**UNIT–I UML(11 Hrs.)**

History of UML, Goals of UML, nature & purpose of models.

UML views & diagrams – static, design, use case, state machine, activity, interaction deployment, model management, profile.

Relationships in UML – association, dependency, generalization, realization.

UML extensibility mechanisms – constraints, stereotypes, tagged values.

Unified Process (UP): UP structure, phases of UP.

[UML](http://www.omg.org/spec/UML/), short for Unified Modeling Language, is a standardized modeling language consisting of an integrated set of diagrams, developed to help system and software developers for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems. The UML is a very important part of developing object oriented software and the software development process. The UML uses mostly graphical notations to express the design of software projects. Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

The Origin of UML

The goal of UML is to provide a standard notation that can be used by all object-oriented methods and to select and integrate the best elements of precursor notations. UML has been designed for a broad range of applications. Hence, it provides constructs for a broad range of systems and activities (e.g., distributed systems, analysis, system design and deployment).

UML is a notation that resulted from the unification of OMT from

1. [Object Modeling Technique OMT](https://en.wikipedia.org/wiki/Object-modeling_technique) [[James Rumbaugh](https://en.wikipedia.org/wiki/James_Rumbaugh) 1991] - was best for analysis and data-intensive information systems.
2. Booch [[Grady Booch](https://en.wikipedia.org/wiki/Grady_Booch) 1994] - was excellent for design and implementation. Grady Booch had worked extensively with the [Ada](https://en.wikipedia.org/wiki/Ada_(programming_language)) language, and had been a major player in the development of Object Oriented techniques for the language. Although the Booch method was strong, the notation was less well received (lots of cloud shapes dominated his models - not very tidy)
3. OOSE (Object-Oriented Software Engineering [[Ivar Jacobson](https://en.wikipedia.org/wiki/Ivar_Jacobson) 1992]) - featured a model known as Use Cases. Use Cases are a powerful technique for understanding the behaviour of an entire system (an area where OO has traditionally been weak).

In 1994, Jim Rumbaugh, the creator of OMT, stunned the software world when he left General Electric and joined Grady Booch at Rational Corp. The aim of the partnership was to merge their ideas into a single, unified method (the working title for the method was indeed the "Unified Method").

By 1995, the creator of OOSE, Ivar Jacobson, had also joined Rational, and his ideas (particularly the concept of "Use Cases") were fed into the new Unified Method - now called the Unified Modelling Language1. The team of Rumbaugh, Booch and Jacobson are affectionately known as the "Three Amigos"

UML has also been influenced by other object-oriented notations:

* Mellor and Shlaer [1998]
* Coad and Yourdon [1995]
* Wirfs-Brock [1990]
* Martin and Odell [1992]

UML also includes new concepts that were not present in other major methods at the time, such as extension mechanisms and a constraint language.

History of UML

Object-oriented modeling languages emerged during the late 1970s and 1980s. From the late 1980s through the beginning of the next decade, the number of modeling languages exploded. The consolidation of these different systems was inevitable, and in the mid-1990s, UML emerged as a combination of three methods: Booch, OOSE and OMT. This created stability and predictability, allowing developers to rely on a solid system.

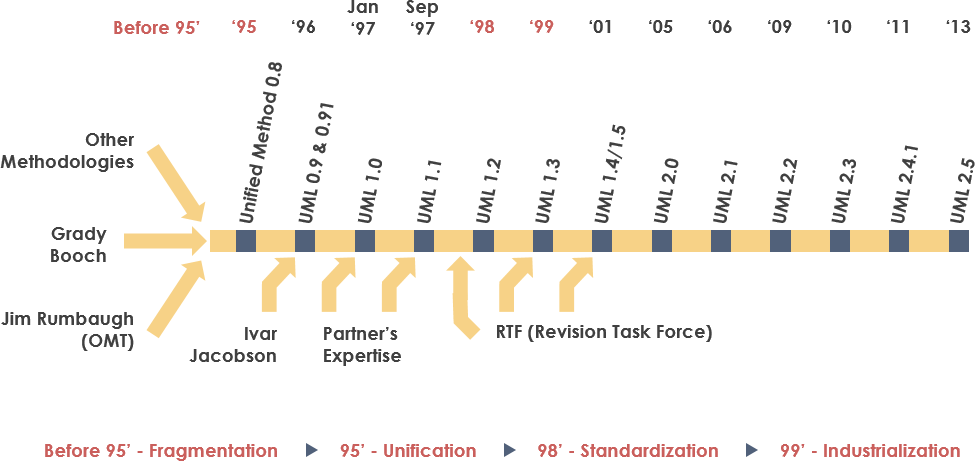
At the same time, the entire industry was seeking a modeling language that would act as an industry standard. In 1996, the Object Management Group asked for proposals from major industry players, such as IBM, Microsoft and Oracle. They combined efforts to create UML 1.0, which was submitted for consideration to the Object Management Group. This quickly became a standard and continues to be regularly updated.

UML is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems. UML stands for **Unified Modeling Language**. UML is different from the other common programming languages such as C++, Java, COBOL, etc. as it can be described as a general purpose visual modeling language to visualize, specify, construct, and document software system.

UML is a pictorial language used to make software blueprints. 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.

