UNIFIED MODELLING LANGUAGE

Subject : Unified Modelling Language (Elective-III)

Subject Code : CSE608

Total Hours : 42

Full Marks : 80 + 20 = 100

Content:

1. Introduction of UML: Overview, Conceptual Model of UML, Software Architecture, Software Development Life Cycle, Classes, Relationships, Common Mechanisms of UML. The importance of modelling, principles of modelling. 04 Hrs

2. Class Diagrams: Terms and Concepts, Common Modelling Techniques, Advanced Classes, Advanced Relationships, Interfaces, Types and Roles, Packages 08 Hrs

3. Instances, Object Diagrams, Basic Behavioural Modelling: Interactions, Use cases, Use Case Diagrams, Interaction Diagrams, Activity Diagrams. 08 Hrs

4. Advanced Behavioural Modelling: Events and Signals, State Machines, State Diagrams, Architectural Modelling: Components, Collaborations, Component Diagrams, Deployment Diagrams. 08 Hrs

5. Generate Use-case Diagram, Class Diagram, Sequence Diagram, Collaboration Diagram, Activity Diagram, State Chart Diagram, Component Diagram, Deployment Diagram for the following systems. 14 Hrs

- Student Registration System
- Courier Tracking System
- Online Shopping System
- Online Pizza ordering System
- Online Job Portal **System**

-Srishti Priya

-CSE

MODULE-1

Introduction of UML

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

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

- Complex applications need collaboration and planning from multiple teams and hence require a clear and concise way to communicate amongst them.
- Businessmen do not understand code. So UML becomes essential to communicate with non programmers essential requirements, functionalities and processes of the system.
- A lot of time is saved down the line when teams are able to visualize processes, user interactions and static structure of the system.

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

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

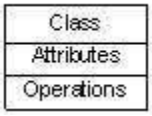
Things are the most important building blocks of UML. Things can be −

1) Structural
2) Behavioral
3) Grouping
4) Annotational

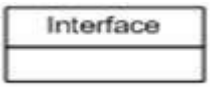
Structural Things

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.



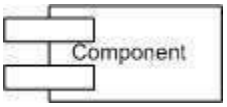
Interface − Interface defines a set of operations, which specify the responsibility of a class.



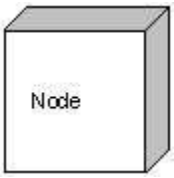
Collaboration −Collaboration defines an interaction between elements.

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.



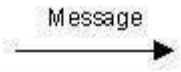
Node − A node can be defined as a physical element that exists at run time.



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

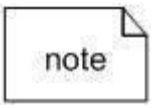
Grouping Things

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.



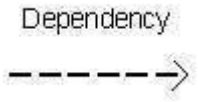
Relationship

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.

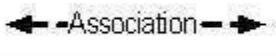
Dependency

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



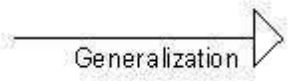
Association

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.



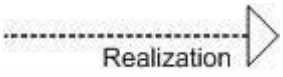
Generalization

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

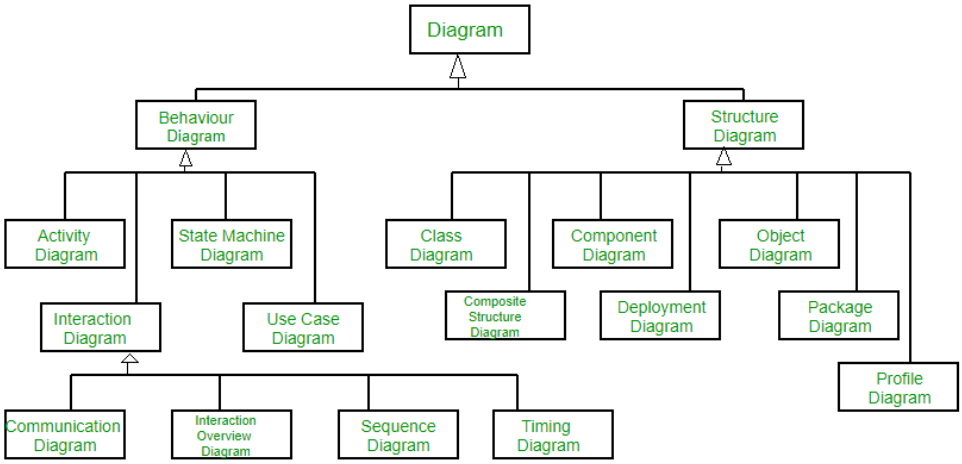
Realization can be defined as a relationship in which two elements are connected. One element describes some responsibility, which is not implemented and the other one implements them. This relationship exists in case of interfaces.



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

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

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



Object Oriented Concepts Used in UML –

1.Class – A class defines the blue print i.e. structure and functions of an object.

2.Objects – Objects help us to decompose large systems and help us to modularize our system. Modularity helps to divide our system into understandable components so that we can build our system piece by piece. An object is the fundamental unit (building block) of a system which is used to depict an entity.

3.Inheritance – Inheritance is a mechanism by which child classes inherit the properties of their parent classes.

4.Abstraction – Mechanism by which implementation details are hidden from user.

5.Encapsulation – Binding data together and protecting it from the outer world is referred to as encapsulation.

6.Polymorphism – Mechanism by which functions or entities are able to exist in different forms.

Additions in UML 2.0 –

- Software development methodologies like agile have been incorporated and scope of original UML specification has been broadened.

- Originally UML specified 9 diagrams. UML 2.x has increased the number of diagrams from 9 to 13. The four diagrams that were added are : timing diagram, communication diagram, interaction overview diagram and composite structure diagram. UML 2.x renamed statechart diagrams to state machine diagrams.
- UML 2.x added the ability to decompose software system into components and sub-components.

Structural UML Diagrams –

1.Class Diagram – The most widely use UML diagram is the class diagram. It is the building block of all object oriented software systems. We use class diagrams to depict the static structure of a system by showing system's classes,their methods and attributes. Class diagrams also help us identify relationship between different classes or objects.

2.Composite Structure Diagram – We use composite structure diagrams to represent the internal structure of a class and its interaction points with other parts of the system. A composite structure diagram represents relationship between parts and their configuration which determine how the classifier (class, a component, or a deployment node) behaves. They represent internal structure of a structured classifier making the use of parts, ports, and connectors. We can also model collaborations using composite structure diagrams. They are similar to class diagrams except they represent individual parts in detail as compared to the entire class.

3.Object Diagram – An Object Diagram can be referred to as a screenshot of the instances in a system and the relationship that exists between them. Since object diagrams depict behaviour when objects have been instantiated, we are able to study the behaviour of the system at a particular instant. An object diagram is similar to a class diagram except it shows the instances of classes in the system. We depict actual classifiers and their relationships making the use of class diagrams. On the other hand, an Object Diagram represents specific instances of classes and relationships between them at a point of time.

4.Component Diagram – Component diagrams are used to represent the how the physical components in a system have been organized. We use them for modelling implementation details. Component Diagrams depict the structural relationship between software system elements and help us in understanding if functional requirements have been covered by planned development. Component Diagrams become essential to use when we design and build complex systems. Interfaces are used by components of the system to communicate with each other.

5.Deployment Diagram – Deployment Diagrams are used to represent system hardware and its software.It tells us what hardware components exist and what software components run on them.We illustrate system architecture as distribution of software artifacts over distributed

targets. An artifact is the information that is generated by system software. They are primarily used when a software is being used, distributed or deployed over multiple machines with different configurations.

6.Package Diagram – We use Package Diagrams to depict how packages and their elements have been organized. A package diagram simply shows us the dependencies between different packages and internal composition of packages. Packages help us to organise UML diagrams into meaningful groups and make the diagram easy to understand. They are primarily used to organise class and use case diagrams.

Behavior Diagrams –

1.State Machine Diagrams – A state diagram is used to represent the condition of the system or part of the system at finite instances of time. It's a behavioral diagram and it represents the behavior using finite state transitions. State diagrams are also referred to as State machines and State-chart Diagrams . These terms are often used interchangeably.So simply, a state diagram is used to model the dynamic behavior of a class in response to time and changing external stimuli.

2.Activity Diagrams – We use Activity Diagrams to illustrate the flow of control in a system. We can also use an activity diagram to refer to the steps involved in the execution of a use case. We model sequential and concurrent activities using activity diagrams. So, we basically depict workflows visually using an activity diagram.An activity diagram focuses on condition of flow and the sequence in which it happens. We describe or depict what causes a particular event using an activity diagram.

3.Use Case Diagrams – Use Case Diagrams are used to depict the functionality of a system or a part of a system. They are widely used to illustrate the functional requirements of the system and its interaction with external agents(actors). A use case is basically a diagram representing different scenarios where the system can be used. A use case diagram gives us a high level view of what the system or a part of the system does without going into implementation details.

4.Sequence Diagram – A sequence diagram simply depicts interaction between objects in a sequential order i.e. the order in which these interactions take place.We can also use the terms event diagrams or event scenarios to refer to a sequence diagram. Sequence diagrams describe how and in what order the objects in a system function. These diagrams are widely used by businessmen and software developers to document and understand requirements for new and existing systems.

5.Communication Diagram – A Communication Diagram(known as Collaboration Diagram in UML 1.x) is used to show sequenced messages exchanged between objects. A communication diagram

focuses primarily on objects and their relationships. We can represent similar information using Sequence diagrams,however, communication diagrams represent objects and links in a free form.

6.Timing Diagram – Timing Diagram are a special form of Sequence diagrams which are used to depict the behavior of objects over a time frame. We use them to show time and duration constraints which govern changes in states and behavior of objects.

Interaction Overview Diagram – An Interaction Overview Diagram models a sequence of actions and helps us simplify complex interactions into simpler occurrences. It is a mixture of activity and sequence diagrams.

Conceptual Model of UML

The Unified Modeling Language (UML) is a standard visual language for describing and modelling software blueprints. The UML is more than just a graphical language. Stated formally, the UML is for: Visualizing, Specifying, Constructing, and Documenting.
The artifacts of a software-intensive system (particularly systems built using the object-oriented style).

Three Aspects of UML:

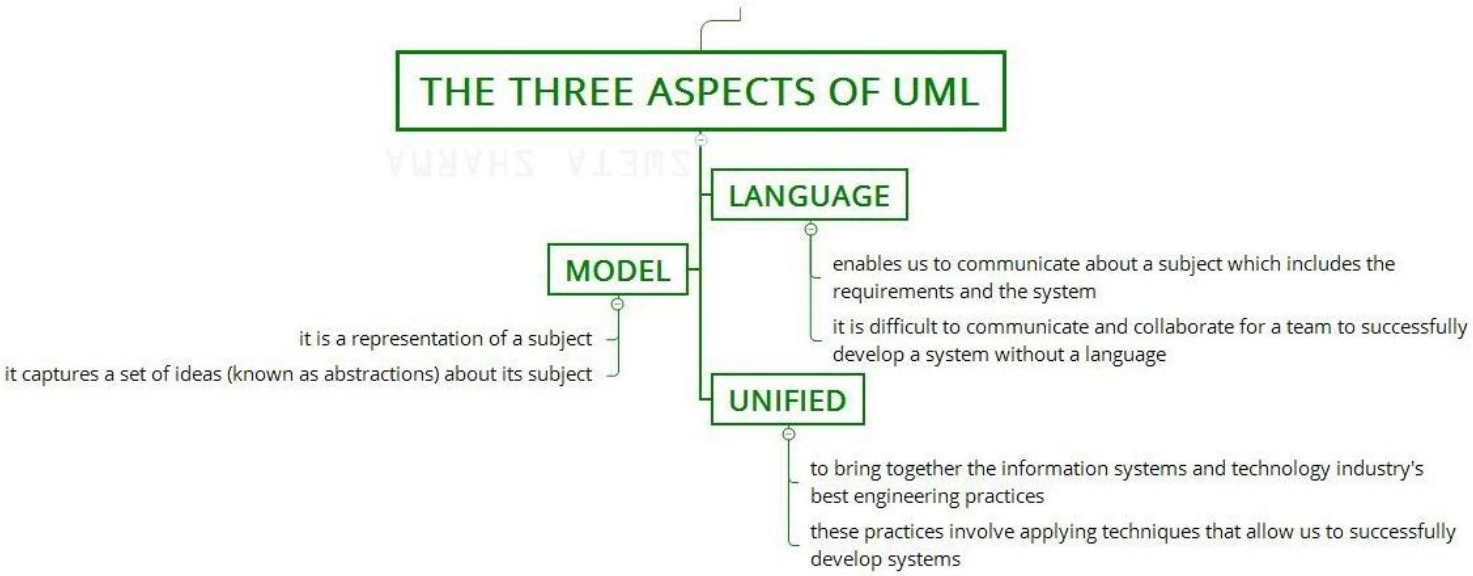


Figure – Three Aspects of UML

Note – Language, Model, and Unified are the important aspect of UML as described in the map above.

1. Language:

- It enables us to communicate about a subject which includes the requirements and the system.
- It is difficult to communicate and collaborate for a team to successfully develop a system without a language.

2. Model:

- It is a representation of a subject.
- It captures a set of ideas *(known as abstractions)* about its subject.

3. Unified:

- It is to bring together the information systems and technology industry's best engineering practices.
- These practices involve applying techniques that allow us to successfully develop systems.

A Conceptual Model:

A conceptual model of the language underlines the three major elements:

```
• The Building Blocks
• The Rules
• Some Common Mechanisms
```

Once you understand these elements, you will be able to read and recognize the models as well as create some of them.

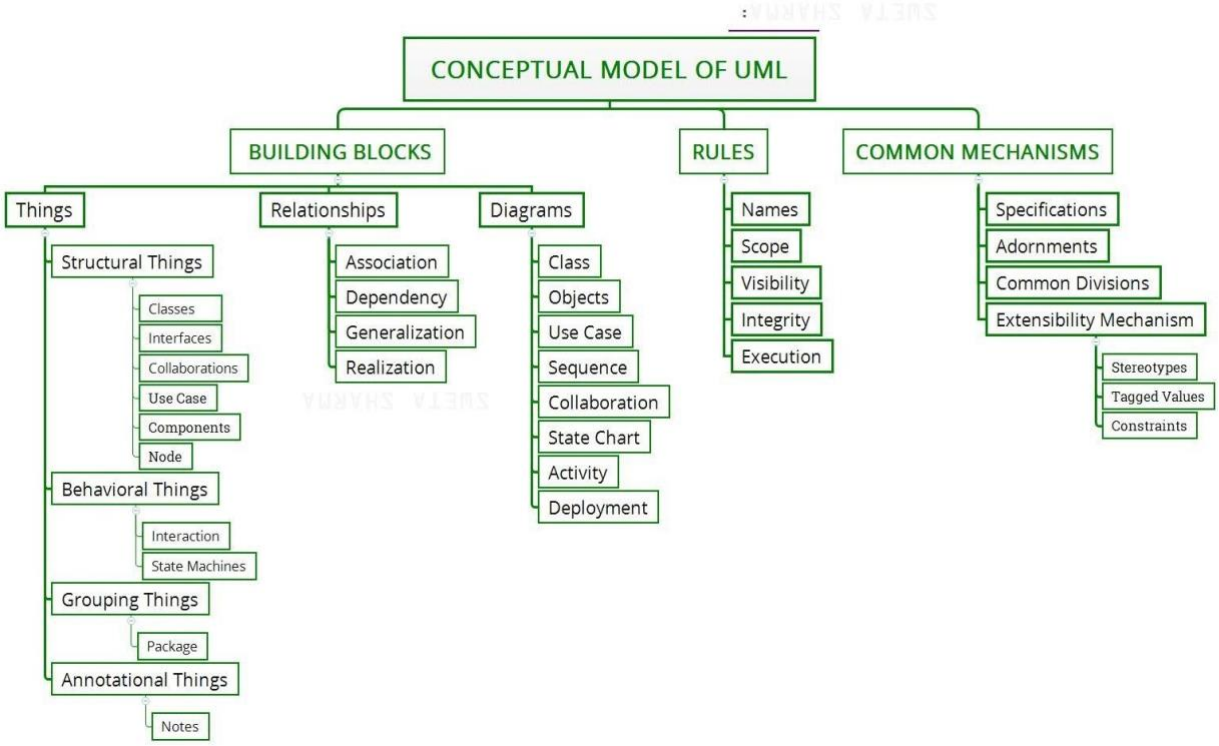


Figure – A Conceptual Model of the UML

Building Blocks:

The vocabulary of the UML encompasses three kinds of building blocks:

1. **Things:**
   Things are the abstractions that are first-class citizens in a model; relationships tie these things together; diagrams group interesting collections of things.
   There are 4 kinds of things in the UML:

```
1. Structural things
2. Behavioral things
3. Grouping things
4. Annotational things
```

   These things are the basic object-oriented building blocks of the UML. You use them to write well-formed models.

2. **Relationships:**
   There are 4 kinds of relationships in the UML:

```
1. Dependency
2. Association
3. Generalization
4. Realization
```

   These relationships are the basic relational building blocks of the UML.

3. **Diagrams:**
   It is the graphical presentation of a set of elements. It is rendered as a connected graph of vertices (things) and arcs (relationships).

```
1. Class diagram
2. Object diagram
3. Use case diagram
4. Sequence diagram
5. Collaboration diagram
6. Statechart diagram
7. Activity diagram
8. Component diagram
9. Deployment diagram
```

```
import java.io.*;
```

Rules:

The UML has a number of rules that specify what a well-formed model should look like. A well-formed model is one that is semantically self-consistent and in harmony with all its related models. The UML has semantic rules for:

1. **Names –** What you can call things, relationships, and diagrams.
2. **Scope –** The context that gives specific meaning to a name.
3. **Visibility –** How those names can be seen and used by others.
4. **Integrity –** How things properly and consistently relate to one another.
5. **Execution –** What it means to run or simulate a dynamic model.

Common Mechanisms:

The UML is made simpler by the four common mechanisms. They are as follows:

1. Specifications
2. Adornments
3. Common divisions
4. Extensibility mechanisms

Specifications: Behind every graphical notation in UML there is a precise specification of the details that element represents. For example, a class icon is a rectangle and it specifies the name, attributes and operations of the class.

Adornments: The mechanism in UML which allows the users to specify extra information with the basic notation of an element is the adornments.

In the above example, the access specifiers: + (public), # (protected) and – (private) represent the visibility of the attributes which is extra information over the basic attribute representation.

Common Divisions: In UML there is clear division between semantically related elements like: separation between a class and an object and the separation between an interface and its implementation.

Extensibility Mechanisms :UMLs extensibility mechanisms allow the user to extend (new additions) the language in a controlled way. The extensibility mechanisms in UML are:

- Stereotypes – Extends the vocabulary of UML. Allows users to declare new building blocks (icons) or extend the basic notations of the existing building blocks by stereotyping them using guillemets.
- Tagged Values – Extends the properties of an UML building block. Allows us to specify extra information in the elements specification. Represented as text written inside braces and placed under the element name. The general syntax of a property is:

  { property name = value }

- Constraints – Extends the semantics of a UMLs building block such as specifying new rules or modifying existing rules. Represented as text enclosed in braces and placed adjacent or beside the element name.

Example:

In the above example, we are specifying the exception “Overflow” using the class symbol and stereo typing it with “exception”. Also under the class name, “EventQueue” we are specifying additional properties like “version” and “author” using tagged values.

Finally, we are constraining the operation named “add” that before adding a new event to the EventQueue object, all the events must be “ordered” in some manner. This is specified using constraints in UML.

Software Architecture:

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

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

- Design View
- Implementation View
- Process View
- Deployment View
- Usecase View

And the centre is the Use Case view which connects all these four. A Use case represents the functionality of the system. So the other perspectives are connected with use case.

1) **Design** of a system consists of classes, interfaces and collaboration. UML provides class diagram, object diagram to support this.
2) **Implementation** defines the components assembled together to make a complete physical system. UML component diagram is used to support implementation perspective.
3) **Process** defines the flow of the system. So the same elements as used in Design are also used to support this perspective.
4) **Deployment** represents the physical nodes of the system that forms the hardware. UML deployment diagram is used to support this perspective.

Software Development Life Cycle:

The Unified Software Development Process

A software development process is the set of activities needed to transform a user„s requirements into a software system.

Basic properties:

• use case driven

• architecture centric

• iterative and incremental

Use case Driven

Use cases

• capture requirements of the user,

• divide the development project into smaller subprojects,

• are constantly refined during the whole development process

• are used to verify the correctness of the implemented software

Architecture Centric:

• Find structures which are suitable to achive the function specified in the use cases,

• understandable,

• maintainable,

• reusable for later extensions or newly discovered use cases and describe them, so that they can be communicated between developers and users.

Inception establishes the business rationale for the project and decides on the scope of the project.

Elaboration is the phase where you collect more detailed requirements, do high-level analysis and design to establish a baseline architecture and create the plan for construction.

Construction is an iterative and incremental process. Each iteration in this phase builds production- quality software prototypes , tested and integrated as subset of the requirements of the project.

Transition contains beta testing , performance tuning and user training.

Importance of Modeling

Modeling is a proven & well accepted engineering techniques. In building architecture, we develop architectural models of houses & high rises to help visualize the final products. In Unified Modeling Language (UML), a model may be structural, emphasizing the organization of the system or it may be behavioral, emphasizing the dynamics of the system. A model is a simplification of reality, providing blueprints of a system. UML, in specific:

- Permits you to specify the structure or behavior of a system.
- Helps you visualize a system.
- Provides template that guides you in constructing a system.
- Helps to understand complex system part by part.
- Document the decisions that you have made.

We build model so that we can better understand the system we are developing. A model may encompass an overview of the system under consideration, as well as a detailed planning for system design, implementation and testing.

Advantages:

- Provides standard for software development.
- Reducing of costs to develop diagrams of UML using supporting tools.
- Development time is reduced.
- The past faced issues by the developers are no longer exists.
- Has large visual elements to construct and easy to follow.

Types of modeling

The three types of modeling in UML are as follows:

1. Structural modeling:

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

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

2. Behavioral modeling:

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

3. Architectural modeling:

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

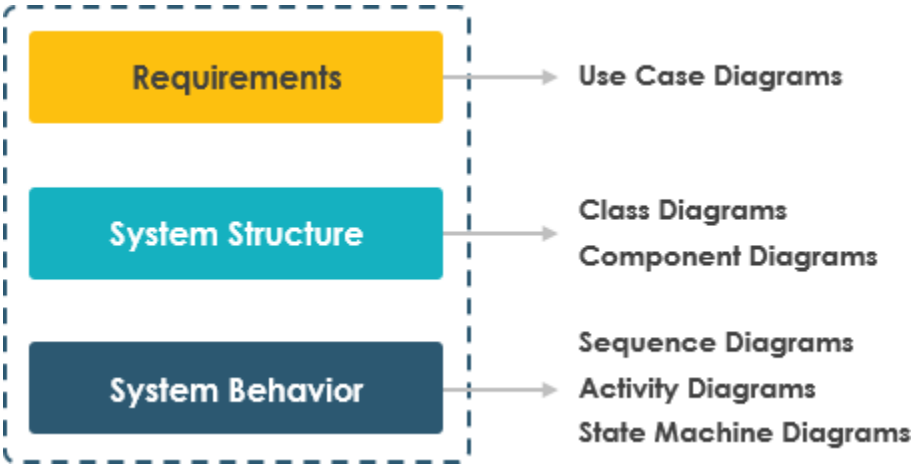
📄 Principles of UML Modeling

1. The choice of model is important

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

- The right models will highlight the most critical development problems.
- Wrong models will mislead you, causing you to focus on irrelevant issues.

For Example: We can use different types of diagrams for different phases in software development.



2. Every model may be expressed at different levels of precision

For Example,

- If you are building a high rise, sometimes you need a 30,000-foot view for instance, to help your investors visualize its look and feel.
- Other times, you need to get down to the level of the studs for instance, when there's a tricky pipe run or an unusual structural element.

3. The best models are connected to reality

All models simplify reality and a good model reflects important key characteristics.

4. No single model is sufficient

Every non-trivial system is best approached through a small set of nearly independent models.

Create models that can be built and studied separately, but are still interrelated. In the case of a building:

- You can study electrical plans in isolation
- But you can also see their mapping to the floor plan and perhaps even their interaction with the routing of pipes in the plumbing plan.

MODULE-2

Class Diagram

Class diagrams are the main building blocks of every object oriented methods. The class diagram can be used to show the classes, relationships, interface, association, and collaboration. UML is standardized in class diagrams. Since classes are the building block of an application that is based on OOPs, so as the class diagram has appropriate structure to represent the classes, inheritance, relationships, and everything that OOPs have in its context. It describes various kinds of objects and the static relationship in between them.
The main purpose to use class diagrams are:

- This is the only UML which can appropriately depict various aspects of OOPs concept.
- Proper design and analysis of application can be faster and efficient.
- It is base for deployment and component 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 −

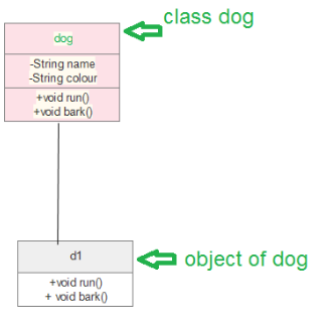
- The name of the class diagram should be meaningful to describe the aspect of the system.
- Each element and their relationships should be identified in advance.

- 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.

There are several software available which can be used online and offline to draw these diagrams Like Edraw max, lucid chart etc. There are several points to be kept in focus while drawing the class diagram. These can be said as its syntax:

- Each class is represented by a rectangle having a subdivision of three compartments name, attributes and operation.
- There are three types of modifiers which are used to decide the visibility of attributes and operations.
  - + is used for public visibility(for everyone)
  - # is used for protected visibility (for friend and derived)
  - – is used for private visibility (for only me)

Below is the example of Animal class (parent) having two child class as dog and cat both have object d1, c1 inheriting properties of the parent class.



Process to design class diagram: In Edraw max (or any other platform where class diagrams can be drawn) follow the steps:

- Open a blank document in the class diagram section.
- From the library select the class diagram and click on create option.
- Prepare the model of the class in the opened template page.
- After editing according to requirement save it.

There are several diagram components which can be efficiently used while making/editing the model. These are as follows:

- Class { name, attribute, method}

- Objects
- Interface
- Relationships {inheritance, association, generalization}
- Associations {bidirectional, unidirectional}

Class diagrams are one of the most widely used diagrams in the fields of software engineering as well as businesses modelling.

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 −

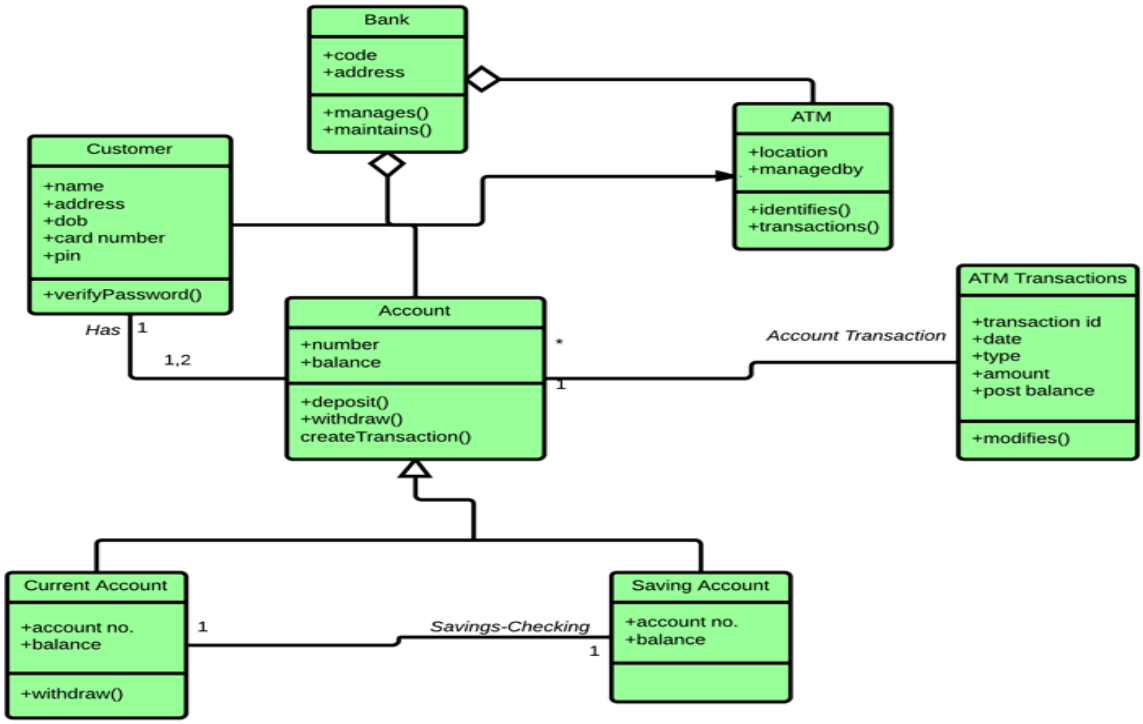
- Describing the static view of the system.
- Showing the collaboration among the elements of the static view.
- Describing the functionalities performed by the system.
- Construction of software applications using object oriented languages.

Example of UML Class Diagram

Creating a class diagram is a straightforward process. It does not involve many technicalities. Here, is an example:

ATMs system is very simple as customers need to press some buttons to receive cash. However, there are multiple security layers that any ATM system needs to pass. This helps to prevent fraud and provide cash or need details to banking customers.

Below given is a UML Class Diagram example:



Class Diagram in Software Development Lifecycle

Class diagrams can be used in various software development phases. It helps in modeling class diagrams in three different perspectives.

1. Conceptual perspective: Conceptual diagrams are describing things in the real world. You should draw a diagram that represents the concepts in the domain under study. These concepts related to class and it is always language-independent.

2. Specification perspective: Specification perspective describes software abstractions or components with specifications and interfaces. However, it does not give any commitment to specific implementation.

3. Implementation perspective: This type of class diagrams is used for implementations in a specific language or application. Implementation perspective, use for software implementation

Common Modelling Techniques

1.To model the vocabulary of a system

Modeling the vocabulary of a system involves making a decision about which abstractions are a part of the system under consideration and which fall outside its boundaries. You use class diagrams to specify these abstractions and their responsibilities.

2. To model simple collaborations

A collaboration is a society of classes, interfaces, and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements. For example, when you re modeling the semantics of a transaction in a distributed system, you can't just stare at a single class to understand what's going on. Rather, these semantics are carried out by a set of classes that work together. You use class diagrams to visualize and specify this set of classes and their relationships.

To model a collaboration,

• Identify the mechanism you'd like to model. A mechanism represents some function or
behavior of the part of the system you are modeling that results from the interaction of a society
of classes, interfaces, and other things.
• For each mechanism, identify the classes, interfaces, and other collaborations that participate in
this collaboration. Identify the relationships among these things as well.
• Use scenarios to walk through these things. Along the way, you'll discover parts of your model
that were missing and parts that were just plain semantically wrong.
• Be sure to populate these elements with their contents. For classes, start with getting a good
balance of responsibilities. Then, over time, turn these in to concrete attributes and operations.

3. To model a logical database schema

Think of a schema as the blueprint for the conceptual design of a database. In many domains, you'll want to store persistent information in a relational database or in an object-oriented database. You can model schemas for these databases using class diagrams.

To model a schema,

• Identify those classes in your model whose state must transcend the lifetime of their
applications.
• Create a class diagram that contains these classes. You can define your own set of stereotypes

and tagged values to address database-specific details.
• Expand the structural details of these classes. In general, this means specifying the details of
their attributes and focusing on the associations and their multiplicities that relate these classes.
• Watch for common patterns that complicate physical database design, such as cyclic
associations and one-to-one associations. Where necessary, create intermediate abstractions to
simplify your logical structure.
• Consider also the behavior of these classes by expanding operations that are important for data
access and data integrity. In general, to provide a better separation of concerns, business rules
concerned with the manipulation of sets of these objects should be encapsulated in a layer above
these persistent classes.
• Where possible, use tools to help you transform your logical design into a physical design.

Advanced Classes

A classifier is a mechanism that has structural features (in theform of attributes), as well as behavioral features (in the form of operations). Classifiers include classes, interfaces, datatypes, signals, components, nodes, use cases, and subsystems. Those modeling elements that can have instances are called classifiers. Every instance of a given classifier shares the same features. The most important kind of classifier in UML is class. The other kinds of classifiers are given in Table: 1.

• Interface	A collection of operations that are used to specify a service of a class or a component
• Datatype	A type whose values have no identity, including primitive built-in types (such as numbers and strings), as well as enumeration types (such as Boolean)
• Signal	The specification of an asynchronous stimulus communicated between instances
• Component	A physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces
• Node	A physical element that exists at run time and that represents a computational resource, generally having at least some memory and often processing capability
• Use case	A description of a set of a sequence of actions, including variants, that a system performs that yields an observable result of value to a particular actor
• Subsystem	A grouping of elements of which some constitute a specification of the behavior offered by the other contained elements

Table:1 Classifier Description

Classifiers represented graphically are shown in Figure: 1.

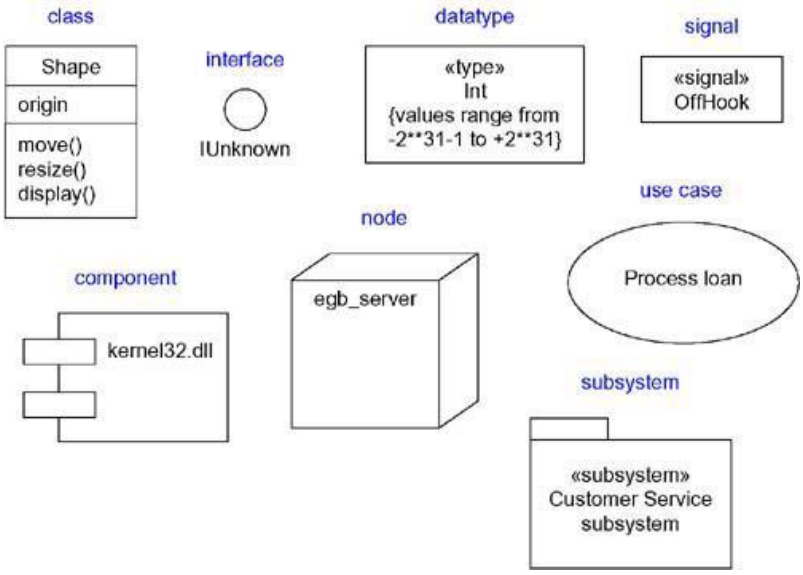


Figure 1: Graphical representation of classifiers

Visibility of a Classifier

Visibility indicates whether the attributes and operations of a classifier can be used by any other classifiers. There are three levels of visibility in UML public, protected and private. A classifier can see another classifier if it is in scope and if there is an explicit or implicit relationship to the target. Default visibility of a feature is public in UML. Table 2 shows the various visibilities of the classifiers.

1. public	Any outside classifier with visibility to the given classifier can use the feature; specified by prepending the symbol +
2. protected	Any descendant of the classifier can use the feature; specified by prepending the symbol #
3. private	Only the classifier itself can use the feature; specified by prepending the symbol

Table:2 Possibile Visibilities of Classifier

Figure: 2 indicates the various visibilities of class Toolbar

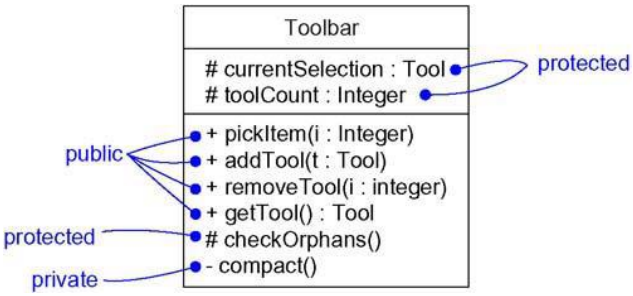


Figure:2 Class showing visibility

Scope(Owner Scope) of a Classifier

The owner scope of a feature(attribute/operations) specifies whether the feature appears in each instance of the classifier or whether there is just a single instance of the feature for all instances of the classifier. Two kinds of owner scope – classifier scope and instance scope. An instance scope is an owner scope in which each instance of the classifier holds its own value for the feature whereas classifier scopeis the one which have just one value of the feature for all instances of the classifier. classifier scope is rendered(shown) by underlining the feature's name. No adornment means that the feature is instance scoped. Figure:3 shows the scope of a classifier.

