Jaipur Engineering College & Research Centre, Jaipur

Notes
Software Engineering
[3CS4 - 07]

Prepared By:
Manju Vyas
Abhishek Jain
Geerija Lavania

VISION AND MISSION OF INSTITUTE

VISION

To become renowned centre of outcome based learning and work towards academic, professional, cultural and social enrichments of the lives of individual and communities"

MISSION

M1. Focus on evaluation of learning outcomes and motivate students to inculcate research aptitude by project based learning.

M2. Identify areas of focus and provide platform to gain knowledge and solutions based on informed perception of Indian, regional and global needs.

M3. Offer opportunities for interaction between academia and industry.

M4. Develop human potential to its fullest extent so that intellectually capable and imaginatively gifted leaders can emerge in a range of professions.

VISION AND MISSION OF DEPARTMENT

VISION

To become renowned Centre of excellence in computer science and engineering and make competent engineers & professionals with high ethical values prepared for lifelong learning.

MISSION

M1: To impart outcome based education for emerging technologies in the field of computer science and engineering.

M2: To provide opportunities for interaction between academia and industry.

M3: To provide platform for lifelong learning by accepting the change in technologies

M4: To develop aptitude of fulfilling social responsibilities.

COURSE OUTCOMES

CO1) understand the purpose of designing a system and evaluate the various models suitable as per its requirement analysis

CO2) understand and apply software project management, effort estimation and project scheduling.

CO3) formulate requirement analysis, process behaviour and software designing.

CO4) Implement the concept of object oriented analysis modelling with the reference of UML and advance SE tools

Program Outcomes (PO)

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problemsand design system components or processes that meet thespecified needs with appropriate consideration for the public health and safety, andthe cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issuesand the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Educational Objectives (PEO)

1.To provide students with the fundamentals of Engineering Sciences with more emphasis in Computer Science & Engineering by way of analyzing and exploiting engineering challenge

2. To train students with good scientific and engineering knowledge so as to comprehend, analyze, design, and create novel products and solutions for the real life problems.

3. To inculcate professional and ethical attitude, effective communication skills, teamwork skills, multidisciplinary approach, entrepreneurial thinking and an ability to relate engineering issues with social issues.

4. To provide students with an academic environment aware of excellence, leadership, written ethical codes and guidelines, and the self-motivated life-long learning needed for a successful professional career.

5. To prepare students to excel in Industry and Higher education by Educating Students along with High moral values and Knowledge.

MAPPING CO-PO

Cos/POs	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	3	3	3	3	3	2	1	2	1	1	2	3
CO2	3	3	3	3	2	2	1	2	2	2	3	3
CO3	3	3	3	2	2	2	1	2	1	2	2	3
CO4	3	3	3	3	3	1	0	1	1	2	2	3

PSO

PSO1: Ability to interpret and analyze network specific and cyber security issues, automation in real word environment.

PSO2: Ability to Design and Develop Mobile and Web-based applications under realistic constraints.

SYLLABUS

UNIT 1: Introduction, software life-cycle models, software requirements specification, formal requirements specification, verification and validation.

UNIT 2: Software Project Management: Objectives, Resources and their estimation, LOC and FP estimation, effort estimation, COCOMO estimation model, risk analysis, software project scheduling.

UNIT 3: Requirement Analysis: Requirement analysis tasks, Analysis principles. Software prototyping and specification data dictionary, Finite State Machine (FSM) models. **Structured Analysis**: Data and control flow diagrams, control and process specification behavioral modeling

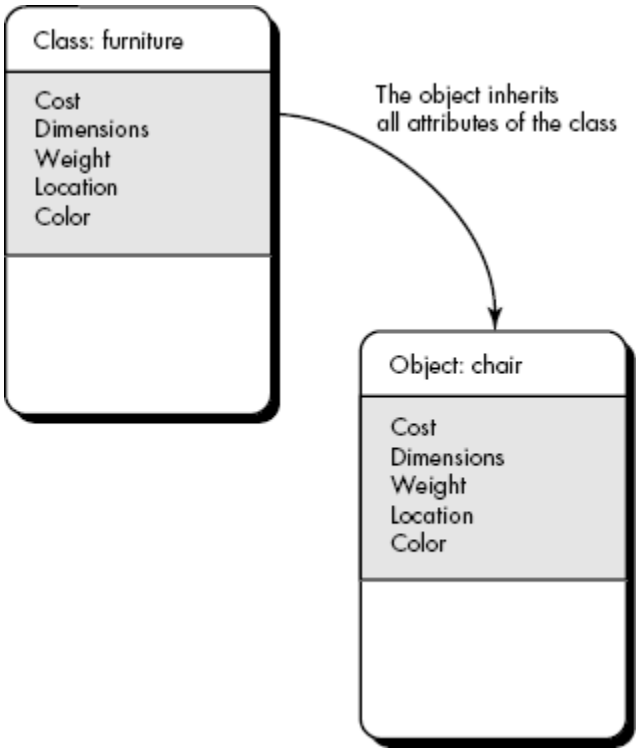
UNIT 4: Software Design: Design fundamentals, Effective modular design: Data architectural and procedural design, design documentation.

UNIT 5: Object Oriented Analysis: Object oriented Analysis Modeling, Data modeling. **Object Oriented Design**: OOD concepts, Class and object relationships, object modularization, Introduction to Unified Modeling Language

UNIT -5 OBJECT –ORIENTED ANALYSIS

Object Oriented Analysis and Design

Software is primarily used to represent real-life players and processes inside a computer. In the past, software was considered to be a collection of information and procedures to transform that information from input to the output format. There was no explicit relationship between the information and the processes that operate on that information. The mapping between software components and their corresponding real-life objects and processes was hidden in the implementation details. There was no mechanism for sharing information and procedures among the objects that had similar properties.



To understand the object-oriented point of view, consider an example of a real world object—the thing you are sitting in right now—a chair. **Chair** is a member (the term *instance*

is also used) of a much larger class of objects that we call **furniture**. A set of generic attributes can be associated with every object in the class **furniture**. For example, all furniture has a cost, dimensions, weight, location, and color, among many possible *attributes.* These apply whether we are talking about a table or a chair, a sofa or an armoire. Because **chair** is a member of **furniture, chair** *inherits* all attributes defined for the class. Once the class has been defined, the attributes can be reused when new instances of the class are created. For example, assume that we were to define a new object called a **chable** (a cross between a chair and a table) that is a member of the class **furniture. Chable** inherits all of the attributes of **furniture**.

Classes and Objects

The fundamental concepts that lead to high-quality design apply equally to systems developed using conventional and object-oriented methods. For this reason, an OO model of computer software must exhibit data and procedural abstractions that lead to effective modularity. A class is an OO concept that encapsulates the data and procedural abstractions required to describe the content and behavior of some real world entity.

The data abstractions (attributes) that describe the class are enclosed by a "wall" of procedural abstractions (called *operations, methods,* or *services*) that are capable of manipulating the data in some way. The only way to reach the attributes (and operate on them) is to go through one of the methods that form the wall. Therefore, the class encapsulates data (inside the wall) and the processing that manipulates the data (the methods that make up the wall). This achieves information hiding and reduces the impact of side effects associated with change. Since the methods tend to manipulate a limited number of attributes, they are cohesive; and because communication occurs only through the methods that make up the "wall," the class tends to be decoupled from other elements of a system. All of these design characteristics lead to highquality software.

Encapsulation, Inheritance, and Polymorphism

As we have already noted, the OO class and the objects spawned from the class encapsulate data and the operations that work on the data in a single package. This provides a number of important benefits:

• The internal implementation details of data and procedures are hidden from the outside world (information hiding). This reduces the propagation of side effects when changes occur.