1. During 1996, the first Request for Proposal (RFP) issued by the [Object Management Group (OMG)](http://www.omg.org/) provided the catalyst for these organizations to join forces around producing a joint RFP response.
2. Rational established the UML Partners consortium with several organizations willing to dedicate resources to work toward a strong UML 1.0 definition. Those contributing most to the UML 1.0 definition included:
   * Digital Equipment Corp
   * HP
   * i-Logix
   * IntelliCorp
   * IBM
   * ICON Computing
   * MCI Systemhouse
   * Microsoft
   * Oracle
   * Rational Software
   * TI
   * Unisys
3. This collaboration produced UML 1.0, a modeling language that was well-defined, expressive, powerful, and generally applicable. This was submitted to the OMG in January 1997 as an initial RFP response.1
4. In January 1997 IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies and Softeam also submitted separate RFP responses to the OMG. These companies joined the UML partners to contribute their ideas, and together the partners produced the revised UML 1.1 response. The focus of the UML 1.1 release was to improve the clarity of the UML 1.0 semantics and to incorporate contributions from the new partners. It was submitted to the OMG for their consideration and adopted in the fall of 1997.1 and enhanced 1.1 to 1.5, and subsequently to UML 2.1 from 01 to 06 (now the UML current version is 2.5)



**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.

Why UML

As the strategic value of software increases for many companies, the industry looks for techniques to automate the production of software and to improve quality and reduce cost and time-to-market. These techniques include component technology, visual programming, patterns and frameworks. Businesses also seek techniques to manage the complexity of systems as they increase in scope and scale. In particular, they recognize the need to solve recurring architectural problems, such as physical distribution, concurrency, replication, security, load balancing and fault tolerance. Additionally, the development for the World Wide Web, while making some things simpler, has exacerbated these architectural problems. The Unified Modeling Language (UML) was designed to respond to these needs. The primary goals in the design of the UML summarizes as follows:

1. Provide users with a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
2. Provide extensibility and specialization mechanisms to extend the core concepts.
3. Be independent of particular programming languages and development processes.
4. Provide a formal basis for understanding the modeling language.
5. Encourage the growth of the OO tools market.
6. Support higher-level development concepts such as collaborations, frameworks, patterns and components.
7. Integrate best practices.

**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**

The building blocks of UML can be defined as Things, Relationships and Diagrams

## Things are the most important building blocks of UML. Things can be Structural, Behavioral, Grouping and Annotational.

### ***Structural Things:***

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

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

class

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

Interface

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

Collaboration

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

Use case

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

Component

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



### ***Behavioral Things***

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

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

Interaction

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



### ***Grouping Things***

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

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



### ***Annotational Things***

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

Note

## Relationship: Relationship is another most important building block of UML. It shows how the elements are associated with each other and this association describes the functionality of an application. There are four kinds of relationships available.

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

Dependency

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

Association

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

Generalization

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

Realization

UML - An Overview

The first thing to notice about the UML is that there are a lot of different diagrams (models) to get used to. The reason for this is that it is possible to look at a system from many different viewpoints. A software development will have many stakeholders playing a part.

For Example: Analysts, Designers, Coders, Testers, QA, Customer, Technical Authors

All of these people are interested in different aspects of the system, and each of them require a different level of detail. For example, a coder needs to understand the design of the system and be able to convert the design to a low level code. By contrast, a technical writer is interested in the behavior of the system as a whole, and needs to understand how the product functions. The UML attempts to provide a language so expressive that all stakeholders can benefit from at least one UML diagram.

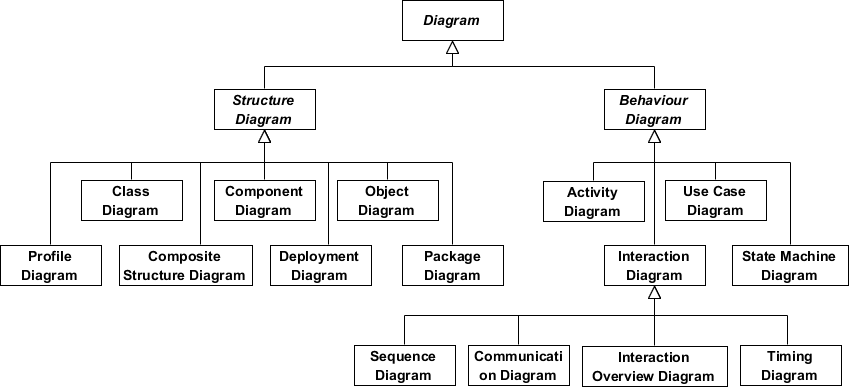
Here's a quick look at each one of these 13 diagrams in as shown in the UML 2 Diagram Structure below:

Structure diagrams show the static structure of the system and its parts on different abstraction and implementation levels and how they are related to each other. The elements in a structure diagram represent the meaningful concepts of a system, and may include abstract, real world and implementation concepts, **there are seven types of structure diagram** as follows:

* [Class Diagram](https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/#class-diagram)
* [Component Diagram](https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/#component-diagram)
* [Deployment Diagram](https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/#deployment-diagram)
* [Object Diagram](https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/#object-diagram)
* [Package Diagram](https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/#package-diagram)
* [Composite Structure Diagram](https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/#composite-structure-diagram)
* [Profile Diagram](https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/#profile-diagram)

Behavior diagrams show the **dynamic behavior** of the objects in a system, which can be described as a series of changes to the system over **time**, **there are seven types of behavior diagrams** as follows:

* [Use Case Diagram](https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/#use-case-diagram)
* [Activity Diagram](https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/#activity-diagram)
* [State Machine Diagram](https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/#state-machine-diagram)
* [Sequence Diagram](https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/#sequence-diagram)
* [Communication Diagram](https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/#communication-diagram)
* [Interaction Overview Diagram](https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/#interaction-overview-diagram)
* [Timing Diagram](https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/#timing-diagram)



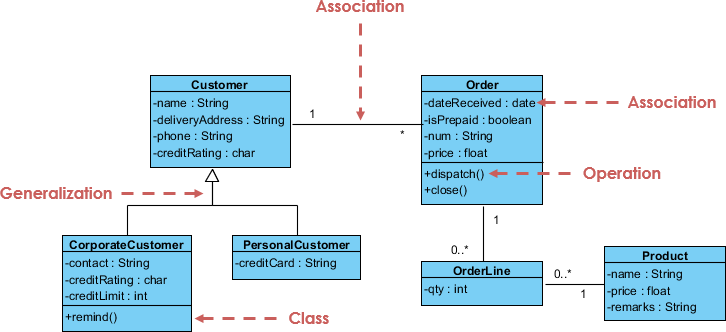
What is a Class Diagram?

