition	———◆

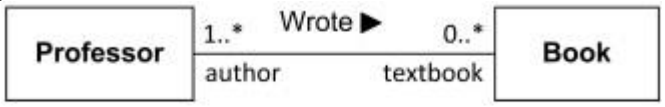
Association

Association represents a relationship between two or more objects where all objects have their own life cycle and there is no owner. We take an example of relationship between Teacher and Student. Many students can have one teacher and one student can have many teachers. But there is no ownership between the objects and both have their own lifecycle. Both can be created and deleted independently.



Other types of relationship in OOP include

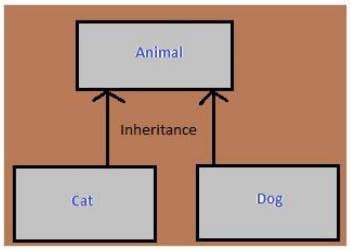
Association is a relationship between classes, which is used to show that instances of class could be either linked to each other or combined logically or physically into some aggregation.
An association is usually drawn as a solid line connecting two classifiers or a single classifier to itself.
Name of the association can be shown somewhere near the middle of the association line but not too close to any of the ends of the line. Each end of the line could be decorated with the name of the association end.



Association **Wrote** between **Professor** and **Book** with association ends **author** and **textbook**.

Inheritance

In OOAD **inheritance** is usually defined as a mechanism by which more specific classes (called subclasses or derived classes) incorporate structure and behavior of more general classes (called superclass's or base classes).

Example: a dog and a cat are types of animal and can hesitate some features the class animal as shown in the diagram below 

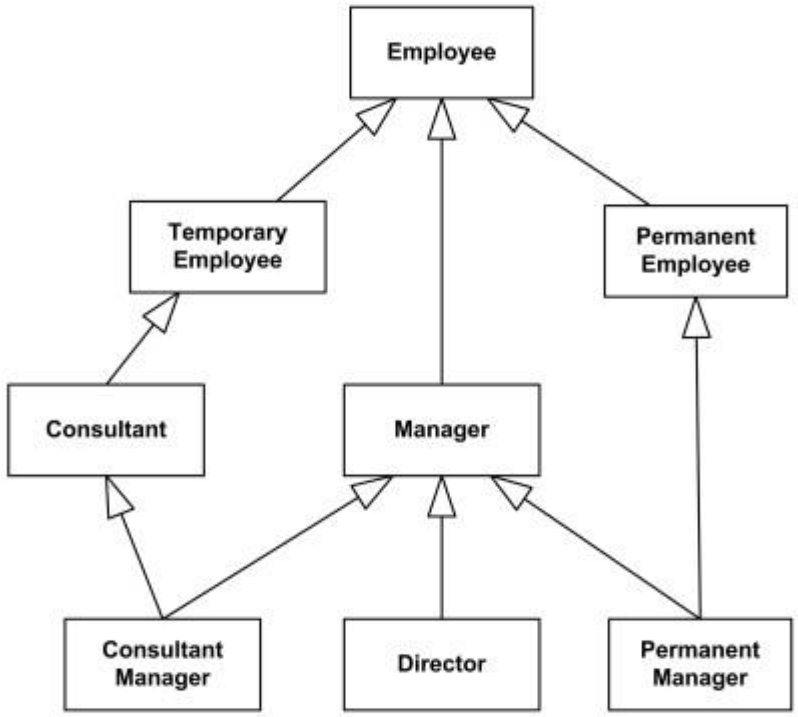
Single Inheritance

When a class inherits from a single base class, it is known as single inheritance.

Multiple Inheritance

Multiple inheritance is implicitly allowed by UML standard, while the standard provides no definition of what it is.