• Data structures and the operations that manipulate them are merged in a single named entity—the class. This facilitates component reuse.

• Interfaces among encapsulated objects are simplified. An object that sends a message need not be concerned with the details of internal data structures.

Hence, interfacing is simplified and the system coupling tends to be reduced. Inheritance is one of the key differentiators between conventional and OO systems. A subclass **Y** inherits all of the attributes and operations associated with its superclass, **X**. This means that all data structures and algorithms originally designed and implemented for **X** are immediately available for **Y**—no further work need be done. Reuse has been accomplished directly.

Object Oriented Design – Why?

There was a need for technology that could bridge the gap between the real-life objects and their counter-parts in the computer. Object oriented technology evolved to bridge the gap. Object-oriented technology helps in software modeling of real-life objects in a direct and explicit fashion, by encapsulating data and processes related to a real-life object or process in a single software entity. It also provides a mechanism through which the objects can inherit properties from their ancestors, just like real-life objects.

A complex system that works is invariably found to have evolved from a simple system that worked. The structure of a system also plays a very important role. It is likely that we understand only those systems that have hierarchical structure and where intracomponent linkages are generally stronger than inter-component linkages. This leads to loose coupling, high cohesion and ultimately more maintainability, which are the basic design considerations. Instead of being a collection of loosely bound data structures and functions, an object-oriented software system consists of objects that are, generally, hierarchical, highly cohesive, and loosely coupled. Some of the key advantages that make the object-oriented technology significantly attractive than other technologies include:

- Clarity and understandability of the system, as object-oriented approach is closer to the working of human cognition.

- Reusability of code resulting from low inter-dependence among objects, and provision of generalization and specialization through inheritance.

- Reduced effort in maintenance and enhancement, resulting from inheritance, encapsulation, low coupling, and high cohesion.

Object Oriented Design Components - What?

The Object and the Class

The basic unit of object-oriented design is an object. An object can be defined as a tangible entity that exhibits some well-defined behavior. An object represents an individual,

identifiable item, unit, or entity, either real or abstract, with a well-defined role in the problem domain. An object has a state, behavior, and identity. The state of an object encompasses all of the properties of the object and their current values. A property is an inherent or distinctive characteristic. Properties are usually static. All properties have some value. The state of an object is encapsulated within the object.

Behavior is how an object acts and reacts in terms of its state changes and message passing. The behavior of an object is completely defined by its actions. A message is some action that one object performs upon another in order to elicit a reaction. The operations that clients may perform upon an object are called methods. The structure and behavior of similar objects are defined in their common class. A class represents an abstraction - the essence or the template of an object, specifies an interface (the outside view - the public part) and defines an implementation (the inside view - the private part). The interface primarily consists of the declaration of all the operations applicable to instances of this class. The implementation of a class primarily consists of the implementation of all the operations defined in the interface of the class.

Classification

The most important and critical stage in OOA and OOD is the appropriate classification of objects into groups and classes. Proper classification requires looking at the problem from different angles and with an open mind. When looked at from different perspectives and analyzed with different sets of characteristics, the same object can be classified into different categories.

The Object Model

The elements of object-oriented design are collectively called the Object Model. The

object model encompasses the principles of abstraction, encapsulation, and hierarchy

or inheritance.

Abstraction is an extremely powerful technique for dealing with complexity. Unable to master the entirety of a complex object, we ignore its essential details, dealing instead with generalized, and the idealized model of the object. An abstraction focuses on the outside view of an object and hence serves to separate an object's external behavior from its implementation. Deciding upon the right set of abstractions for a given domain is the central problem in object-oriented design.

Relationship Among Objects

The object model presents a static view of the system and illustrates how different objects collaborate with one another through patterns of interaction. Inheritance, association and aggregation are the three inter-object relationships specified by the object model. Inheritance defines a "kind of" hierarchy among classes. By inheritance, we specify generalization/specialization relationship among objects. In this relationship, a class (called the subclass) shares the structure and behavior defined in another class (called the super class). A subclass augments or redefines the existing structure and behavior of its super class. By classifying objects into groups of related abstractions, we come to explicitly distinguish the common and distinct properties of different objects, which further helps us to master their inherent complexity. Identifying the hierarchy within a complex system requires the discovery of patterns among many objects.

Object Oriented Analysis

The intent of OOA is to define all classes, their relationships, and their behavior.

A number of tasks must occur:

1) Static Model

a) Identify classes (i.e. attributes and methods are defined)

b) Specify class hierarchy

c) Identify object-to-object relationships

d) Model the object behavior

2) Dynamic Model

a) Scenario Diagrams

Object Oriented Design

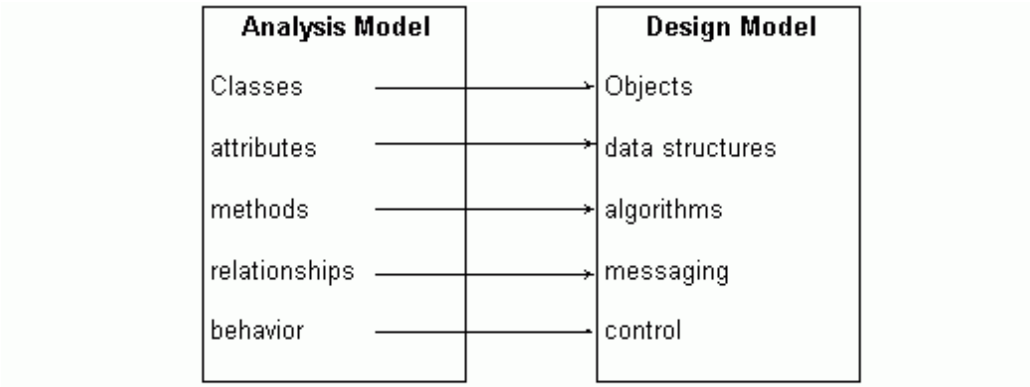
OOD transforms the analysis model into design model that serves as a blueprint for software construction. OOD results in a design that achieves a number of different levels of modularity. The four layers of the OO design pyramid are:

1) **The subsystem layer**: Contains a representation of each of the subsystems that enable the software to achieve its customer-defined requirements and to implement the technical infrastructure that supports customer requirements.

2) **The class and object layer**: Contains the class hierarchies that enable the system to be created using generalization and increasingly more targeted specializations. The layer also contains design representations for each object.

3) **The message layer**: Contains the details that enable each object to communicate with its collaborators. This layer establishes the external and internal interfaces for the system.

4) **The responsibility layer**: Contains the data structures and algorithmic design for all attributes and operations for each object.



Translating the analysis model into a design model during object design

OOA—object-oriented analysis — is based upon concepts that we first learned in kindergarten:

objects and attributes, classes and members, wholes and parts. Why it has taken us so long to apply these concepts to the analysis and specification of information systems is anyone's guess.

OOA is grounded in a set of basic principles that were introduced in earlier chapters.

In order to build an analysis model, five basic principles were applied:

(1) the information domain is modeled.

(2) function is described.

(3) behavior is represented.

(4) data, functional, and behavioral models are partitioned to expose greater detail.

(5) early models represent the essence of the problem while later models provide implementation details. These principles form the foundation for the approach to OOA presented in this chapter.

A Unified Approach to OOA

Over the past decade, Grady Booch, James Rumbaugh, and Ivar Jacobson have collaborated to combine the best features of their individual object-oriented analysis and design methods into a unified method. The result, called the *Unified Modeling Language* (UML), has become widely used throughout the industry.