The class diagram is a central modeling technique that runs through nearly all object-oriented methods. This diagram describes the types of objects in the system and various kinds of static relationships which exist between them.

***Relationships***

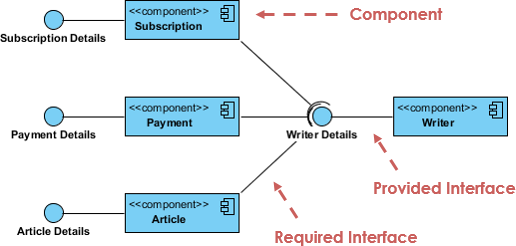
There are three principal kinds of relationships which are important:

1. **Association** - represent relationships between instances of types (a person works for a company, a company has a number of offices.
2. **Inheritance** - the most obvious addition to ER diagrams for use in OO. It has an immediate correspondence to inheritance in OO design.
3. **Aggregation** - Aggregation, a form of object composition in object-oriented design.



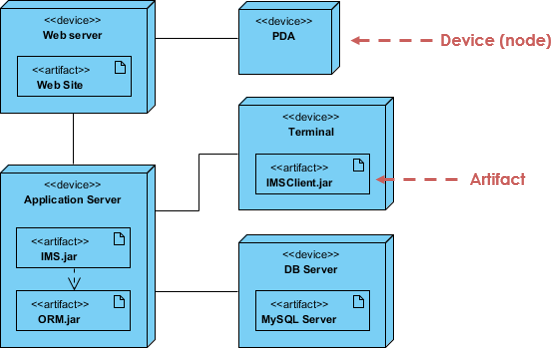
What is Component Diagram?

In the Unified Modeling Language, a component diagram depicts how components are wired together to form larger components or software systems. It illustrates the architectures of the software components and the dependencies between them. Those software components including run-time components, executable components also the source code components.



What is a Deployment Diagram?

The Deployment Diagram helps to model the physical aspect of an Object-Oriented software system. It is a structure diagram which shows architecture of the system as deployment (distribution) of software artifacts to deployment targets. Artifacts represent concrete elements in the physical world that are the result of a development process. It models the run-time configuration in a static view and visualizes the distribution of artifacts in an application. In most cases, it involves modeling the hardware configurations together with the software components that lived on.



What is an Object Diagram?

An object diagram is a graph of instances, including objects and data values. A static object diagram is an instance of a class diagram; it shows a snapshot of the detailed state of a system at a point in time. 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. The use of object diagrams is fairly limited, namely to show examples of data structure.

Class Diagram vs Object Diagram - An Example

Some people may find it difficult to understand the difference between a UML Class Diagram and a UML Object Diagram as they both comprise of named "rectangle blocks", with attributes in them, and with linkages in between, which make the two UML diagrams look similar. Some people may even think they are the same because in the UML tool they use both the notations for Class Diagram and Object Diagram are put inside the same diagram editor - Class Diagram.

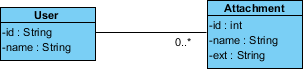
But in fact, Class Diagram and Object Diagram represent two different aspects of a code base. In this article, we will provide you with some ideas about these two UML diagrams, what they are, what are their differences and when to use each of them.

Relationship between Class Diagram and Object Diagram

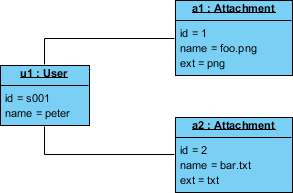
You create "classes" when you are programming. For example, in an online banking system you may create classes like 'User', 'Account', 'Transaction', etc. In a classroom management system you may create classes like 'Teacher', 'Student', 'Assignment', etc. In each class, there are attributes and operations that represent the characteristic and behavior of the class. Class Diagram is a UML diagram where you can visualize those classes, along with their attributes, operations and the inter-relationship.

UML Object Diagram shows how object instances in your system are interacting with each other at a particular state. It also represents the data values of those objects at that state. In other words, a UML Object Diagram can be seen as a representation of how classes (drawn in UML Class Diagram) are utilized at a particular state.

The following Class Diagram example represents two classes - User and Attachment. A user can upload multiple attachment so the two classes are connected with an association, with 0..\* as multiplicity on the Attachment side.

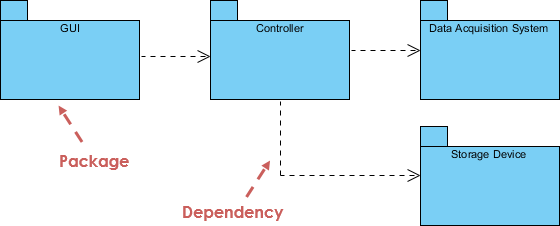


The following Object Diagram example shows you how the object instances of User and Attachment class "look like" at the moment Peter (i.e. the user) is trying to upload two attachments. So there are two Instance Specification for the two attachment objects to be uploaded.