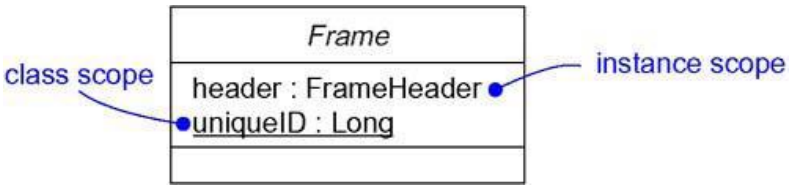


Figure:3 Class showing its Scope

Abstract, Root, Leaf and Polymorphic Elements

Abstract classes are those that do not have any direct instances and is specified in UML by writing its name in italics. A leaf class is a class that have no children and is specified in UML by writing the property leaf below the class's name. A root class is a class that has no parents and is specified in UML by writing the property root below the class's name.
An operation is polymorphic if it is specified with the same signature at different places in the hierarchy of classes. Which operation to invoke is done polymorphically, that is a match is determined at run time according to the type of the object. These are indicated in Figure:4

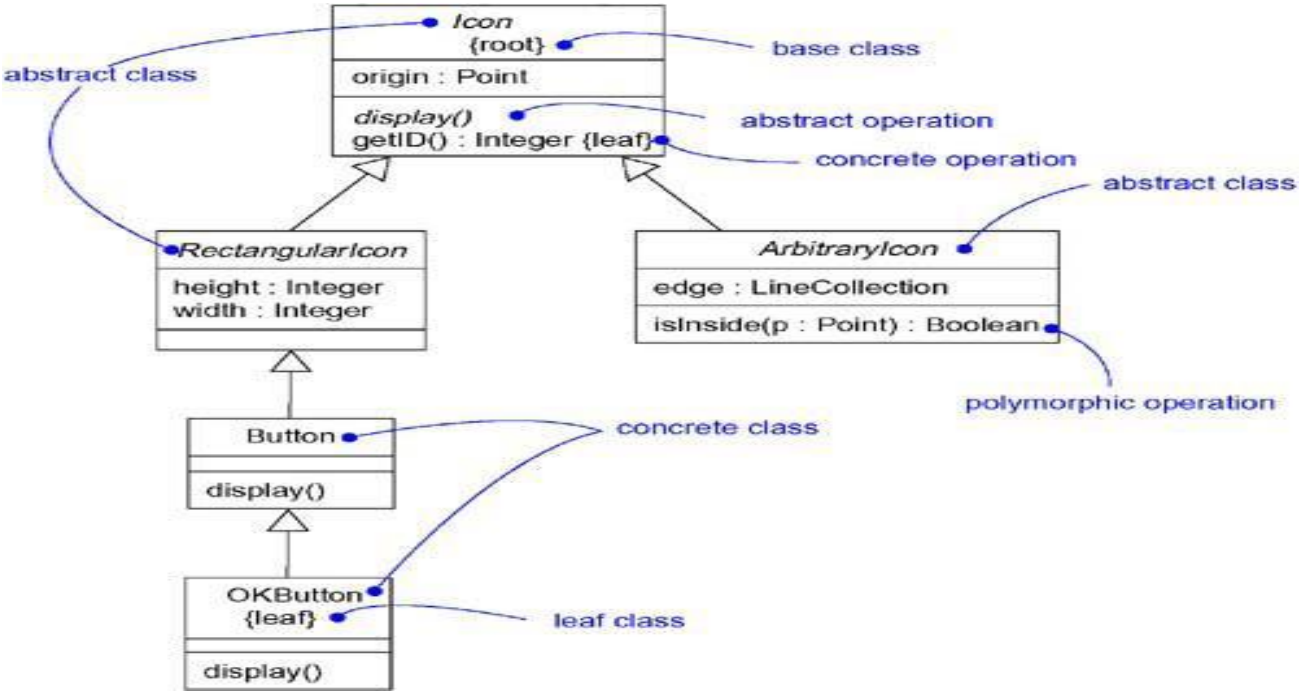


Figure:4 class Diagram indicating root,leaf,polymorphism and abstract

For example, display and isInside are both polymorphic operations. Furthermore, the operation Icon::display() is abstract, meaning that it is incomplete and requires a child to supply an implementation of the operation. In the UML, you specify an abstract operation by writing its name in italics, just as you do for a class. By contrast, Icon::getID() is a leaf operation as indicated by the property leaf. This means that the operation is not polymorphic and may not be overridden.

Multiplicity

The number of instances a class may have is called its multiplicity. Multiplicity applies to attributes, as well. A class having single instance is called as a singleton class. It is indicated in Figure:5

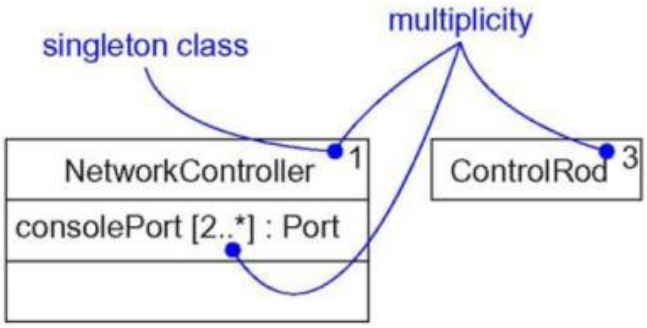


Figure:5 Multiplicity

Attributes

A class's structural features are indicated by its attributes.

The *syntax of an attribute* in UML is

[visibility] name [multiplicity] [: type] [= initial-value] [{property-string}]

Some legal attribute declarations are given in Table:3.

• origin	Name only
• + origin	Visibility and name
• origin : Point	Name and type
• head : *Item	Name and complex type
• name [0..1] : String	Name, multiplicity, and type
• origin : Point = (0,0)	Name, type, and initial value
• id : Integer {frozen}	Name and property

Table:3 Attributes

Three defined *properties that can be used with attribute* values are given in Table:4.

1. changeable	There are no restrictions on modifying the attribute's value.
2. addOnly	For attributes with a multiplicity greater than one, additional values may be added, but once created, a value may not be removed or altered.
3. frozen	The attribute's value may not be changed after the object is initialized.

Table:4 Attribute properties

Where default property is ‘changeable’.

Operations

A class’s behavioral features are indicated by its operations.
The UML *distinguishes between operation and method*. An operation specifies a service that can be requested from any object of the class to affect behavior; a method is an implementation of an operation.
The *syntax of an operation* in the UML is
[visibility] name [(parameter-list)][: return-type] [{property-string}]
All legal operation declarations are indicated in Table:5

• display	Name only
• + display	Visibility and name
• set(n : Name, s : String)	Name and parameters
• getID() : Integer	Name and return type
• restart() {guarded}	Name and property

Table:5 Legal Operations

In an *operation*‘s signature, you may provide zero or more parameters, each of which follows the *syntax.*
[direction] name : type [= default-value]
Direction of a parameter may be any of the following values given in Table:6.

• in	An input parameter; may not be modified
• out	An output parameter; may be modified to communicate information to the caller
• inout	An input parameter; may be modified

Table:6 Direction Parameter

In addition to the leaf property described earlier, there are four defined properties that can be used with operations. They are given in Table:7.

1. isQuery	Execution of the operation leaves the state of the system unchanged. In other words, the operation is a pure function that has no side effects.
2. sequential	Callers must coordinate outside the object so that only one flow is in the object at a time. In the presence of multiple flows of control, the semantics and integrity of the object cannot be guaranteed.
3. guarded	The semantics and integrity of the object is guaranteed in the presence of multiple flows of control by sequentializing all calls to all of the object's guarded operations. In effect, exactly one operation at a time can be invoked on the object, reducing this to sequential semantics.
4. concurrent	The semantics and integrity of the object is guaranteed in the presence of multiple flows of control by treating the operation as atomic. Multiple calls from concurrent flows of control may occur simultaneously to one object on any concurrent operation, and all may proceed concurrently with correct semantics; concurrent operations must be designed so that they perform correctly in the case of a concurrent sequential or guarded operation on the same object.

Table: 7 Operation_property

Template Classes

A template is a parameterized element. A template includes slots for classes,objects, and values, and these slots serve as the template's parameters. Every templates should be instantiated first. Instantiation involves binding these formal template parameters to actual ones. For a template class, the result is a concrete class that can be used just like any ordinary class.
The *instantiation of a template class can be modelled in two ways*.
First done implicitly, by declaring a class whose name provides the binding. Second, explicitly by using a dependency stereotyped as bind, which specifies that the source instantiates the target template using the actual parameters. A template class is indicated in Figure: 6.

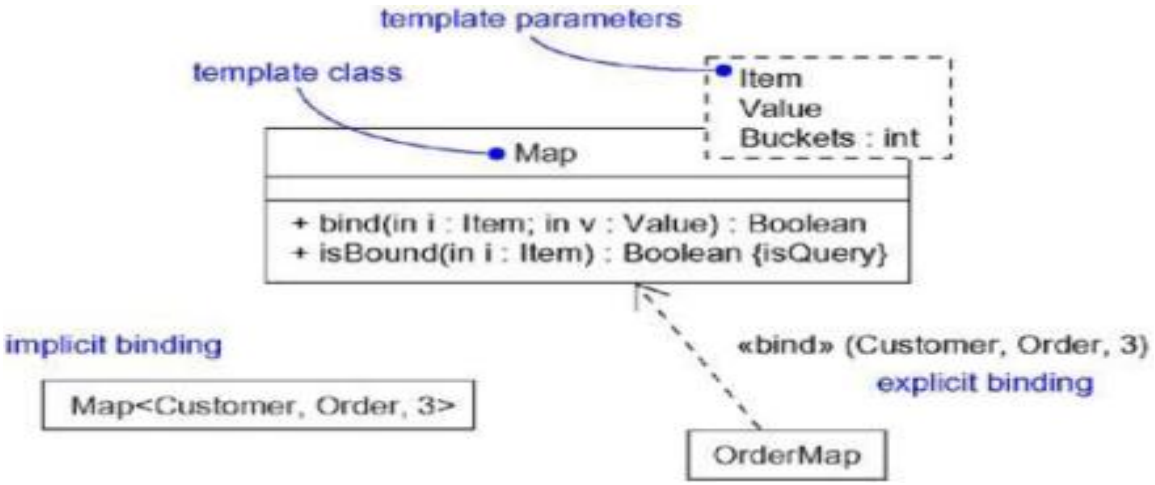


Figure:6 Template class

UML defines four standard stereotypes that apply to classes. They are indicated in Table:8.

1. metaclass	Specifies a classifier whose objects are all classes
2. powertype	Specifies a classifier whose objects are the children of a given parent
3. stereotype	Specifies that the classifier is a stereotype that may be applied to other elements
4. utility	Specifies a class whose attributes and operations are all class scoped

Table:8 Standard Prototypes

Advanced Relationship

A relationship is a connection among things(things can be any of, structural, behavioural, annotational or grouping. This module contains mainly structural things). In object-oriented modeling, the four most important relationships are dependencies, generalizations, associations, and realizations. Diagram indicating advanced relationship is shown in Figure: 1

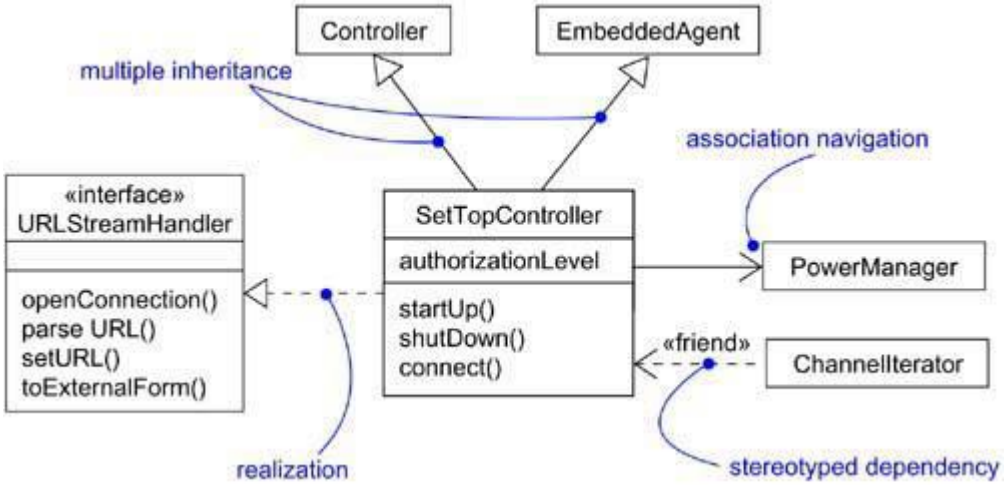


Figure: 1 Advanced Relationship

Dependency

A dependency is a using relationship, specifying that a change in the specification of one thing may affect another thing that uses it. The dependency relationship is also known as using relationship i.e., if the specification of one thing (for example Graphics class code) changes then it will automatically affect the behavior of another thing (for example Hello class using the drawstring method of Graphics class) that uses it. A dependency relationship is graphically represented as a dashed arrow.

Stereotypes for Dependency relationship (17 stereotypes in 6 groups)

✓dependency relationships *among classes and objects* in class diagrams

«***bind***» – Specifies that the source instantiates the target template using the given actual parameters

«***derive***» – Specifies that the source may be computed from the target

«***friend***» – Specifies that the source is given special visibility into the target

«***instanceOf***» – Specifies that the source object is an instance of the target classifier

«***instantiate***» – Specifies that the source creates instances of the target

«***powertype***» – Specifies that the target is a powertype of the source; a powertype is a classifier whose objects are all the children of a given parent

«***refine***» – Specifies that the source is at a finer degree of abstraction than the target

«***use***» – Specifies that the semantics of the source element depends on the semantics of the public part of the target

✓dependency relationships *among packages*

«***access***» – Specifies that the source package is granted the right to reference the elements of the target package

«***import***» – A kind of access that specifies that the public contents of the target package enter the flat namespace of the source, as if they had been declared in the source

✓dependency relationships *among use cases*

«***extend***» – Specifies that the target use case extends the behavior of the source

«***include***» – Specifies that the source use case explicitly incorporates the behavior of another use case at a location specified by the source

✓when modeling interactions *among objects*

«***become***» – Specifies that the target is the same object as the source but at a later point in time and with possibly different values, state, or roles

«***call***» – Specifies that the source operation invokes the target operation

«***copy***» – Specifies that the target object is an exact, but independent, copy of the source

✓in the *context of state machines*

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

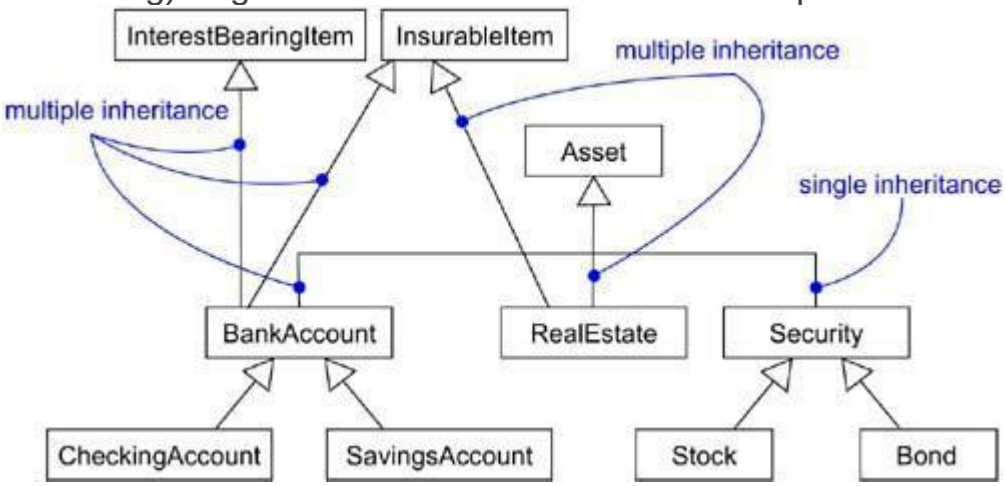
✓in the *context of organizing the elements of your system into subsystems and models* is

«***trace***» – Specifies that the target is an historical ancestor of the source.

Generalization

A generalization is a relationship between a general thing and a more specific kind of that thing. Here child will inherit all the structure and behaviour of the parent and can even add new structure and behavior, or it may modify the behavior of the parent(i.e,

overriding). Figure: 2 shows various inheritance possible in UML.



UML defines *one stereotype and four constraints* that may be applied to generalization relationships.

«***implementation***» – Specifies that the child inherits the implementation of the parent but does not make public nor support its interfaces, thereby violating substitutability.
Note: use implementation when you want to model private inheritance

{**complete**} – Specifies that all children in the generalization have been specified in the model and that no additional children are permitted
Note: use the complete constraint when you want to show explicitly that you've fully specified a hierarchy in the model

{**incomplete**} – Specifies that not all children in the generalization have been specified and that additional children are permitted
Note: use incomplete to show explicitly that you have not stated the full specification of the hierarchy in the model

{**disjoint**} – Specifies that objects of the parent may have no more than one of the children as a type

{**overlapping**} – Specifies that objects of the parent may have more than one of the children
as a type
Note: use disjoint and overlapping when you want to distinguish between static classification (disjoint) and dynamic classification (overlapping).

Association

An association is a structural relationship, specifying that objects of one thing are connected to objects of another.
Navigation :Unless otherwise specified, navigation across an association is bidirectional. Unidirectional navigation is also possible where reverse navigation is not desirable such as where security concern is an important thing. A Unidirectional navigation is shown in Figure:3.

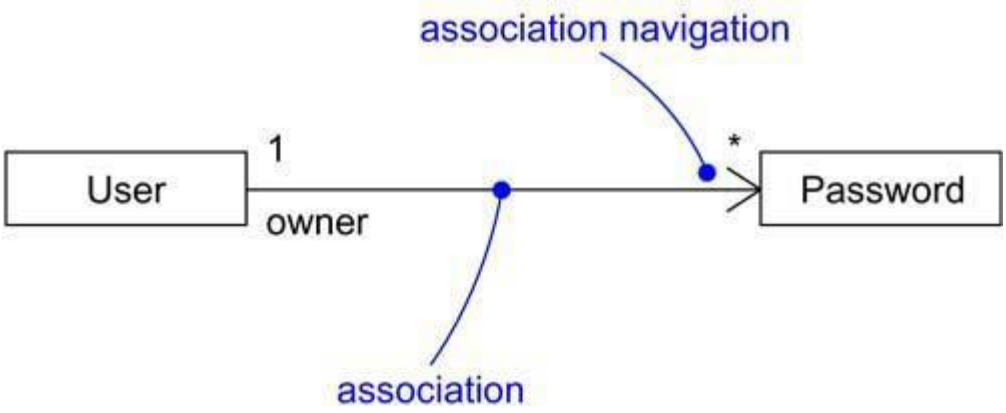


Figure:3 unidirectional-navigation

Visibility : For an association between two classes, objects of one class can see and navigate to objects of the other, unless otherwise restricted by an explicit statement of navigation. *Three levels of visibility* for an association possible in UML. Default visibility of a role is public. *Private visibility* indicates that objects at that end are not accessible to any objects outside the association; *protected visibility* indicates that objects at that end are not accessible to any objects outside the association, except for children of the other end.

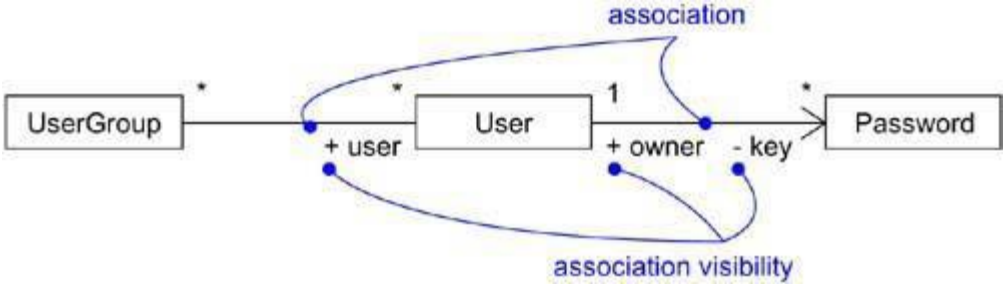


Figure:4 Association_Visibility

Qualification :A qualifier is an association attribute whose values partition the set of objects related to an object across an association. It is graphically represented as a small rectangle attached to the end of an association, placing the attributes in the rectangle. For example If you can devise a lookup data structure at one end of an association (for example, a hash table or b-tree), then manifest that index as a qualifier. In most cases, the source end's multiplicity will be many and the target end's multiplicity will be 0..1.

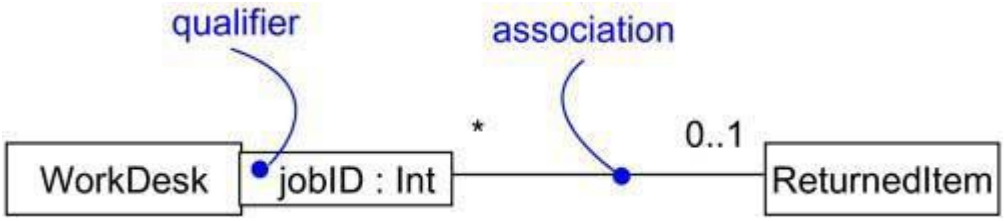


Figure:5 Qualification

Interface Specifier : An interface is a collection of operations that are used to specify a service of a class or a component; every class may realize many interfaces. In the context of an association with another target class, a source class may choose to present only part of its face(interfaces) to the world and this part that is chosen is called as the interface specifier. Figure: 6 illustrates this.

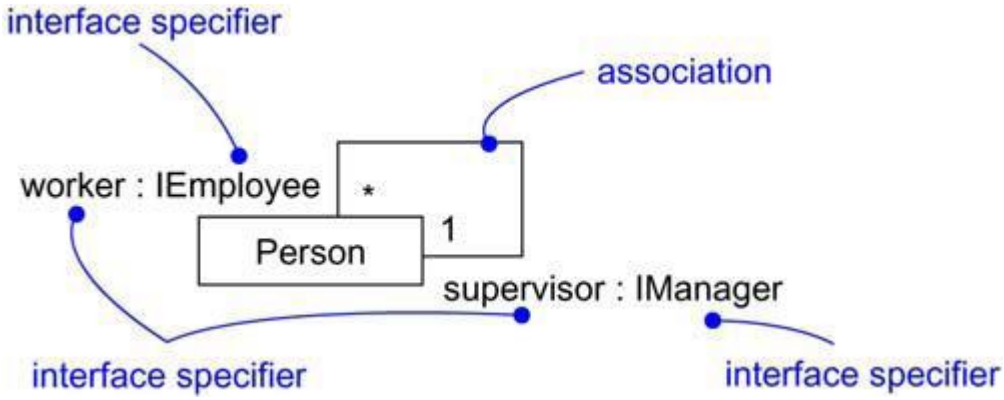


Figure:6 Interface Specifier

Here in Figure:6, In the vocabulary of a HR system, a Person class may realize many interfaces such as IManager, IEmployee, IOfficer, and so on. A relationship between a supervisor and workers with a one-to-many association is shown by explicitly labeling the roles of this association as supervisor and worker. In the context of this association, a Person in the role of supervisor presents only the IManager face to the worker; a Person in the role of worker presents only the IEmployee face to the supervisor. We can explicitly show the type of role using the syntax rolename : iname, where iname is some interface of the other classifier.

Composition : In a composite aggregation, an object may be a part of only one composite at a time. For example, in a windowing system, a Frame belongs to exactly one Window. In a composite aggregation, the whole is responsible for the disposition

of its parts, which means that the composite must manage the creation and destruction of its parts. For example, when you create a Frame in a windowing system, you must attach it to an enclosing Window. Similarly, when you destroy the Window, the Window object must in turn destroy its Frame parts.

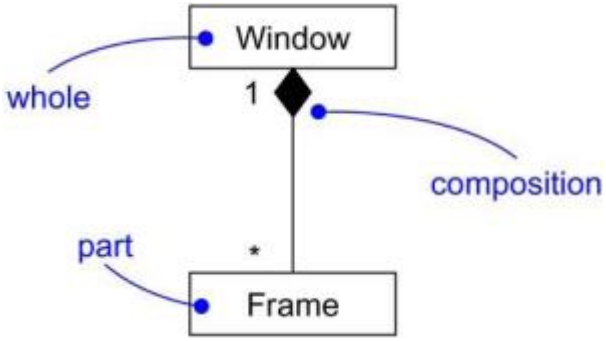


Figure:7 Composition

Association Classes :An association class can be seen as an association that also has class properties, or as a class that also has association properties. An association class is represented in Figure:8.

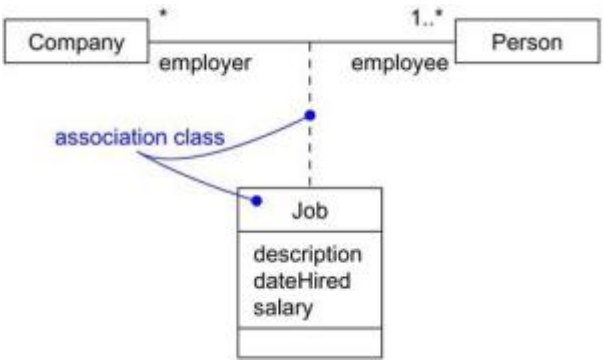


Figure:8 Association class

Here in figure:8 the associations on both ends can be a many to many that indicates a person can be an employee in more than one company at a time. So in this case it is not possible to place the job details in any of the class(Company or Person). In such a situation we go for association class.

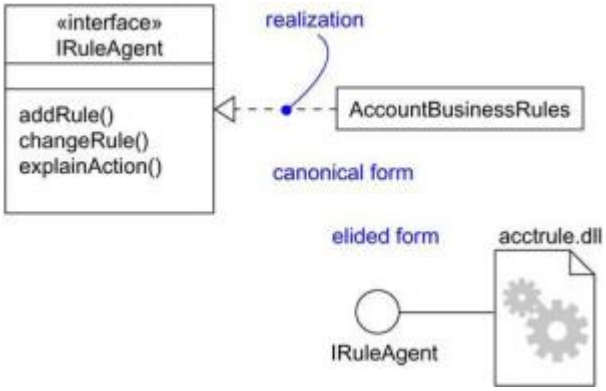
five constraints that can be applied to association relationships.

{**implicit**} – Specifies that the relationship is not manifest but, rather,is only conceptual
{**ordered**} – Specifies that the set of objects at one end of an association are in an explicit order
constraints that *relate to the changeability of the instances* of an association.
{**changeable**} – Links between objects may be added, removed, and changed freely
{**addOnly**} – New links may be added from an object on the opposite end of the association
{**frozen**} – A link, once added from an object on the opposite end of the association, may not be modified or deleted

constraint *for managing related sets* of associations
{**xor**} – Specifies that, over a set of associations, exactly one is manfest for each associated object.

Realization

A realizationis a semantic relationship between classifiers in which one classifier specifies a contract that another classifier guarantees to carry out. Realization can be represented in two ways: in the canonical form (using the interface stereotype and the dashed directed line with a large open arrowhead) and in an elided form (using the interface lollipop notation). It is represented in Figure:9.



Common Modeling Techniques

Modeling Webs of Relationships

To model webs of relationships,

1. Apply use cases and scenarios to find the relationships between the abstractions in the system.
2. Start by modeling the structural relationships (associations) between the things. These specify the structure of the system.
3. Then, model the generalization-specialization relationships.
4. Finally, after modeling the remaining relationships go for dependency relationships.
5. After representing all the relationships, transform their basic representation by applying the advanced features to your intent.

Interface, Types and Roles

Interface

An interface is a collection of operations that are used to specify a service of a class or a component. ***Graphically***, an interface is rendered(represented) as a circle; in its expanded form, an interface may be rendered as a stereotyped class(a class with sterotype interface) as shown in Figure:1. An interface name must be unique within its enclosing package. ***Two naming mechanism***; a simple name(only name of the interface), a path name is the interface name prefixed by the name of the package in which that interface lives represented in Figure: 2. To distinguish an interface from a class, prepend an 'I' to every interface name. Operations in an interface may be adorned with visibility properties, concurrency properties, stereotypes, tagged values, and constraints.interface dont have attributes. interfaces span model boundaries and it does'nt have direct instances.

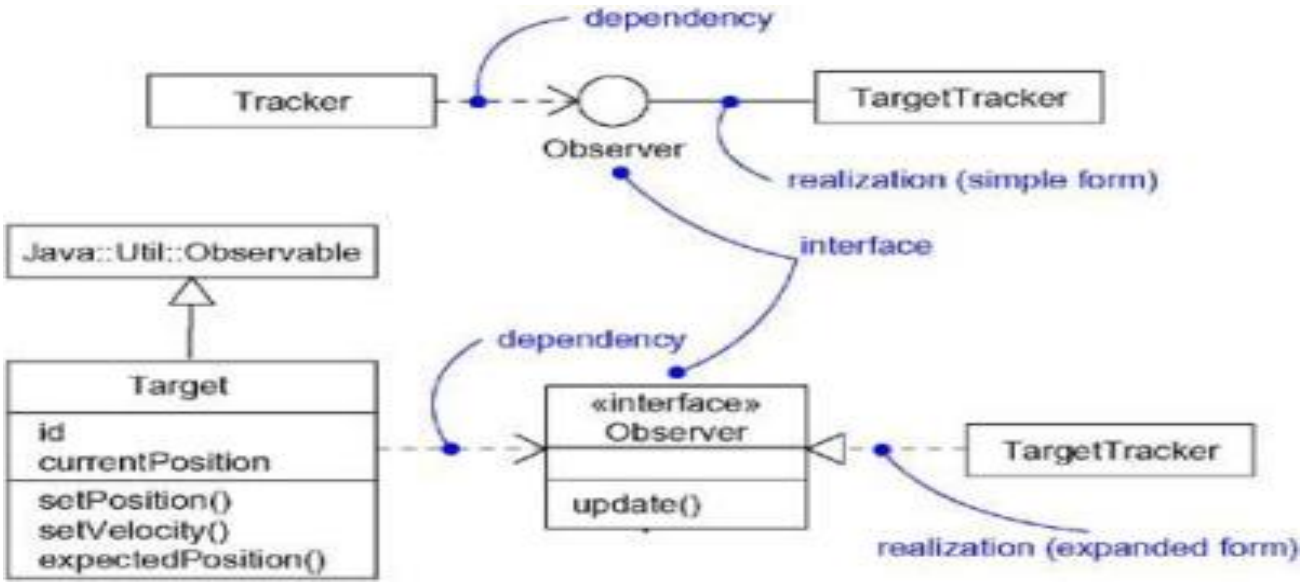


Figure:1

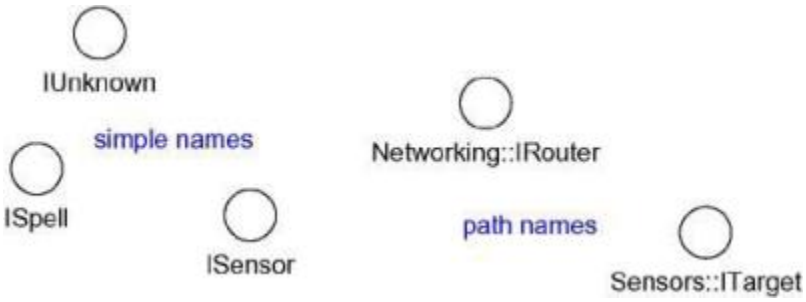


Figure:2

Interface relationships

An interface may participate in generalization, association, dependency and realization

relationships.
Note: Interfaces may also be used to specify a contract for a use case or subsystem.

Type

A type is a stereotype of a class used to specify a domain of objects, together with the operations (but not the methods) applicable to the object of that type. To distinguish a type from an interface or a class, prepend a 'T' to every type. Stereotype type is used to formally model the semantics of an abstraction and its conformance to a specific interface.

Role

A role names(indicates) a behavior of an entity participating in a particular context. Or, a role is the face that an abstraction presents to the world. For example, consider an instance of the class Person. Depending on the context, that Person instance may play the role of Mother, Comforter, PayerOfBills, Employee, Customer, Manager, Pilot, Singer, and so on. When an object plays a particular role, it presents a face to the world, and clients that interact with it expect a certain behavior depending on the role that it plays at the time. For example, an instance of Person in the role of Manager would present a different set of properties than if the instance were playing the role of Mother. Figure:3 indicates a role employee played by person and is represented statically there.

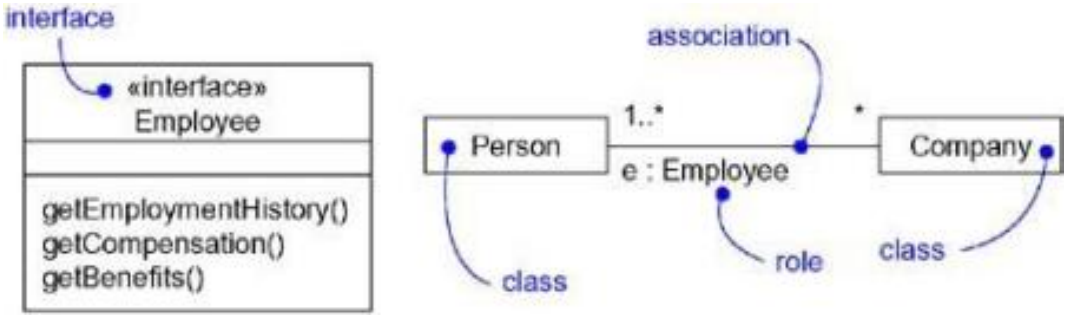


Figure:3 Roles

In Figure:3 the Person presents the role of Employee to the Company, and in that context, only the properties specified by Employee are visible and relevant to the Company.

Static and Dynamic modeling in UML

A class diagram that indicates a particular role is useful for modeling the static binding of an abstraction to its interface. To model the dynamic binding of an abstraction to its interface by using the become stereotype in an interaction diagram, showing an object changing from one role to another.

To model a dynamic type,

- Specify the different possible types of that object by rendering each type as a class stereotyped as type (if the abstraction requires structure and behavior) or as interface (if the abstraction requires only behavior).
- Model all the roles the class of the object may take on at any point in time. It can be done in two ways:

1. First, in a class diagram, explicitly type each role that the class plays in its association with other classes. Doing this specifies the face instances of that class put on in the context of the associated object.
2. Second, also in a class diagram, specify the class-to-type relationships using generalization

- In an interaction diagram, properly render each instance of the dynamically typed class. Display the role of the instance in brackets below the object's name.
- To show the change in role of an object, render the object once for each role it plays in the interaction, and connect these objects with a message stereotyped as become.

They are represented in the following figures:

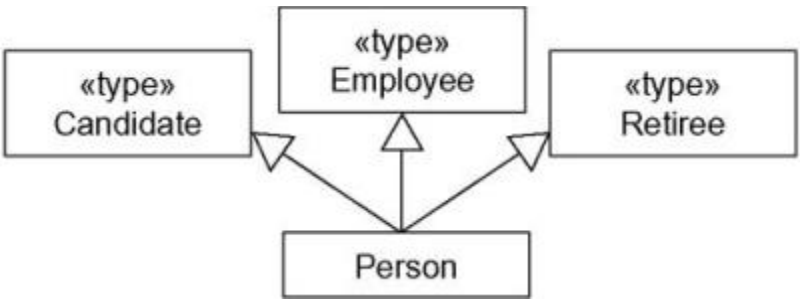


Figure:4 Static modeling

Figure:4 shows statically that instances of the Person class may be any of the three types namely, Candidate, Employee, or Retiree.

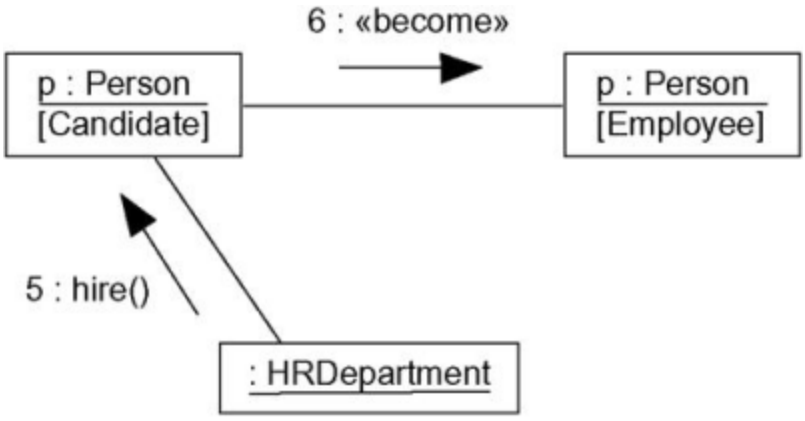


Figure:5 Dynamic-modeling

Figure:5 shows the dynamic nature of a person's type. In this fragment of an interaction diagram, p (the Person object) changes its role from Candidate to Employee.

Packages

Package

Package is the UML mechanism for grouping things. It can be used to:

- group semantically related elements;
- define a "semantic boundary" in the model;
- provide units for parallel working and configuration management;
- package is used to provide an encapsulated namespace within which all names must be unique.