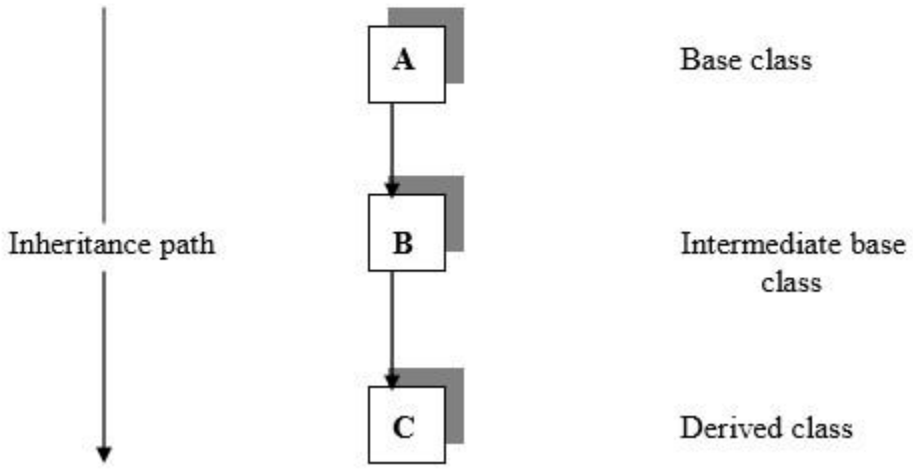
Classifier in UML can have zero, one or many generalization relationships to more general classifiers. In OOAD **multiple inheritance** refers to the ability of a class to inherit behaviors and features from more than one superclass.



Multiple inheritance for Consultant Manager and Permanent Manager - both inherit from two classes

Multilevel Inheritance

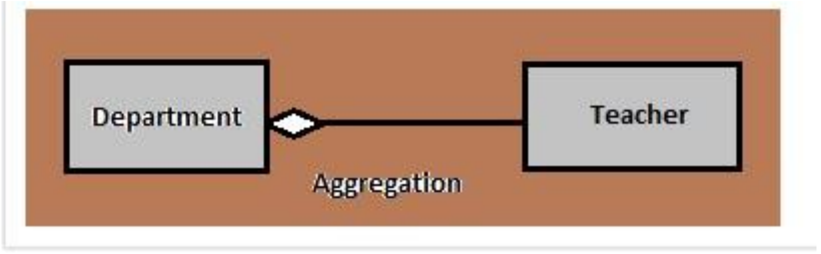
When the inheritance is such that, the class A serves as a base class for a derived class B which in turn serves as a base class for the derived class C. This type of inheritance is called 'MULTILEVEL INHERITENCE'. The class B is known as the 'INTERMEDIATE BASE CLASS' since it provides a link for the inheritance between A and C. The chain ABC is called 'ITNHERITENCE*PATH' for e.g



Aggregation

- ➢ **Aggregation** is a special form of Association where all objects have their own life cycle but there is ownership. This represents **whole-part** or **a-part-of** relationship. This is represented by a **hollow diamond** followed by a line.

We can take an example of relationship between Department and Teacher. A Teacher may belong to many departments. Hence Teacher is a part of many departments. But if we delete a Department, Teacher Object will not destroy.

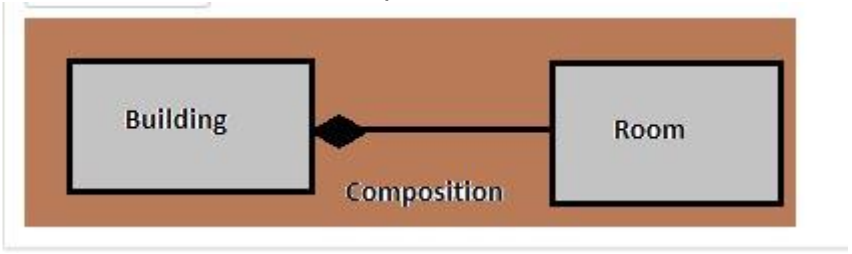


Shared aggregation (**aggregation**) is a binary association between a property and one or more composite objects which group together a set of instances. It is a "weak" form of aggregation when part instance is independent of the composite. Shared aggregation has the following characteristics:

- it is binary association,
- it is asymmetric - only one end of association can be an aggregation,

Composition

- ➢ **Composition** is a special form of Aggregation. In this relationship child objects do not have their life cycle without Parent object. If a parent object is deleted, all its child objects will also be deleted. Composition is based on **PART-OF - death** relationship.



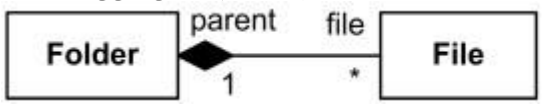
Composite aggregation (**composition**) is a "strong" form of aggregation with the following characteristics:

- it is binary association,
- it is a *whole/part* relationship,
- a part could be included in *at most one* composite (whole) at a time, and
- if a composite (whole) is deleted, all of its composite parts are "normally" deleted with it.