What is a Package Diagram?

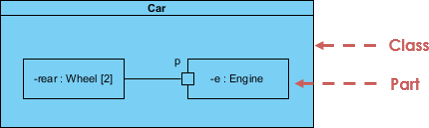
Package diagram is UML structure diagram which shows packages and dependencies between the packages. Model diagrams allow to show different views of a system, for example, as multi-layered (aka multi-tiered) application - multi-layered application model.



What is a Composite Structure Diagram?

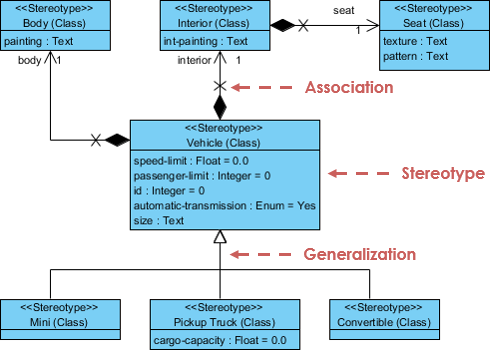
Composite Structure Diagram is one of the new artifacts added to UML 2.0. A composite structure diagram is similar to a class diagram and is a kind of component diagram mainly used in modeling a system at micro point-of-view, but it depicts individual parts instead of whole classes. It is a type of static structure diagram that shows the internal structure of a class and the collaborations that this structure makes possible.

This diagram can include internal parts, ports through which the parts interact with each other or through which instances of the class interact with the parts and with the outside world, and connectors between parts or ports. A composite structure is a set of interconnected elements that collaborate at runtime to achieve some purpose. Each element has some defined role in the collaboration.



What is a Profile Diagram?

A profile diagram enables you to create domain and platform specific stereotypes and define the relationships between them. You can create stereotypes by drawing stereotype shapes and relate them with composition or generalization through the resource-centric interface. You can also define and visualize tagged values of stereotypes.



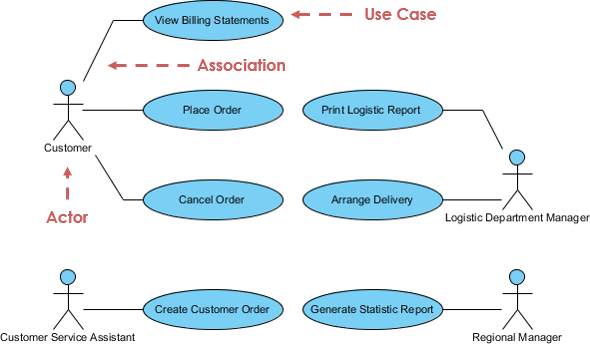
What is a Use Case Diagram?

A use-case model describes a system's functional requirements in terms of use cases. It is a model of the system's intended functionality (use cases) and its environment (actors). Use cases enable you to relate what you need from a system to how the system delivers on those needs.

Think of a use-case model as a menu, much like the menu you'd find in a restaurant. By looking at the menu, you know what's available to you, the individual dishes as well as their prices. You also know what kind of cuisine the restaurant serves: Italian, Mexican, Chinese, and so on. By looking at the menu, you get an overall impression of the dining experience that awaits you in that restaurant. The menu, in effect, "models" the restaurant's behavior.

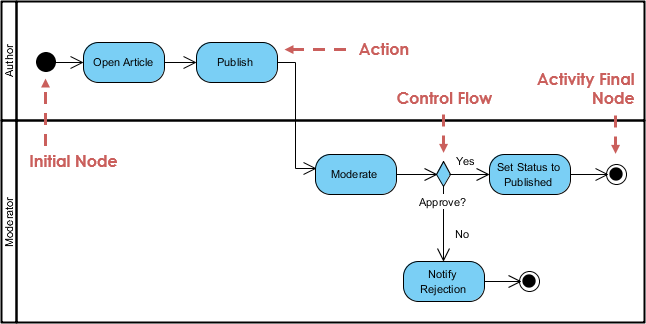
Because it is a very powerful planning instrument, the use-case model is generally used in all phases of the development cycle by all team members.

Use Case Diagram Example



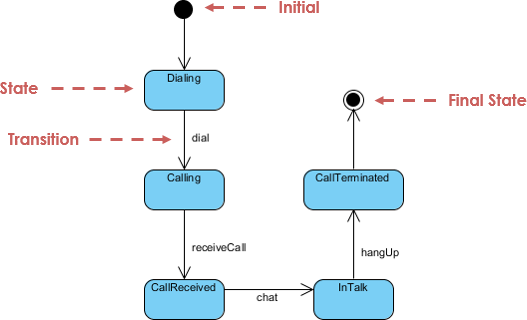
What is an Activity Diagram?

Activity diagrams are graphical representations of workflows of stepwise activities and actions with support for choice, iteration and concurrency. It describes the flow of control of the target system, such as the exploring complex business rules and operations, describing the use case also the business process. In the Unified Modeling Language, activity diagrams are intended to model both computational and organizational processes (i.e. workflows).