UML allows a software engineer to express an analysis model using a modeling notation that is governed by a set of syntactic, semantic, and pragmatic rules. Eriksson and Penker [ERI98] explain these rules in the following way:

The syntax tells us how the symbols should look and how the symbols are combined. The syntax is compared to words in natural language; it is important to know how to spell them correctly and how to put different words together to form a sentence. The semantic rules tell us what each symbol means and how it should be interpreted by itself and in the context of other symbols; they are compared to the meanings of words in a natural language.

The pragmatic rules define the intentions of the symbols through which the purpose of the model is achieved and becomes understandable for others. This corresponds in natural language to the rules for constructing sentences that are clear and understandable. In UML, a system is represented using five different "views" that describe the system from distinctly different perspectives. Each view is defined by a set of diagrams.

The following views are present in UML:

User model view. This view represents the system (product) from the user's (called *actors* in UML) perspective. The use-case is the modeling approach of choice for the user model view. This important analysis representation describes a usage scenario from the end-user's perspective and has been discussed earlier.

Structural model view. Data and functionality are viewed from inside the system. That is, static structure (classes, objects, and relationships) is modeled.

Behavioral model view. This part of the analysis model represents the dynamic or behavioral aspects of the system. It also depicts the interactions or collaborations between various structural elements described in the user model and structural model views.

Implementation model view. The structural and behavioral aspects of the system are represented as they are to be built.

Environment model view. The structural and behavioral aspects of the environment in which the system is to be implemented are represented.

THE OOA PROCESS

The OOA process does not begin with a concern for objects. Rather, it begins with an understanding of the manner in which the system will be used—by people, if the system is human-interactive; by machines, if the system is involved in process control; or by other programs, if the system coordinates and controls applications. Once the scenario of usage has been defined, the modeling of the software begins. The sections that follow define a series of techniques that may be used to gather basic customer requirements and then define an analysis model for an object oriented system.

Use-Cases

As we noted in earlier chapter, use-cases model the system from the end-user's point of view. Created during requirements elicitation, use-cases should achieve the following objectives:

• To define the functional and operational requirements of the system (product) by defining a scenario of usage that is agreed upon by the end-user and the software engineering team.

• To provide a clear and unambiguous description of how the end-user and the system interact with one another.

• To provide a basis for validation testing.During OOA, use-cases serve as the basis for the first element of the analysis model. Using UML notation, a diagrammatic representation of a use-case, called a *use-case diagram,* can be created. Like many elements of the analysis model, the use-case diagram can be represented at many levels of abstraction. The use-case diagram contains actors and use-cases. *Actors* are entities that interact with the system. They can be human users or other machines or systems that have defined interfaces to the software.

Class-Responsibility-Collaborator Modeling

Once basic usage scenarios have been developed for the system, it is time to identify candidate classes and indicate their responsibilities and collaborations. Classresponsibility-collaborator (CRC) modeling [WIR90] provides a simple means for identifying and organizing the classes that are relevant to system or product requirements.

Ambler describes CRC modeling in the following way:

A CRC model is really a collection of standard index cards that represent classes. The cards are divided into three sections. Along the top of the card you write the name of the class. In the body of the card you list the class responsibilities on the left and the collaborators on the right.

In reality, the CRC model may make use of actual or virtual index cards. The intent is to develop an organized representation of classes. Responsibilities are the attributes and operations that are relevant for the class. Stated simply, a responsibility is “anything the class knows or does” [AMB95]. Collaborators are those classes that are required to provide a class with the information needed to complete a responsibility.

In general, collaboration implies either a request for information or a request for some action.

Classes

To summarize, objects manifest themselves in a variety of forms: external entities, things, occurrences, or events; roles; organizational units; places; or structures. One technique for identifying these in the context of a software problem is to perform a grammatical parse on the processing narrative for the system. All nouns become potential objects. However, not every potential object makes the cut.

Six selection characteristics were defined:

1. Retained information. The potential object will be useful during analysis only if information about it must be remembered so that the system can function.

2. Needed services. The potential object must have a set of identifiable operations that can change the value of its attributes in some way.

3. Multiple attributes. During requirements analysis, the focus should be on "major" information; an object with a single attribute may, in fact, be useful during design but is probably better represented as an attribute of another object during the analysis activity.

4. Common attributes. A set of attributes can be defined for the potential object and these attributes apply to all occurrences of the object.

5. Common operations. A set of operations can be defined for the potential object and these operations apply to all occurrences of the object.

6. Essential requirements. External entities that appear in the problem space and produce or consume information that is essential to the operation of any solution for the system will almost always be defined as objects in the requirements model.

Class name:	
Class type: (e.g., device, property, role, event)	
Class characteristic: (e.g., tangible, atomic, concurrent)	
responsibilities:	collaborations:

A CRC MODEL INDEX CARD

A potential object should satisfy all six of these selection characteristics if it is to be considered for inclusion in the CRC model.

System intelligence should be evenly distributed.

Every application encompasses a certain degree of intelligence; that is, what the system knows and what it can do. This intelligence can be distributed across classes in a number of different ways. “Dumb” classes (those that have few responsibilities) can be modeled to act as servants to a few “smart” classes (those having many responsibilities. Although this approach makes the flow of control in a system straightforward, it has a few disadvantages:

(1) It concentrates all intelligence within a few classes, making changes more difficult

(2) it tends to require more classes, hence more development effort.

Therefore, system intelligence should be evenly distributed across the classes in an application. Because each object knows about and does only a few things (that are generally well focused), the cohesiveness of the system is improved. In addition, side effects due to change tend to be dampened because system intelligence has been decoupled across many objects.

Each responsibility should be stated as generally as possible.

This guideline implies that general responsibilities (both attributes and operations) should reside high in the class hierarchy (because they are generic, they will apply to all subclasses). In addition, polymorphism should beused in an effort to define operations that generally apply to the superclass but are implemented differently in each of the subclasses.

Information and the behavior related to it should reside within the same class.

This achieves the OO principle that we have called *encapsulation* .Data and the processes that manipulate the data should be packaged as a cohesive unit.

Information about one thing should be localized with a single class, not distributed across multiple classes.

A single class should take on the responsibility for storing and manipulating a specific type of information. This responsibility should not, in general, be shared across a number of classes.

If information is distributed, software becomes more difficult to maintain and more challenging to test.

Responsibilities should be shared among related classes, when appropriate.

There are many cases in which a variety of related objects must all exhibit the same behavior at the same time. As an example, consider a video game that must display the following objects: player, player-body, player-arms, player-legs, player-head. Each of these objects has its own attributes (e.g., position, orientation, color, speed) and all must be updated and displayed as the user manipulates a joy stick. The responsibilities *update* and *display* must therefore be shared by each of the objects noted. Player knows when something has changed and *update* is required. It collaborates with the other objects to achieve a new position or orientation, but each object controls its own display.

Object-oriented analysis methods enable a software engineer to model a problem by representing both static and dynamic characteristics of classes and their relationships as the primary modeling components. Like earlier OO analysis methods, the Unified Modeling Language builds an analysis model that has the following characteristics:

(1) representation of classes and class hierarchies.

(2) creation of objectrelationship models

(3) derivation of object-behavior models.

Analysis for object-oriented systems occurs at many different levels of abstraction. At the business or enterprise level, the techniques associated with OOA can be coupled with a business process engineering approach. This technique is often called domain analysis. At an application level, the object model focuses on specific customer requirements as those

requirements affect the application to be built. The OOA process begins with the definition of use-cases—scenarios that describe how the OO system is to be used. The class-responsibility-collaborator modeling technique is then applied to document classes and their attributes and operations. It also provides an initial view of the collaborations that occur among objects. The next step in the OOA process is classification of objects and the creation of a class hierarchy.

Subsystems (packages) can be used to encapsulate related objects. The objectrelationship model provides an indication of how classes are connected to one another, and the object-behavior model indicates the behavior of individual objects and the overall behavior of the OO system.