Note, that UML does not define how, when and specific order in which parts of the composite are created. Also, in some cases a part can be removed from a composite before the composite is deleted, and so is not necessarily deleted as part of the composite.

Notation

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



Folder could contain many files, while each File has exactly one Folder parent. If Folder is deleted, all contained Files are deleted as well.

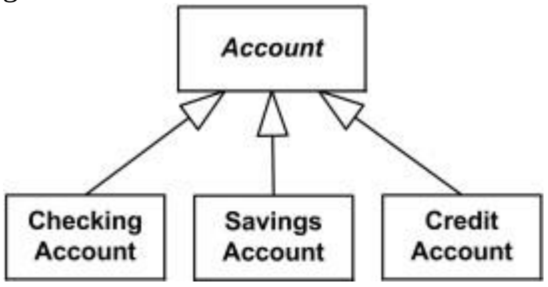
Generalization in UML

A **generalization** is a binary taxonomic (i.e. related to classification) directed relationship between a more general classifier (superclass) and a more specific classifier (subclass).

Each instance of the specific classifier is also an indirect instance of the general classifier, so that we can say "Patient is a Person", "Savings account is an Account", etc. Because of this, **generalization** relationship is also informally called "**Is A**" relationship.

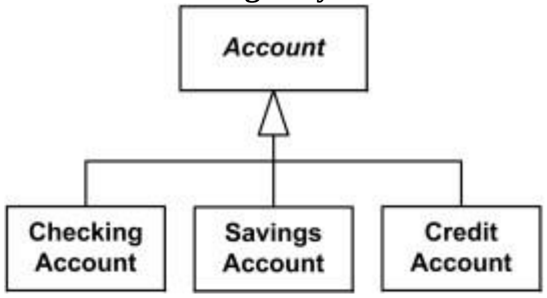
Generalization is **owned by** the specific classifier.

A **generalization** is shown as a line with a hollow triangle as an arrowhead between the symbols representing the involved classifiers. The arrowhead points to the symbol representing the general classifier. This notation is referred to as the **"separate target style."**

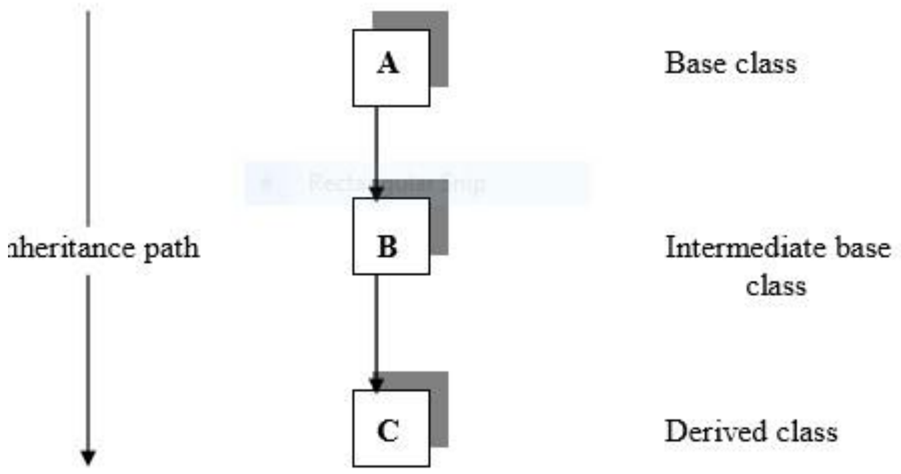


Checking, Savings, and Credit Accounts are **generalized** by Account

Generalization relationships that reference the same general classifier can also be connected together in the **"shared target style."**



Checking, Savings, and Credit Accounts are **generalized**



Dependency in UML

Dependency is a directed relationship which is used to show that some UML element or a set of elements requires, needs or depends on other model elements for **specification** or **implementation**. Because of this, dependency is called a **supplier - client** relationship, where supplier provides something to the client, and thus the client is in some sense incomplete while semantically or structurally dependent on the supplier element(s). Modification of the supplier may impact the client elements.