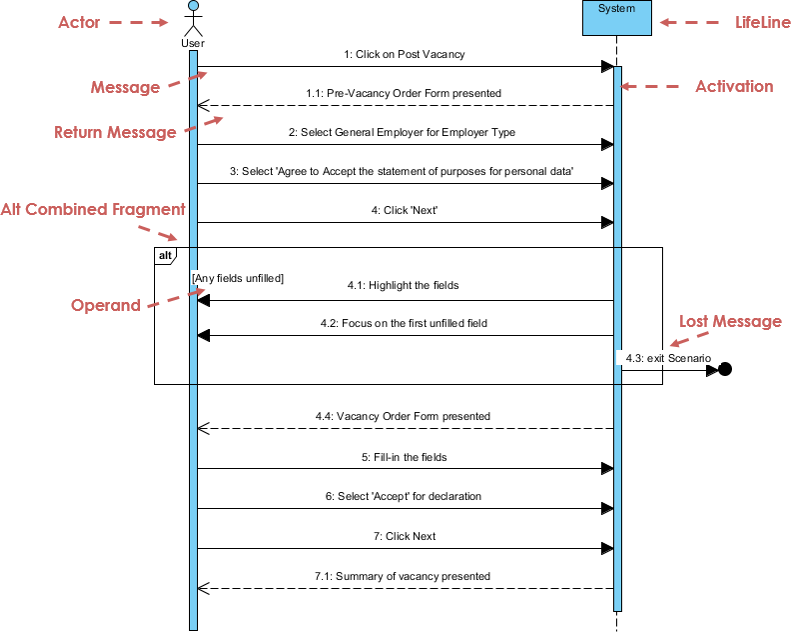
What is a State Machine Diagram?

A state diagram is a type of diagram used in UML to describe the behavior of systems which is based on the concept of state diagrams by David Harel. State diagrams depict the permitted states and transitions as well as the events that effect these transitions. It helps to visualize the entire lifecycle of objects and thus help to provide a better understanding of state-based systems.



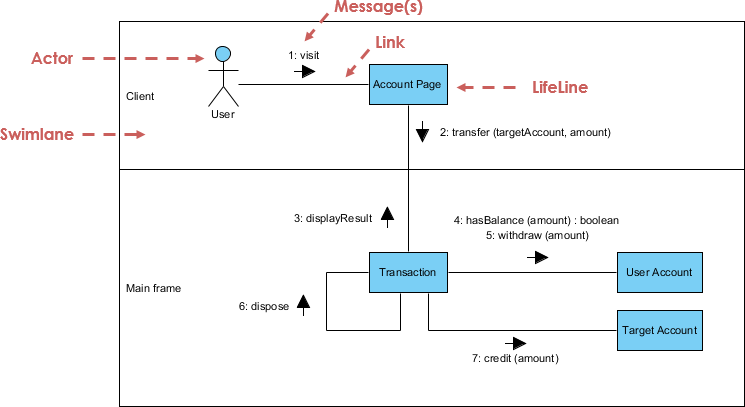
What is a Sequence Diagram?

The Sequence Diagram models the collaboration of objects based on a time sequence. It shows how the objects interact with others in a particular scenario of a use case. With the advanced visual modeling capability, you can create complex sequence diagram in few clicks. Besides, some modeling tool such as Visual Paradigm can generate sequence diagram from the flow of events which you have defined in the use case description.



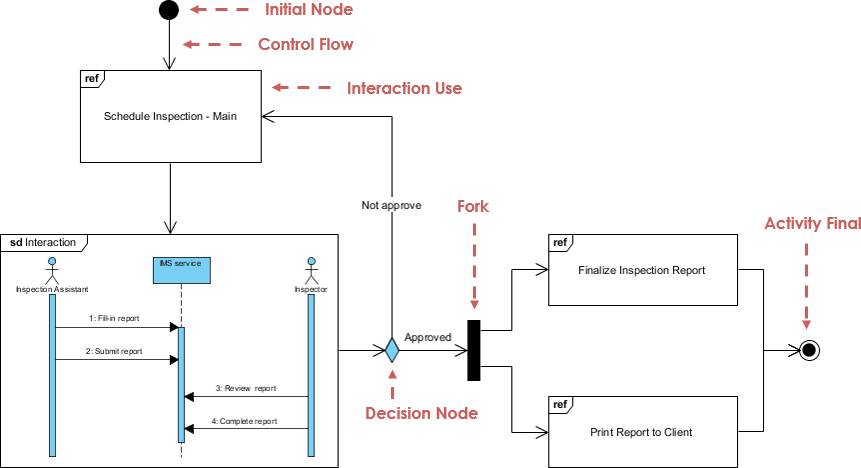
What is a Communication Diagram?

Similar to Sequence Diagram, the Communication Diagram is also used to model the dynamic behavior of the use case. When compare to Sequence Diagram, the Communication Diagram is more focused on showing the collaboration of objects rather than the time sequence. They are actually semantically equivalent, so some of the modeling tool such as, Visual Paradigm allows you to generate it from one to the other.



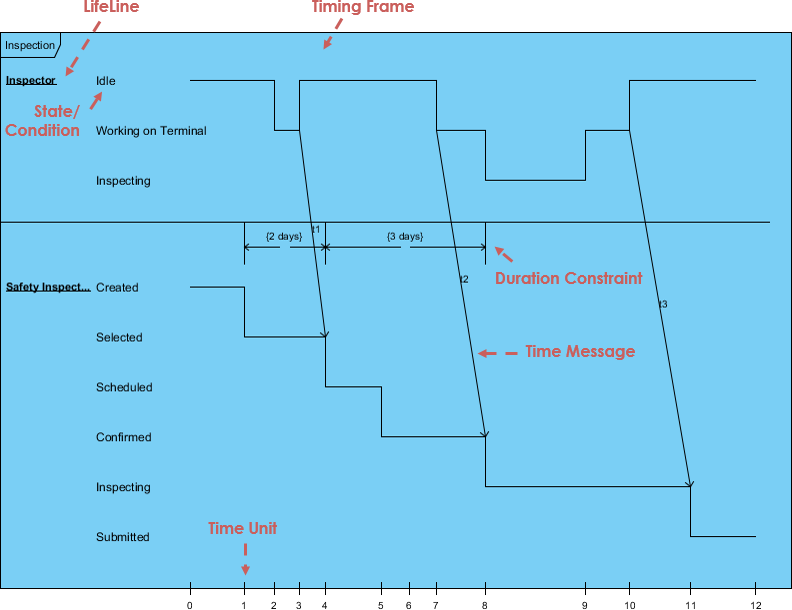
What is Interaction Overview Diagram?