UML – Overview:

UML was created by the Object Management Group (OMG) and UML 1.0 specification draft was proposed to the OMG in January 1997.

OMG is continuously making efforts to create a truly industry standard.

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

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

Goals of UML

A picture is worth a thousand words, this idiom absolutely fits describing UML. Object-oriented concepts were introduced much earlier than UML. At that point of time, there were no standard methodologies to organize and consolidate the object-oriented development. It was then that UML came into picture.

There are a number of goals for developing UML but the most important is to define some general purpose modeling language, which all modelers can use and it also needs to be made simple to understand and use.

UML diagrams are not only made for developers but also for business users, common people, and anybody interested to understand the system. The system can be a software or non-software system. Thus it must be clear that UML is not a development method rather it accompanies with processes to make it a successful system.

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

A Conceptual Model of UML

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

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

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

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

Object-Oriented Concepts

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

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

Objects are the real-world entities that exist around us and the basic concepts such as abstraction, encapsulation, inheritance, and polymorphism all can be represented using UML.

UML is powerful enough to represent all the concepts that exist in object-oriented analysis and design. UML diagrams are representation of object-oriented concepts only. Thus, before learning UML, it becomes important to understand OO concept in detail.

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

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

OO Analysis and Design

OO can be defined as an investigation and to be more specific, it is the investigation of objects. Design means collaboration of identified objects.

Thus, it is important to understand the OO analysis and design concepts. The most important purpose of OO analysis is to identify objects of a system to be designed. This analysis is also done for an existing system. Now an efficient analysis is only possible when we are able to start thinking in a way where objects can be identified. After identifying the objects, their relationships are identified and finally the design is produced.

The purpose of OO analysis and design can described as −

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

There are three basic steps where the OO concepts are applied and implemented. The steps can be defined as

OO Analysis → OO Design → OO implementation using OO languages

The above three points can be described in detail as −

- During OO analysis, the most important purpose is to identify objects and describe them in a proper way. If these objects are identified efficiently, then the next job of design is easy. The objects should be identified with responsibilities. Responsibilities are the functions performed by the object. Each and every object has some type of responsibilities to be performed. When these responsibilities are collaborated, the purpose of the system is fulfilled.

- The second phase is OO design. During this phase, emphasis is placed on the requirements and their fulfilment. In this stage, the objects are collaborated according to their intended association. After the association is complete, the design is also complete.

- The third phase is OO implementation. In this phase, the design is implemented using OO languages such as Java, C++, etc.

Role of UML in OO Design

UML is a modeling language used to model software and non-software systems. Although UML is used for non-software systems, the emphasis is on modeling OO software applications. Most of the UML diagrams discussed so far are used to model different aspects such as static, dynamic, etc. Now whatever be the aspect, the artifacts are nothing but objects.

If we look into class diagram, object diagram, collaboration diagram, interaction diagrams all would basically be designed based on the objects.

Hence, the relation between OO design and UML is very important to understand. The OO design is transformed into UML diagrams according to the requirement. Before understanding the UML in detail, the OO concept should be learned properly. Once the OO analysis and design is done, the next step is very easy. The input from OO analysis and design is the input to UML diagrams.

UML - Building Blocks

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

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

This chapter describes all the UML building blocks. The building blocks of UML can be defined as −

- Things

- Relationships
- Diagrams

Things

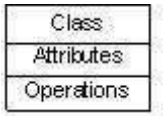
Things are the most important building blocks of UML. Things can be −

- Structural
- Behavioral
- Grouping
- Annotational

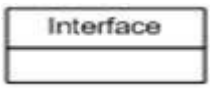
Structural Things

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

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



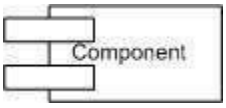
Interface − Interface defines a set of operations, which specify the responsibility of a class.



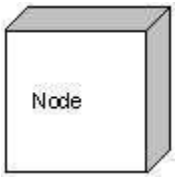
Collaboration −Collaboration defines an interaction between elements.

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

Component −Component describes the physical part of a system.



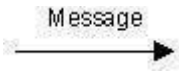
Node − A node can be defined as a physical element that exists at run time.



Behavioral Things

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

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



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

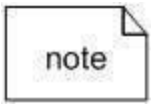
Grouping Things

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

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

Annotational Things

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



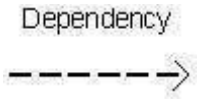
Relationship

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

There are four kinds of relationships available.

Dependency

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

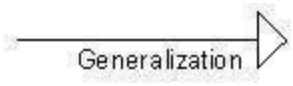


Association

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

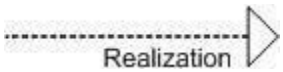
Generalization

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



Realization

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



UML Diagrams

UML diagrams are the ultimate output of the entire discussion. All the elements, relationships are used to make a complete UML diagram and the diagram represents a system.

The visual effect of the UML diagram is the most important part of the entire process. All the other elements are used to make it complete.

UML includes the following nine diagrams, the details of which are described in the subsequent chapters.

- Class diagram
- Object diagram
- Use case diagram
- Sequence diagram

- Collaboration diagram
- Activity diagram
- Statechart diagram
- Deployment diagram
- Component diagram

UML - Architecture

Any real-world system is used by different users. The users can be developers, testers, business people, analysts, and many more. Hence, before designing a system, the architecture is made with different perspectives in mind. The most important part is to visualize the system from the perspective of different viewers. The better we understand the better we can build the system.

UML plays an important role in defining different perspectives of a system. These perspectives are −

- Design
- Implementation
- Process
- Deployment

The center is the **Use Case** view which connects all these four. A **Use Case** represents the functionality of the system. Hence, other perspectives are connected with use case.

Design of a system consists of classes, interfaces, and collaboration. UML provides class diagram, object diagram to support this.

Implementation defines the components assembled together to make a complete physical system. UML component diagram is used to support the implementation perspective.

Process defines the flow of the system. Hence, the same elements as used in Design are also used to support this perspective.

Deployment represents the physical nodes of the system that forms the hardware. UML deployment diagram is used to support this perspective.

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 object-oriented languages.

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

Purpose of Class Diagrams

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

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

The purpose of the class diagram can be summarized as −

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

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.

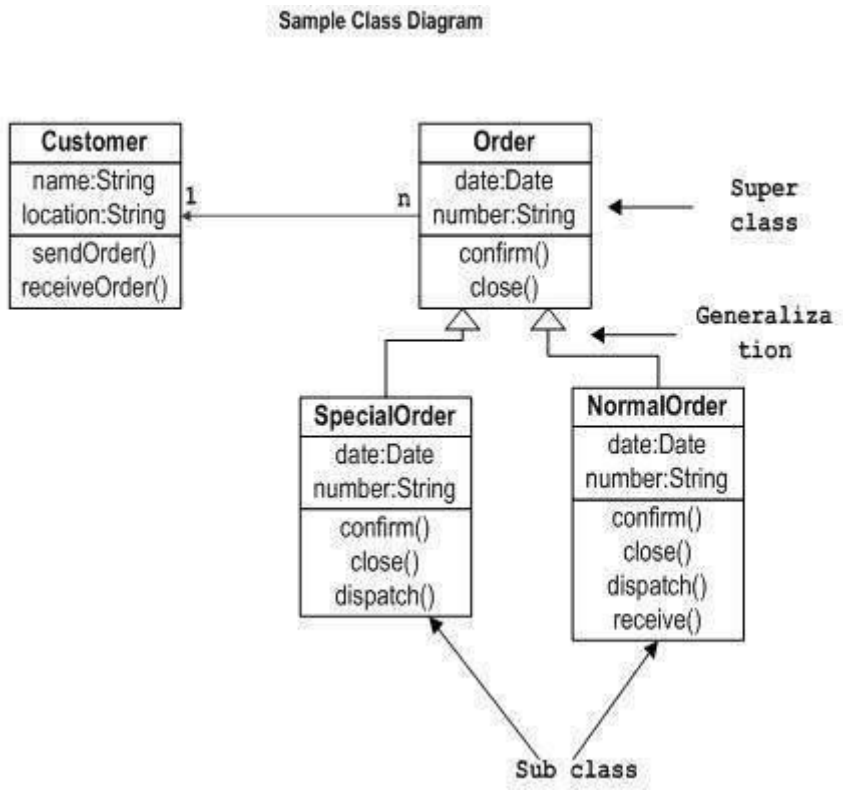
The following points should be remembered while drawing a class diagram −