Analysis packages contain:

- use cases
- analysis classes
- use case realizations

A package syntax is shown in any of the form as in Figure:1.

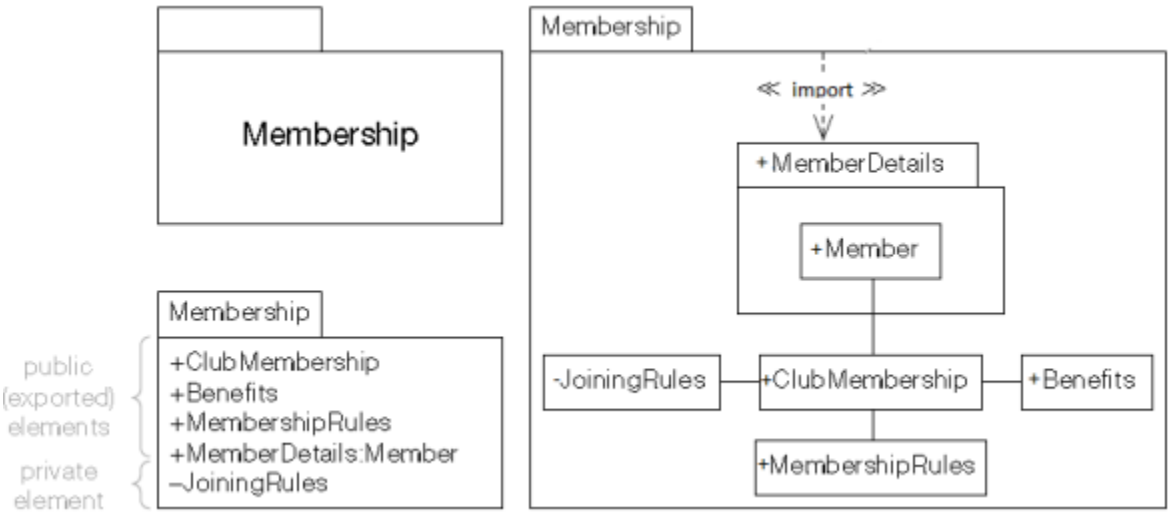


Figure:1 Package syntax

Visibilities of elements inside a package

An element with public visibility are visible to elements outside the package and they are exported by the package and are represented by pre-pending '+'.

An element with private visibility are completely hidden inside the package and are represented by pre-pending '-'.

Standard package stereotypes

<<framework>> – if the package contains model elements that specify a reusable

architecture.
<<modelLibrary>> – if the package contains elements that are intented to be reused by other packages.

A package define automatically an ***encapsulated namespace*** which means within the package boundary all elements name are unique. A ***qualified name*** is the representation by which an element inside a package is accessed. It is indicated with the help of '::'. for eg:- a class Librarian inside a package users which again is inside in Library is accessed as Library:: Users:: Librarian

Nested packages
Packages may be nested inside other packages to any depth.
Two possible representations of nested packages are shown in Figure:2. First one is commonly used.

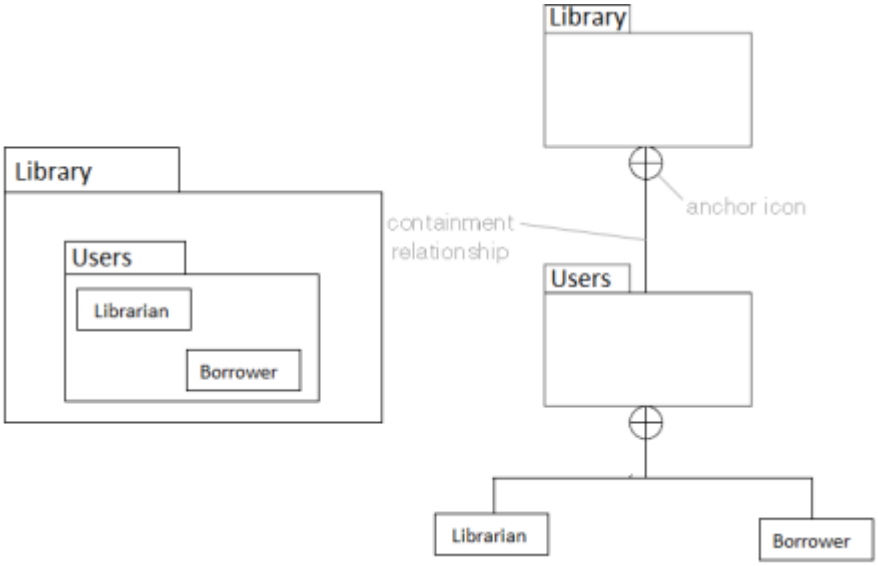


Figure:2 Nested Packages

Package dependencies
A dependency relationship indicates that one package depends in some way on another. Different dependency are explained below.
«use»– This means that an element in the client uses a public element in the supplier in some way i.e the client depends on the supplier. If a package dependency is shown without a stereotype, then «use» should be assumed. Shown in Figure:3.

Figure:3 Use dependency

«import»– Public elements of the supplier namespace are added as public elements to the client namespace. Elements in the client can access all public elements in the supplier using unqualified names. Shown in Figure:4.

Figure:4 import dependency

«access»– Public elements of the supplier namespace are added as private elements to the client namespace. Elements in the client can access all public elements in the supplier using unqualified names. Shown in Figure:5.

Figure:5 access dependency

«trace»– «trace» usually represents an historical development of one element into another more developed version – it is usually a relationship between models rather than elements.A complete UML model can be represented by a package with a small triangle in its top right hand corner. Shown in Figure:6.

Figure:6 trace dependency

«merge»– Public elements of the supplier package are merged with elements of the client package. used only in meta modeling, not used in OO analysis and design. Shown in Figure:7.

Figure:7 merge dependency

Transitivity

Transitivity means that if there is a relationship between thing A and thing B and a relationship between thing B and thing C, then there is an implicit relationship between

thing A and thing C. «access» dependency is not transitive and «import» dependency is transitive. This is illustrated in Figure: 8.

Figure:8 package-transitivity

Lack of transitivity in «access»means that:
public elements in package C become private elements in package B;
public elements in package B become private elements in package A;
Therefore elements in package A can't see elements in package C.

Package generalization

In package generalization, the more specialized child packages inherit the public and protected elements from their parent package. Child packages may add new elements, and may override elements in the parent package by providing an alternative implementation with the same name. This is repreasented in Figure: 9.

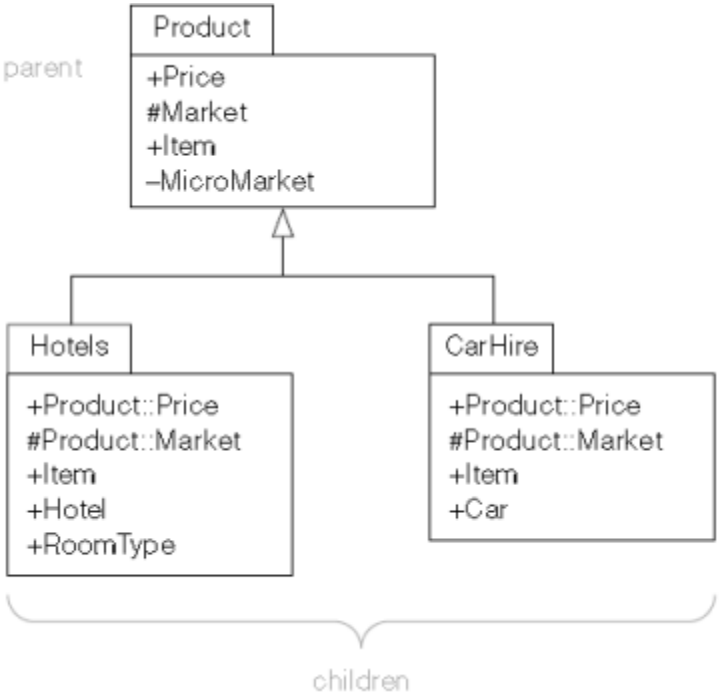


Figure: 9 Package Generalization

Architectural analysis

Architectural analysis partitions related classes into analysis packages, and then layers the packages.Represented in Figure: 10.

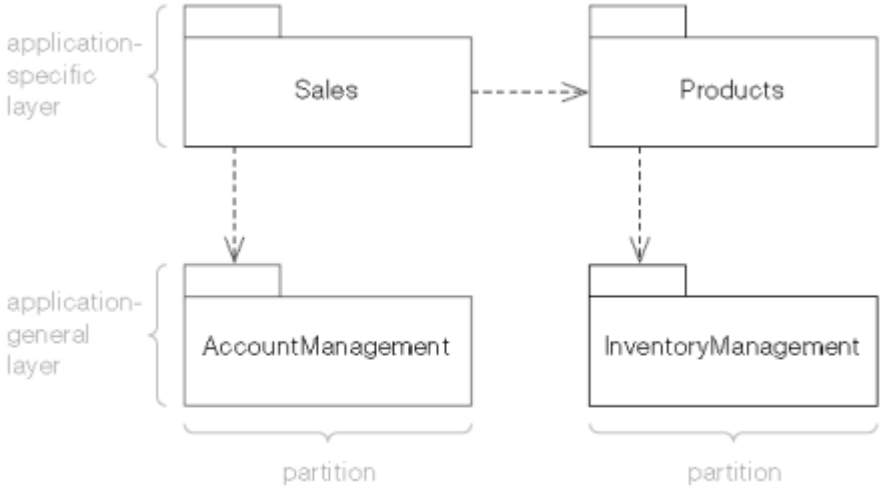


Figure: 10 Architectural analysis

One of the goals in architectural analysis is to try to minimize the amount of coupling in the system. It can be done in three ways:

- minimize the dependencies between analysis packages;
- minimize the number of public and protected elements in each analysis package;
- maximize the number of private elements in each analysis package.

Systems that exhibit a high degree of coupling are typically complex and difficult to build and maintain.

Finding analysis packages

Analysis packages are found by identifying groupings of model elements that have strong semantic connections. They are often discovered over a period of time as the model develops and matures. The keys to good package structure are high cohesion within a package, and low coupling between packages. A package should contain a group of closely related classes.

Cyclic package dependencies

Avoid cyclic dependencies in the analysis package model.If package A depends in some way on package B, and vice versa, there is a very strong argument for just merging the two packages and this is a perfectly valid way of removing cyclic dependencies.This is shown in Figure: 11.

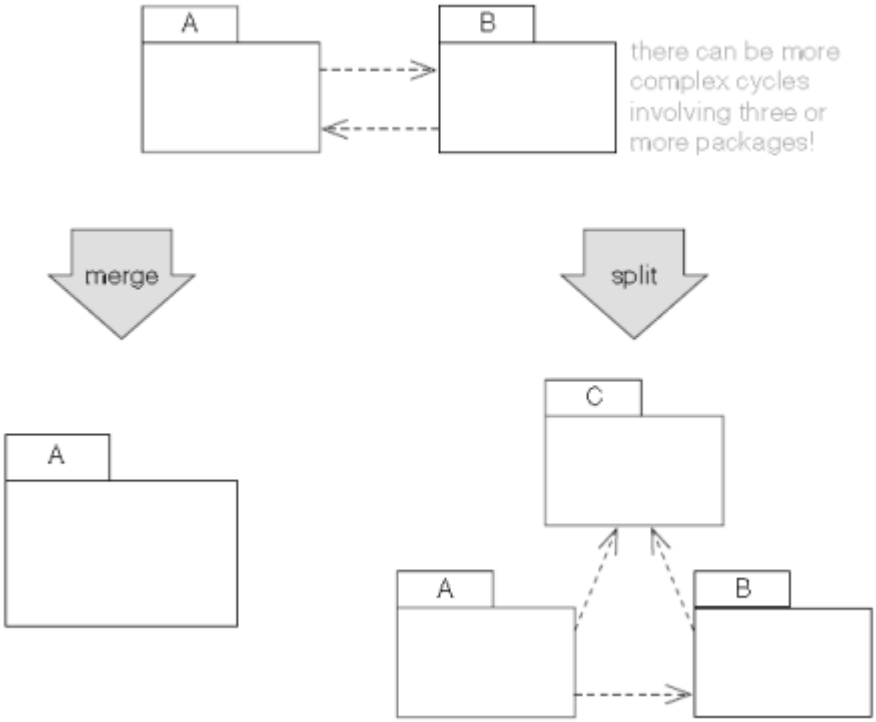


Figure:11 Cyclic package dependencies

Suppose we have a very simple model with one class in package A and another class in package B. If the class in package A has a bidirectional relationship with the class in package B, then package A depends on package B, but package B also depends on package A – we have a cyclic dependency between the two packages. The only ways to remove this violation are to refine the relationship between A and B by making it unidirectional(split), or to put the two classes in the same package(merge). Try to factor the common elements out into a third package C as shown in figure above.

MODULE-3

Instances

- Instance is a concrete manifestation of an abstraction to which a set of operations can be applied and which has a state that stores the effects of the operations.
- instances and objects are almost same (we cannot say an object of association, it is an instance of association called link)
- an instance is rendered(shown) by underlining its name
- Graphical representation for instances – named instances, as well as anonymous Figure 1.
- Instances don't stand alone; they are tied to an abstraction
- instances can be of classes, components, nodes, use cases, and associations
- Every instance must have a name that distinguishes it from other instances within its context. Figure 2.
- an instance can be used to things dynamically eg:- calling an operation using a class instance.
- an instance/ object can have a state -> it's static properties along with current(dynamic) values it holds. so when you visualize its state, you are really specifying the value of its state at a given moment in time and space. Figure 3.
- Active objects are represented as shown in Figure 4.

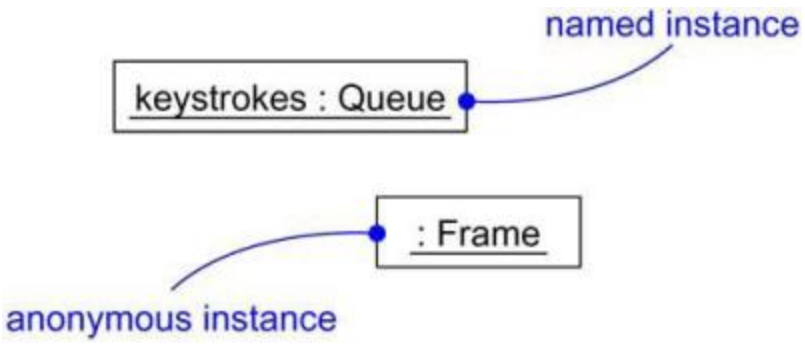


Figure1: instances

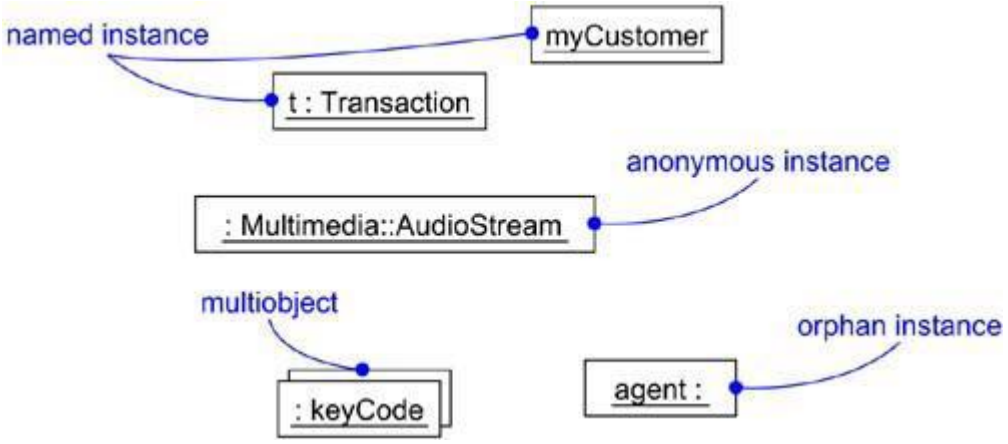


Figure: 2 Named, Anonymous, Multiple, and Orphan Instances

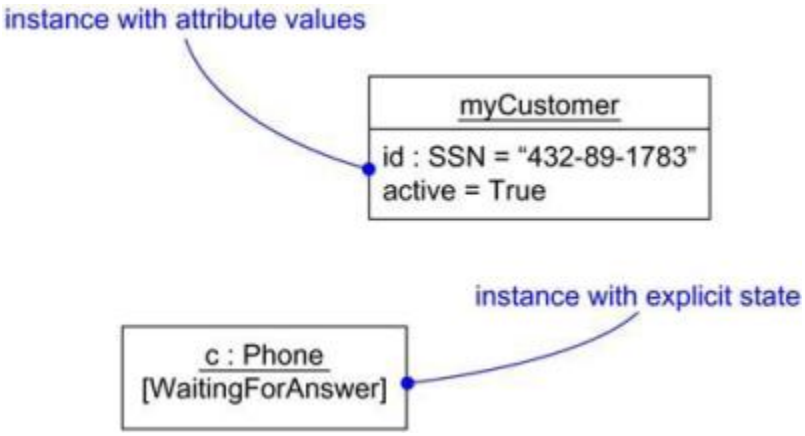


Figure 3 Object or instnace State

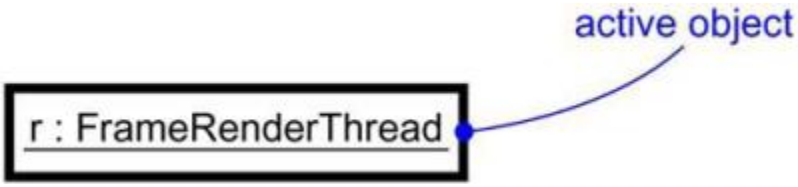


Figure 4 Active Objects

two standard stereotypes that apply to the dependency relationships among objects and among classes:
1. instanceOf – Specifies that the client object is an instance of the supplier classifier
2. instantiate – Specifies that the client class creates instances of the supplier class
two stereotypes related to objects that apply to messages and transitions:
1. become – Specifies that the client is the same object as the supplier, but at a later time and with possibly different values, state, or roles
2. copy – Specifies that the client object is an exact but independent copy of the supplier
standard constraint that applies to objects:
1.transient – Specifies that an instance of the role is created during execution of the enclosing
2.interaction – but is destroyed before completion of execution

Modeling Concrete Instances
To model concrete instances,

- Identify those instances necessary and sufficient to visualize, specify, construct, or document the problem you are modeling.
- Render these objects in the UML as instances. Where possible, give each object a name. If there is no meaningful name for the object, render it as an anonymous object.
- Expose the stereotype, tagged values, and attributes (with their values) of each instance necessary and sufficient to model your problem.
- Render these instances and their relationships in an object diagram or other diagram appropriate to the kind of the instance.

Figure 5 shows an object diagram drawn from the execution of a credit card validation system

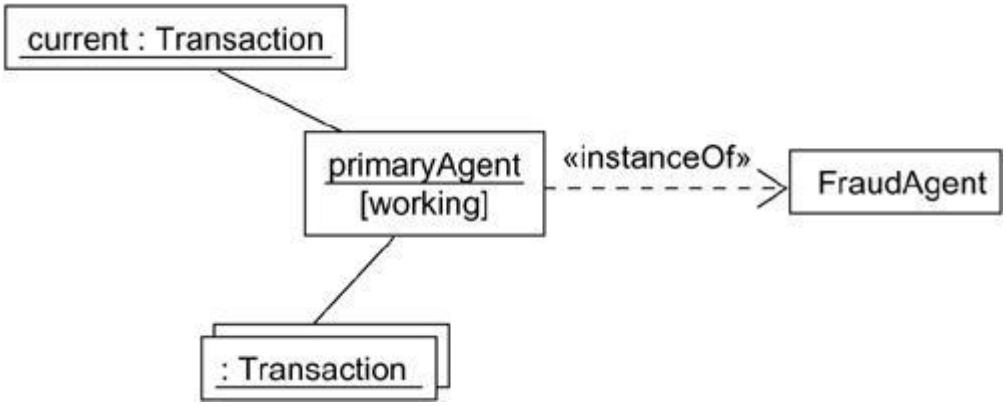


Figure 5 Modeling Concrete Instances

Modeling Prototypical Instances

To model prototypical instances,

- Identify those prototypical instances necessary and sufficient to visualize, specify, construct, or document the problem you are modeling.
- Render these objects in the UML as instances. Where possible, give each object a name. If there is no meaningful name for the object, render it as an anonymous object.
- Expose the properties of each instance necessary and sufficient to model your problem.
- Render these instances and their relationships in an interaction diagram or an activity diagram.

Figure 6 shows an interaction diagram illustrating a partial scenario for initiating a phone call in the context of a switch. There are four prototypical objects: a (a CallingAgent), c (a Connection), and t1 and t2 (both instances of Terminal). All four of these objects are prototypical; all represent conceptual proxies for concrete objects that may exist in the real world.

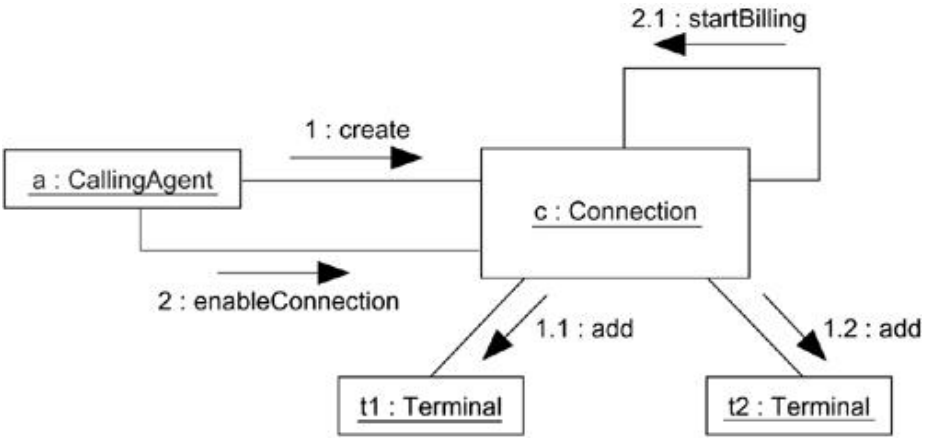


Figure 6 Modeling Prototypical Instances

Object Diagram

An **Object Diagram** can be referred to as a screenshot of the instances in a system and the relationship that exists between them. Since object diagrams depict behaviour when objects have been instantiated, we are able to study the behavior of the system at a particular instant. Object diagrams are vital to portray and understand functional requirements of a system.
In other words, "An object diagram in the Unified Modeling Language (UML), is a diagram that shows a **complete or partial view** of the structure of a modeled system **at a specific time.**"

Difference between an Object and a Class Diagram –
An object diagram is similar to a class diagram except it shows the instances of classes in the system. We depict actual classifiers and their relationships making the use of class diagrams. On the other hand, an Object Diagram represents specific instances of classes and relationships between them at a point of time.

What is a classifier?

In UML a classifier refers to a group of elements that have some common features like methods, attributes and operations. A classifier can be thought of as an abstract metaclass which draws a boundary for a group of instances having common static and dynamic features. For example, we refer a class, an object, a component, or a deployment node as classifiers in UML since they define a common set of properties.

An Object Diagram is a structural diagram which uses notation similar to that of class diagrams. We are able to design object diagrams by instantiating classifiers.

Object Diagrams use **real world examples** to depict the nature and structure of the system at a particular **point in time**. Since we are able to use data available within objects, Object diagrams provide a **clearer view** of the relationships that exist **between objects**.

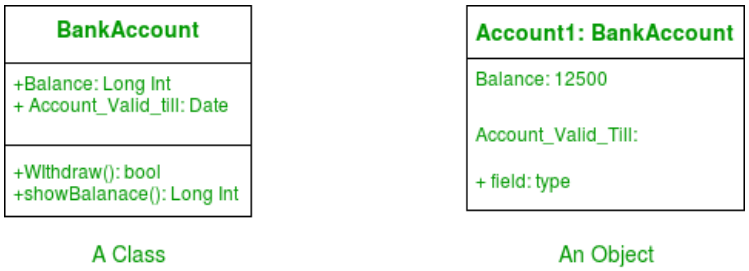


Figure – a class and its corresponding object

Notations Used in Object Diagrams –

- Objects or Instance specifications – When we instantiate a classifier in a system, the object we create represents an entity which exists in the system.We can represent the changes in object over time by creating multiple instance specifications. We use a rectangle to represent an object in an Object Diagram. An object is generally linked to other objects in an object diagram.

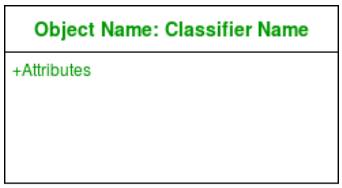


Figure – notation for an Object

For example – In the figure below, two objects of class Student are linked to an object of class College.

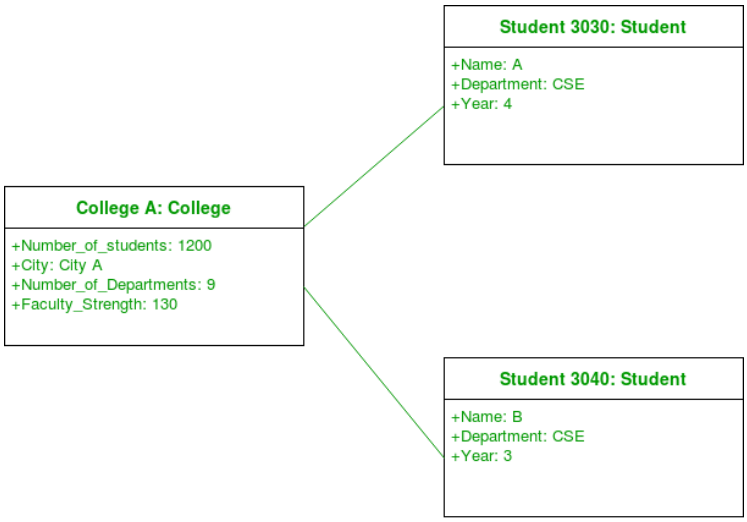


Figure – an object diagram using a link and 3 objects

- Links – We use a link to represent a relationship between two objects.

Figure – notation for a link

We represent the number of participants on the link for each end of the link. We use the term association for a relationship between two classifiers. The term link is used to specify a relationship between two instance specifications or objects. We use a solid line to represent a link between two objects.

Notation	Meaning
0..1	Zero or one
1	One only
0..*	Zero or more
*	Zero or more
1..*	One or more
7	Seven only
0..2	Zero or two
4..7	Four to seven

- Dependency Relationships – We use a dependency relationship to show when one element depends on another element.

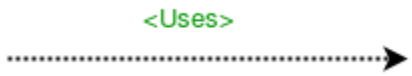


Figure – notation for dependency relationship

Class diagrams, component diagrams, deployment and object diagrams use dependency relationships. A dependency is used to depict the relationship between dependent and independent entities in the system.Any change in the definition or structure of one element may cause changes to the other. This is a unidirectional kind of relationship between two objects.
Dependency relationships are of various types specified with keywords (sometimes within angular brackets").

Abstraction,Binding, Realization, Substitution and Usage are the types of dependency relationships used in UML.
For example – In the figure below, an object of Player class is dependent (or uses) an object of Bat class.

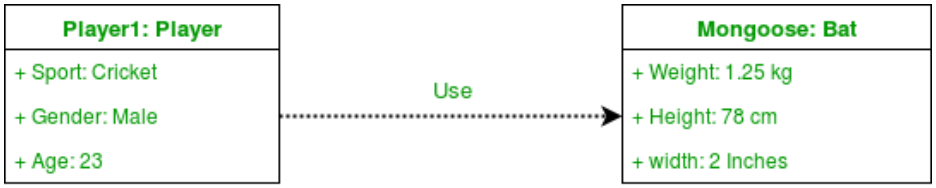


Figure – an object diagram using a dependency relationship

- Association – Association is a reference relationship between two objects (or classes).

Figure – notation for association

Whenever an object uses another it is called an association.We use association when one object references members of the other object. Association can be uni-directional or bi-directional. We use an arrow to represent association.
For example – The object of Order class is associated with an object of Customer class.

Figure – an object diagram using association

- Aggregation – Aggregation represents a “has a” relationship.

Figure – notation for aggregation

Aggregation is a specific form of association.on relationship; aggregation is more specific than ordinary association. It is an association that represents a part-whole or part-of relationship. It is a kind of parent -child relationship however it isn’t inheritance. Aggregation occurs when the lifecycle of the contained objects does not strongly depend on the lifecycle of container objects.

Figure – an object diagram using aggregation

For example – A library has an aggregation relationship with books. Library has books or books are a part of library. The existence of books is independent of the existence of the library.While implementing, there isn’t a lot of difference between aggregation and association. We use a hollow diamond on the containing object with a line which joins it to the contained object.

- Composition – Composition is a type of association where the child cannot exist independent of the other.

Figure – notation for composition

Composition is also a special type of association. It is also a kind of parent child relationship but it is not inheritance. Consider the example of a boy Gurkaran: Gurkaran is composed of legs and arms. Here Gurkaran has a composition relationship with his legs and arms. Here legs and arms cant exist without the existence of their parent object. So whenever independent existence of the child is not possible we use a composition relationship. We use a filled diamond on the containing object with a line which joins it to the contained object.

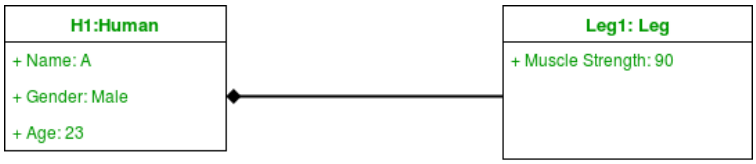


Figure – an object diagram using composition

For Example – In the figure below, consider the object Bank1. Here an account cannot exist without the existence of a bank.

Figure – a bank is composed of accounts

Difference between Association and Dependency –

Association and dependency are often confused in their usage. A source of confusion was the use of transient links in UML 1. Meta-models are now handled differently in UML 2 and the issue has been resolved.

There are a large number of dependencies in a system. We only represent the ones which are essential to convey for understanding the system. We need to understand that every association implies a dependency itself. We , however, prefer not to draw it separately. An association implies a dependency similar to a way in which generalization does.

How to draw an Object Diagram?

- Draw all the necessary class diagrams for the system.
- Identify the crucial points in time where a system snapshot is needed.
- Identify the objects which cover crucial functionality of the system.
- Identify the relationship between objects drawn.

Uses of an Object Diagram –

5) Model the static design(similar to class diagrams) or structure of a system using prototypical instances and real data.
6) Helps us to understand the functionalities that the system should deliver to the users.
7) Understand relationships between objects.
8) Visualise, document, construct and design a static frame showing instances of objects and their relationships in the dynamic story of life of a system.
9) Verify the class diagrams for completeness and accuracy by using Object Diagrams as specific test cases.
10) Discover facts and dependencies between specific instances and depicting specific examples of classifiers.

UML Behavioral Models

UML Behavioral Diagrams depict the elements of a system that are dependent on time and that convey the dynamic concepts of the system and how they relate to each other. The elements in these diagrams resemble the verbs in a natural language and the relationships that connect them typically convey the passage of time. For example, a behavioral diagram of a vehicle reservation system might contain elements such as Make a Reservation, Rent a Car, and Provide Credit Card Details. Experienced modelers will show the relationship to structural elements on these diagrams.

UML behavioral diagrams visualize, specify, construct, and document the dynamic aspects of a system. The behavioral diagrams are categorized as follows: use case diagrams, interaction diagrams, state–chart diagrams, and activity diagrams.

Interaction

- An interaction is a behavior that is composed of a set of messages exchanged among a set of objects within a context (it can be class, an operation, classifier, system or subsystem) to accomplish a purpose.
- A message specifies the communication between objects for an activity to happen. It has following parts: its name, parameters (if any), and sequence number. Figure 1: Messages, Links, and Sequencing.
- objects in an interaction can be concrete things(represents something in the real world.) or prototypical things (any instance of Person- like a professional)

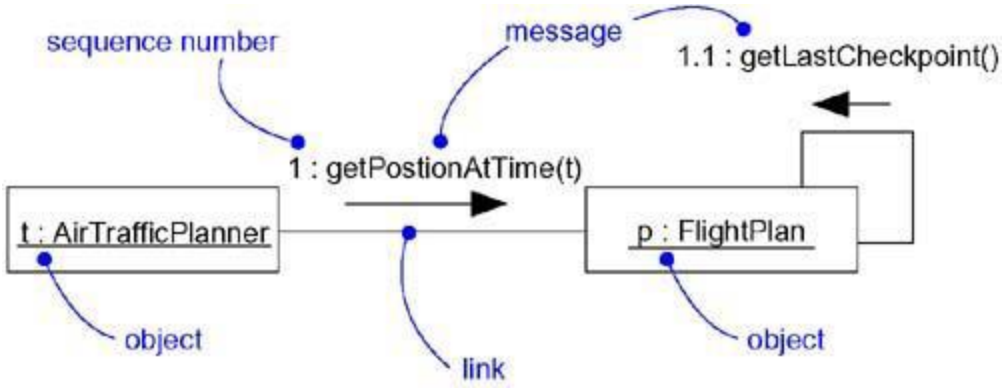


Figure 1 Messages, Links, and Sequencing

A link is a semantic connection(path) among objects through which a message/s can be send. In general, a link is an instance of an association. Figure 2: Links and Associations. The semantics of link can be enhanced by using following prototypes as adornments

- <<association>> – Specifies that the corresponding object is visible by association
- <<self>> – Specifies that the corresponding object is visible because it is the dispatcher of the operation
- <<global>> – Specifies that the corresponding object is visible because it is in an enclosing scope
- <<local>> – Specifies that the corresponding object is visible because it is in a local scope
- <<parameter>> – Specifies that the corresponding object is visible because it is a parameter

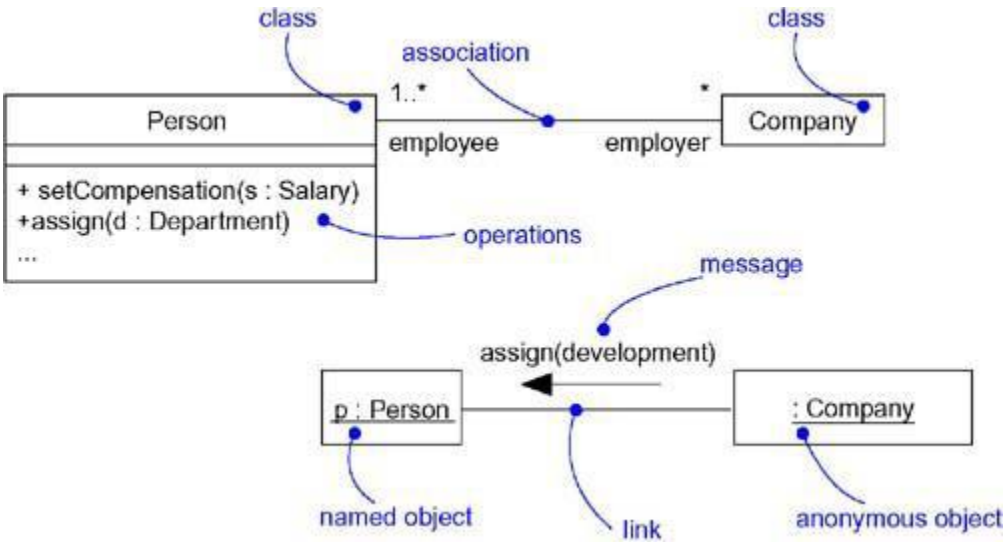


Figure 2: Links and Associations

message – indicates an action to be done. Complex expressions can be written on arbitrary string of message.Different types of messages are,

11) Call -Invokes an operation on an object; an object may send a message to itself, resulting in the local invocation of an operation

12) Return – Returns a value to the caller

13) Send – Sends a signal to an object

14) Create – Creates an object

15) Destroy – Destroys an object; an object may commit suicide by destroying itself

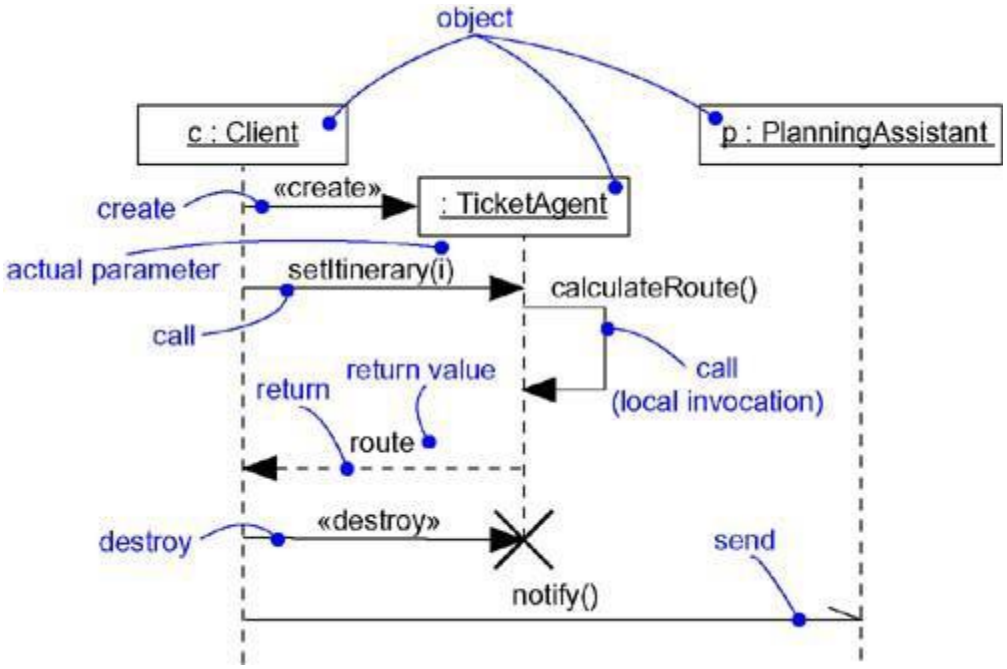


Figure 3 Messages

Sequencing

- a sequence is a stream of messages exchange between objects
- sequence must have a beginning and is rooted in some process or thread
- sequence will continue as long as the process or thread that owns it lives

Flow of control (2 types)
In each flow of control, messages are ordered in sequence by time and are visualized by prefixing the message with a sequence number set apart by a colon separator

A procedural or nested flow of control is rendered by using a filled solid arrowhead, as in Figure 4: shows Procedural Sequence
A flat flow of control is rendered by using a stick arrowhead as in Figure 5: Flat Sequence
Distinguishing one flow of control from another by prefixing a message's sequence number with the name of the process or thread that sits at the root of the sequence

more-complex forms of sequencing, such as iteration, branching, and guarded messages can be modeled in UML.