The Interaction Overview Diagram focuses on the overview of the flow of control of the interactions. It is a variant of the Activity Diagram where the nodes are the interactions or interaction occurrences. The Interaction Overview Diagram describes the interactions where messages and lifelines are hidden. You can link up the "real" diagrams and achieve high degree navigability between diagrams inside the Interaction Overview Diagram.



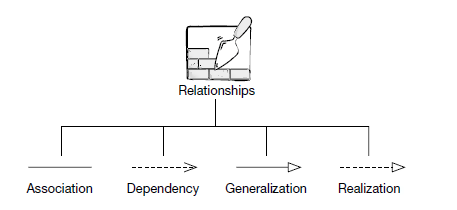
What is Timing Diagram?

Timing Diagram shows the behavior of the object(s) in a given period of time. Timing diagram is a special form of a sequence diagram. The differences between timing diagram and sequence diagram are the axes are reversed so that the time are increase from left to right and the lifelines are shown in separate compartments arranged vertically.



**Relationships in UML – association, dependency, generalization, realization.**

Relationships in UML are used to represent a connection between various things. It is also called a link that describes how two or more things can relate to each other during the execution of a system. A relationship is a connection amongst things such as structural, behavioral, or grouping things in the unified modeling language. Following are the different types of standard relationships in UML:

[](https://www.guru99.com/images/1/062819_0548_UMLRelation1.png)

1. **Association:** It is a set of links that connects elements of the UML model. It also defines how many objects are taking part in that relation.
2. **Dependency:** In a dependency relationship, as the name suggests, two or more elements are dependent on each other. In this kind of a relationship, if we make a change to a particular element, then it is likely possible that all the other elements will also get affected by the change.
3. **Generalization:** It is also called a parent-child relationship. In generalization, one element is a specialization of another general component. It may be substituted for it. It is mostly used to represent inheritance.
4. **Realization:** In a realization relationship of UML, one entity denotes some responsibility which is not implemented by itself and the other entity that implements them. This relationship is mostly found in the case of **interfaces.**

## *Association*

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

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

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

### **Reflexive association:** The reflexive association is a subtype of association relationship in UML. In a reflexive association, the instances of the same class can be related to each other. An instance of a class is also said to be an object. Reflexive association states that a link or a connection can be present within the objects of the same class. Let us consider an example of a class fruit. The fruit class has two instances, such as mango and apple. Reflexive association states that a link between mango and apple can be present as they are instances of the same class, such as fruit.

### **Directed association:** As the name suggests, the directed association is related to the direction of flow within association classes. In a directed association, the flow is directed. The association from one class to another class flows in a single direction only. It is denoted using a solid line with an arrowhead. Example: You can say that there is a directed association relationship between a server and a client. A server can process the requests of a client. This flow is unidirectional, that flows from server to client only. Hence a directed association relationship can be present within servers and clients of a system.

## *Dependency*

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

### **Stereotypes**

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

### **Stereotypes among state machine**

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

## *Generalization*

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

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

### Stereotypes and their constraints

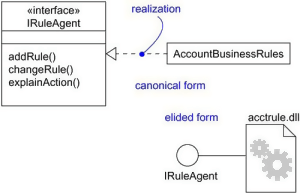
* **«implementation»** – This stereotype is used to represent that the child entity is being implemented by the parent entity by inheriting the structure and behavior of a parent object without violating the rules.**Note** This stereotype if widely used in a singl**e inheritance**.

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

## *Realization*

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

* Using a canonical form
* Using an elided form

[](https://www.guru99.com/images/1/062819_0548_UMLRelation2.png)

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

### Types of realization:

1. Canonical form

In a realization relationship of UML, the canonical form is used to realize interfaces across the system. It uses an interface stereotype to create an interface and realization relationship is used to realize the particular interface. In a canonical form, the realization relationship is denoted using the dashed directed line with a sizeable open arrowhead. In the above diagram, interface Iruleagent is realized using an object called as Account Business Rules.

1. Elided form

Realization in the UML class diagram can also be shown using an elided form. In an elided form, the interface is denoted using a circle which is also called as a lollipop notation. This interface, when realized using anything present inside the system, creates an elided structure. In the above diagram, the interface Iruleagent is denoted using an elided form which is being realized by acctrule.dll.

## *Composition*

It is not a standard UML relationship, but it is still used in various applications. Composite aggregation is a subtype of aggregation relation with characteristics as:

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

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

[https://www.guru99.com/images/1/062819_0548_UMLRelation3.png](https://www.guru99.com/images/1/062819_0548_UMLRelation3.png)

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

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

## *Aggregation*

An aggregation is a subtype of an association relationship in UML. Aggregation and composition are both the types of association relationship in UML. An aggregation relationship can be described in simple words as " an object of one class can own or access the objects of another class." In an aggregation relationship, the dependent object remains in the scope of a relationship even when the source object is destroyed. Let us consider an example of a car and a wheel. A car needs a wheel to function correctly, but a wheel doesn't always need a car. It can also be used with the bike, bicycle, or any other vehicles but not a particular car. Here, the wheel object is meaningful even without the car object. Such type of relationship is called an aggregation relation.