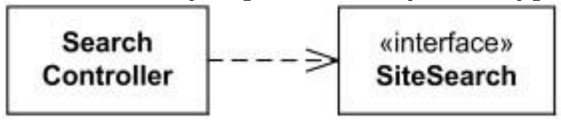
Dependency is a relationship between named elements, which in UML includes a lot of different elements, e.g. classes, interfaces, components, artifacts, packages, etc. There are several kinds of dependencies shown on the diagram below.

Dependency relationship overview diagram - usage, abstraction, deployment.

Usage is a dependency in which one named element (client) requires another named element (supplier) for its full **definition** or **implementation**.

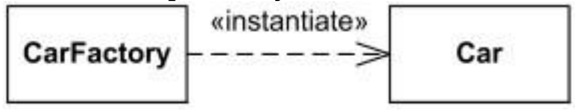
The abstraction relates two elements representing the same concept but at different levels of abstraction.

A dependency is generally shown as a dashed arrow pointing from the **client** (dependent) at the tail to the **supplier** (provider) at the arrowhead. The arrow may be labeled with an optional stereotype and an optional name. Because the direction of the arrow goes opposite to what we would normally expect, I usually stereotype it as client «depends on» supplier.



Class SearchController depends on (requires) SiteSearch interface.

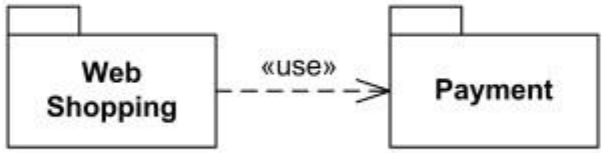
For many years UML specifications provide contradictory example of the dependency shown below. The explanation for the Figure 7.38 of UML 2.4.1 specification states: "In the example below, the Car class has a dependency on the CarFactory class. In this case, the dependency is an instantiate dependency, where the Car class is an instance of the CarFactory class."



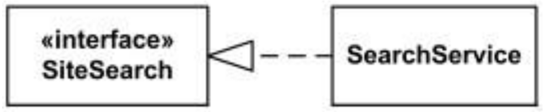
Wrong: Car class has a dependency on the CarFactory class. Right:
CarFactory class depends on the Car class.
This example in fact shows opposite to what UML specification states. CarFactory depends on the Car class. Car class could be defined without the knowledge of CarFactory class, but CarFactory requires Car for its definition because it produces Cars. It is also wrong to say that "... the Car class is an instance of the CarFactory class." The Car class is **instantiated** by the CarFactory class.
It is possible to have a set of elements for the client or supplier. In this case, one or more arrows with their tails on the clients are connected to the tails of one or more arrows with their heads on the suppliers. A small dot can be placed on the junction if desired. A note on the dependency should be attached at the junction point.
Dependency could be used on several kinds of UML diagrams:

- class diagram
- composite structure diagram
- package diagram
- component diagram
- deployment diagram

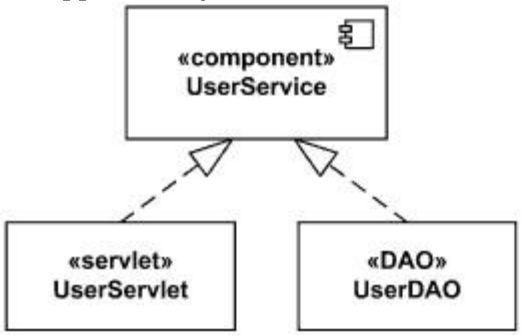
Here's some examples of dependencies:



Web Shopping package uses (depends on) Payment package.



Interface SiteSearch is **realized** (implemented) by SearchService.
Note, that abstraction dependency has a convention to have more abstract element as supplier and more specific or implementation element as client but UML specification also allows to draw it the opposite way.



Component UserService realized by UserServlet and UserDAO.

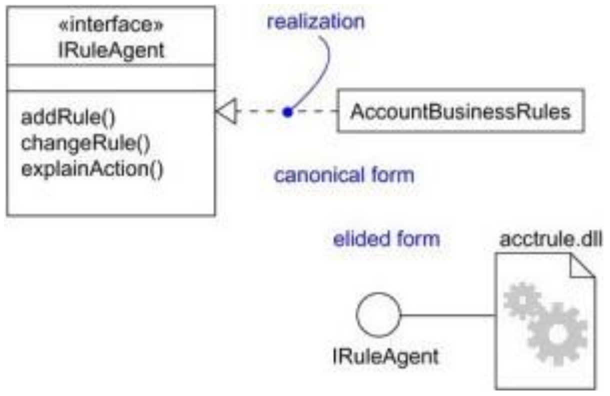
Realization

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

Realization can be represented in two ways:

Using a canonical form Using

an elided form



Visibility in UML

Visibility allows to constrain the usage of a **named element**, either in namespaces or in access to the element. It is used with classes, packages, generalizations, element import, package import.