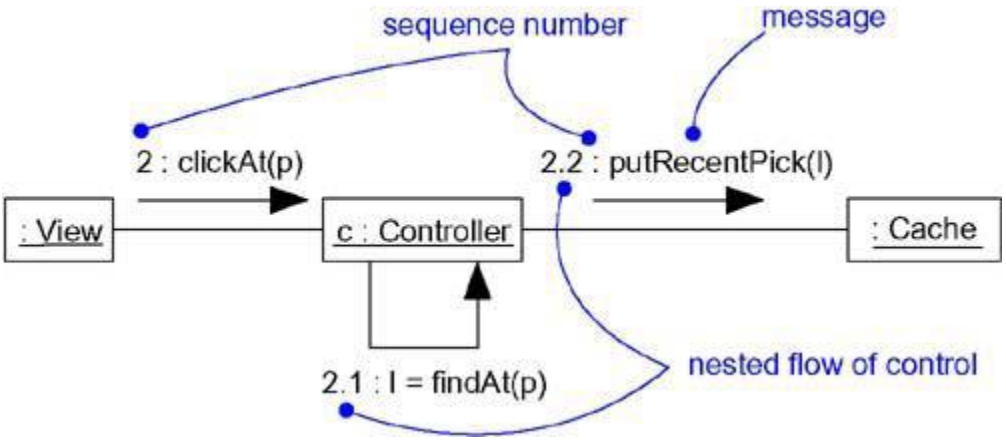


Figure 4: Procedural Sequence

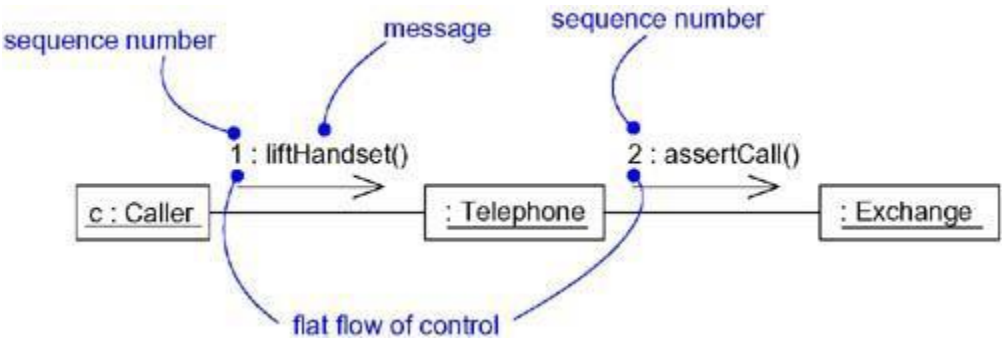


Figure 5: Flat Sequence

Creation, Modification, and Destruction of links
Enabled by adding the following constraints to the element

- new – Specifies that the instance or link is created during execution of the enclosing interaction
- destroyed – Specifies that the instance or link is destroyed prior to completion of execution of the enclosing interaction
- transient – Specifies that the instance or link is created during execution of the enclosing interaction but is destroyed before completion of execution

Representation of interactions
interaction goes together with objects and messages

represented by time ordering of its messages (sequence diagram), and by emphasizing the structural organization of these objects that send and receive messages (collaboration diagram)

Modeling a Flow of Control

To model a flow of control,

- Set the context for the interaction, whether it is the system as a whole, a class, or an individual operation
- Set the stage for the interaction by identifying which objects play a role; set their initial properties, including their attribute values, state, and role
- If your model emphasizes the structural organization of these objects, identify the links that connect them, relevant to the paths of communication that take place in this interaction Specify the nature of the links using the UML's standard stereotypes and constraints, as necessary
- In time order, specify the messages that pass from object to object As necessary, distinguish the different kinds of messages; include parameters and return values to convey the necessary detail of this interaction
- Also to convey the necessary detail of this interaction, adorn each object at every moment in time with its state and role

For example, Figure 6: shows Flow of Control by Time which is a set of objects that interact in the context of a publish and subscribe mechanism

Figure 7: Flow of Control by Organization is shown below for the same context given above.

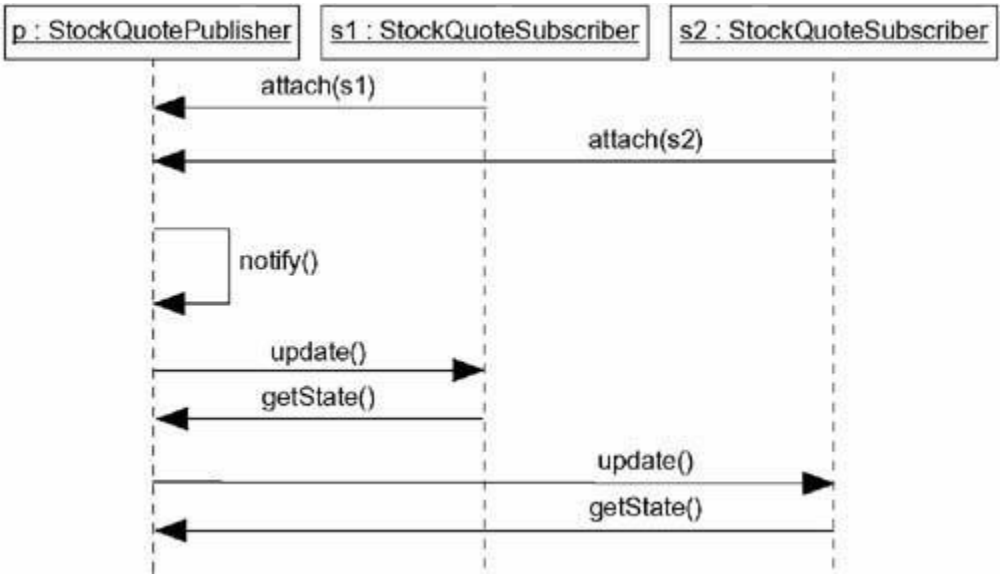


Figure 6: Flow of Control by time

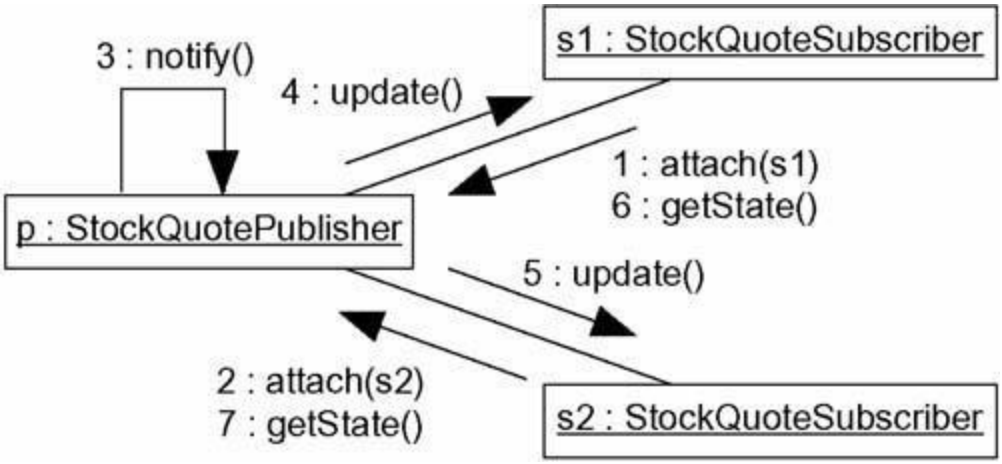


Figure 7: Flow of Control by Organization

Use Cases

- use case is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor
- use case captures the intended behavior of the system (or subsystem, class, or interface) without having to specify how that behavior is implemented
- a use case is rendered(represented) as an ellipse Figure 1: Actors and Use Cases
- Every use case must have a name that distinguishes it from other use cases: simple name and path name Figure 2: Simple and Path Names

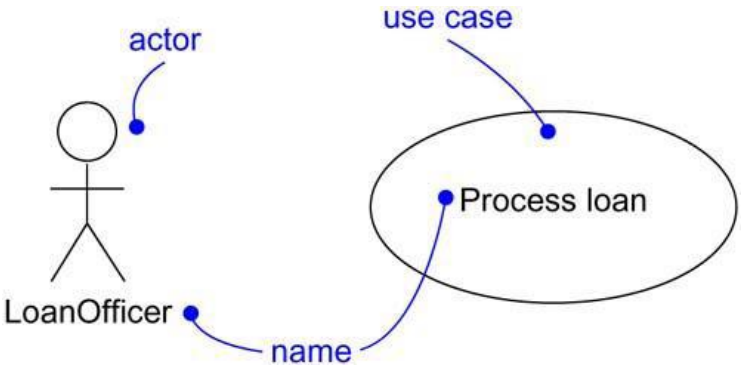


Figure 1: Actors and Use Cases

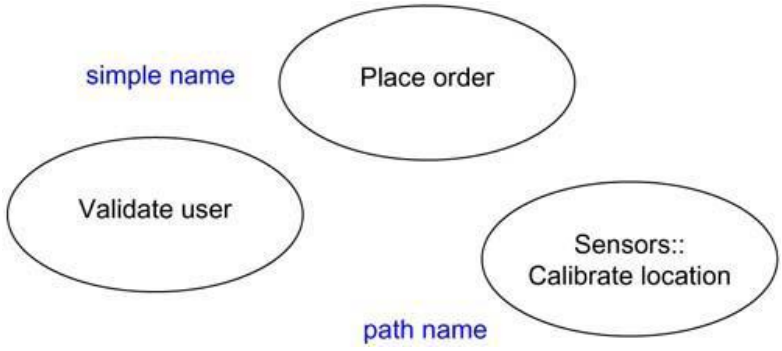


Figure 2: Simple and Path Names

Actors

- actor represents a coherent set of roles that users of use cases play when interacting with these use cases
- an actor represents a role that a human, a hardware device, or even another system plays with a system Figure 3: Actors

Actors may be connected to use cases by association

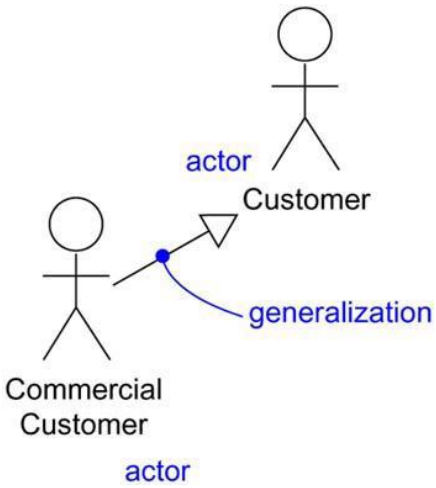


Figure 3: Actors

Use Cases & Flow of Events

flow of events include how and when the use case starts and ends when the use case interacts with the actors and what objects are exchanged, and the basic flow and alternative flows of the behavior.

The behavior of a use case can be specified by describing a flow of events in text.

There can be Main flow of events and one or more Exceptional flow of events.

Use Cases and Scenarios

A scenario is a specific sequence of actions that illustrates behavior

Scenarios are to use cases, as instances are to classes means that scenario is basically one instance of a use case

for each use case, there will be primary scenarios and secondary scenarios.

Use Cases and Collaborations

- Collaborations are used to implement the behaviour of use cases with society of classes and other elements that work together
- It includes static and dynamic structure
- Figure 4: Use Cases and Collaborations

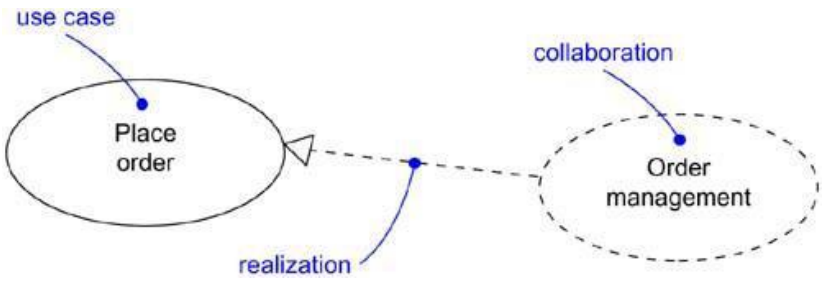


Figure 4: Use Cases and Collaborations

Organizing Use Cases

- organize use cases by grouping them in packages
- organize use cases by specifying generalization, include, and extend relationships among them

Generalization, Include and Extend

An include relationship between use cases means that the base use case explicitly incorporates the behavior of another use case at a location specified in the base.
An include relationship can be rendered as a dependency, stereotyped as include

An extend relationship between use cases means that the base use case implicitly incorporates the behavior of another use case at a location specified indirectly by the extending use case
An extend relationship can be rendered as a dependency, stereotyped as extend. Figure:5

extension points are just labels that may appear in the flow of the base use case

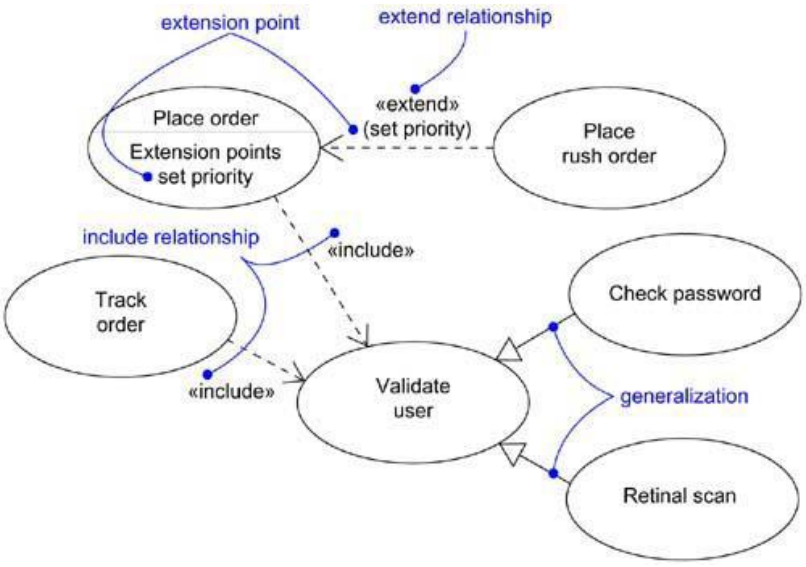


Figure 5: Generalization, Include and Extend

Modeling the Behavior of an Element

To model the behavior of an element,

- Identify the actors that interact with the element Candidate actors include groups that require certain behavior to perform their tasks or that are needed directly or indirectly to perform the element's functions
- Organize actors by identifying general and more specialized roles
- For each actor, consider the primary ways in which that actor interacts with the element Consider also interactions that change the state of the element or its environment or that involve a response to some event
- Consider also the exceptional ways in which each actor interacts with the element
- Organize these behaviors as use cases, applying include and extend

Figure 6: shows Modeling the Behavior of an Element

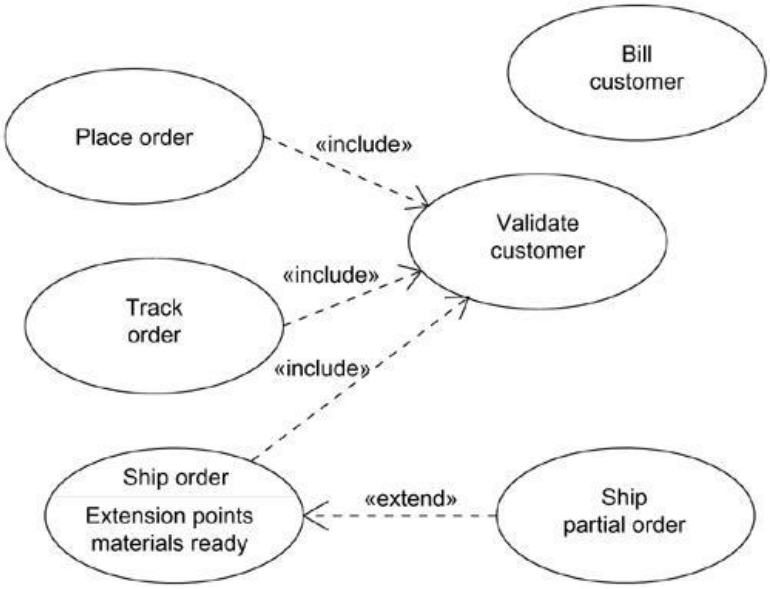
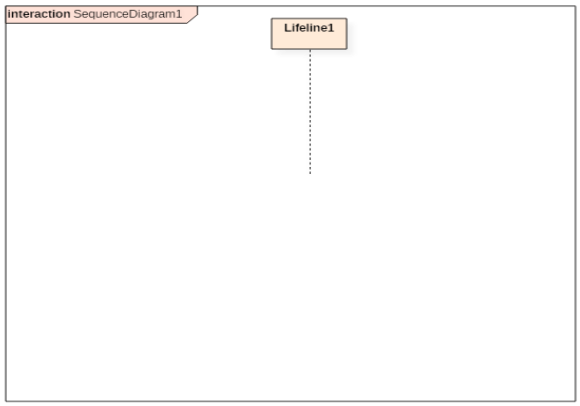


Figure 6: Modeling the Behavior of an Element

Interaction Diagram

INTERACTION DIAGRAM are used in UML to establish communication between objects. It does not manipulate the data associated with the particular communication path. Interaction diagrams mostly focus on message passing and how these messages make up one functionality of a system. Interaction diagrams are designed to display how the objects will realize the particular requirements of a system. The critical component in an interaction diagram is lifeline and messages.

Various UML elements typically own interaction diagrams. The details of interaction can be shown using several notations such as sequence diagram, timing diagram, communication/collaboration diagram. Interaction diagrams capture the dynamic behavior of any system.



Notation of an interaction diagram

Following are the different types of interaction diagrams defined in UML:

- Sequence diagram
- Collaboration diagram
- Timing diagram

The purpose of a sequence diagram in UML is to visualize the sequence of a message flow in the system. The sequence diagram shows the interaction between two lifelines as a time-ordered sequence of events.

The collaboration diagram is also called as a communication diagram. The purpose of a collaboration diagram is to emphasize structural aspects of a system, i.e., how various lifelines in the system connects.

Timing diagrams focus on the instance at which a message is sent from one object to another object.

Purpose of an Interaction Diagram

Interaction diagrams help you to visualize the interactive behavior of a system. Interaction diagrams are used to represent how one or more objects in the system connect and communicate with each other.

Interaction diagrams focus on the dynamic behavior of a system. An interaction diagram provides us the context of an interaction between one or more lifelines in the system.

In UML, the interaction diagrams are used for the following purposes:

- Interaction diagrams are used to observe the dynamic behavior of a system.
- Interaction diagram visualizes the communication and sequence of message passing in the system.
- Interaction diagram represents the structural aspects of various objects in the system.
- Interaction diagram represents the ordered sequence of interactions within a system.
- Interaction diagram provides the means of visualizing the real time data via UML.
- Interaction diagrams can be used to explain the architecture of an object-oriented or a distributed system.

Important terminology

An interaction diagram contains lifelines, messages, operators, state invariants and constraints.

Lifeline

A lifeline represents a single participant in an interaction. It describes how an instance of a specific classifier participates in the interaction.

A lifeline represents a role that an instance of the classifier may play in the interaction. Following are various attributes of a lifeline,

- Name
  - It is used to refer the lifeline within a specific interaction.
  - A name of a lifeline is optional.
- Type
  - It is the name of a classifier of which the lifeline represents an instance.
- Selector
  - It is a Boolean condition which is used to select a particular instance that satisfies the requirement.
  - Selector attribute is also optional.

The notation of lifeline is explained in the notation section.

Messages

A message is a specific type of communication between two lifelines in an interaction. A message involves following activities,

- A call message which is used to call an operation.
- A message to create an instance.
- A message to destroy an instance.
- For sending a signal.

When a lifeline receives a call message, it acts as a request to invoke an operation that has a similar signature as specified in the message. When a lifeline is executing a message, it has a focus of control. As the interaction progresses over time, the focus of control moves between various lifelines. This movement is called a flow of control.

Following are the messages used in an interaction diagram:

Message Name	Meaning
Synchronous message	The sender of a message keeps waiting for the receiver to return control from the message execution.
Asynchronous message	The sender does not wait for a return from the receiver; instead, it continues the execution of a next message.
Return message	The receiver of an earlier message returns the focus of control to the sender.
Object creation	The sender creates an instance of a classifier.
Object destruction	The sender destroys the created instance.
Found message	The sender of the message is outside the scope of interaction.
Lost message	The message never reaches the destination, and it is lost in the interaction.

State invariants and constraints

When an instance or a lifeline receives a message, it can cause it to change the state. A state is a condition or a situation during a lifetime of an object at which it satisfies some constraint, performs some operations, and waits for some event.

In interaction diagram, not all messages cause to change the state of an instance. Some messages do not the values of some attribute. It has no side effects on the state of an object.

Operator

An operator specifies an operation on how the operands are going to be executed. The operators in UML supports operations on data in the form of branching as well as an iteration.

Various operators can be used to ensure the use of iteration and branching in the UML model. The opt and alt operators are used for branching operations. The loop operator is used to ensure the iteration operations in which a condition is executed repeatedly until the satisfying result is produced. Break operator is used inside the loop or iteration operations. It ensures that the loop is terminated whenever a break operator is encountered. If a break condition is not specified, then the loop executes the infinite number of times, which results in crashing the program.

Following are the operators used in an interaction diagram:

Operator	Name	Meaning
Opt	Option	An operand is executed if the condition is true. e.g., If else
Alt	Alternative	The operand, whose condition is true, is executed. e.g., switch
Loop	Loop	It is used to loop an instruction for a specified period.
Break	Break	It breaks the loop if a condition is true or false, and the next instruction is executed.
Ref	Reference	It is used to refer to another interaction.
Par	Parallel	All operands are executed in parallel.

Iteration

In an interaction diagram, we can also show iteration using an iteration expression. An iteration expression consists of an iteration specifier and an optional iteration clause. There is no pre-specified syntax for UML iteration.

In iteration to show that messages are being sent in parallel, parallel iteration specifier is used. A parallel iteration specifier is denoted by *//. Iteration in UML is achieved by using the loop operator.

Branching

In an interaction diagram, we can represent branching by adding guard conditions to the messages. Guard conditions are used to check if a message can be sent forward or not. A message is sent forward only when its guard condition is true. A message can have multiple guard conditions, or multiple messages can have the same guard condition. Branching in UML is achieved with the help of alt and opt, operators.

These are some of the most important terminologies used in UML interaction diagram.

Types of Interaction diagram and Notations

Following are the different types of interaction diagrams defined in UML:

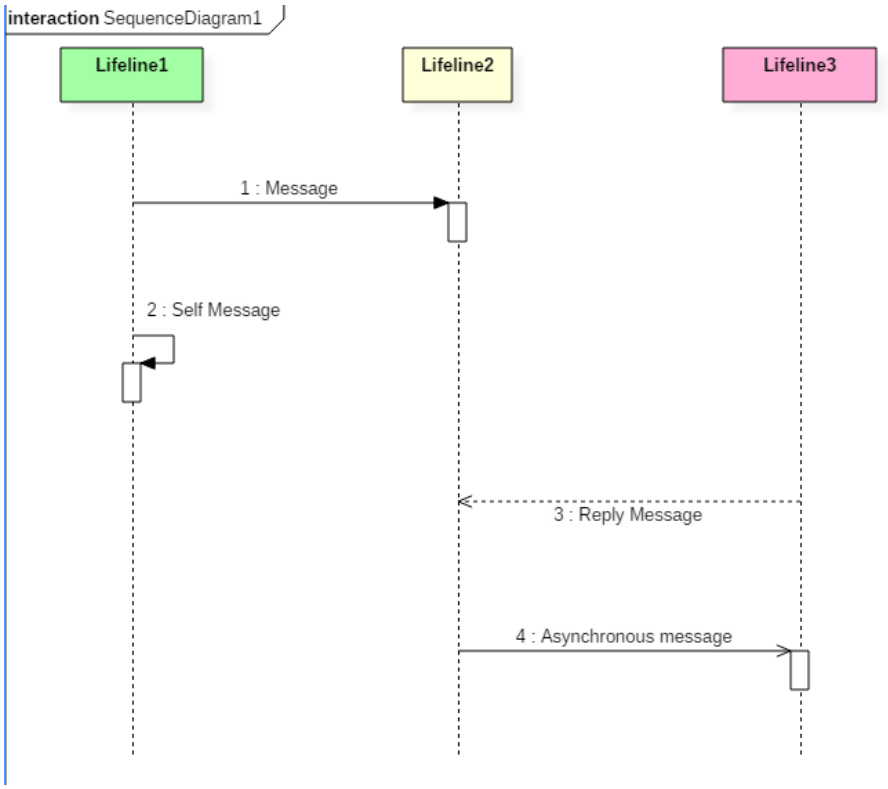
- Sequence diagram
- Collaboration diagram
- Timing diagram

The basic notation of interaction is a rectangle with a pentagon in the upper left corner of a rectangular box.

What is a Sequence Diagram?

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

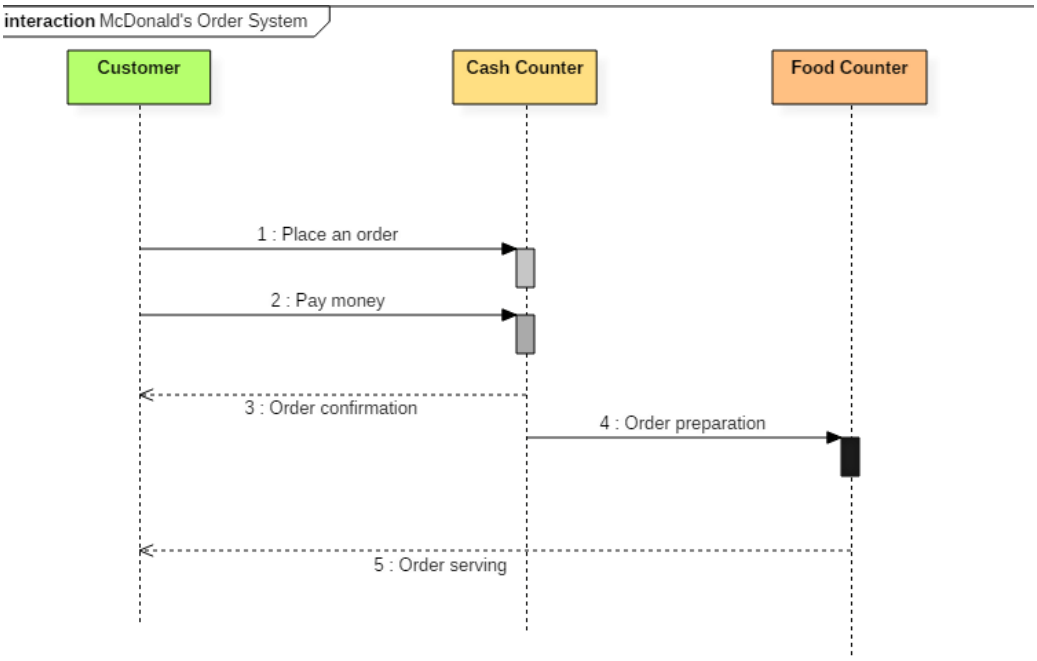
- A sequence diagram shows an implementation of a scenario in the system. Lifelines in the system take part during the execution of a system.
- In a sequence diagram, a lifeline is represented by a vertical bar.
- A message flow between two or more objects is represented using a vertical dotted line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



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

Sequence diagram example

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



Sequence diagram of Mcdonald's ordering system

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

- Place an order.
- Pay money to the cash counter.
- Order Confirmation.
- Order preparation.
- Order serving.

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

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

Benefits of a Sequence Diagram

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

Drawbacks of a sequence diagram

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

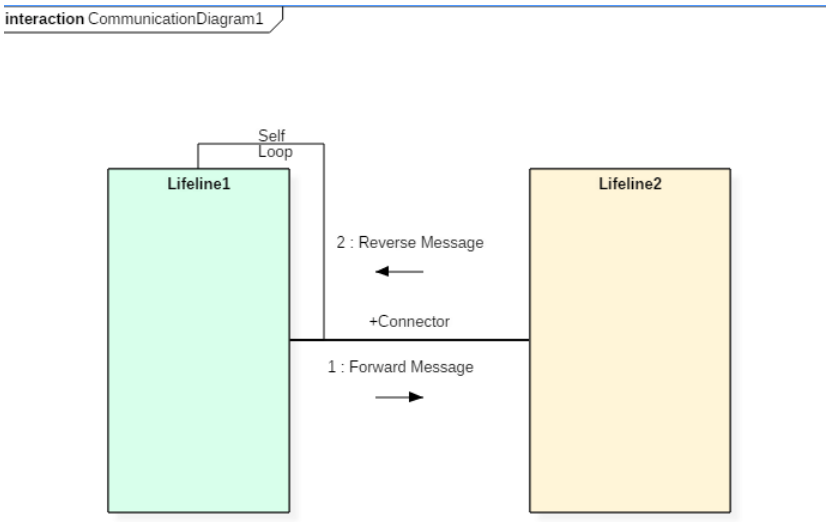
What is the Collaboration diagram?

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

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

Benefits of Collaboration Diagram

- It is also called as a communication diagram.
- It emphasizes the structural aspects of an interaction diagram - how lifeline connects.
- Its syntax is similar to that of sequence diagram except that lifeline don't have tails.
- Messages passed over sequencing is indicated by numbering each message hierarchically.
- Compared to the sequence diagram communication diagram is semantically weak.
- Object diagrams are special case of communication diagram.
- It allows you to focus on the elements rather than focusing on the message flow as described in the sequence diagram.
- Sequence diagrams can be easily converted into a collaboration diagram as collaboration diagrams are not very expressive.
- While modeling collaboration diagrams w.r.t sequence diagrams, some information may be lost.



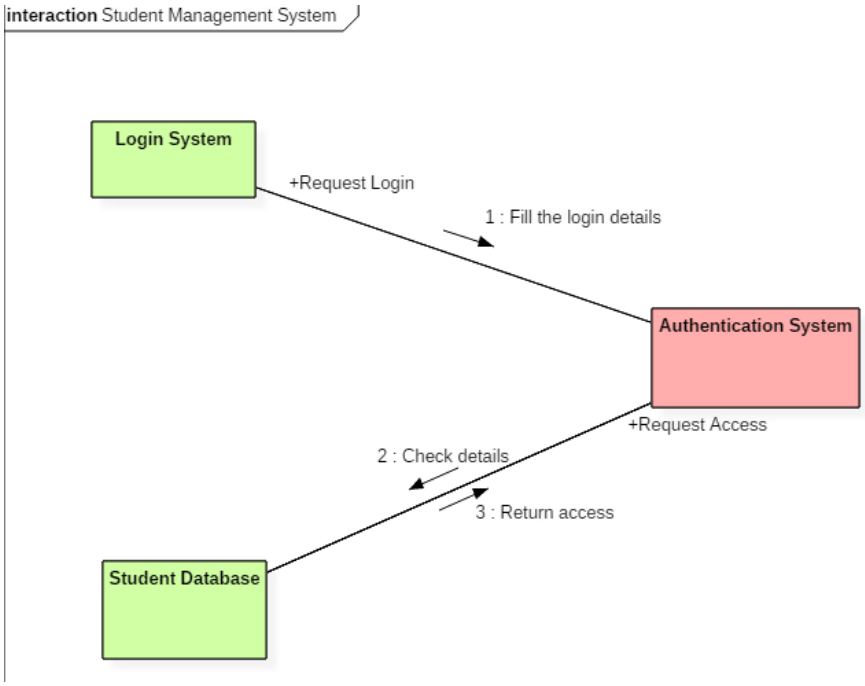
The above collaboration diagram notation contains lifelines along with connectors, self-loops, forward, and reverse messages used in a collaboration diagram.

Drawbacks of a Collaboration Diagram

- Collaboration diagrams can become complex when too many objects are present within the system.
- It is hard to explore each object inside the system.
- Collaboration diagrams are time consuming.
- The object is destroyed after the termination of a program.
- The state of an object changes momentarily, which makes it difficult to keep track of every single change the occurs within an object of a system.

Collaboration diagram Example

Following diagram represents the sequencing over student management system:



Collaboration diagram for student management system

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

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

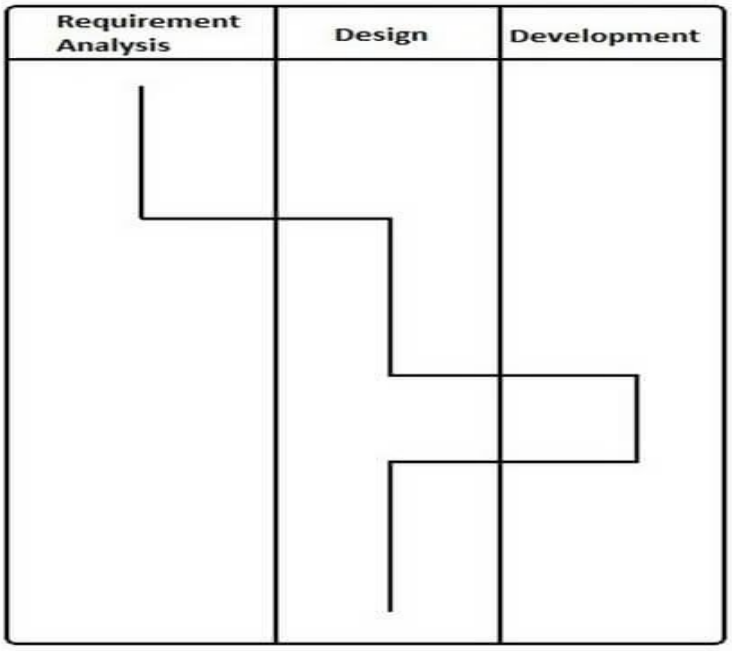
What is Timing diagram?

TIMING DIAGRAM is a waveform or a graph that is used to describe the state of a lifeline at any instance of time. It is used to denote the transformation of an object from one form into another form. Timing diagram does not contain notations as required in the sequence and collaboration diagram. The flow between the software program at various instances of time is represented using a waveform.

- It is a proper representation of interactions that focuses upon the specific timings of messages sent between various objects.
- Timing diagrams are used to explain the detailed time processing of a particular object.
- Timing diagrams are used to explain how an object changes within its lifetime.
- Timing diagrams are mostly used with distributed and embedded systems.
- In UML, timing diagrams are read from left to right according to the name of a lifeline specified at the left edge.
- Timing diagrams are used to represent various changes that occur within a lifeline from time to time.
- Timing diagrams are used to display a graphical representation of various states of a lifeline per unit time.
- UML provides various notations to simplify the transition state between two lifelines per unit time.