## Note:

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

**Extensibility Mechanisms**

The UML provides a standard language for writing software blueprints, but it is not possible for one closed language to ever be sufficient to express all possible nuances of all models across all domains across all time. For this reason, the UML is opened-ended, making it possible for you to extend the language in controlled ways.

The UML's extensibility mechanisms include Stereotypes, Tagged values and Constraints

A **stereotype** extends the vocabulary of the UML, allowing you to create new kinds of building blocks that are derived from existing ones but that are specific to your problem. For example, if you are working in a programming language, such as Java or C++, you will often want to model exceptions. In these languages, exceptions are just classes, although they are treated in very special ways. Typically, you only want to allow them to be thrown and caught, nothing else. You can make exceptions first-class citizens in your models meaning that they are treated like basic building blocks by marking them with an appropriate stereotype, as for the class Overflow in Figure 2-19.

A screenshot of a cell phone

Description automatically generated

A **tagged value** extends the properties of a UML stereotype, allowing you to create new information in the stereotype's specification. For example, if you are working on a shrink- wrapped product that undergoes many releases over time, you often want to track the version and author of certain critical abstractions. Version and author are not primitive UML concepts. They can be added to any building block, such as a class, by introducing new tagged values to that building block. In Figure 2-19, for example, the class EventQueue is extended by marking its version and author explicitly.

A **constraint** extends the semantics of a UML building block, allowing you to add new rules or modify existing ones. For example, you might want to constrain the EventQueue class so that all additions are done in order. As Figure 2-22 shows, you can add a constraint that explicitly marks these for the operation add. Figure 2-22. Extensibility Mechanisms Collectively, these three extensibility mechanisms allow you to shape and grow the UML to your project's needs. These mechanisms also let the UML adapt to new software technology, such as the likely emergence of more powerful distributed programming languages. You can add new building blocks, modify the specification of existing ones, and even change their semantics. Naturally, it's important that you do so in controlled ways so that through these extensions, you remain true to the UML's purpose the communication of information.

A screenshot of a cell phone

Description automatically generated

***UNIFIED PROCESS MODEL***

The **unified process model** (or **UPM**) is an iterative, incremental, architecture-centric, and use-case driven approach to software development. Unified process can be applied to different software systems with different levels of technical and managerial complexity across various domains and organizational cultures.  
  
UP is also referred to as the unified software development process.

### Use-Case Driven Approach

A **use-case** defines the interaction between two or more entities. The list of requirements specified by a customer are converted to functional requirements by a business analyst and generally referred to as use-cases. A use-case describes the operation of a software as interactions between the customer and the system, resulting in a specific output or a measurable return. For example, the online cake shop can be specified in terms of use cases such as 'add cake to cart,' 'change the quantity of added cakes in cart,' 'cake order checkout,' and so on. Each use case represents a significant functionality and could be considered for an iteration.

### Architecture-Centric Approach

This approach creates a blueprint of the organization of the software system. It would include taking into account the different technologies, programming languages, operating systems, development and release environments, server capabilities, and other such areas for developing the software.

### Iterative and Incremental Approach

Using an iterative and incremental approach means treating each iteration as a mini-project. Therefore, always develop the software as a number of small mini-projects, working in cycles. You'd develop small working versions of the software at the end of each cycle. Each iteration would add some functionality to the software according to the requirements specified by the customer.

## Unified Process Model Phases

The four different phases are:

1. ***Inception***

The inception phase is similar to the requirements collection and analysis stage of the waterfall model of software development. In this phase, collect the requirements from the customer and analyze the project's feasibility, its cost, risks, and profits. The Inception Phase is the part of the framework when the why of the development effort is defined. This includes such activities as:

* Defining the business case
* Creating a vision document with core requirements, features, and constraints
* Creating an initial risk assessment
* Creating early use cases (10-20% complete, mostly use-case models)
* Creating a initial project plan
* And the creation of one or more prototypes (especially architectural prototypes)

The milestones (that together comprise the Lifecycle Objectives Milestone) that show completion of the Inception phase are:

* Stakeholder agreement on business case, scope, and project cost and schedule estimates
* Agreement that the content of the primary use cases is an accurate representation of what the software will deliver (at a high level)
* That the final prototypes are sufficient indications of the correct future development goals

1. ***Elaboration***

In this phase, expand the activities undertaken in the inception phase. The major goals of this phase include creating fully functional requirements (use-cases) and creating a detailed architecture for fulfillment of the requirements. In this phase, prepare a business case document for the customer. The Elaboration Phase is the part of the framework when more detailed analysis and planning are undertaken to better understand the problem domain, develop a more concrete project plan, identify and eliminate the high-risk elements of the effort, and to establish a solid architectural foundation for the software to be developed.  The goal is to develop a "mile wide and inch deep" view of the system to be developed. The specific activities of this phase include:

* The identification of all actors and use cases, with most use cases having been defined to at least 80% completion (use-case descriptions rather than models)
* Supplementary requirements detailing the non-functional requirements and any requirements not related to a use case are completed
* A Software Architecture Description has been completed
* The business case and risk lists have been updated with higher-confidence information
* The project and development plans have been defined to at least a level that shows all iterations and the evaluation criteria for each iteration
* An executable architecture prototype has been created and approved for use (this may involve creating more than one)
* A preliminary user manual has been created (optional)

The milestones (that together comprise the Lifecycle Architecture Milestone) that show completion of the Elaboration phase are:

* The product vision is stable and approved
* The product architecture is stable and approved
* The executable architecture prototype shows that the major risk elements have been identified and credibly resolved
* The project and development plans sufficiently detailed, accurate, and credible
* All stakeholders agree that the vision can be achieved is the project and development plans are executed with the architecture specified

1. ***Construction***

In this phase, write the actual code and implement the features for each iteration. The first iteration of the software depending on the key use-cases that make up the core functionalities of the software system. The Construction Phase is the part of the framework where software development, integration, and testing takes place. Because of the emphasis on component-based architectures and the significant attention paid to the architectural plan in the Inception and Elaboration phases, it should be possible to initiate multiple Construction Phases within a single cycle if the software to be developed is complex enough to support multiple discreet components. The specific activities of this phase include:

* The software is built, integrated, and tested
* The user manuals have been created (or updated)
* The details of the software developed are documented and ready to be provided to end users or support staff (including changes, etc.)

The milestones (that together comprise the Initial Operational Capability Milestone) that show completion of the Construction phase are:

* The software product is stable and mature enough to be deployed to end users
* All stakeholders are ready to transition to the new / updated software
* Actual versus planned expenditures are still acceptable enough to move forward with the project

The outcome of the construction phase should be a product that is ready to put into the hands of end-users in at least a beta release state.

1. ***Transition***

In this phase, roll out the next iterations to the customer and fixing bugs for previous releases. You would also deploy builds of the software to the customer. The Transition Phase of the framework is where the software is deployed to end users and is essentially a broad beta test of the application. Users begin to use the new software, issues are identified and potentially corrected, and any features that were delayed are finished and deployed. The transition phase can include multiple iterations of the software, including beta releases, bug fixes, and enhancements. The specific activities of this phase include:

* beta testing or user acceptance testing by end users to validate the new software against user expectations
* Parallel operation with legacy systems (if in existence) that will be replaced
* Operational databases are converted (if necessary)
* Users and maintainers of the software are fully trained
* The software is fully rolled-out

The milestones (that together comprise the Product Release Milestone) that show completion of the Transition phase are:

* Users are satisfied with the software
* Actual versus planned expenditures are still acceptable enough to move forward with the project

[Unified Process](https://www.sciencedirect.com/topics/computer-science/unified-process) is based on the enlargement and refinement of a system through multiple iterations, with cyclic feedback and adaptation. The system is developed incrementally over time, iteration by iteration, and thus this approach is also known as iterative and incremental [software development](https://www.sciencedirect.com/topics/computer-science/software-development). The iterations are spread over four phases where each phase consists of one or more iterations:

1. Inception—the first and the shortest phase in the project. It is used to prepare basis for the project, including preparation of business case, establishing project scope and setting boundaries, outlining key requirements, and possible [architecture solution](https://www.sciencedirect.com/topics/computer-science/solution-architecture) together with design tradeoffs, identifying risks, and development of initial project plan—schedule with main milestones and cost estimates. If the [inception phase](https://www.sciencedirect.com/topics/computer-science/inception-phase) lasts for too long, it is like an indicator stating that the project vision and goals are not clear to the stakeholders. With no clear goals and vision the project most likely is doomed to fail. At this scenario it is better to take a pause at the very beginning of the project to refine the vision and goals. Otherwise it could lead to unnecessary make-overs and schedule delays in further phases.
2. Elaboration—during this phase the project team is expected to capture a majority of system’s requirements (e.g., in the form of use cases), to perform [identified risk](https://www.sciencedirect.com/topics/computer-science/identified-risk) analysis and make a plan of risk management to reduce or eliminate their impact on final schedule and product, to establish design and architecture (e.g., using basic class diagrams, [package diagrams](https://www.sciencedirect.com/topics/computer-science/package-diagram), and deployment diagrams), to create a plan (schedule, cost estimates, and achievable milestones) for the next (construction) phase.
3. Construction—the longest and largest phase within Unified Process. During this phase, the design of the system is finalized and refined and the system is built using the basis created during [elaboration phase](https://www.sciencedirect.com/topics/computer-science/elaboration-phase). The construction phase is divided into multiple iterations, for each iteration to result in an executable release of the system. The final iteration of construction phase releases fully completed system which is to be deployed during transition phase, and
4. Transition—the final project phase which delivers the new system to its end-users. Transition phase includes also [data migration](https://www.sciencedirect.com/topics/computer-science/data-migration) from [legacy systems](https://www.sciencedirect.com/topics/computer-science/legacy-systems) and user trainings.

Each phase and its iteration consists of a set of [predefined activities](https://www.sciencedirect.com/topics/computer-science/predefined-activity). The Unified Process describes work activities as disciplines—a discipline is a set of activities and related artifacts in one subject area (e.g., the activities within requirements analysis). The disciplines described by Unified Process are as follows:

1. Business modeling—domain object modeling and dynamic modeling of the business processes,
2. Requirements—requirements analysis of system under consideration. Includes activities like writing use cases and identifying [nonfunctional requirements](https://www.sciencedirect.com/topics/computer-science/non-functional-requirement),
3. Analysis and design—covers aspects of design, including the overall architecture,
4. Implementation—programming and building the system (except the deployment),
5. Test—involves testing activities such as test planning, development of test scenarios, alpha and beta testing, [regression](https://www.sciencedirect.com/topics/computer-science/regression) testing, acceptance testing, and
6. Deployment—the deployment activities of developed system.

The disciplines and phases of Unified Process are given in Figure where the phases are columns and the disciplines are rows. It clearly shows that the relative effort across disciplines changes over time from iteration to iteration, e.g., initial iterations apply greater relative effort on requirements and design while the latter—more on testing and deployment.