- 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

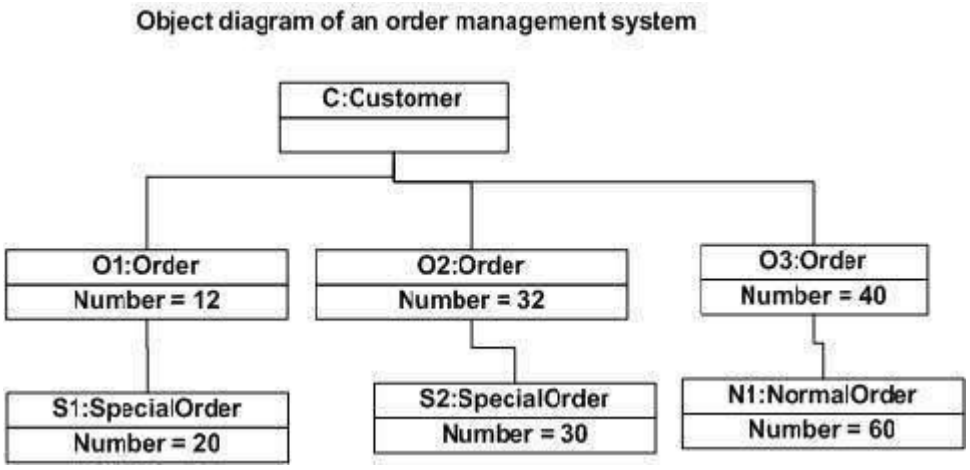
Now the customer object (C) is associated with three order objects (O1, O2, and O3). These order objects are associated with special order and normal order objects (S1, S2, and N1). The customer has the following three orders with different numbers (12, 32 and 40) for the particular time considered.

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



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.

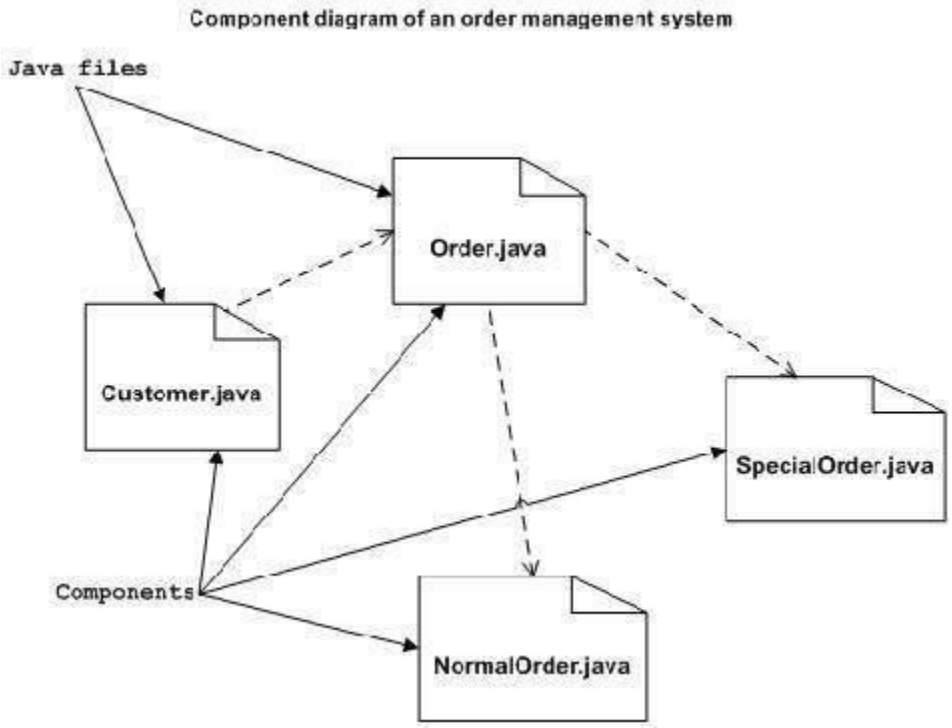
After identifying the artifacts, the following points need to be kept in mind.

- 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

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

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

Purpose of Deployment Diagrams

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

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

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

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

The purpose of deployment diagrams can be described as −

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

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 −

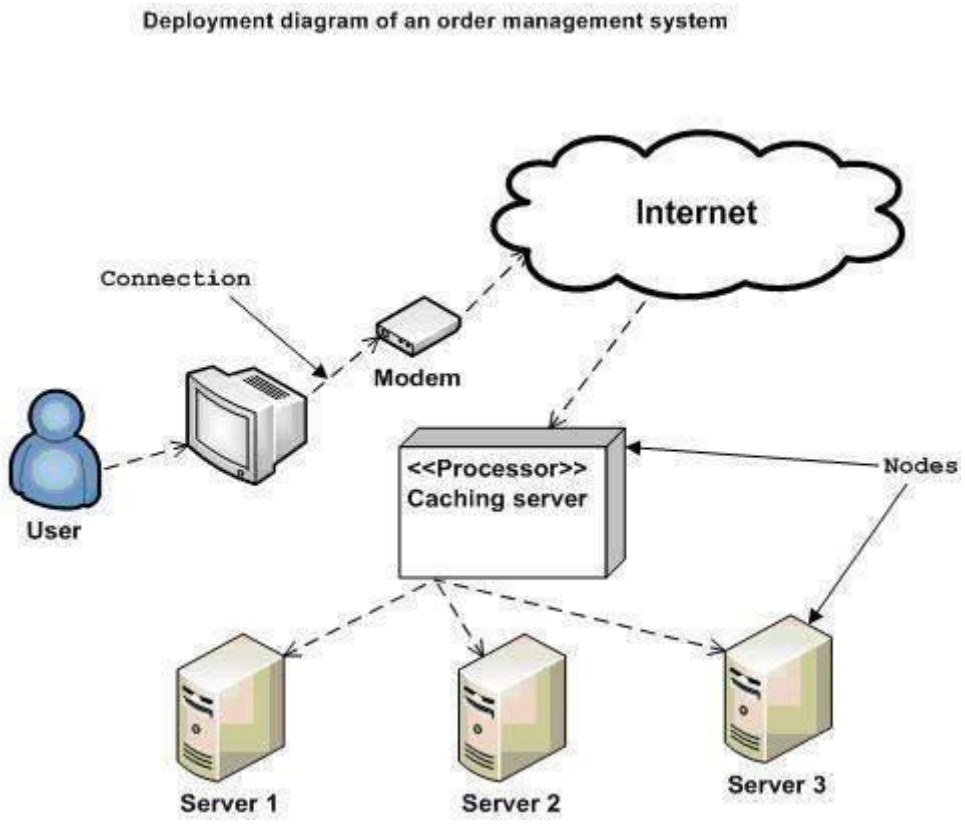
- Nodes
- Relationships among nodes

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.



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.