UML has the following types of **visibility**:

- public
- package
- protected private

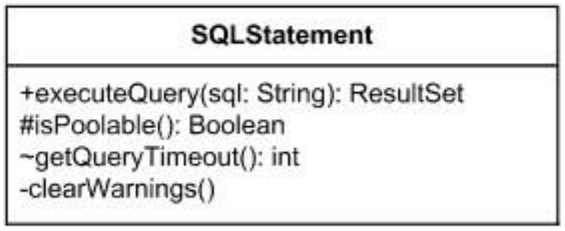
Note, that if a **named element** is not owned by any namespace, then it does not have a visibility.

A **public** element is visible to all elements that can access the contents of the namespace that owns it. **Public** visibility is represented by '+' literal.

A **package** element is owned by a namespace that is not a package, and is visible to elements that are in the same package as its owning namespace. Only named elements that are not owned by packages can be marked as having package visibility. Any element marked as having package visibility is visible to all elements within the nearest enclosing package (given that other owning elements have proper visibility). Outside the nearest enclosing package, an element marked as having package visibility is not visible. **Package** visibility is represented by '~' literal.

A **protected** element is visible to elements that have a generalization relationship to the namespace that owns it. **Protected** visibility is represented by '#' literal.

A **private** element is only visible inside the namespace that owns it. **Private** visibility is represented by '-' literal.

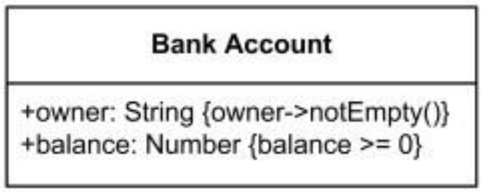


Operation executeQuery is public, isPoolable - protected, getQueryTimeout - with package visibility, and clearWarnings is private.
If some named element appears to have multiple visibilities, for example, by being imported multiple times, public visibility overrides private visibility. If an element is imported twice into the same namespace, once using a public import and another time using a private import, resulting visibility is public.

UML Constraint
A **constraint** is a packageable element which represents some **condition**, **restriction** or **assertion** related to some element (that owns the constraint) or several elements. Constraint is usually specified by a Boolean expression which must evaluate to a true or false. Constraint must be satisfied (i.e. evaluated to **true**) by a correct design of the system. Constraints are commonly used for various elements on class diagrams.
In general there are many possible kinds of **owners** for a constraint. Owning element must have access to the constrained elements to verify constraint. The **owner** of the constraint will determine when the constraint is evaluated. For example, operation can have pre-condition and/or a post-condition constraints.
Constraint could have an optional name, though usually it is anonymous. A constraint is shown as a text string in curly braces according to the following syntax:
constraint ::= '{' [name ':'] boolean-expression '}'
UML specification does not restrict languages which could be used to express constraint. Some examples of **constraint languages** are:

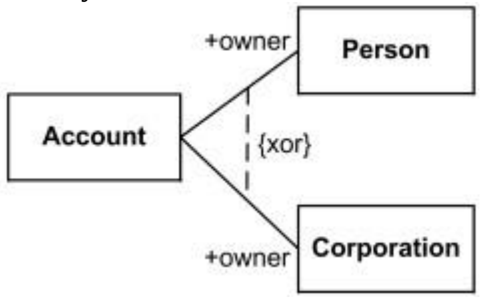
- OCL
- Java
- some machine readable language
- natural language

OCL is a constraint language predefined in UML but if some UML tool is used to draw diagrams, any constraint language supported by that tool could be applied.
For an element whose notation is a text string (such as a class attribute), the constraint string may follow the element text string in curly braces.