Timing diagram Example

The timing diagram given below represents a few phases of a software development life cycle.



In the above diagram, first, the software passes through the requirements phase then the design and later the development phase. The output of the previous phase at that given instance of time is given to the second phase as an input. Thus, the timing diagram can be used to describe SDLC (Software Development Life Cycle) in UML.

Benefits of a Timing Diagram

- Timing diagrams are used to represent the state of an object at a particular instance of time.
- Timing diagram allows reverse as well as forward engineering.
- Timing diagram can be used to keep track of every change inside the system.

Drawbacks of a Timing Diagram

- Timing diagrams are difficult to understand.
- Timing diagrams are difficult to maintain.

How to draw a Interaction diagram?

Interaction diagrams are used to represent the interactive behavior of a system. Interaction diagrams focus on the dynamic behavior of a system. An interaction diagram provides us the context of an interaction between one or more lifelines in the system.

To draw an interaction diagram, you have first to determine the scenario for which you have to draw an interaction diagram. After deciding the situation, identify various lifelines that are going to be involved in the interaction. Categorize all the lifeline elements and explore them to identify possible connections and how the lifelines are related to one another. To draw an interaction diagram, the following things are required:

- The total number of lifelines that are going to be part of an interaction
- is a sequence of message flow within various objects of a system.
- Various operators to ease the functionality of an interaction diagram.
- Various types of messages to display the interaction more clearly and in a precise manner.
- The ordered sequence of messages.
- Organization and a structure of an object.
- Various time constructs of an object.

Use of an interaction diagram

Interaction diagrams consist of a sequence diagram, collaboration diagram, and timing diagrams. Following is the specific purpose of an interaction diagram:

- Sequence diagrams are used to explore any real application or a system.
- Interaction diagrams are used to explore and compare the use of sequence, collaborations, and timing diagrams.
- Interaction diagrams are used to capture the behavior of a system. It displays the dynamic structure of a system.
- Sequence diagrams are used to represent message flow from one object to another object.
- Collaboration diagrams are used to understand the object architecture of a system rather than message flow.
- Interaction diagrams are used to model a system as a time-ordered sequence of events.
- Interaction diagrams are used in reverse as well as forward engineering.
- Interaction diagrams are used to organize the structure of interactive elements.

Use cases Diagrams

Use Case Diagram captures the system's functionality and requirements by using actors and use cases. Use Cases model the services, tasks, function that a system needs to perform. Use

cases represent high-level functionalities and how a user will handle the system. Use-cases are the core concepts of Unified Modelling language modeling.

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

Use-case diagram notations

Following are the common notations used in a use case diagram:

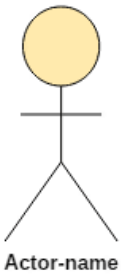
Use-case:

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

UML UseCase Notation

Actor:

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



UML Actor Notation

How to draw a use-case diagram?

To draw a use case diagram in UML first one need to analyse the entire system carefully. You have to find out every single function that is provided by the system. After all the functionalities of a system are found out, then these functionalities are converted into various use cases which will be used in the use case diagram.

A use case is nothing but a core functionality of any working system. After organizing the use cases, we have to enlist the various actors or things that are going to interact with the system. These actors are responsible for invoking the functionality of a system. Actors can be a person or a thing. It can also be a private entity of a system. These actors must be relevant to the functionality or a system they are interacting with.

After the actors and use cases are enlisted, then you have to explore the relationship of a particular actor with the use case or a system. One must identify the total number of ways an actor could interact with the system. A single actor can interact with multiple use cases at the same time, or it can interact with numerous use cases simultaneously.

Following rules must be followed while drawing use-case for any system:

1. The name of an actor or a use case must be meaningful and relevant to the system.
2. Interaction of an actor with the use case must be defined clearly and in an understandable way.
3. Annotations must be used wherever they are required.
4. If a use case or an actor has multiple relationships, then only significant interactions must be displayed.

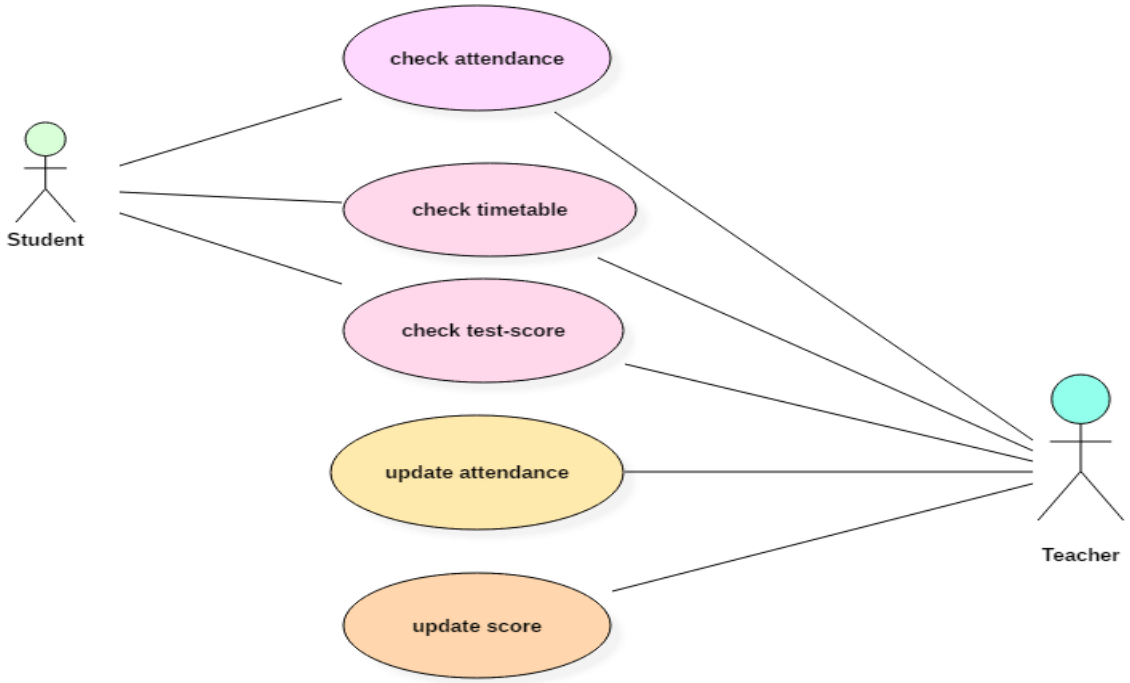
Tips for drawing a use-case diagram

1. A use case diagram should be as simple as possible.
2. A use case diagram should be complete.
3. A use case diagram should represent all interactions with the use case.

4. If there are too many use cases or actors, then only the essential use cases should be represented.
5. A use case diagram should describe at least a single module of a system.
6. If the use case diagram is large, then it should be generalized.

An example of a use-case diagram

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



UML UseCase Diagram

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

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

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

When to use a use-case diagram?

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

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

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

In general use case diagrams are used for:

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

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

Activity Diagram

Activity diagram is defined as a UML diagram that focuses on the execution and flow of the behavior of a system instead of implementation. It is also called **object-oriented flowchart**. Activity diagrams consist of activities that are made up of actions which apply to behavioral modeling technology.

Components of Activity Diagram

Activities

It is a behavior that is divided into one or more actions. Activities are a network of nodes connected by edges. There can be action nodes, control nodes, or object nodes. Action nodes represent some action. Control nodes represent the control flow of an activity. Object nodes are used to describe objects used inside an activity. Edges are used to show a path or a flow of execution. Activities start at an initial node and terminate at a final node.

Activity partition/swimlane

An activity partition or a swimlane is a high-level grouping of a set of related actions. A single partition can refer to many things, such as classes, use cases, components, or interfaces.

If a partition cannot be shown clearly, then the name of a partition is written on top of the name of an activity.

Fork and Join nodes

Using a fork and join nodes, concurrent flows within an activity can be generated. A fork node has one incoming edge and numerous outgoing edges. It is similar to one too many decision parameters. When data arrives at an incoming edge, it is duplicated and split across numerous outgoing edges simultaneously. A single incoming flow is divided into multiple parallel flows.

A join node is opposite of a fork node as It has many incoming edges and a single outgoing edge. It performs logical AND operation on all the incoming edges. This helps you to synchronize the input flow across a single output edge.

Pins

An activity diagram that has a lot of flows gets very complicated and messy.

Pins are used to clearing up the things. It provides a way to manage the execution flow of activity by sorting all the flows and cleaning up messy thins. It is an object node that represents one input to or an output from an action.

Both input and output pins have precisely one edge.

Why use Activity Diagrams?

Activity diagram allows you to create an event as an activity which contains a collection of nodes joined by edges. An activity can be attached to any modeling element to model its behavior. Activity diagrams are used to model,

- Use cases
- Classes
- Interfaces
- Components
- Collaborations

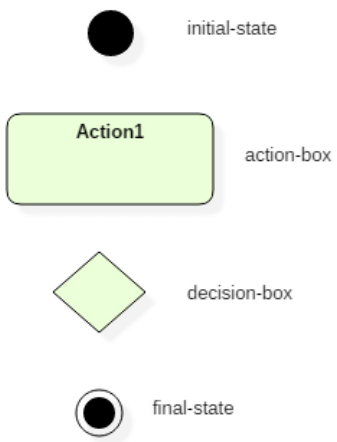
Activity diagrams are used to model processes and workflows. The essence of a useful activity diagram is focused on communicating a specific aspect of a system's dynamic behavior. Activity diagrams capture the dynamic elements of a system.

Activity diagram is similar to a flowchart that visualizes flow from one activity to another activity. Activity diagram is identical to the flowchart, but it is not a flowchart. The flow of activity can be controlled using various control elements in the UML diagram. In simple words, an activity diagram is used to activity diagrams that describe the flow of execution between multiple activities.

Activity Diagram Notations

Activity diagrams symbol can be generated by using the following notations:

- Initial states: The starting stage before an activity takes place is depicted as the initial state
- Final states: The state which the system reaches when a specific process ends is known as a Final State
- State or an activity box:
- Decision box: It is a diamond shape box which represents a decision with alternate paths. It represents the flow of control.



Activity Digram Notation and Symbol

How to draw an activity diagram?

Activity diagram is a flowchart of activities. It represents the workflow between various system activities. Activity diagrams are similar to the flowcharts, but they are not flowcharts. Activity diagram is an advancement of a flowchart that contains some unique capabilities.

Activity diagrams include swimlanes, branching, parallel flow, control nodes, expansion nodes, and object nodes. Activity diagram also supports exception handling.

To draw an activity diagram, one must understand and explore the entire system. All the elements and entities that are going to be used inside the diagram must be known by the user. The central concept which is nothing but an activity must be clear to the user. After analyzing all activities, these activities should be explored to find various constraints that are applied to activities. If there is such a constraint, then it should be noted before developing an activity diagram.

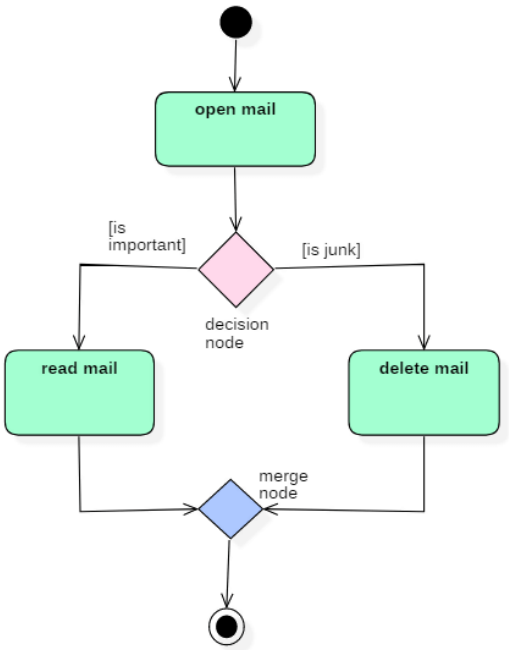
All the activities, conditions, and associations must be known. Once all the necessary things are gathered, then an abstract or a prototype is generated, which is later converted into the actual diagram.

Following rules must be followed while developing an activity diagram,

1. All activities in the system should be named.
2. Activity names should be meaningful.
3. Constraints must be identified.
4. Activity associations must be known.

Example of Activity Diagram

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



Activity diagram

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

When Use Activity Diagram

Activity diagram is used to model business processes and workflows. These diagrams are used in software modeling as well as business modeling.

Most commonly activity diagrams are used to,

1. Model the workflow in a graphical way, which is easily understandable.
2. Model the execution flow between various entities of a system.
3. Model the detailed information about any function or an algorithm which is used inside the system.
4. Model business processes and their workflows.
5. Capture the dynamic behavior of a system.
6. Generate high-level flowcharts to represent the workflow of any application.
7. Model high-level view of an object-oriented or a distributed system.

MODULE-4

Advanced Behaviour Modelling

Events and Signals

Events

- An event is the specification of a significant occurrence that has a location in time and space.
- Any thing that happens is modeled as an event in UML.
- In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition
- four kinds of events – signals, calls, the passing of time, and a change in state.
- Figure 1: Events
- Events may be external or internal and asynchronous or synchronous.

asynchronous events are events that can happen at arbitrary times eg:- signal, the passing of time, and a change of state. *synchronous events*, represents the invocation of an operation eg:- Calls

External events are those that pass between the system and its actors. *Internal events* are those that pass among the objects that live inside the system.

A signal is an event that represents the specification of an asynchronous stimulus communicated between instances.

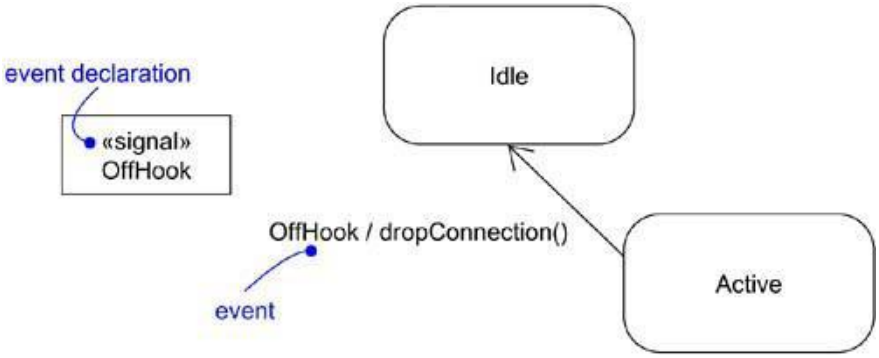


Figure 1: Events

kinds of events

Signal Event

- a signal event represents a named object that is dispatched (thrown) asynchronously by one object and then received (caught) by another. Exceptions are an example of internal signal
- a signal event is an asynchronous event
- signal events may have instances, generalization relationships, attributes and operations. Attributes of a signal serve as its parameters
- A signal event may be sent as the action of a state transition in a state machine or the sending of a message in an interaction
- signals are modeled as stereotyped classes and the relationship between an operation and the events by using a dependency relationship, stereotyped as send
- Figure 2 shows Signal Events

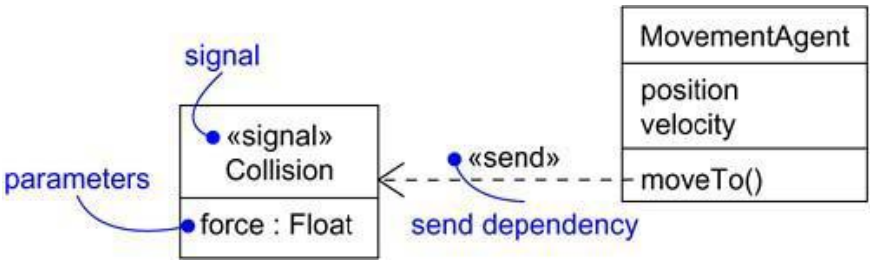


Figure 2: Signals

Call Event

- a call event represents the dispatch of an operation
- a call event is a synchronous event
- Figure 3 shows Call Events

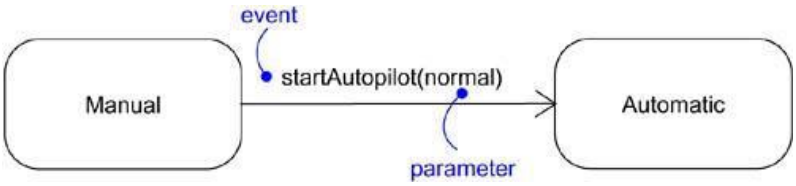


Figure 3: Call Events

Time and Change Events

- A *time event* is an event that represents the passage of time.
- modeled by using the keyword 'after' followed by some expression that evaluates to a period of time which can be simple or complex.
- A *change event* is an event that represents a change in state or the satisfaction of some condition
- modeled by using the keyword 'when' followed by some Boolean expression
- Figure 4

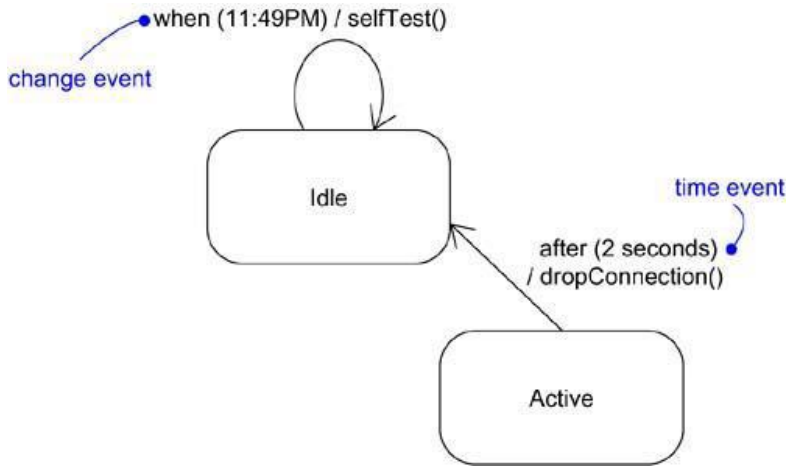


Figure 4: Time and Change Events

Sending and Receiving Events

For synchronous events (Sending or Receiving) like call event, the sender and the receiver are in a rendezvous(the sender dispatches the signal and wait for a response from the receiver) for the duration of the operation. when an object calls an operation, the sender dispatches the operation and then waits for the receiver.

For asynchronous events (Sending or Receiving) like signal event, the sender and receiver do not rendezvous ie,the sender dispatches the signal but does not wait for a response from the receiver. When an object sends a signal, the sender dispatches the signal and then continues along its flow of control, not waiting for any return from the receiver.

call events can be modelled as operations on the class of the object.

named signals can be modelled by naming them in an extra compartment of the class as in Figure 5: Signals and Active Classes

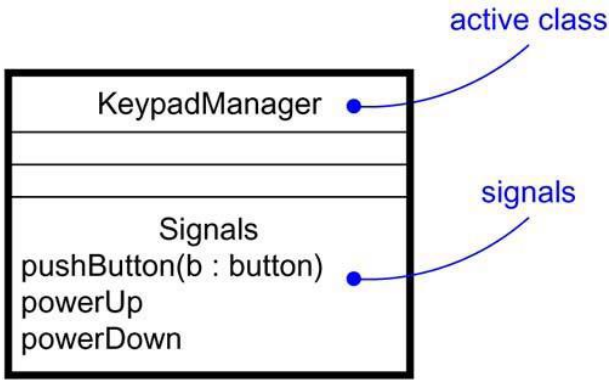


Figure 5: Signals and Active Classes

Modeling family of signals

To model a family of signals,

- Consider all the different kinds of signals to which a given set of active objects may respond.
- Look for the common kinds of signals and place them in a generalization/specialization hierarchy using inheritance. Elevate more general ones and lower more specialized ones.
- Look for the opportunity for polymorphism in the state machines of these active objects. Where you find polymorphism, adjust the hierarchy as necessary by introducing intermediate abstract signals.

Figure 6 models a family of signals that may be handled by an autonomous robot.

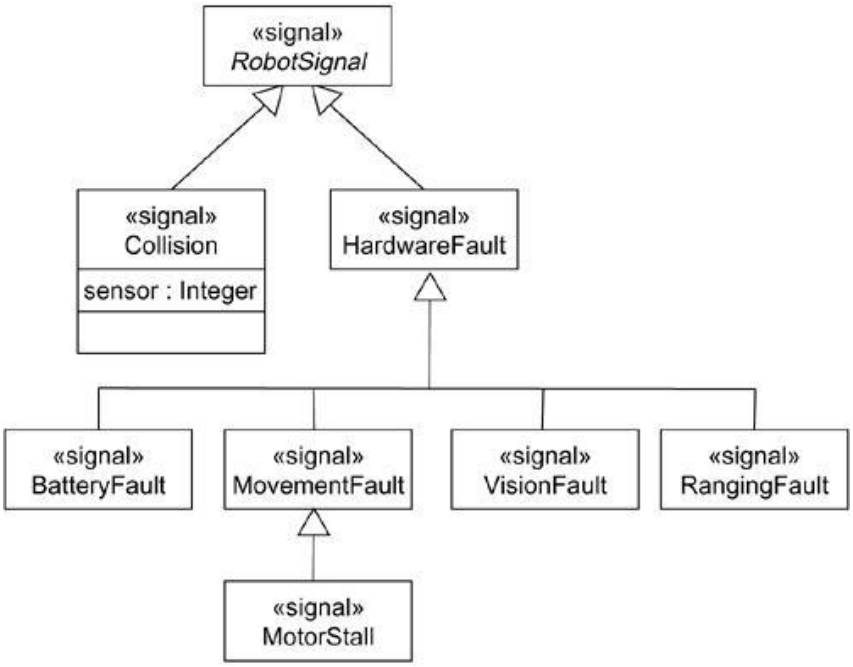


Figure 6: Modeling Families of Signals

Modeling Exceptions

To model exceptions,

- For each class and interface, and for each operation of such elements, consider the exceptional conditions that may be raised.
- Arrange these exceptions in a hierarchy. Elevate general ones, lower specialized ones, and introduce intermediate exceptions, as necessary.
- For each operation, specify the exceptions that it may raise. You can do so explicitly (by showing send dependencies from an operation to its exceptions) or you can put this in the operation's specification.

Figure 7 models a hierarchy of exceptions

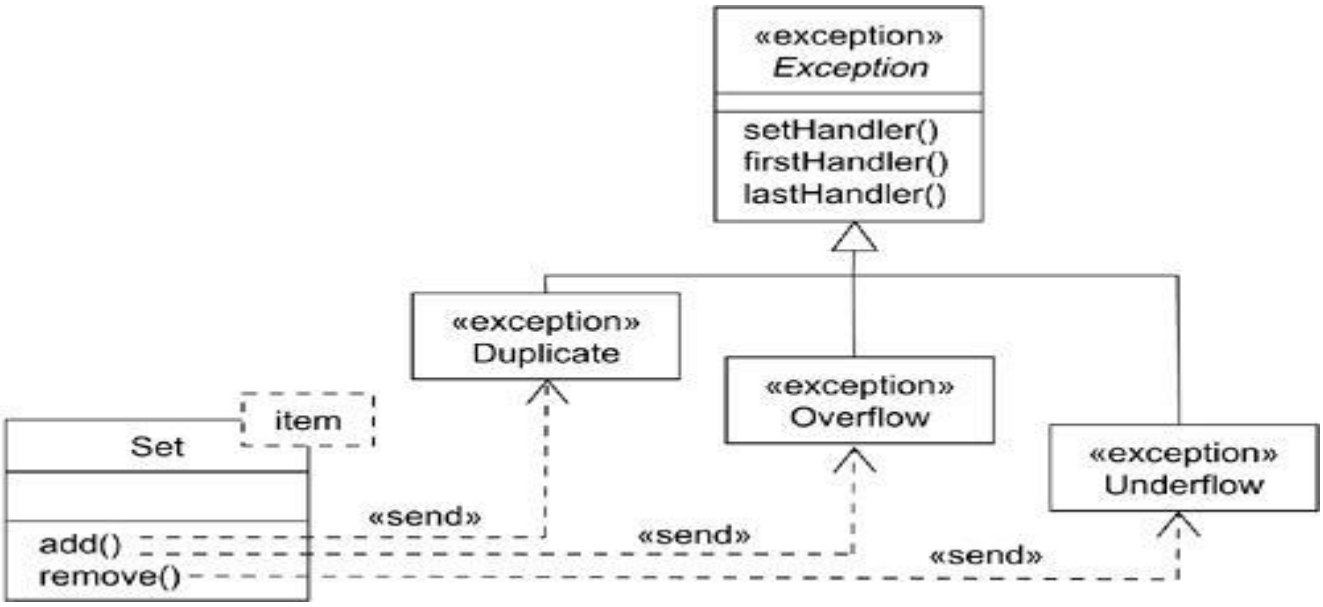


Figure 7: Modeling Exceptions

State Machine

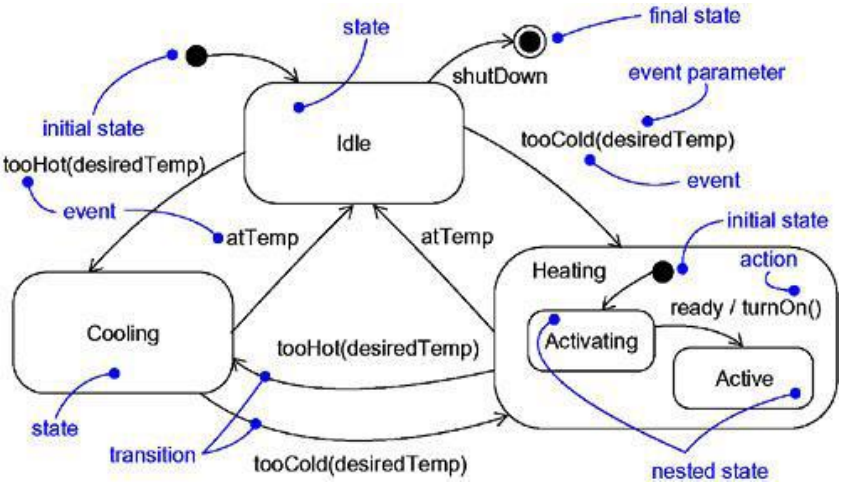


Figure 1: State Machines

- A state machine is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events.
- Graphically, a state is rendered as a rectangle with rounded corners. A transition is rendered as a solid directed line.
- Figure 1 shows State Machines
- state machine are used to specif the behavior of objects that must respond to asynchronous stimulus or whose current behavior depends on their past.

- state machines are used to model the behavior of entire systems, especially reactive systems, which must respond to signals from actors outside the system.

States

- A state is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.
- An object remains in a state for a finite amount of time. For example, a Heater in a home might be in any of four states: Idle, Activating, Active, and ShuttingDown.
- a state name must be unique within its enclosing state
- A state has five parts: *Name, Entry/exit actions, Internal transitions* – Transitions that are handled without causing a change in state, *Substates* – nested structure of a state, involving disjoint (sequentially active) or concurrent (concurrently active) substates, *Deferred events* – A list of events that are not handled in that state but, rather, are postponed and queued for handling by the object in another state
- Figure 2 shows States
- initial state indicates the default starting place for the state machine or substate and is represented as a filled black circle
- final state indicates that the execution of the state machine or the enclosing state has been completed and is represented as a filled black circle surrounded by an unfilled circle
- Initial and final states are pseudo-states

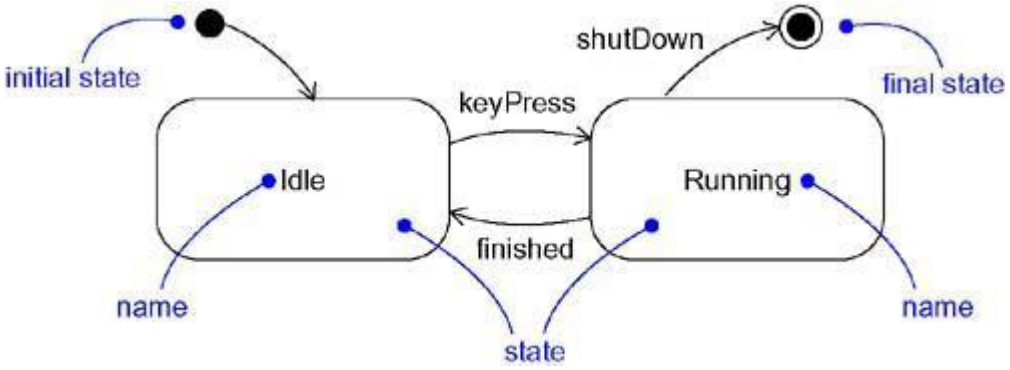


Figure 2: States

Transitions

- A transition is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied.
- Transition fires means change of state occurs. Until transition fires, the object is in the source state; after it fires, it is said to be in the target state.
- A transition has five parts: *Source state* – The state affected by the transition, *Event trigger* – a stimulus that can trigger a source state to fire on satisfying guard condition, *Guard condition* – Boolean expression that is evaluated when the transition is

triggered by the reception of the event trigger, *Action* – An executable atomic computation that may directly act on the object that owns the state machine, and indirectly on other objects that are visible to the object, *Target state* – The state that is active after the completion of the transition.

- Figure 3 shows transitions
- A transition may have multiple sources as well as multiple targets
- A *self-transition*is a transition whose source and target states are the same

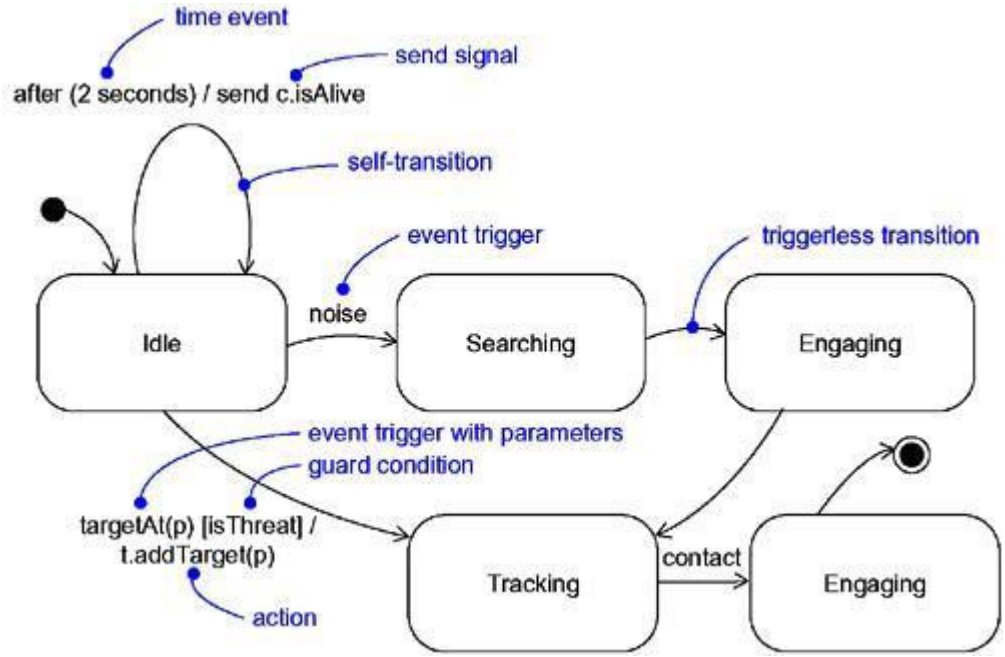


Figure 3:Transitions

Event Trigger

- An event in the context of state machines is an occurrence of a stimulus that can trigger a state transition.
- events may include signals, calls, the passing of time,or a change in state.
- An event – signal or a call – may have parameters whose values are available to the transition, including expressions for the guard condition and action.
- An event trigger may be polymorphic

Guard condition

- a guard condition is rendered as a Boolean expression enclosed in square brackets and placed after the trigger event
- A guard condition is evaluated only after the trigger event for its transition occurs
- A guard condition is evaluated just once for each transition at the time the event occurs, but it may be evaluated again if the transition is retriggered

Action

- An action is an executable atomic computation i.e, it cannot be interrupted by an event and runs to completion.
- Actions may include operation calls, the creation or destruction of another object, or the sending of a signal to an object

An activity may be interrupted by other events.

Advanced States and Transitions

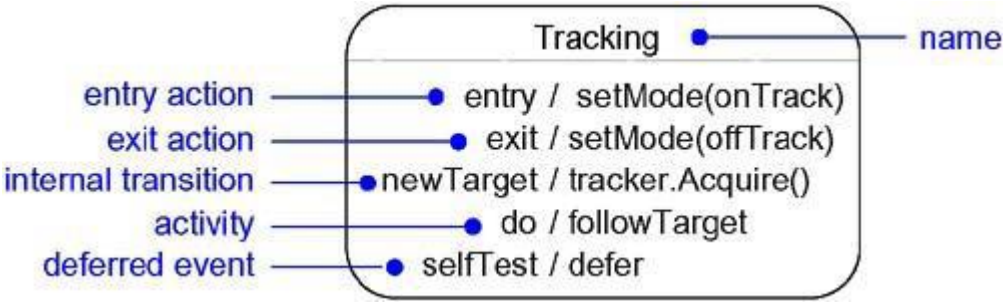


Figure 4: Advanced States and Transitions

Entry and Exit Actions

- Entry Actions are those actions that are to be done upon entry of a state and are shown by the keyword event 'entry' with an appropriate action
- Exit Actions are those actions that are to be done upon exit from a state marked by the keyword event 'exit', together with an appropriate action

Internal Transitions

- Internal Transitions are events that should be handled internally without leaving the state.
- Internal transitions may have events with parameters and guard conditions.

Activities

Activities make use of object's idle time when inside a state. 'do' transition is used to specify the work that's to be done inside a state after the entry action is dispatched.

Deferred Events

A deferred event is a list of events whose occurrence in the state is postponed until a state in which the listed events are not deferred becomes active, at which time they occur and may

trigger transitions as if they had just occurred. A deferred event is specified by listing the event with the special action 'defer'.