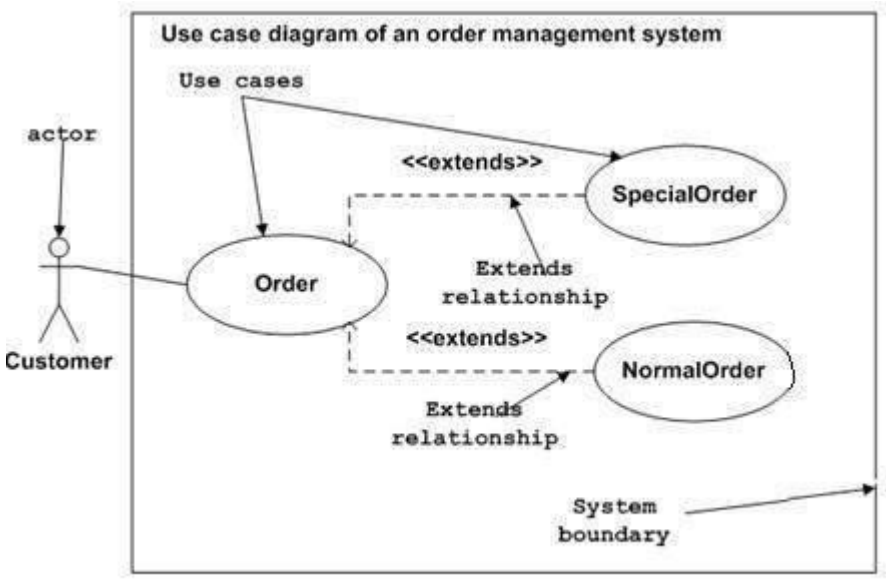


Figure: Sample Use Case diagram

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.

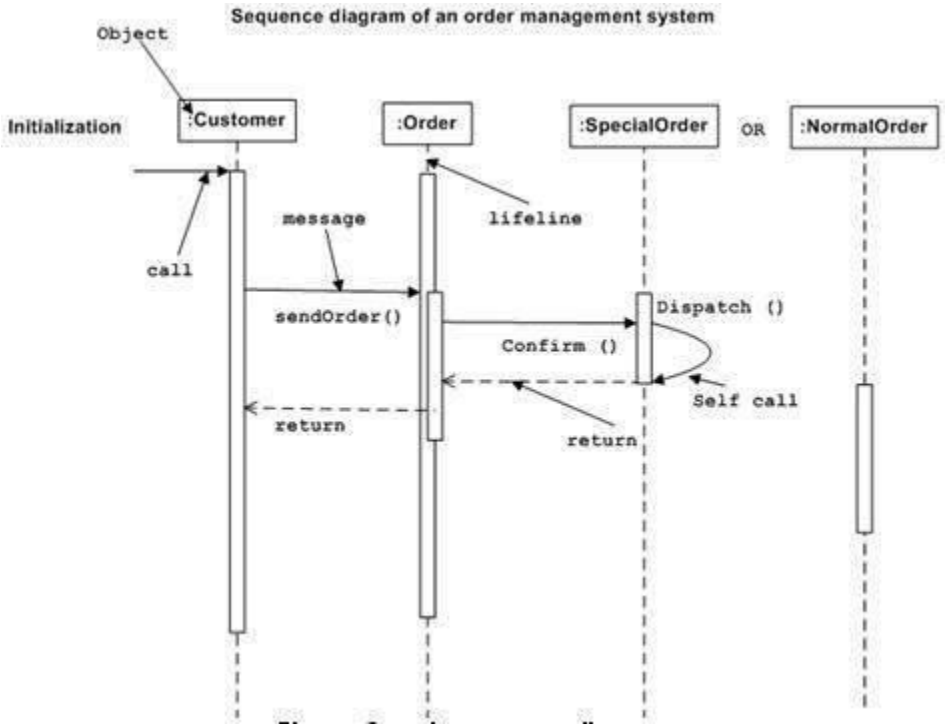
Following are two interaction diagrams modeling the order management system. The first diagram is a sequence diagram and the second is a collaboration diagram

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.

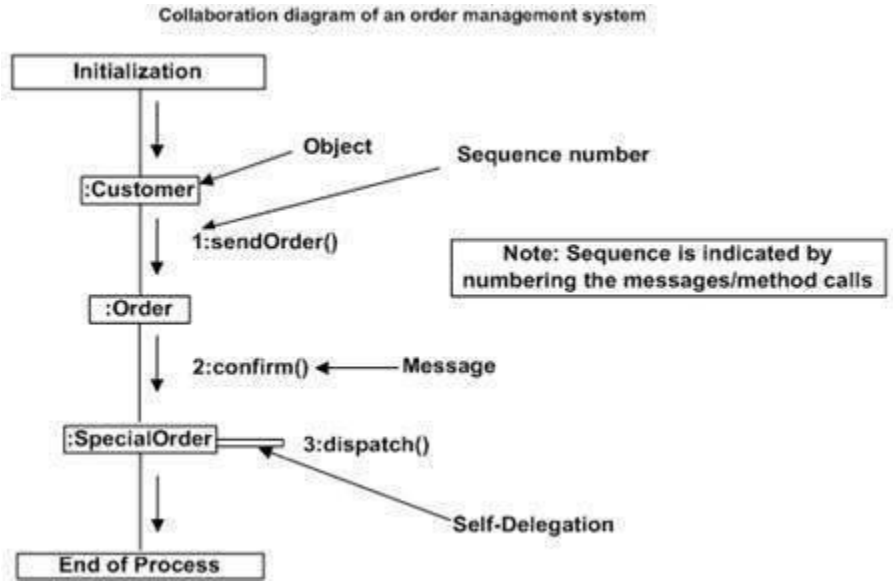


The Collaboration Diagram

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 −

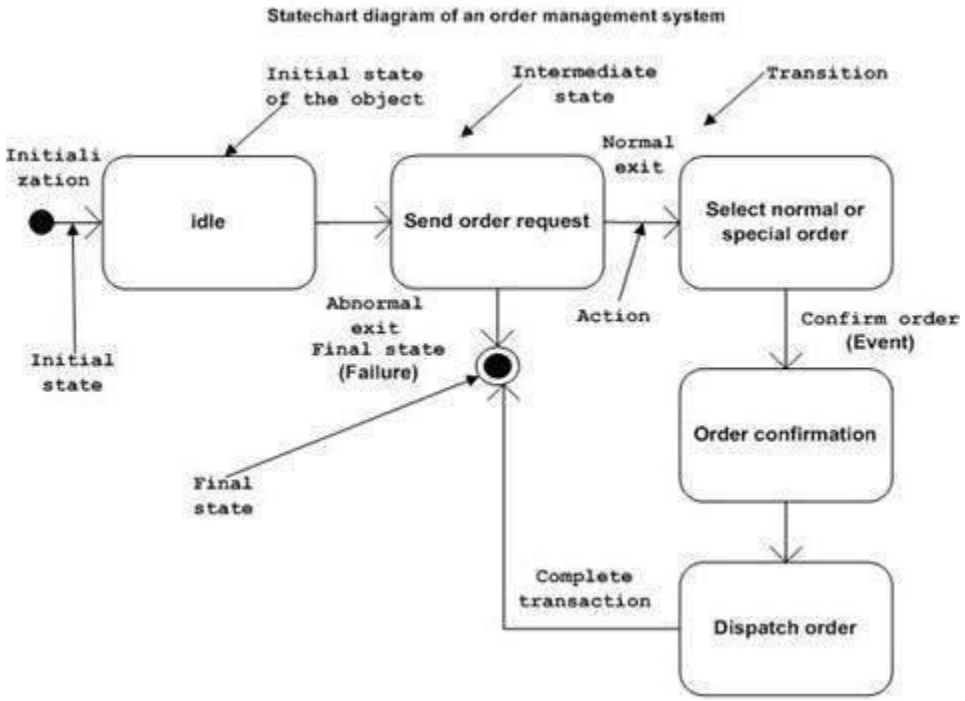
- Identify the important objects to be analyzed.
- Identify the states.
- Identify the events.