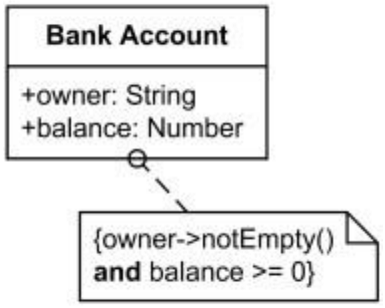

Bank Account
+owner: String {owner->notEmpty()}
+balance: Number {balance >= 0}

Bank account **attribute constraints** - non empty owner and positive balance.

For a constraint that applies to a **single element** (such as a class or an association path), the constraint string may be placed near the symbol for the element, preferably near the name, if any. A UML tool must make it possible to determine the constrained element.
For a constraint that applies to **two elements** (such as two classes or two associations), the constraint may be shown as a **dashed line** between the elements labeled by the constraint string in curly braces.



Account owner is either Person or Corporation, **{xor}** is predefined UML constraint.
If the constraint is shown as a dashed line between two elements, then an arrowhead may be placed on one end. The direction of the arrow is relevant information within the constraint. The element at the tail of the arrow is mapped to the first position and the element at the head of the arrow is mapped to the second position in the constrainedElements collection.
For three or more paths of the same kind (such as generalization paths or association paths), the constraint may be attached to a dashed line crossing all of the paths.
The constraint string may be placed in a **note** symbol (same as used for comments) and attached to each of the symbols for the constrained elements by a dashed line.



Bank account **constraints** - non empty owner and positive balance.

Exercise one

8- Draw a class diagram representing a book defined by the following statement: "A book is composed of a number of parts, which in turn are composed of a number of chapters. Chapters are composed of sections." Focus only on classes and relationships. **(7 marks)**
9- Extend the class diagram of the question above to include the following attributes: **(8 marks)**
    E. abook includes a publisher, publication date, and an ISBN

F. apart includes a title and a number
G. achapter includes a title, a number, and an abstract H. asection includes a title and a number
I.

4) Draw a class diagram representing an airline operates flights. Each airline has an ID. Each flight has an ID a departure airport and an arrival airport: an airport as a unique identifier. Each flight has a pilot and a co-pilot, and it uses an aircraft of a certain type; a flight has also a departure time and an arrival time.
An airline owns a set of aircrafts of different types.
An aircraft can be in a working state or it can be under repair.
In a particular moment an aircraft can be landed or airborne.
A company has a set of pilots: each pilot has an experience level:1 is minimum, 3 is maximum.
A type of aeroplane may need a particular number of pilots, with a different role (e.g.: captain, co-pilot, navigator): there must be at least one captain and one co-pilot, and a captain must have a level 3.

Exercise two

Draw a class diagram representing a university where there are different classrooms, offices and departments. A department has a name and it contains many offices. A person working at the university has a unique ID and can be a professor or an employee.

 A professor can be a full, associate or assistant professor and he/she is enrolled in one department.
Offices and classrooms have a number ID, and a classroom has a number of seats.
Every employee works in an office.

Exercise three

Draw a class diagram representing a system for a movie-shop, in order to handle ordering of movies and browsing of the catalogue of the store, and user subscriptions with rechargeable cards.

Only subscribers are allowed hiring movies with their own card.
 Credit is updated on the card during rent operations.
both users and subscribers can buy a movie and their data are saved in the related order.
When a movie is not available it is ordered.

Exercise four

Draw a class diagram representing a system for management of flights and pilots. An airline operates flights. Each airline has an ID. Each flight has an ID a departure airport and an arrival airport: an airport as a unique identifier. Each flight has a pilot and a co-pilot, and it uses an aircraft of a certain type; a flight has also a departure time and an arrival time. An airline owns a set of aircrafts of different types. An aircraft can be in a working state or it can be under repair. In a particular moment an aircraft can be landed or airborne. A company has a set of pilots: each pilot has an experience level: 1 is minimum, 3 is maximum. A type of aeroplane may need a particular number of pilots, with a different role (e.g.: captain, co-pilot, navigator): there must be at least one captain and one co-pilot, and a captain must have a level 3.

Exercise five

Draw a class diagram representing a bank system which contains data on customers (identified by name and address) and their accounts. Each account has a balance and there are 2 type of accounts: one for savings which offers an interest rate, the other for investments, used to buy stocks. Stocks are bought at a certain quantity for a certain price (ticker) and the bank applies commission on stock orders.