Substate

- A substate is a state that's nested inside another one.
- A state that has substates is called a composite state.
- A composite state may contain either *concurrent (orthogonal) or sequential (disjoint) substates*.
- Substates may be nested to any level

Sequential Substates

- Sequential Substates are those substates in which an event common to the composite states can eaisly be excercised by each states inside it at any time
- sequential substates partition the state space of the composite state into disjoint states
- Figure 5: shows Sequential Substates
- A nested sequential state machine may have at most one initial state and one final state

History States

- A history state allows composite state that contains sequential substates to remember the last substate that was active in it prior to the transition from the composite state.
- a shallow history state is represented as a small circle containing the symbol H
- Figure 6: shows History State
- The first time entry to a composite state doesn't have any history and the process for collecting history is as shown in the figure: 6
- the symbol H designates a *shallow history*, which remembers only the history of the immediate nested state machine.
- the symbol H* designates *deep history*, which remembers down to the innermost nested state at any depth.
- When only one level of nesting, shallow and deep history states are semantically equivalent.
-

Concurrent Substates

- concurrent substates specify two or more state machines that execute in parallel in the context of the enclosing object
- Figure 7: shows Concurrent Substates
- Execution of these concurrent substates continues in parallel. These substates waits for each other to finish to joins back into one flow
- A nested concurrent state machine does not have an initial, final, or history state

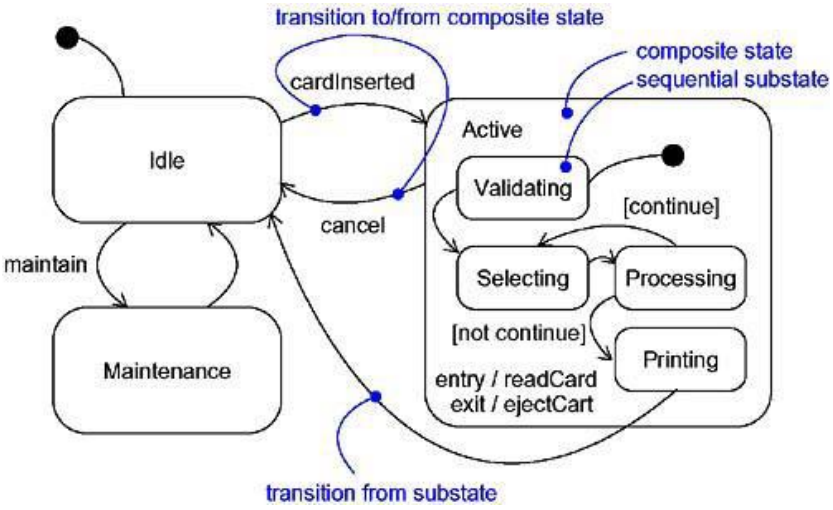


Figure 5: Sequential Substates

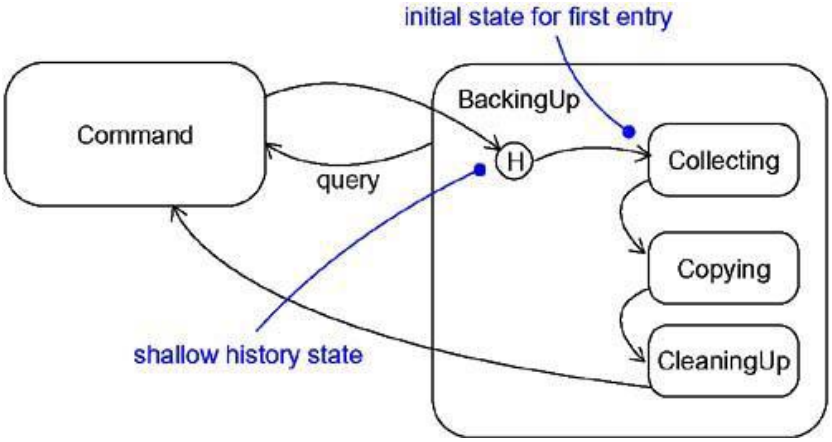


Figure 6: History State

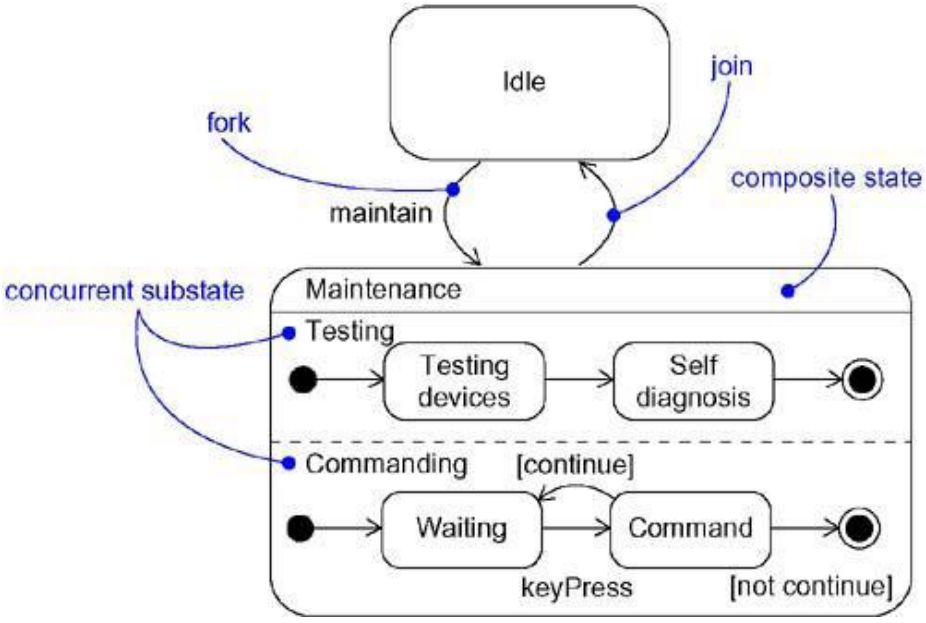


Figure 7: Concurrent Substates

Modeling the Lifetime of an Object

To model the lifetime of an object,

- · Set the context for the state machine, whether it is a class, a use case, or the system as a whole.
- *If the context is a class or a use case*, collect the neighboring classes, including any parents of the class and any classes reachable by associations or dependences. These neighbors are candidate targets for actions and are candidates for including in guard conditions.
- *If the context is the system as a whole*, narrow your focus to one behavior of the system. Theoretically, every object in the system may be a participant in a model of the system's lifetime, and except for the most trivial systems, a complete model would be intractable.
- · Establish the initial and final states for the object. To guide the rest of your model, possibly state the pre- and postconditions of the initial and final states, respectively.
- · Decide on the events to which this object may respond. If already specified, you'll find these in the object's interfaces; if not already specified, you'll have to consider which objects may interact with the object in your context, and then which events they may possibly dispatch.
- · Starting from the initial state to the final state, lay out the top-level states the object may be in. Connect these states with transitions triggered by the appropriate events. Continue by adding actions to these transitions.
- · Identify any entry or exit actions (especially if you find that the idiom they cover is used in the state machine).
- · Expand these states as necessary by using substates.
- · Check that all events mentioned in the state machine match events expected by the interface of the object. Similarly, check that all events expected by the interface of the object are handled by the state machine. Finally, look to places where you explicitly want to ignore events.
- · Check that all actions mentioned in the state machine are sustained by the relationships, methods, and operations of the enclosing object.
- · Trace through the state machine, either manually or by using tools, to check it against expected sequences of events and their responses. Be especially diligent in looking for unreachable states and states in which the machine may get stuck.
- · After rearranging your state machine, check it against expected sequences again to ensure that you have not changed the object's semantics.

For example, Figure 8 shows the state machine for the controller in a home security system, which is responsible for monitoring various sensors around the perimeter of the house

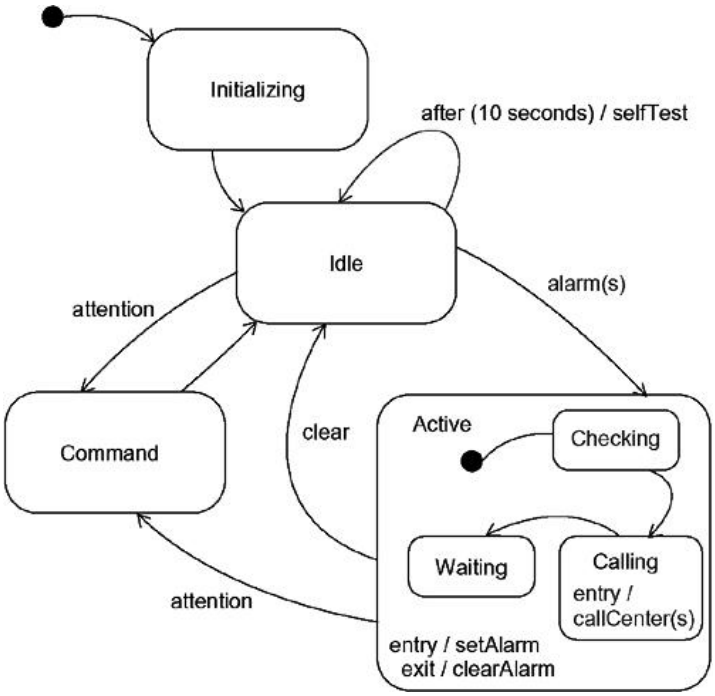


Figure 8: Modeling the Lifetime of An Object

State Diagrams

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

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

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

There are a total of two types of state machine diagrams:

1. Behavioral state machine

- It captures the behavior of an entity present in the system.
- It is used to represent the specific implementation of an element.
- The behavior of a system can be modelled using behavioral state machine diagrams.

2. Protocol state machine

- These diagrams are used to capture the behavior of a protocol.
- It represents how the state of protocol changes concerning the event. It also represents corresponding changes in the system.
- They do not represent the specific implementation of an element.

Statechart diagram is used to capture the dynamic aspect of a system. State machine diagrams are used to represent the behavior of an application. An object goes through various states during its lifespan. The lifespan of an object remains until the program is terminated. The object goes from multiple states depending upon the event that occurs within the object. Each state represents some unique information about the object.

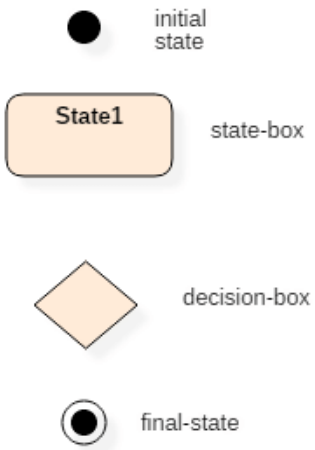
Statechart diagrams are used to design interactive systems that respond to either internal or external event. Statechart diagram visualizes the flow of execution from one state to another state of an object.

It represents the state of an object from the creation of an object until the object is destroyed or terminated.

The primary purpose of a statechart diagram is to model interactive systems and define each and every state of an object. Statechart diagrams are designed to capture the dynamic behavior of an application system. These diagrams are used to represent various states of a system and entities within the system.

Notation and Symbol for State Machine

Following are the various notations that are used throughout the state chart diagram. All these notations, when combined, make up a single diagram.



UML state diagram notations

Initial state

The initial state symbol is used to indicate the beginning of a state machine diagram.

Final state

This symbol is used to indicate the end of a state machine diagram.

Decision box

It contains a condition. Depending upon the result of an evaluated guard condition, a new path is taken for program execution.

Transition

A transition is a change in one state into another state which is occurred because of some event. A transition causes a change in the state of an object.

State box

It is a specific moment in the lifespan of an object. It is defined using some condition or a statement within the classifier body. It is used to represent any static as well as dynamic situations.

It is denoted using a rectangle with round corners. The name of a state is written inside the rounded rectangle.

The name of a state can also be placed outside the rectangle. This can be done in case of composite or submachine states. One can either place the name of a state within the rectangle or outside the rectangle in a tabular box. One cannot perform both at the same time.

A state can be either active or inactive. When a state is in the working mode, it is active, as soon as it stops executing and transits into another state, the previous state becomes inactive, and the current state becomes active.

Types of State

Unified Modeling Language defines three types of states:

- Simple state
    - They do not have any substrate.
- Composite state

    - These types of states can have one or more than one substrate.
    - A composite state with two or more substates is called an orthogonal state.
- Submachine state
    - These states are semantically equal to the composite states.
    - Unlike the composite state, we can reuse the submachine states.

How to draw a Statechart diagram?

Statechart diagrams are used to describe the various state that an object passes through. A transition between one state into another state occurs because of some triggered event. To draw a state diagram, one must identify all the possible states of any particular entity.

The purpose of these UML diagrams is to represent states of a system. States plays a vital role in state transition diagrams. All the essential object, states, and the events that cause changes within the states must be analyzed first before implementing the diagram.

Following rules must be considered while drawing a state chart diagram:

1. The name of a state transition must be unique.
2. The name of a state must be easily understandable and describe the behavior of a state.
3. If there are multiple objects, then only essential objects should be implemented.
4. Proper names for each transition and an event must be given.

When to use State Diagrams?

State diagrams are used to implement real-life working models and object-oriented systems in depth. These diagrams are used to compare the dynamic and static nature of a system by capturing the dynamic behavior of a system.

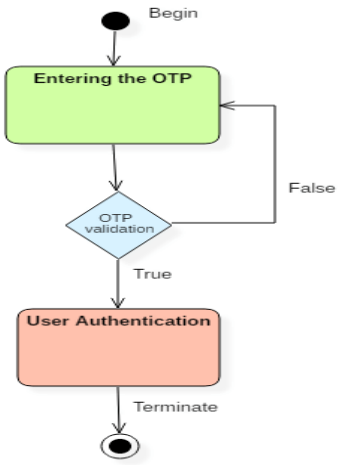
Statechart diagrams are used to capture the changes in various entities of the system from start to end. They are used to analyze how an event can trigger change within multiple states of a system.

State char diagrams are used,

1. To model objects of a system.
2. To model and implement interactive systems.
3. To display events that trigger changes within the states.

Example of State Diagram

Following state chart diagram represents the user authentication process.



UML state diagram

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

State diagram vs. Flowchart

State diagram	**FlowChart**
It represents various states of a system.	The Flowchart illustrates the program execution flow.
The state machine has a WAIT concept, i.e., wait for an action or an event.	The Flowchart does not deal with waiting for a concept.
State machines are used for a live running system.	Flowchart visualizes branching sequences of a system.
The state machine is a modeling diagram.	A flowchart is a sequence flow or a DFD diagram.
The state machine can explore various states of a system.	Flowchart deal with paths and control flow

Architectural Modelling

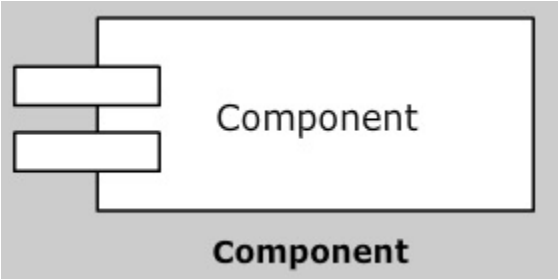
Architectural model represents the overall framework of the system. It contains both structural and behavioral elements of the system. Architectural model can be defined as the blueprint of the entire system. Package diagram comes under architectural modeling.

Components

A component is a physical and replaceable part of a system that conforms to and provide the realization of a set of interfaces.

It is used to model physical things like executable's, libraries, tables, files and documents .

Graphically represented as a rectangle with tabs

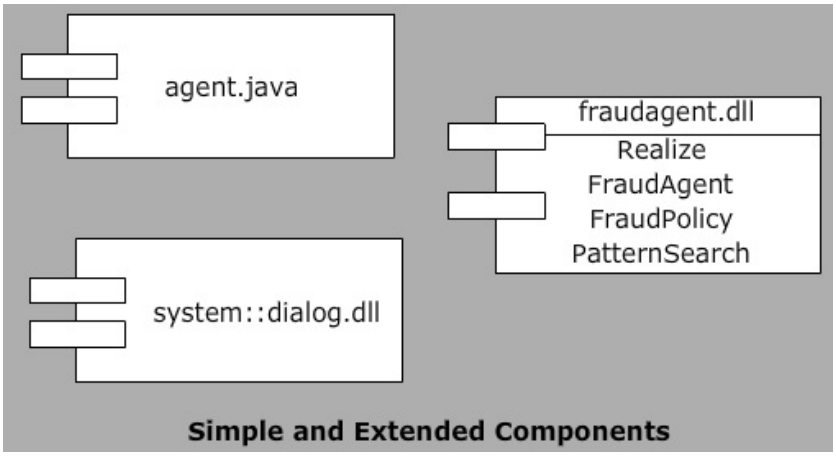


- Names
- Components and Classes
- Components and Interfaces
- Binary Replace ability
- Kinds of Components
- Common Modeling Techniques

Names:

Name of every component should be unique, in order to distinguish it from other components

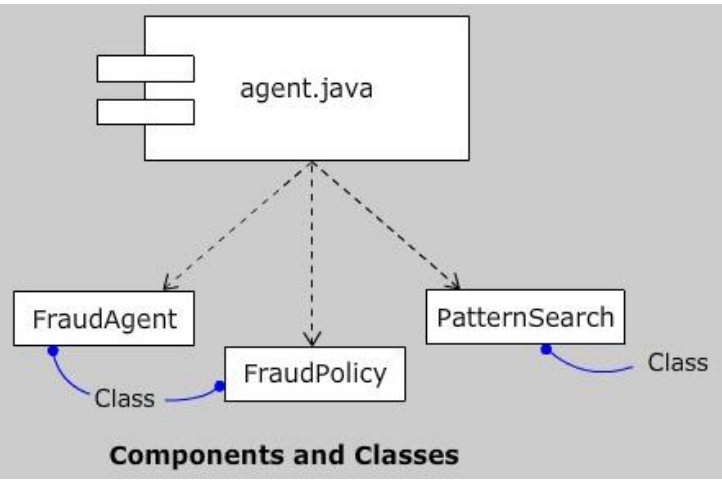
It has simple and path names



Components and Classes:

Classes represent logical abstractions where as components represent physical things representing physical packaging

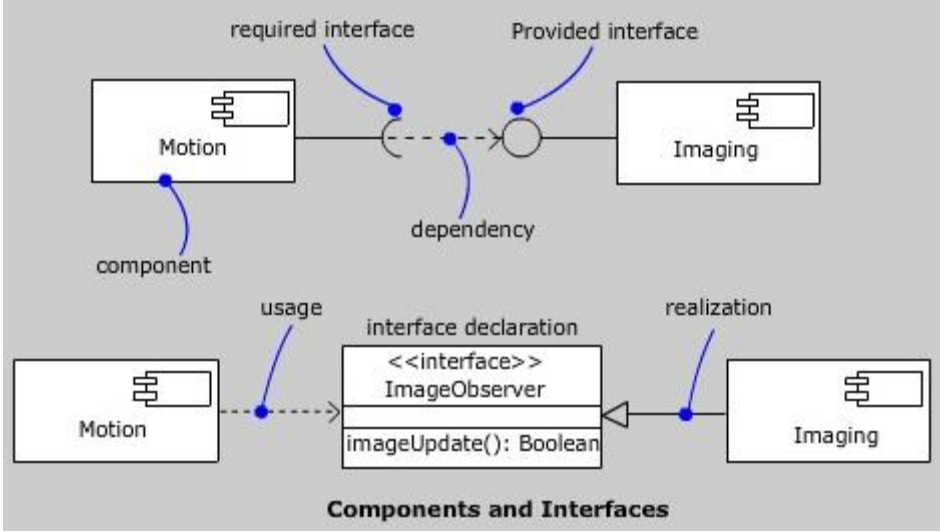
Classes have attributes and operations where as components have only operations which are reachable through their interfaces



Components and Interfaces:

The relationship between component and interfaces can be shown by connecting the component that realizes interface either by an elided or full realization relationship

In both ways, the component to the interface is connected by dependency relationship



Binary Replace ability:

➤A component is

- Physical, it lives in the world of bits not in concepts
- Replaceable, it is possible to replace a component with another that conforms to the same interfaces
- Part of a system, it rarely stands alone

➤A component conforms to and provides the realization of a set of interfaces

Kinds of Components

Three kinds of components may be distinguished.

First, there are deployment components. These are the components necessary and sufficient to form an executable system, such as dynamic libraries (DLLs) and executables (EXEs).

Second, there are work product components. These components are generally the residue of the development process, consisting of things such as the source code files and data files from which deployment components are created.

Third are execution components. These components are created as a consequence of an executing system, such as COM+ object, which is instantiated from a DLL.

Standard Elements

All the UML's extensibility mechanisms apply to components. Most often, we'll use tagged values to extend the component properties and stereotypes to specify new kind of components.

The UML defines five standard stereotypes that apply to components:

Common Modeling Techniques

Modeling Executables and Libraries

To model executables and libraries:

- Identify the partitioning of the physical system. Consider the impact of the technical, configuration management and reuse issues.
- Model any executables and libraries as components, using the appropriate standard elements.
- Model the significant interfaces that some components use and others realize.
- As necessary to communicate your intent, model the relationships among these executables, libraries and interfaces.

Modeling Tables, Files and Documents

To model tables, files and documents:

- Identify the components that are part of the physical implementation of your system.
- Model these things as components.
- As necessary to communicate your intent, model the relationships among these components and other executables, libraries and interfaces in the system.

Modeling an API

To model an API:

- Identify the programmatic seems in the system and model each seem as an interface.
- Expose only those properties of the interface that are important to visualize the given context. Otherwise, hide these properties, keeping them in the interface's specification for reference, as necessary.
- Model the realization of each API only as it is important to show the configuration of a specific implementation.

Modeling Source Code

To model source code:

- Depending on the constraints imposed by your development tools, model the files used to store the details of all your logical elements, along with their compilation dependencies.
- Use tagged values if you want to use configuration management and version control tools.
- As far as possible, let your development tools manage the relationships among these files, and use the UML only to visualize and document these relationships.

Collaboration

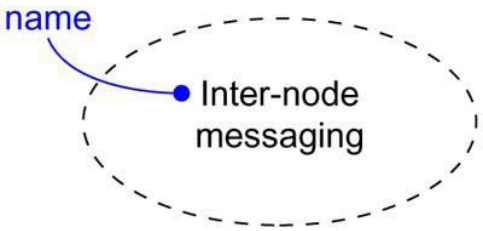


Figure 1: Collaborations

- A collaboration is a society of classes, interfaces, and other elements that work together to provide some cooperative behavior that's bigger than the sum of all its parts.
- A collaboration is also the specification of how an element, such as a classifier (including a class, interface, component, node, or use case) or an operation, is realized by a set of classifiers and associations playing specific roles used in a specific way.
- Collaboration names a conceptual chunk that encompasses both static and dynamic aspects.
- A collaboration gives a name to the conceptual building blocks of your system, encompassing both structural and behavioral elements.
- Graphically, a collaboration is rendered as an ellipse with dashed lines as in Figure 1
- collaboration name must be unique within its enclosing package

Structure

- Collaborations have two aspects: *a structural part* that specifies the classes, interfaces, and other elements that work together to carry out the named collaboration, and a *behavioral part* that specifies the dynamics of how those elements interact.
- a collaboration may cut across many levels of a system
- An element may appear in more than one collaboration
- Figure 2: shows Structural Aspects of a Collaboration

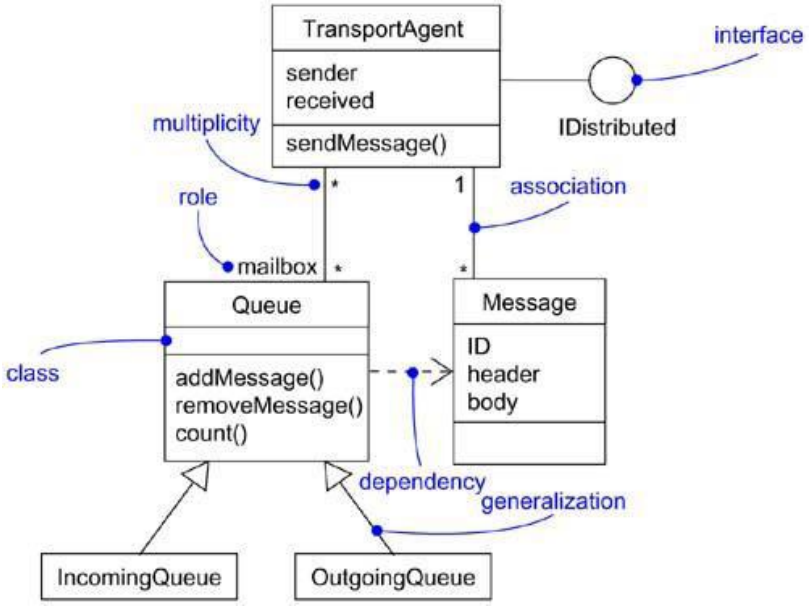


Figure 2: Structural Aspects of a Collaboration

Behavior

- structural part of a collaboration is typically rendered using a class diagram, the behavioral part of a collaboration is typically rendered using an interaction diagram
- to emphasize the time ordering of messages, use a sequence diagram
- to emphasize the structural relationships among these objects as they collaborate, use a collaboration diagram
- Figure 3: shows Behavioral Aspects of a Collaboration
- the objects found in a collaboration's interactions must be instances of classes found in its structural part

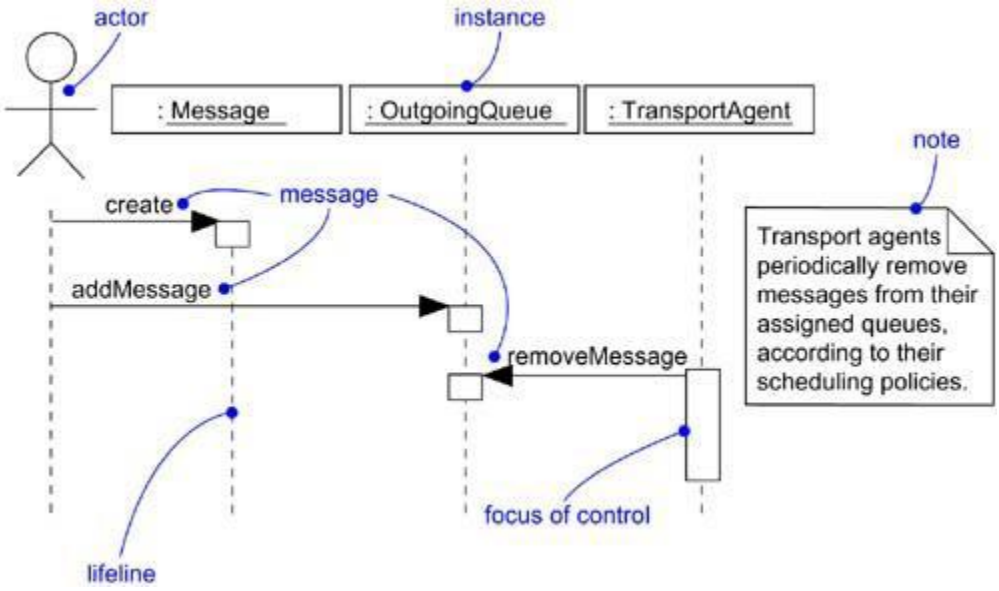


Figure 3: Behavioral Aspects of a Collaboration

Organizing Collaborations

A collaboration may *realize either a classifier or an operation*, which means that the collaboration specifies the structural and behavioral realization of that classifier or operation. The relationship between a classifier or an operation and the collaboration that realizes it is modeled as a realization relationship.
Figure 4: shows Organizing Collaborations

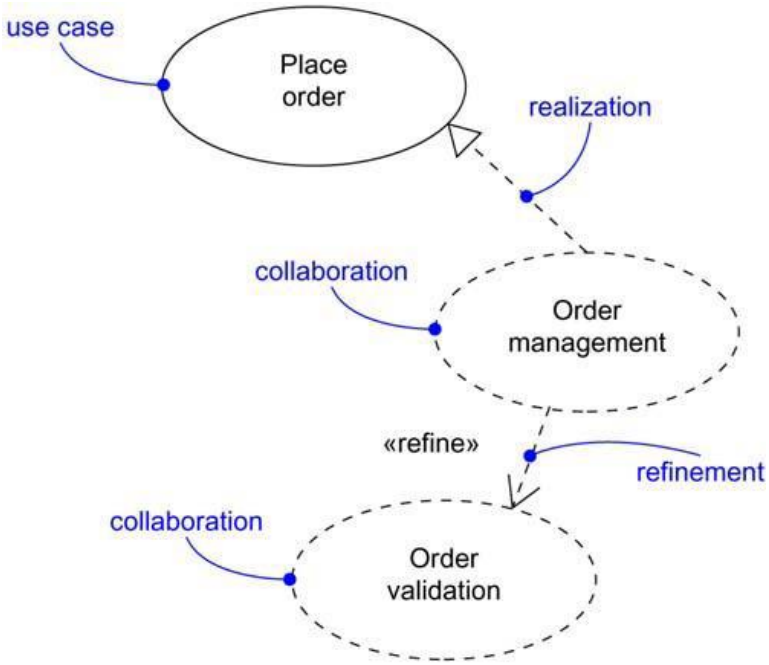


Figure 4: Organizing Collaborations

Modeling the Realization of a Use Case

To model the realization of a use case,

- Identify those structural elements necessary and sufficient to carry out the semantics of the use case.
- Capture the organization of these structural elements in class diagrams.
- Consider the individual scenarios that represent this use case. Each scenario represents a specific path through the use case.
- Capture the dynamics of these scenarios in interaction diagrams. Use sequence diagrams if you want to emphasize the time ordering of messages. Use collaboration diagrams if you want to emphasize the structural relationships among these objects as they collaborate.
- Organize these structural and behavioral elements as a collaboration that you can connect to the use case via realization.

Figure 5: shows a set of use cases drawn from a credit card validation system

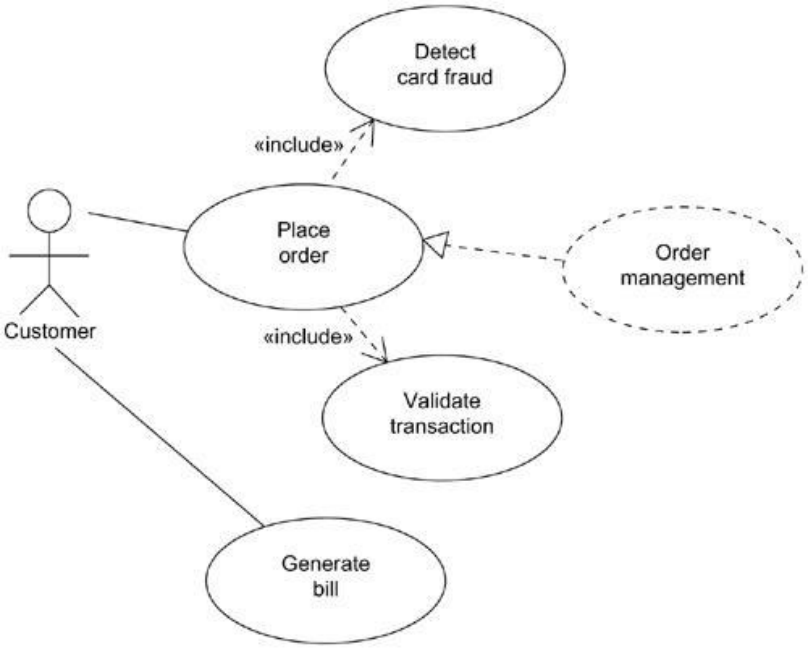


Figure 5: Modeling the Realization of a Use Case

Modeling the Realization of an Operation

To model the implementation of an operation,

- Identify the parameters, return value, and other objects visible to the operation.
- If the operation is trivial, represent its implementation directly in code, which you can keep in the backplane of your model, or explicitly visualize it in a note.
- If the operation is algorithmically intensive, model its realization using an activity diagram.
- If the operation is complex or otherwise requires some detailed design work, represent its implementation as a collaboration. You can further expand the structural and behavioral parts of this collaboration using class and interaction diagrams, respectively.

Figure 6: shows the active class RenderFrame with three of its operations exposed.

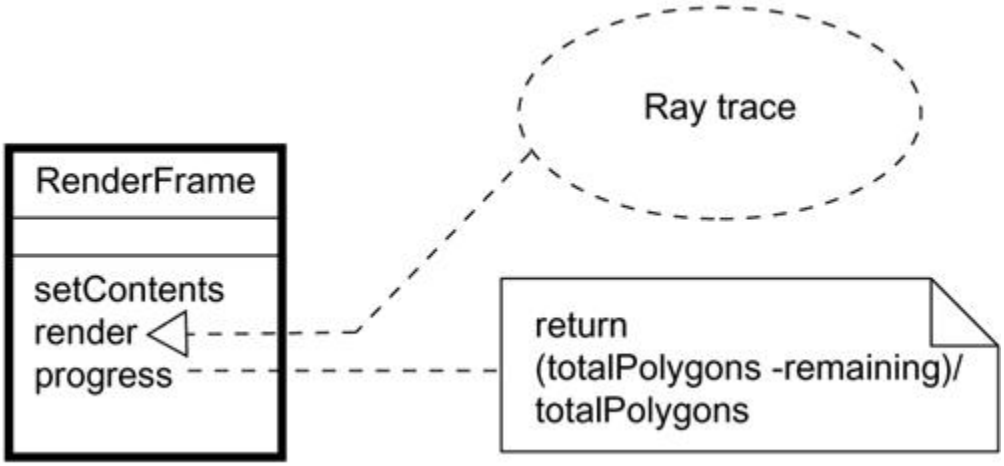


Figure 6: Modeling the Realization of an Operation

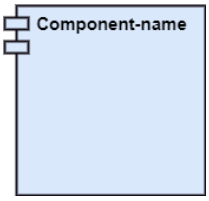
Component Diagram

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

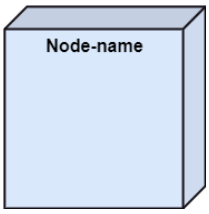
It visualizes the relationships as well as the organization between the components present in the system. It helps in forming an executable system. A component is a single unit of the system, which is replaceable and executable. The implementation details of a component are hidden, and it necessitates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

Notation of a Component Diagram

a) A component



b) A node



Purpose of a Component Diagram

Since it is a special kind of a UML diagram, it holds distinct purposes. It describes all the individual components that are used to make the functionalities, but not the functionalities of the system. It visualizes the physical components inside the system. The components can be a library, packages, files, etc.

The component diagram also describes the static view of a system, which includes the organization of components at a particular instant. The collection of component diagrams represents a whole system.

The main purpose of the component diagram are enlisted below:

1. It envisions each component of a system.

2. It constructs the executable by incorporating forward and reverse engineering.
3. It depicts the relationships and organization of components.

Why use Component Diagram?

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

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

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

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

When to use a Component Diagram?

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

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

How to Draw a Component Diagram?

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

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

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

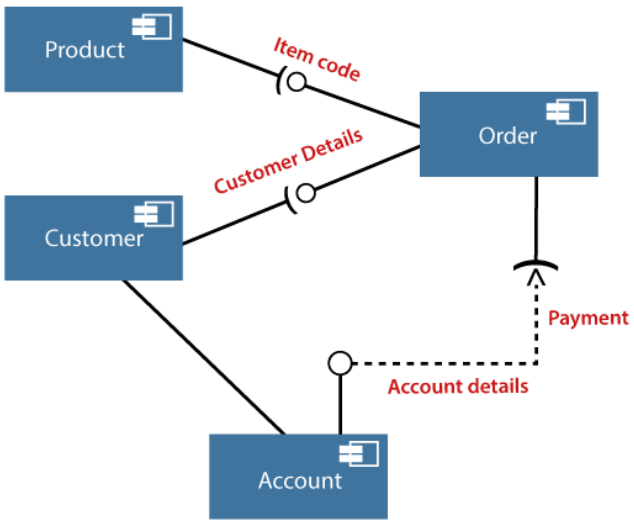
1. What files are used inside the system?
2. What is the application of relevant libraries and artifacts?
3. What is the relationship between the artifacts?

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

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

Example of a Component Diagram

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



Where to use Component Diagrams?

The component diagram is a special purpose diagram, which is used to visualize the static implementation view of a system. It represents the physical components of a system, or we can say it portrays the organization of the components inside a system. The components, such as libraries, files, executables, etc. are first needed to be organized before the implementation.

The component diagram can be used for the followings:

1. To model the components of the system.
2. To model the schemas of a database.
3. To model the applications of an application.
4. To model the system's source code.

Deployment Diagram

The deployment diagram visualizes the physical hardware on which the software will be deployed. It portrays the static deployment view of a system. It involves the nodes and their relationships.

It ascertains how software is deployed on the hardware. It maps the software architecture created in design to the physical system architecture, where the software will be executed as a node. Since it involves many nodes, the relationship is shown by utilizing communication paths.

Purpose of Deployment Diagram

The main purpose of the deployment diagram is to represent how software is installed on the hardware component. It depicts in what manner a software interacts with hardware to perform its execution.

Both the deployment diagram and the component diagram are closely interrelated to each other as they focus on software and hardware components. The component diagram represents the components of a system, whereas the deployment diagram describes how they are actually deployed on the hardware.

The deployment diagram does not focus on the logical components of the system, but it put its attention on the hardware topology.

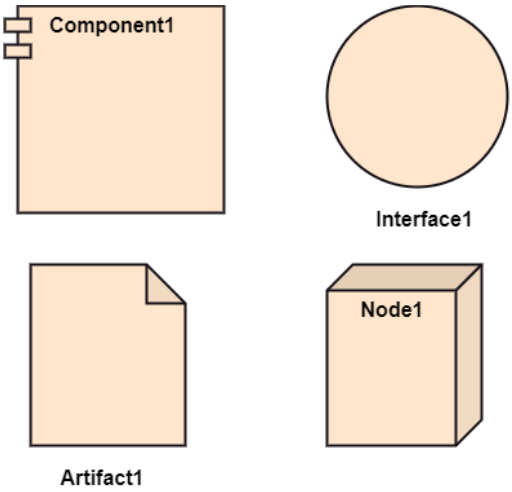
Following are the purposes of deployment diagram enlisted below:

1. To envision the hardware topology of the system.
2. To represent the hardware components on which the software components are installed.
3. To describe the processing of nodes at the runtime.

Symbol and notation of Deployment diagram

The deployment diagram consist of the following notations:

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



How to draw a Deployment Diagram?

The deployment diagram portrays the deployment view of the system. It helps in visualizing the topological view of a system. It incorporates nodes, which are physical hardware. The nodes are used to execute the artifacts. The instances of artifacts can be deployed on the instances of nodes.