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.

UML - Activity Diagrams

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

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

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

Purpose of Activity Diagrams

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

Activity is a particular operation of the system. Activity diagrams are not only used for visualizing the dynamic nature of a system, but they are also used to construct the executable

system by using forward and reverse engineering techniques. The only missing thing in the activity diagram is the message part.

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

The purpose of an activity diagram can be described as −

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

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

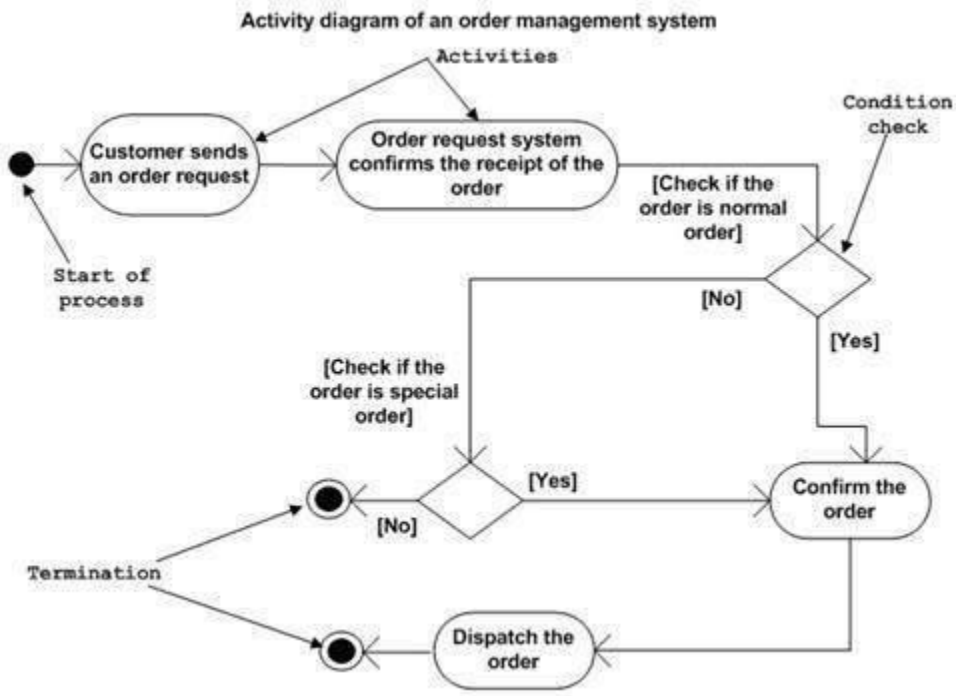
Once the above-mentioned parameters are identified, we need to make a mental layout of the entire flow. This mental layout is then transformed into an activity diagram.

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.



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.

UML 2.0 - Overview

UML 2.0 is totally a different dimension in the world of Unified Modeling Language. It is more complex and extensive in nature. The extent of documentation has also increased compared to UML 1.5 version. UML 2.0 has added new features so that its usage can be more extensive.

UML 2.0 adds the definition of formal and completely defined semantics. This new possibility can be utilized for the development of models and the corresponding systems can be generated from these models. However, to utilize this new dimension, a considerable effort has to be made to acquire knowledge.

New Dimensions in UML 2.0

The structure and documentation of UML was completely revised in the latest version of UML 2.0. There are now two documents available that describe UML −

- UML 2.0 Infrastructure defines the basic constructs of the language on which UML is based. This section is not directly relevant to the users of UML. This is directed more towards the developers of modeling tools. This area is not in the scope of this tutorial.

- UML 2.0 Superstructure defines the user constructs of UML 2.0. It means those elements of UML that the users will use at the immediate level. This is the main focus for the user community of UML.

This revision of UML was created to fulfil a goal to restructure and refine UML so that usability, implementation, and adaptation are simplified.

UML infrastructure is used to −

- Provide a reusable meta-language core. This is used to define UML itself.
- Provide mechanisms to adjustment the language.

UML superstructure is used to −

- Provide better support for component-based development.
- Improve constructs for the specification of architecture.
- Provide better options for the modeling of behavior.

The important point to note is the major divisions described above. These divisions are used to increase the usability of UML and define a clear understanding of its usage.

There is another dimension which is already proposed in this new version. It is a proposal for a completely new Object Constraint Language (OCL) and Diagram Interchange. These features all together form the complete UML 2.0 package.

Modeling Diagrams in UML 2.0

Modeling Interactions

The interaction diagrams described in UML 2.0 is different than the earlier versions. However, the basic concept remains the same as the earlier version. The major difference is the enhancement and additional features added to the diagrams in UML 2.0.

UML 2.0 models object interaction in the following four different ways.

- **Sequence diagram** is a time dependent view of the interaction between objects to accomplish a behavioral goal of the system. The time sequence is similar to the earlier version of sequence diagram. An interaction may be designed at any level of abstraction within the system design, from subsystem interactions to instancelevel.

- **Communication diagram** is a new name added in UML 2.0. Communication diagram is a structural view of the messaging between objects, taken from the Collaboration diagram concept of UML 1.4 and earlier versions. This can be defined as a modified version of collaboration diagram.

- **Interaction Overview diagram** is also a new addition in UML 2.0. An Interaction Overview diagram describes a high-level view of a group of interactions combined into a logic sequence, including flow-control logic to navigate between the interactions.

- **Timing diagram** is also added in UML 2.0. It is an optional diagram designed to specify the time constraints on the messages sent and received in the course of an interaction.

From the above description, it is important to note that the purpose of all the diagrams are to send/receive messages. The handling of these messages are internal to the objects. Hence, the objects also have options to receive and send messages, and here comes another important

aspect called interface. Now these interfaces are responsible for accepting and sending messages to one another.

It can thus be concluded that the interactions in UML 2.0 are described in a different way and that is the reason why the new diagram names have come into picture. If we analyze the new diagrams then it is clear that all the diagrams are created based upon the interaction diagrams described in the earlier versions. The only difference is the additional features added in UML to make the diagrams more efficient and purpose oriented.

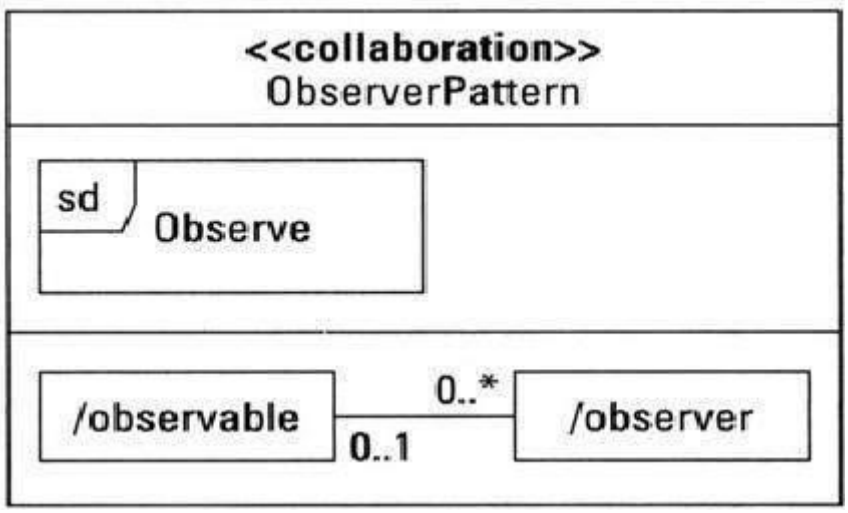
Modeling Collaborations

As we have already discussed, collaboration is used to model common interactions between objects. We can say that collaboration is an interaction where a set of messages are handled by a set of objects having pre-defined roles.

The important point to note is the difference between the collaboration diagram in the earlier version and in UML 2.0 version. To distinguish, the name of the collaboration diagram has been changed in UML 2.0. In UML 2.0, it is named as **Communication diagram.**

Consequently, collaboration is defined as a class with attributes (properties) and behavior (operations). Compartments on the collaboration class can be user defined and may be used for interactions (Sequence diagrams) and structural elements (Composite Structure diagram).

Following figure models the Observer design pattern as collaboration between an object in the role of an observable item and any number of objects as the observers.



Modeling Communication

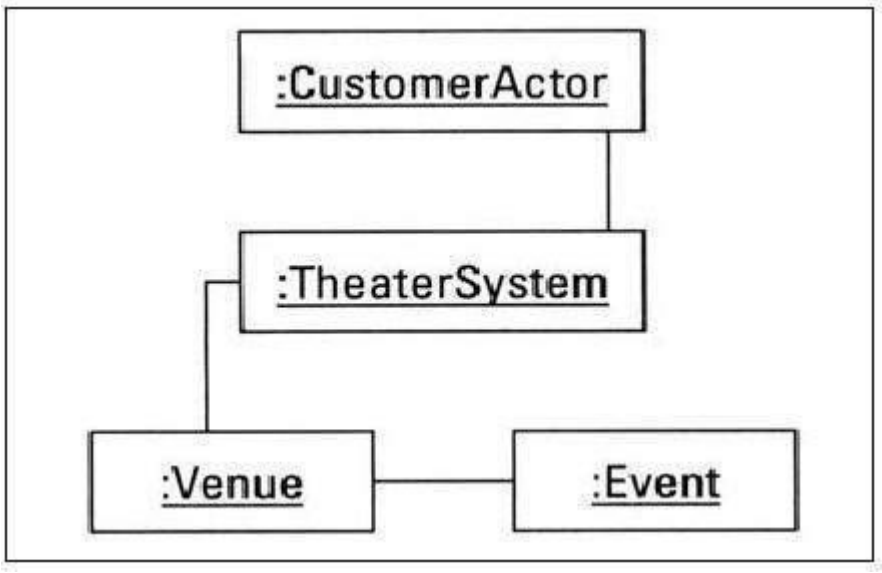
Communication diagram is slightly different than the collaboration diagrams of the earlier versions. We can say it is a scaled back version of the earlier UML versions. The distinguishing factor of the communication diagram is the link between objects.

This is a visual link and it is missing in the sequence diagram. In the sequence diagram, only the messages passed between the objects are shown even if there is no link between them.

Communication diagram is used to prevent the modeler from making this mistake by using an Object diagram format as the basis for messaging. Each object on a Communication diagram is called an object lifeline.

The message types in a Communication diagram are the same as in a Sequence diagram. Communication diagram may model synchronous, asynchronous, return, lost, found, a object-creation messages.

Following figure shows an Object diagram with three objects and two links that form the basis for the Communication diagram. Each object on a Communication diagram is called an object lifeline.



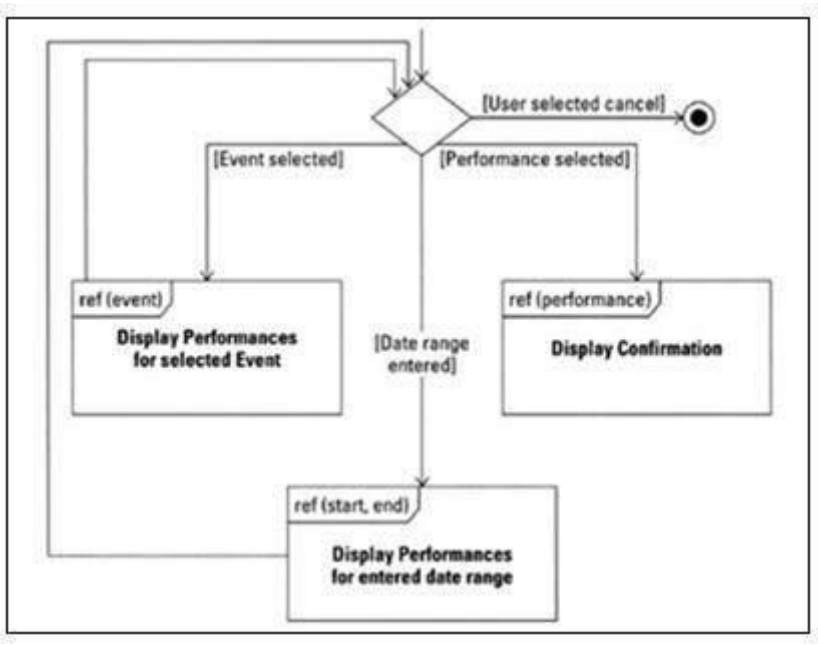
Modeling an Interaction Overview

In practical usage, a sequence diagram is used to model a single scenario. A number of sequence diagrams are used to complete the entire application. Hence, while modeling a single scenario, it is possible to forget the total process and this can introduce errors.

To solve this issue, the new interaction overview diagram combines the flow of control from an activity diagram and messaging specification from the sequence diagram.

Activity diagram uses activities and object flows to describe a process. The Interaction Overview diagram uses interactions and interaction occurrences. The lifelines and messages found in Sequence diagrams appear only within the interactions or interaction occurrences. However, the lifelines (objects) that participate in the Interaction Overview diagram may be listed along with the diagram name.

Following figure shows an interaction overview diagram with decision diamonds, frames, and termination point.



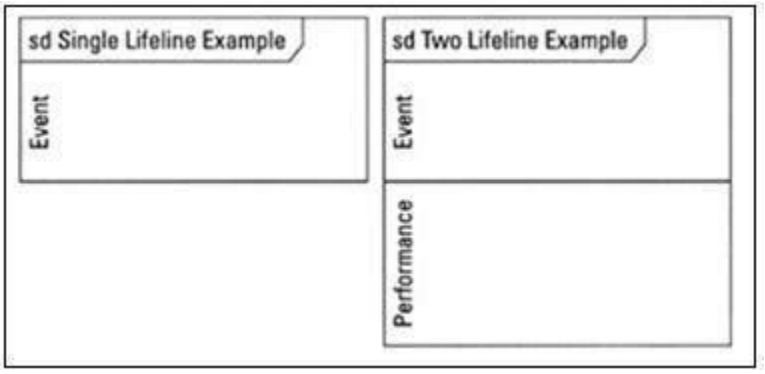
Modeling a Timing Diagram

The name of this diagram itself describes the purpose of the diagram. It basically deals with the time of the events over its entire lifecycle.

A timing diagram can therefore be defined as a special purpose interaction diagram made to focus on the events of an object in its life time. It is basically a mixture of state machine and interaction diagram. The timing diagram uses the following timelines −

- State time line
- General value time line

A lifeline in a Timing diagram forms a rectangular space within the content area of a frame. It is typically aligned horizontally to read from the left to right. Multiple lifelines may be stacked within the same frame to model the interaction between them.



Summary

UML 2.0 is an enhanced version where the new features are added to make it more usable and efficient. There are two major categories in UML 2.0, one is UML super structure and another is UML infrastructure. Although the new diagrams are based on the old concepts, they still have some additional features.

UML 2.0 offers four interaction diagrams, the Sequence diagram, Communication diagram, Interaction Overview diagram, and an optional Timing diagram. All four diagrams utilize the frame notation to enclose an interaction. The use of frames support the reuse of interactions as interaction occurrences