Since it plays a critical role during the administrative process, it involves the following parameters:

1. High performance
2. Scalability
3. Maintainability
4. Portability
5. Easily understandable

One of the essential elements of the deployment diagram is the nodes and artifacts. So it is necessary to identify all of the nodes and the relationship between them. It

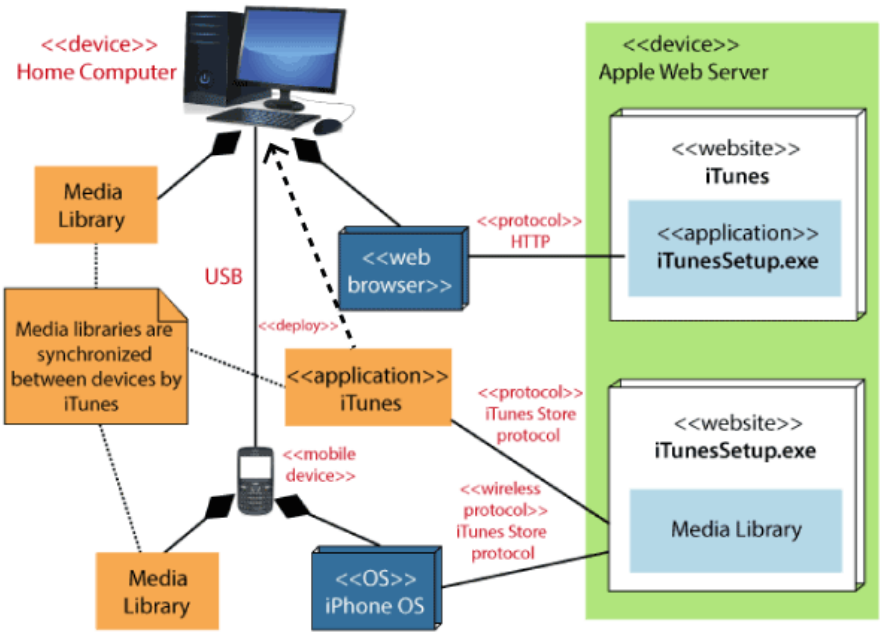
becomes easier to develop a deployment diagram if all of the nodes, artifacts, and their relationship is already known.

Example of a Deployment diagram

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

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

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



When to use a Deployment Diagram?

The deployment diagram is mostly employed by network engineers, system administrators, etc. with the purpose of representing the deployment of software on the hardware system. It envisions the interaction of the software with the hardware to accomplish the execution. The selected hardware must be of good quality so that the software can work more efficiently at a faster rate by producing accurate results in no time.

The software applications are quite complex these days, as they are standalone, distributed, web-based, etc. So, it is very necessary to design efficient software.

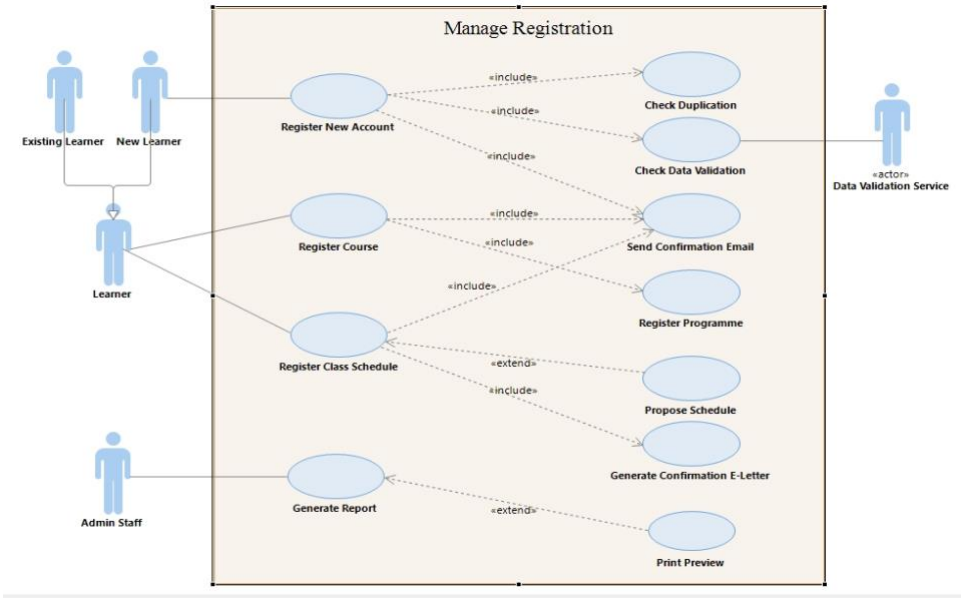
Deployment diagrams can be used for the followings:

1. To model the network and hardware topology of a system.
2. To model the distributed networks and systems.
3. Implement forwarding and reverse engineering processes.
4. To model the hardware details for a client/server system.
5. For modeling the embedded system.

MODULE-5

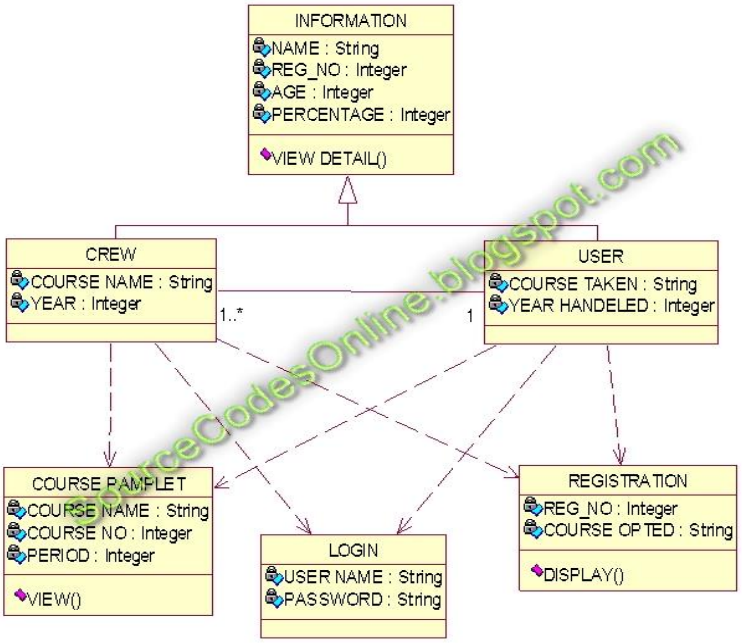
Student's Registration System

Use case Diagram

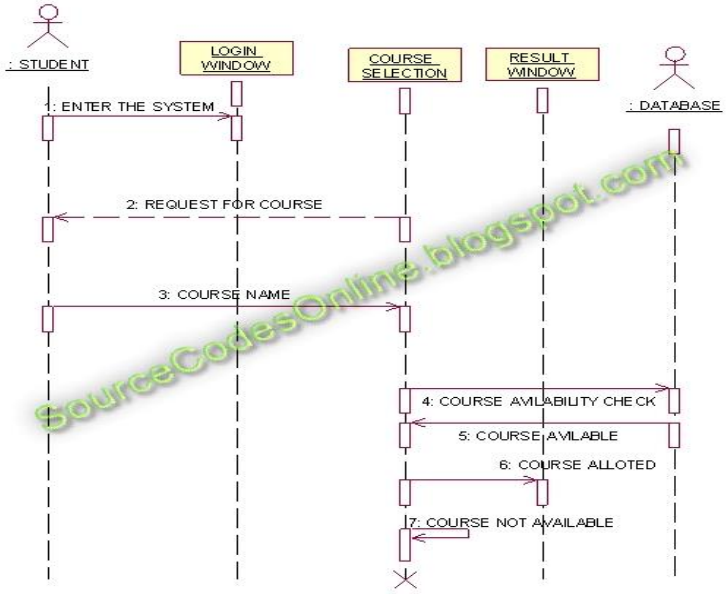


OR

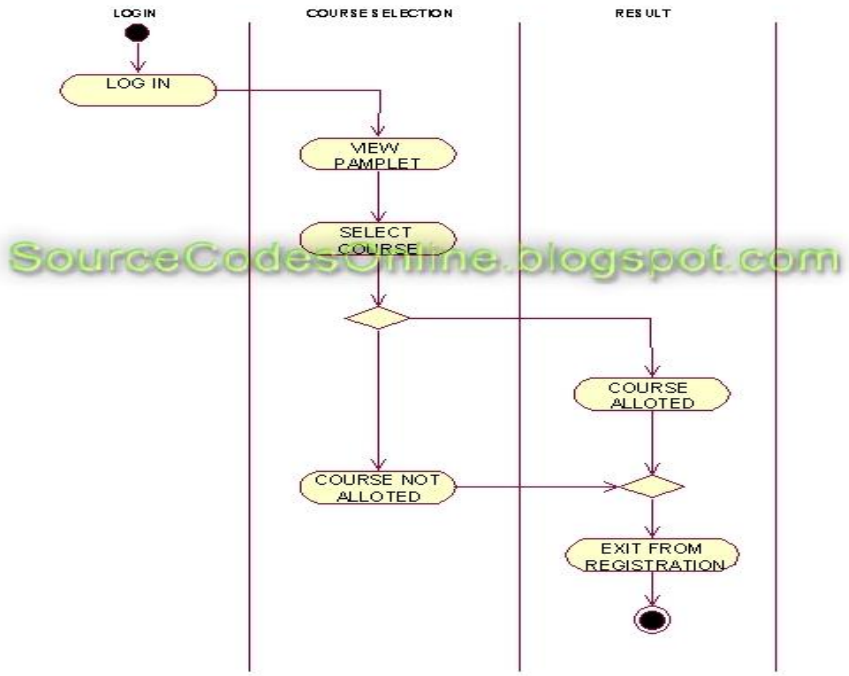
Class Diagram



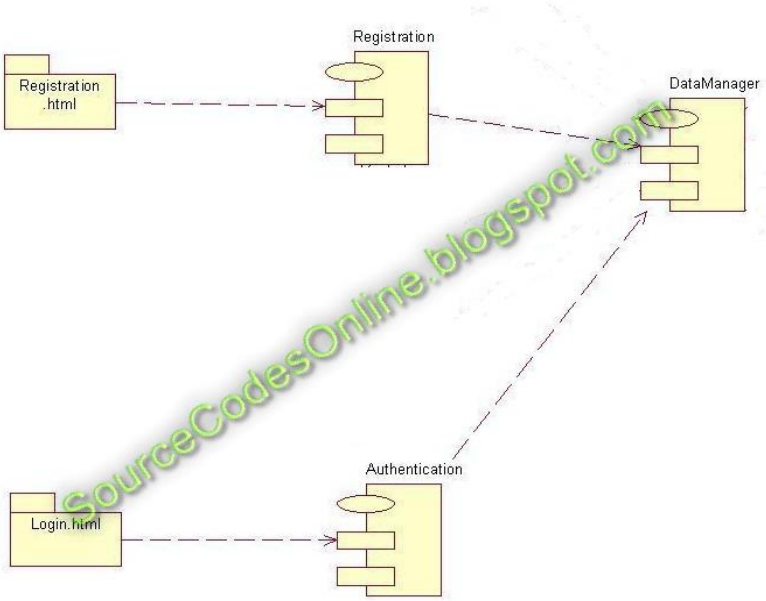
Sequence Diagram



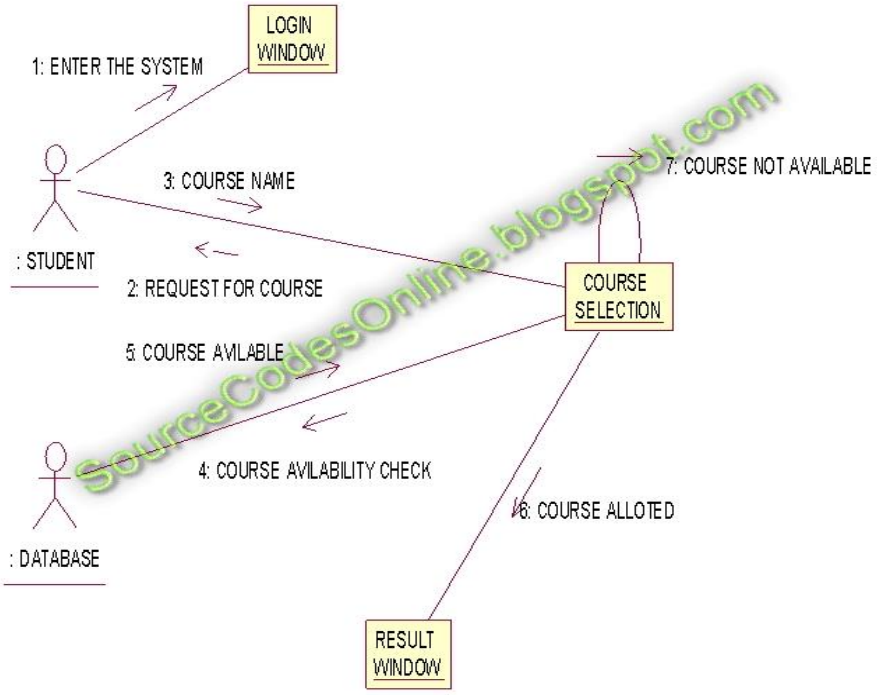
Activity Diagram



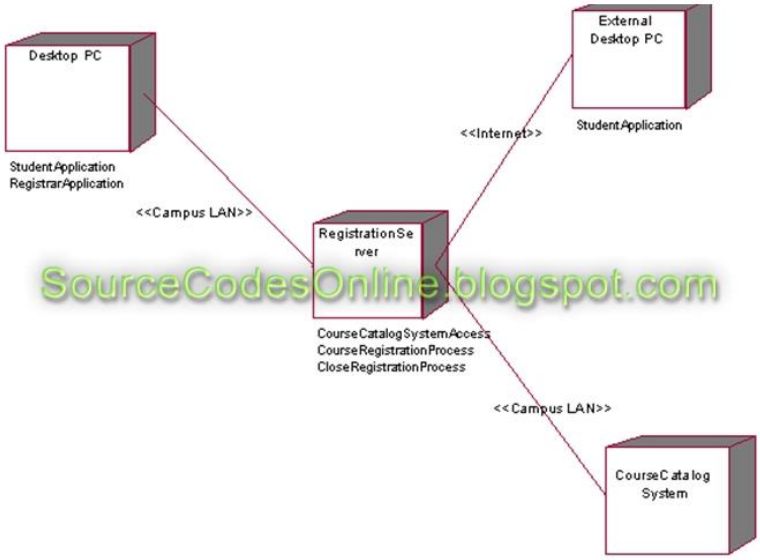
Component Diagram



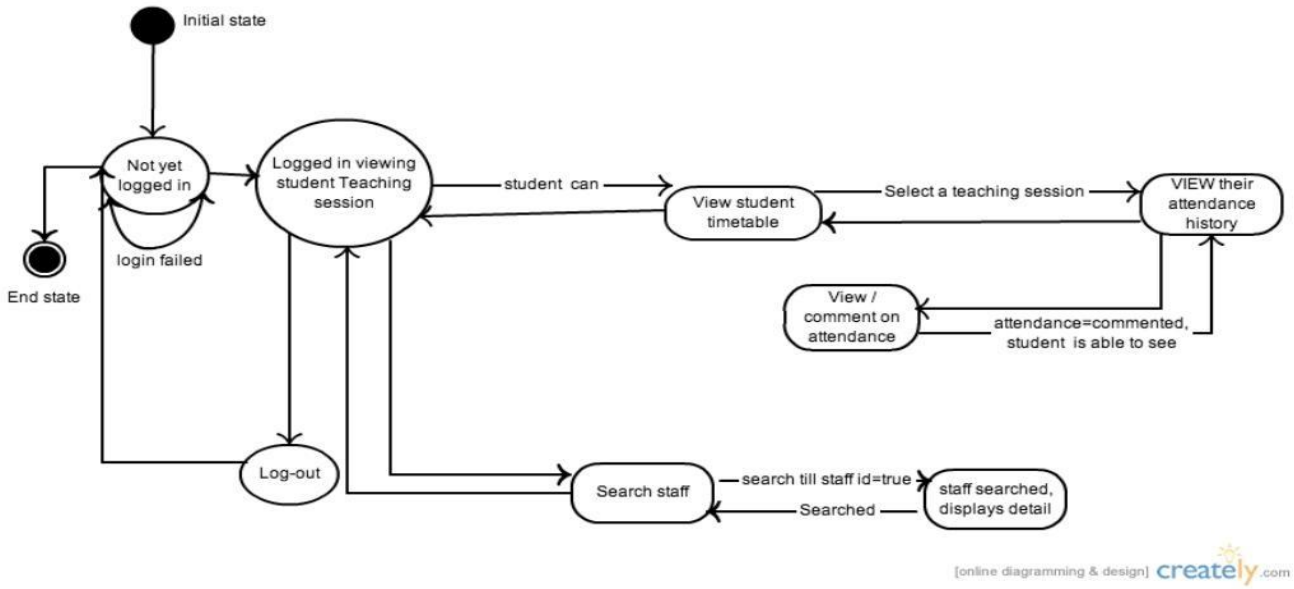
Collaboration Diagram



Deployment Diagram



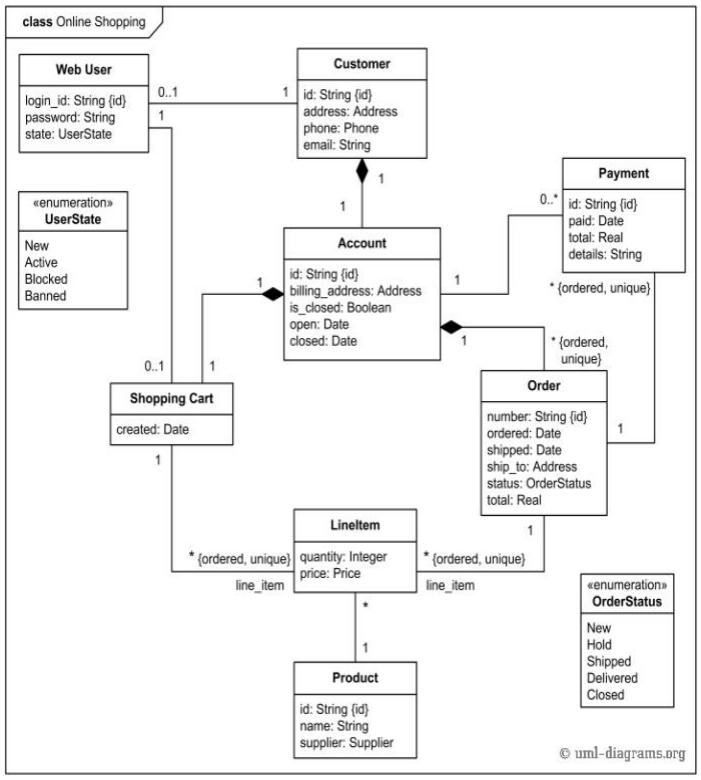
Statechart Diagram

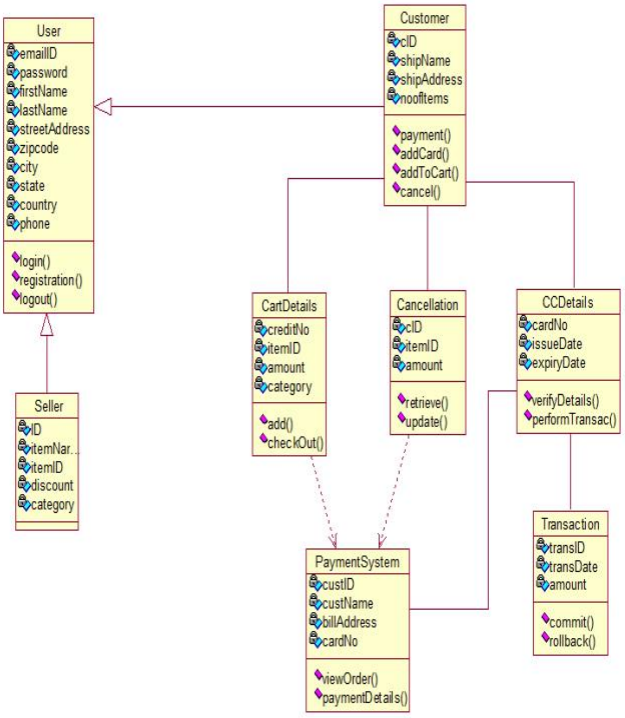


Online Shopping System

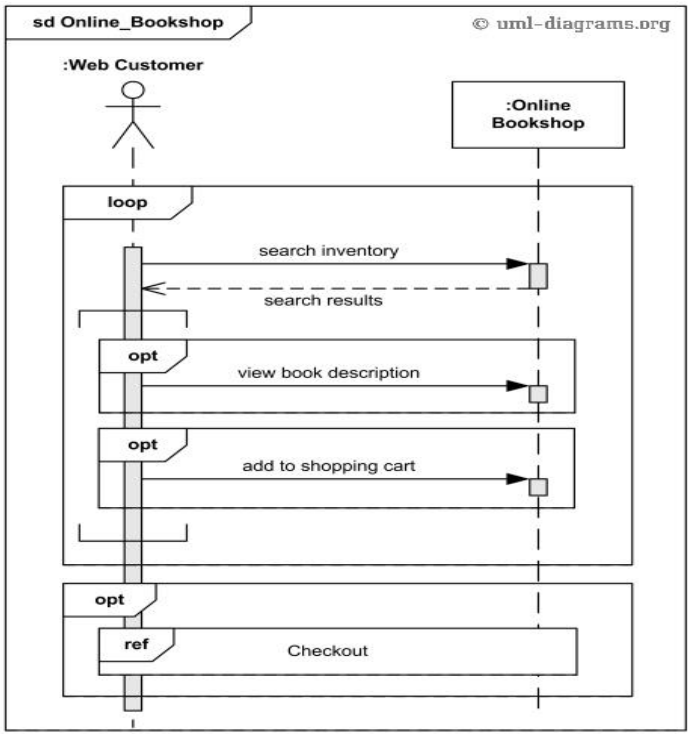
Usecase Diagram

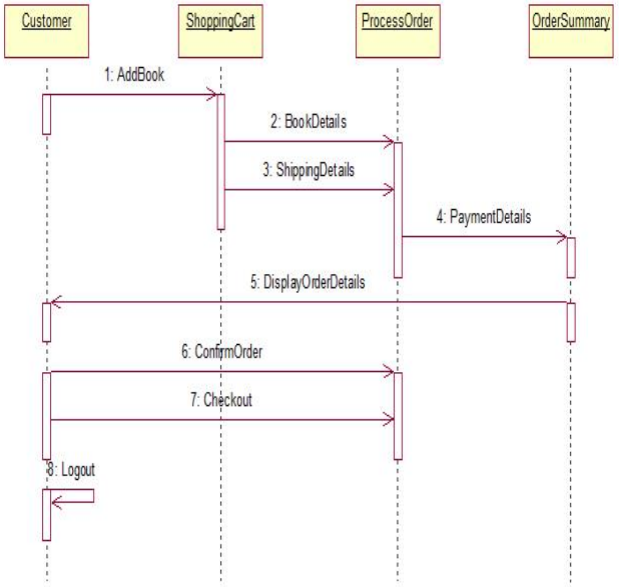
Class Diagram



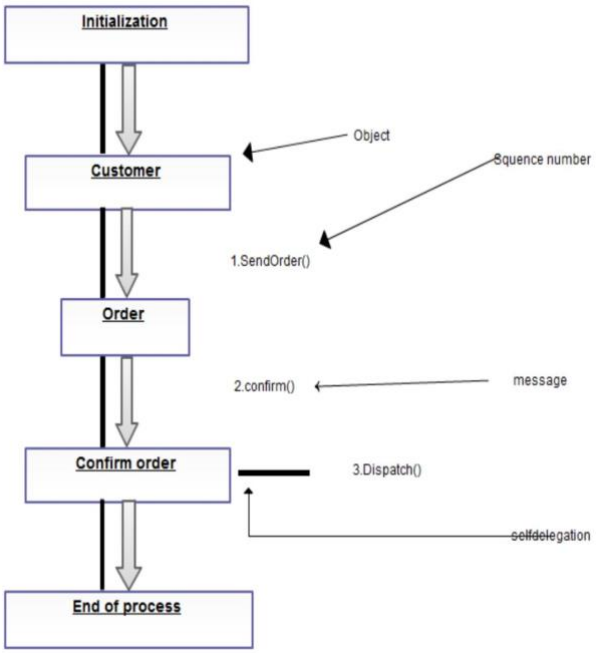


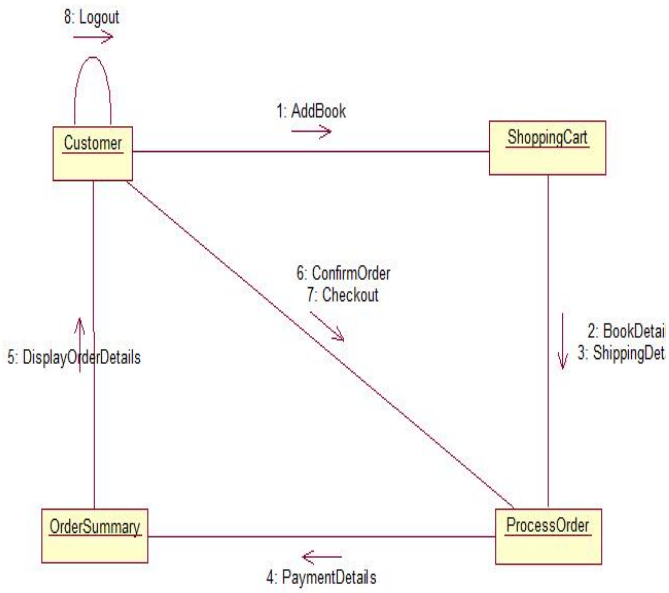
Sequence Diagram



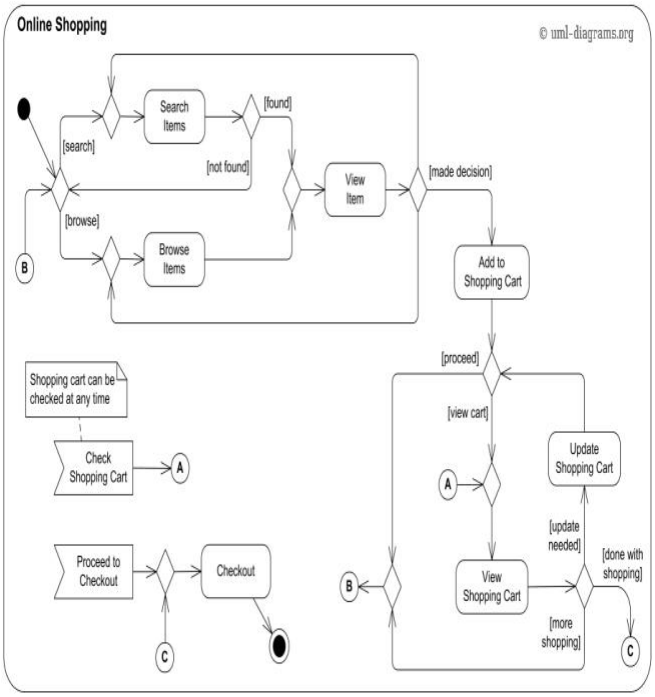


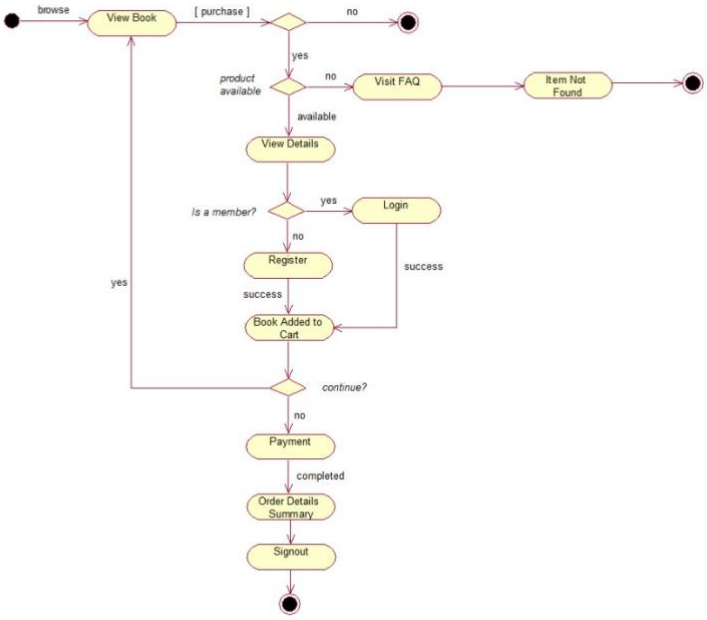
Collaboration Diagram



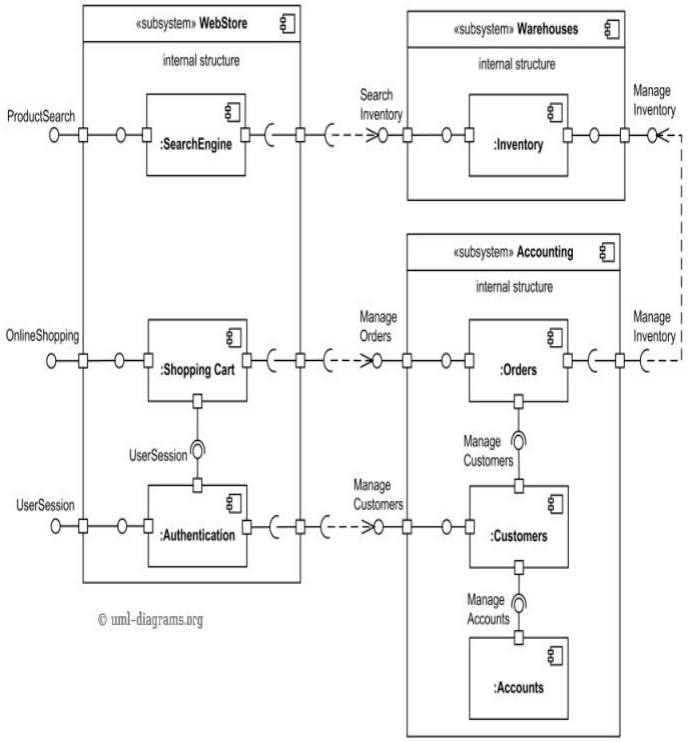


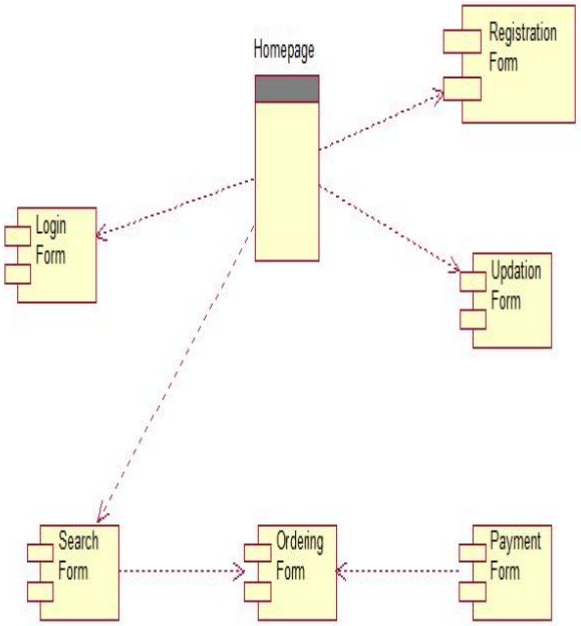
Activity Diagram



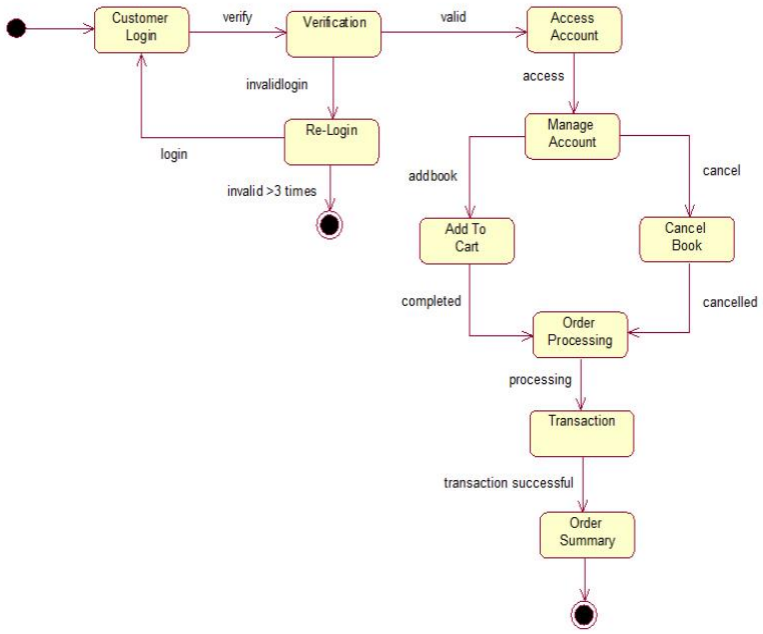


Component Diagram

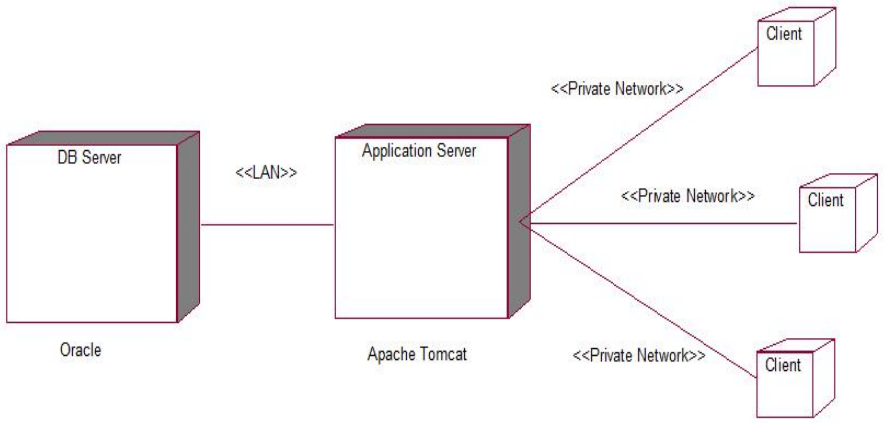




StateChart Diagram

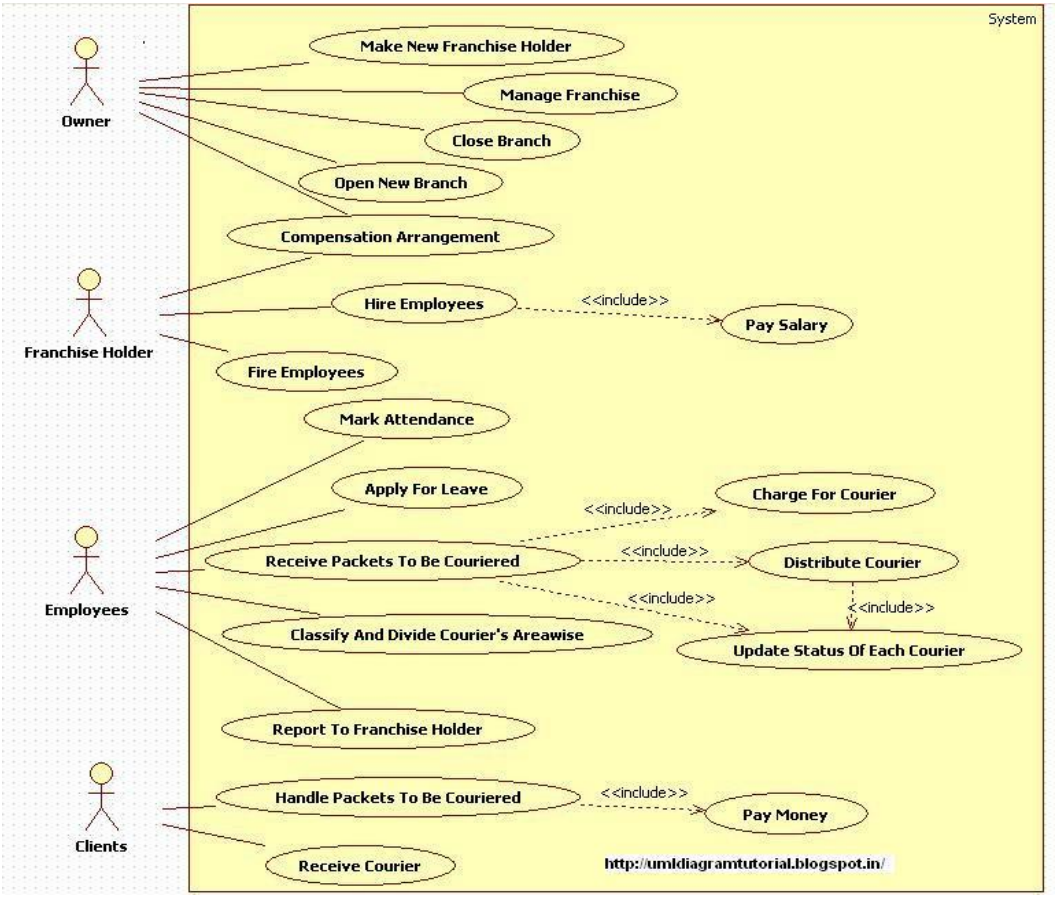


Deployment Diagram

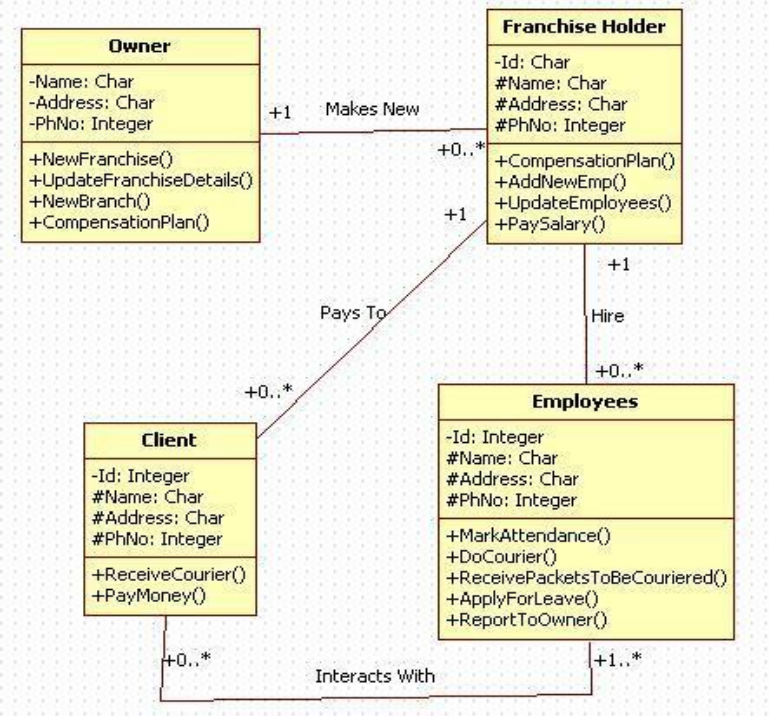


Courier Tracking System

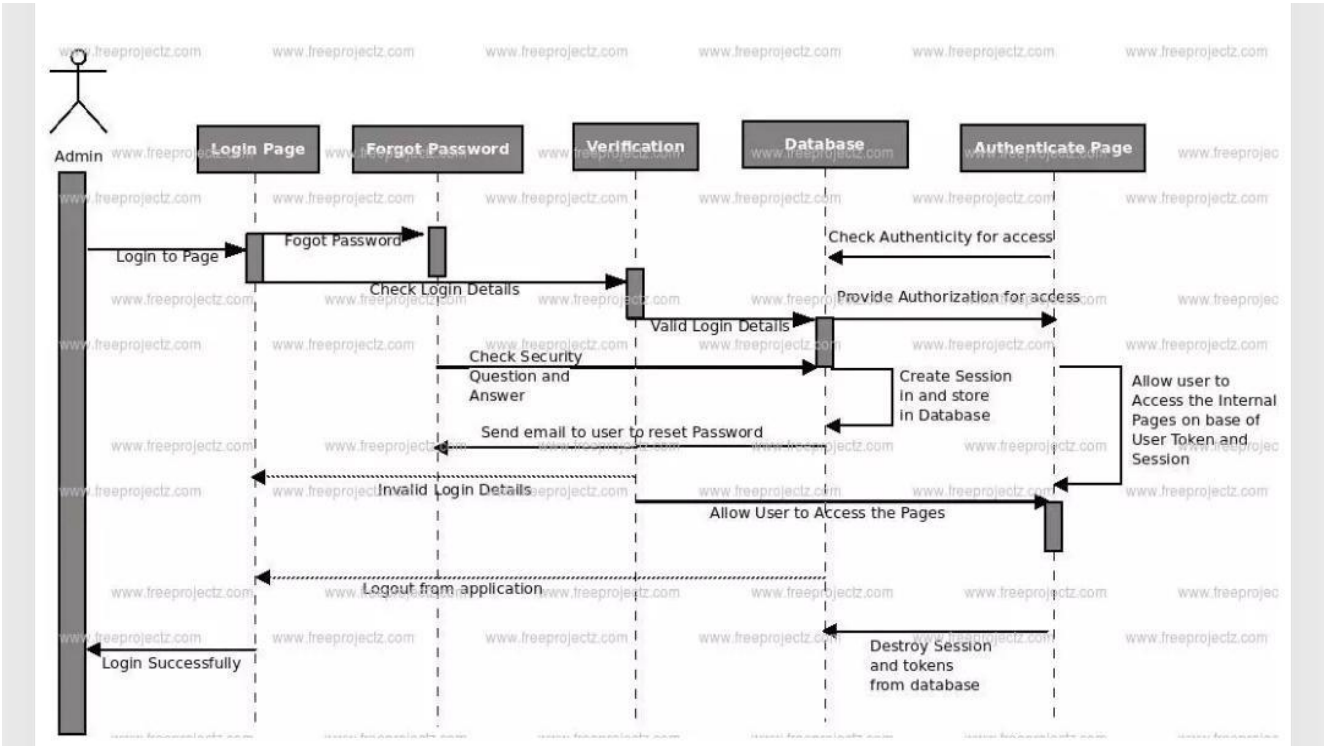
Use case Diagram



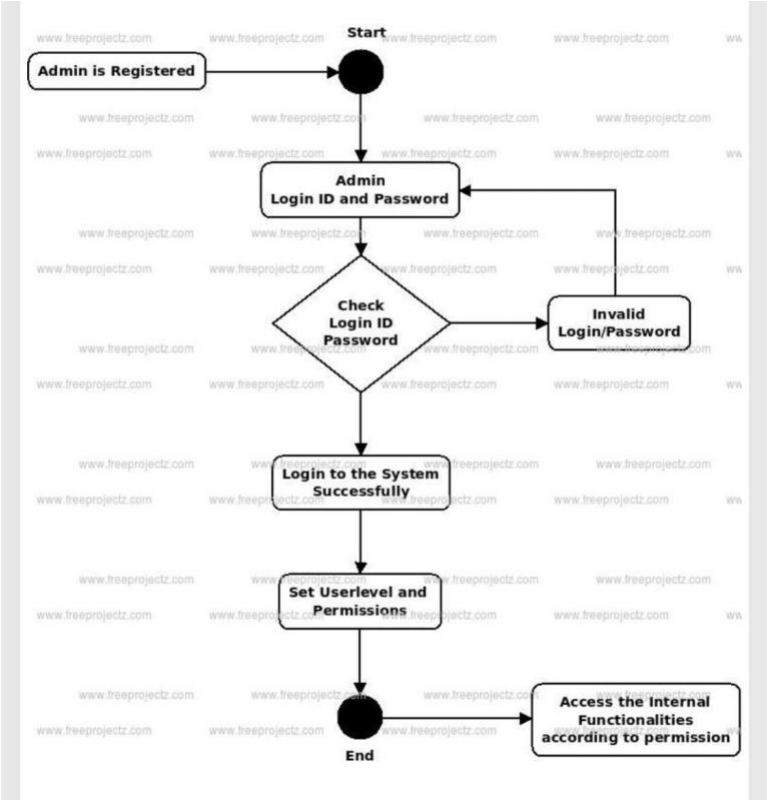
Class Diagram



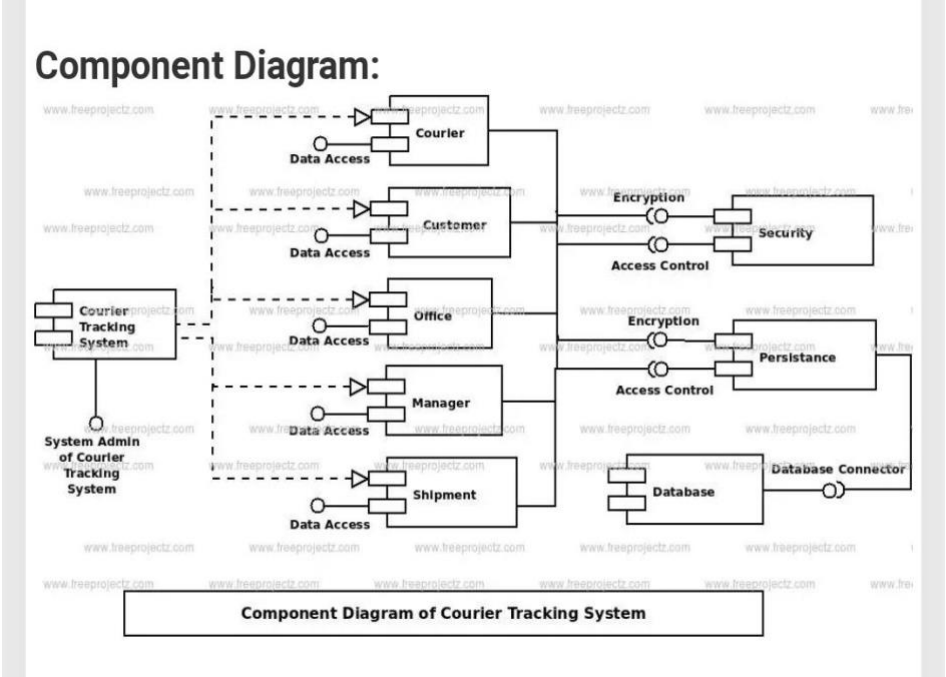
Sequence Diagram



Activity diagram



Component Diagram



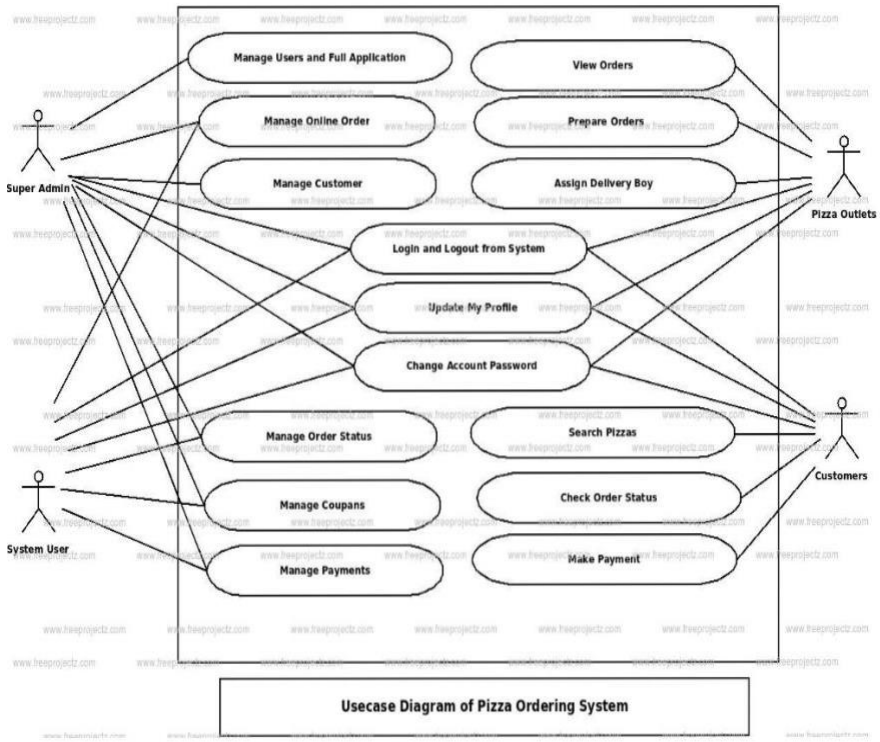
Collaboration Diagram

StateChart Diagram

Deployment Diagram

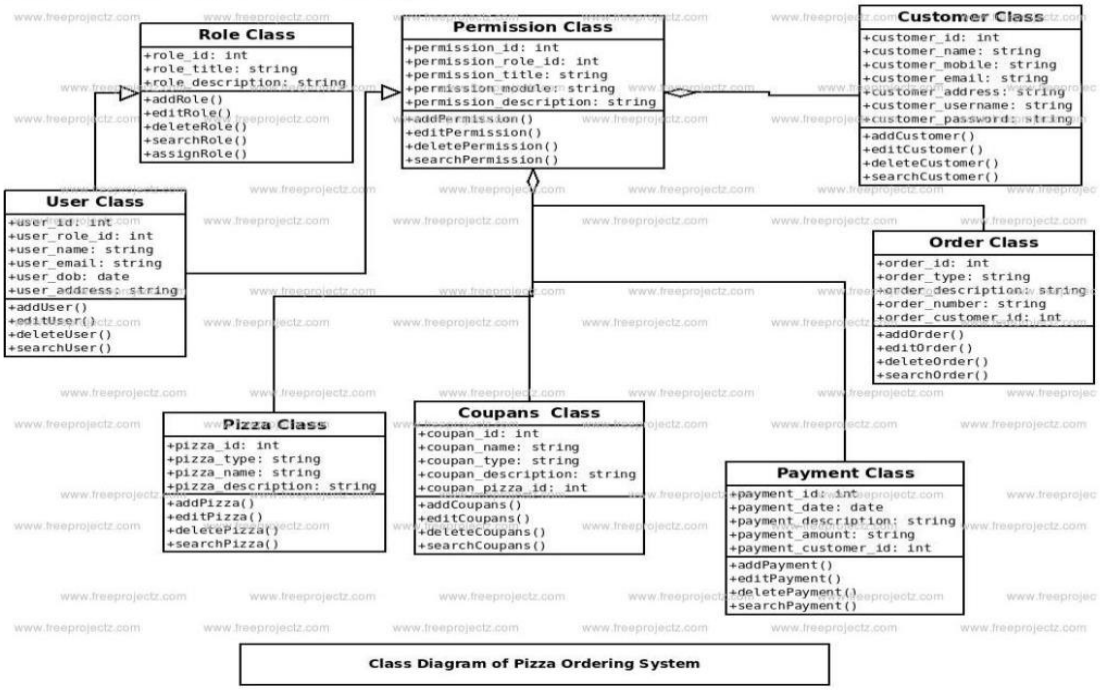
Online pizza ordering System

Use case Diagram

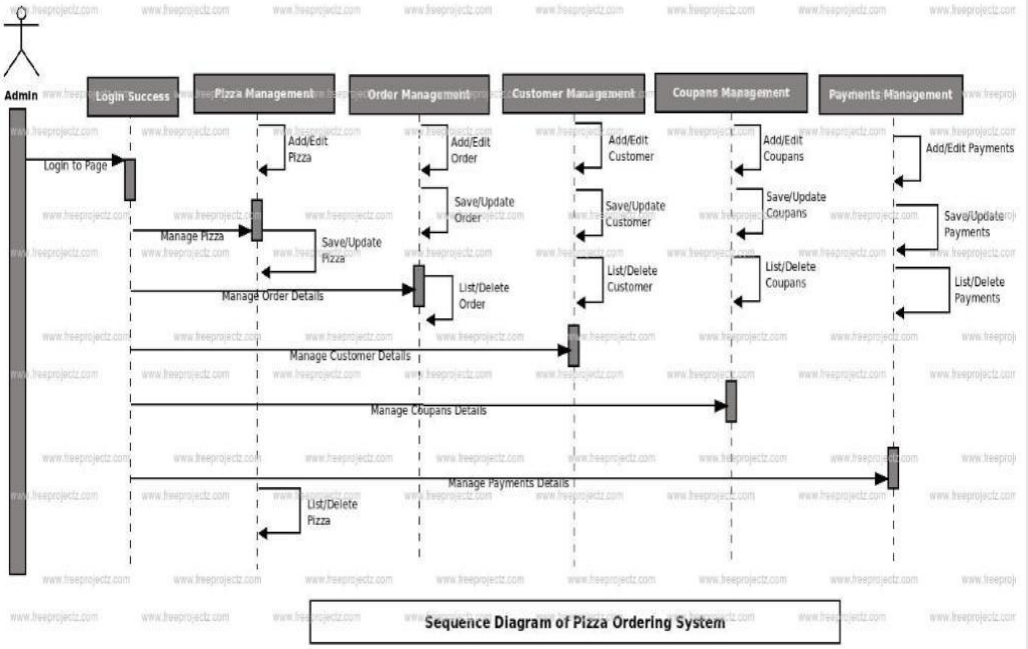


Class Diagram

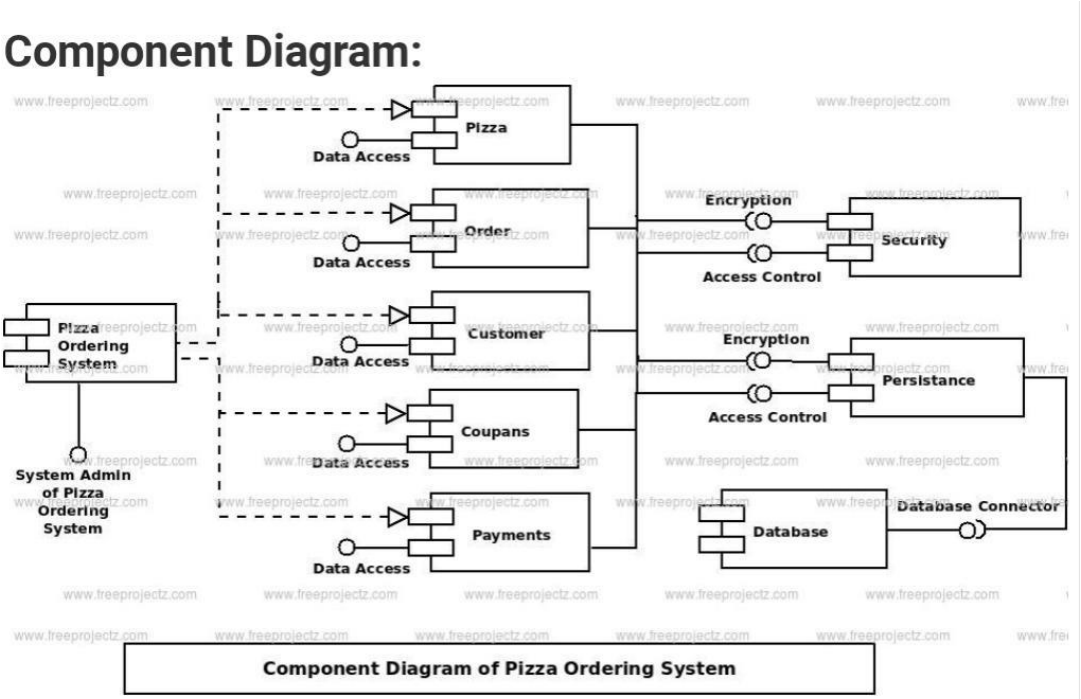
Class Diagram Image:



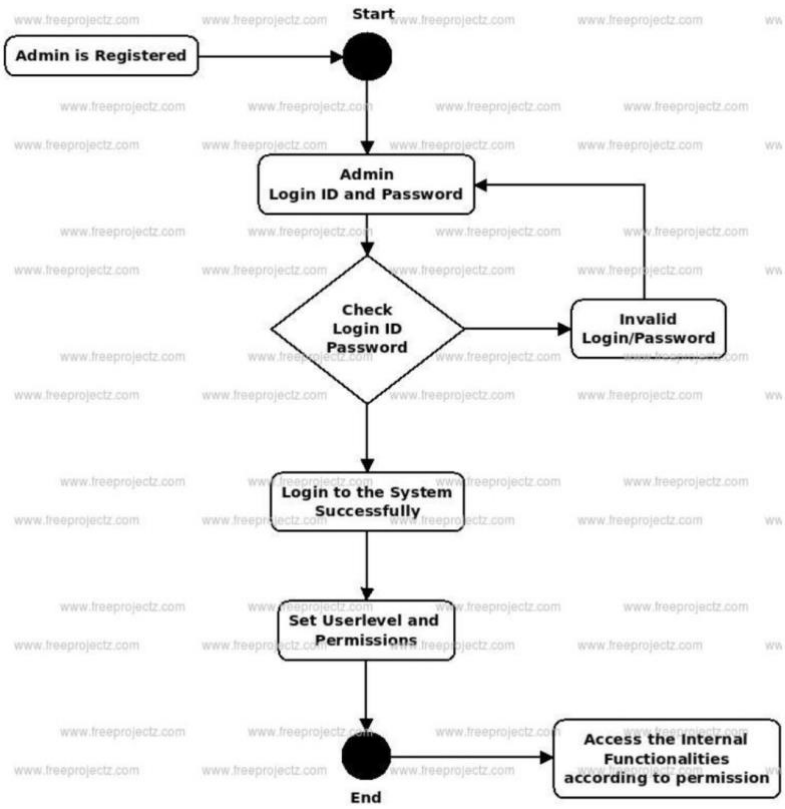
Sequence Diagram



Component Diagram



Activity Diagram



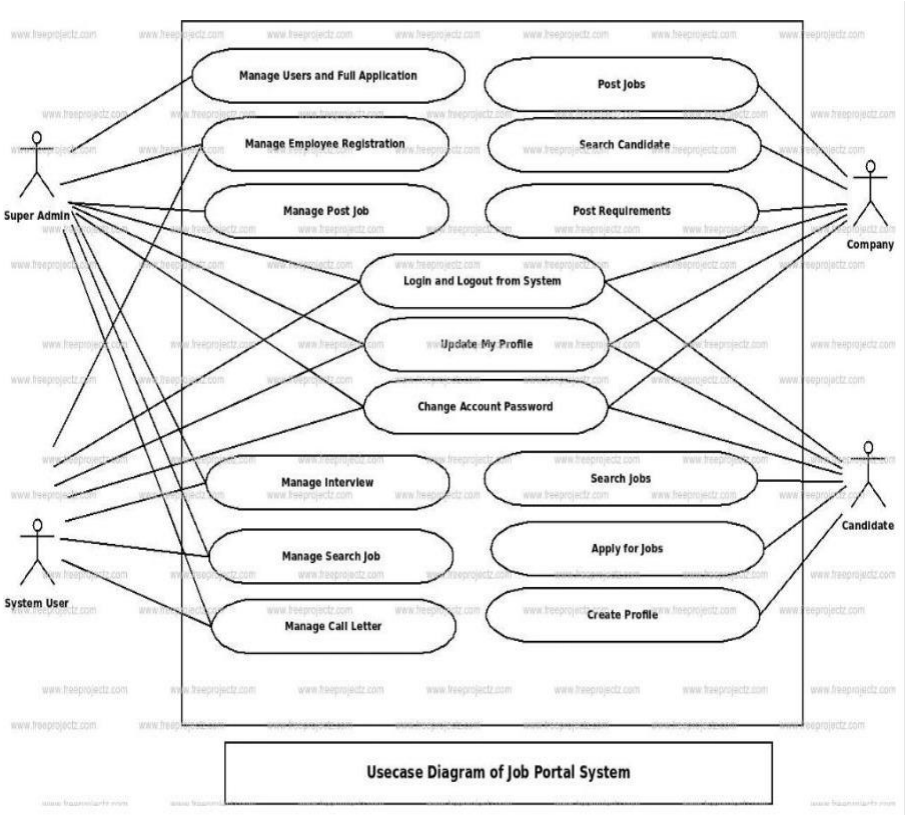
StateChart Diagram

collaboration Diagram

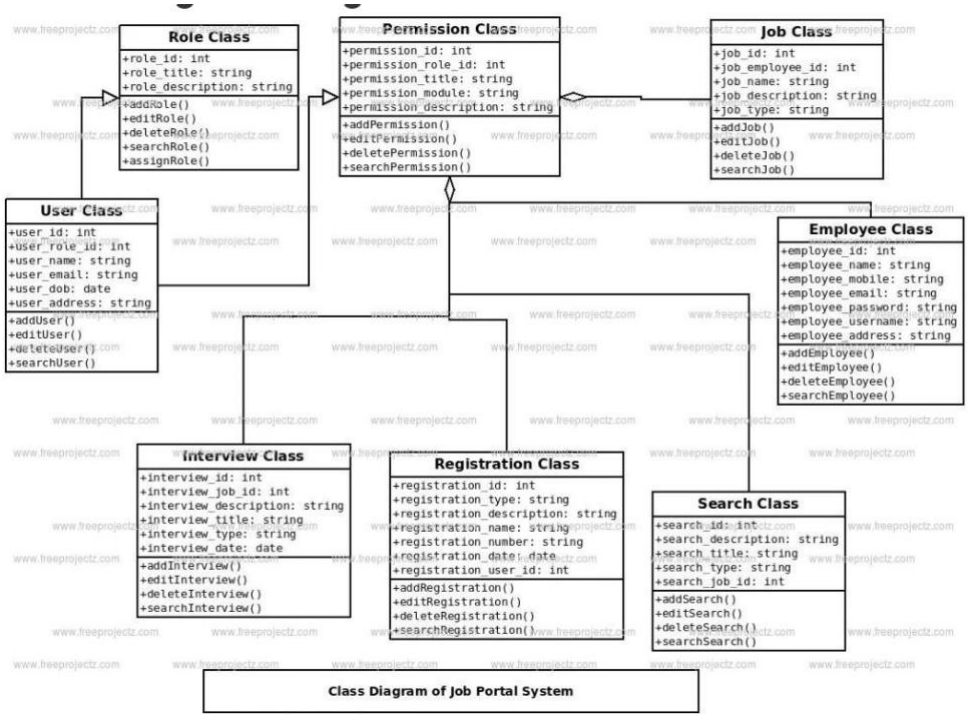
Deployment Diagram

Online Job Portal System

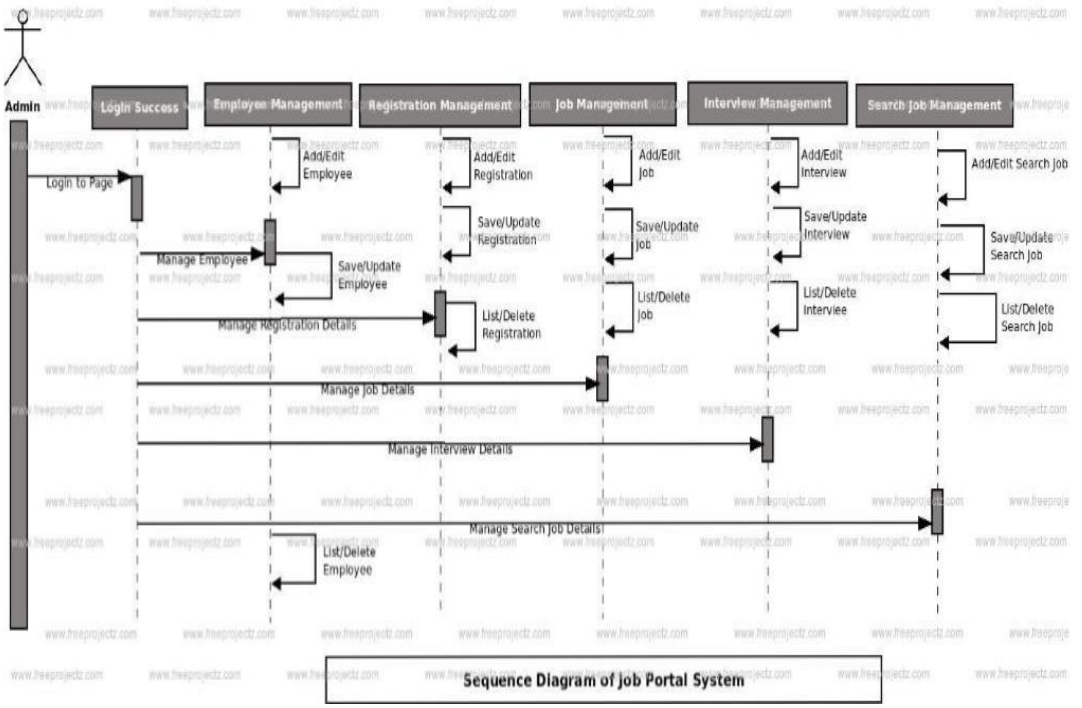
Use case Diagram



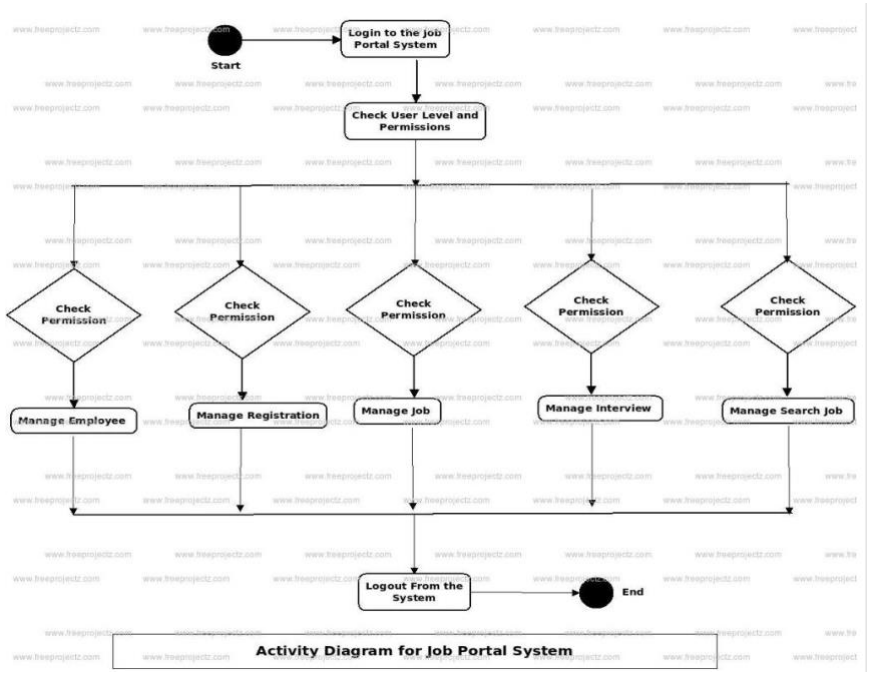
Class Diagram



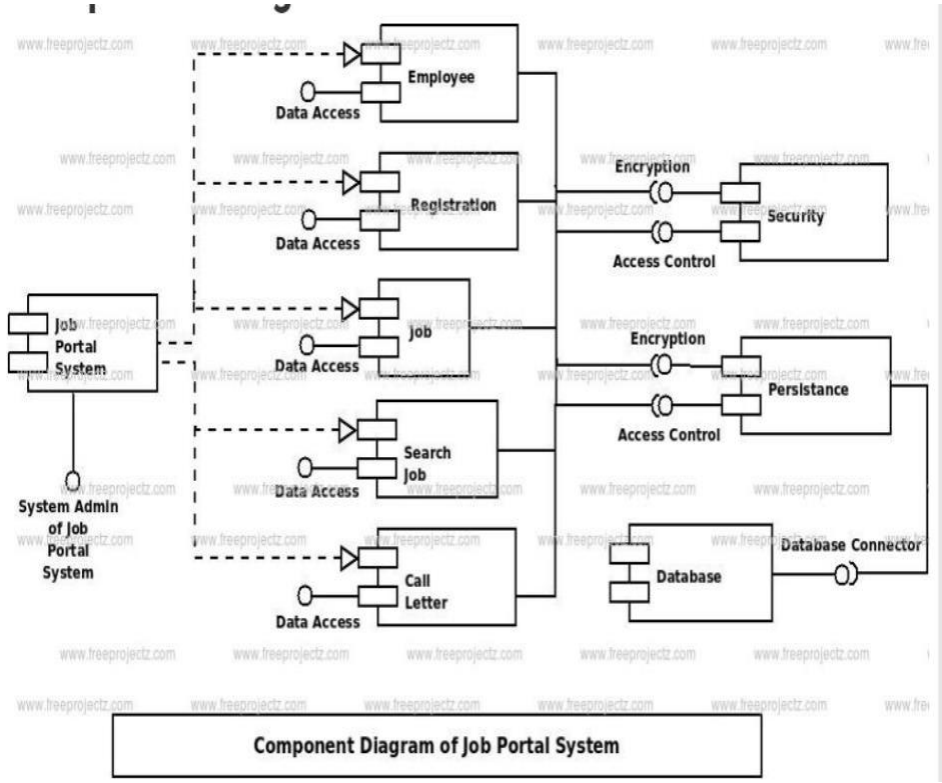
Sequence Diagram



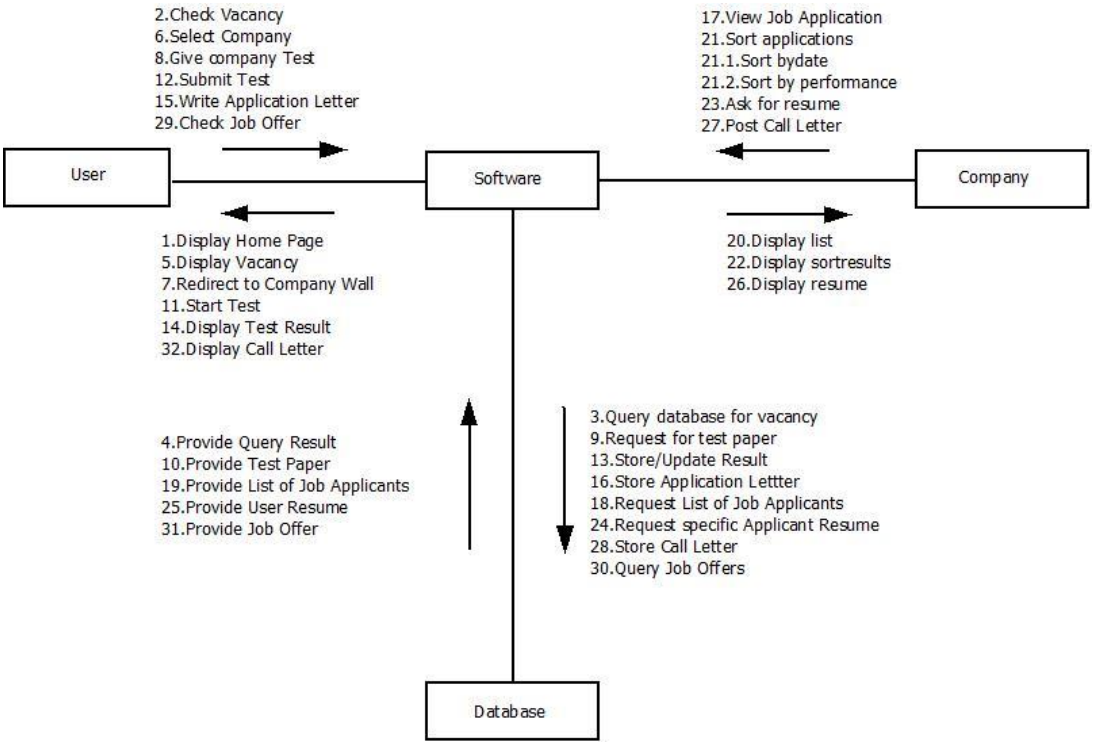
Activity Diagram



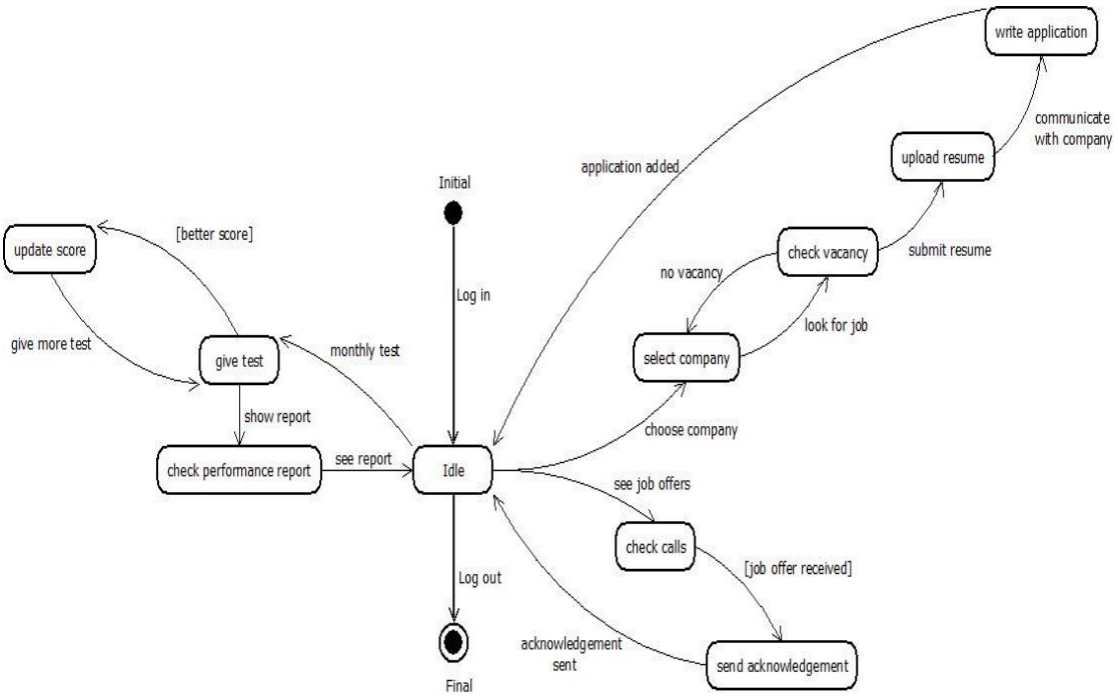
Component Diagram



Collaboration Diagram



State Chart Diagram



Deployment Diagram