11 Unified Modeling Language

11.1 Introduction

UML is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems. UML was created by Object Management Group and UML 1.0 specification draft was proposed to the OMG in January 1997.[1]

OMG is continuously putting effort to make a truly industry standard. UML stands for Unified Modeling Language. It is different from the other common programming languages like C++, Java, COBOL *etc.* UML is a pictorial language used to make software blue prints. So UML can be described as a general purpose visual modeling language to visualize, specify, construct and document software systems. Although UML is generally used to model software systems but it is not limited within this boundary. It is also used to model non software systems as well like process flow in a manufacturing unit *etc.*

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

11.2 Goals of UML

A picture is worth a thousand words, this absolutely fits while discussing about UML. Object oriented concepts were introduced much earlier than UML. So at that time there were no standard methodologies to organize and consolidate the object oriented development. At that point of time 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 also it 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. So it must be clear that UML is not a development method rather it accompanies with processes to make a successful system. The goal of UML can be defined as a simple modeling mechanism to model all possible practical systems in today.s complex environment.

11.3 Conceptual Model

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. 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

11.4 Object Oriented Concepts

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

The objects are the real world entities that exist around us and the basic concepts like abstraction, encapsulation, inheritance, polymorphism *etc.* can be represented using UML. Thus UML is powerful enough to represent all the concepts existing in object oriented analysis and design.

UML diagrams are representation of object oriented concepts only. So before learning UML, it becomes important to understand OO concepts in details.

Following are some fundamental concepts of object oriented world:

- **Objects:** Objects represent an entity and the basic building block.
- **Class:** Class is the blue print of an object.
- **Abstraction:** Abstraction represents the behavior of a real world entity.
- **Encapsulation:** Encapsulation is the mechanism of binding together the data and hiding them from outside world.
- **Inheritance:** Inheritance is the mechanism of making new specialized classes from existing one.
- **Polymorphism:** Polymorphism is the ability to create a variable, a function, or an object that has more than one form.[2]

[2]

11.5 OO Analysis and Design

Object Oriented analysis and design can be defined as the investigation of objects and their inter-relationships. The most important purpose of OO analysis is to identify objects of a system to be designed. This analysis can also be done for an existing system. An efficient analysis is only possible when we are able to start thinking in a way where objects can be identified.

After identifying the objects, their relationships are identified and finally the design is produced. So the purpose of OO analysis and design can be described as:

- Identifying the objects of a system.
- Identify their relationships.
- Make a design which can be converted to executables using OO languages or semi-OO languages.

There are three basic steps where the OO concepts are applied and implemented. These steps are

OO Analysis –> OO Design –> OO implementation

Now the these three steps can be described in details:

- During object oriented analysis, the most important part is to identify objects and describe them in a proper way. If these objects are identified efficiently then the next job of design is easy. The objects should be identified with responsibilities. Responsibilities are the functions performed by the object. Each object must have one or more responsibilities.
- During object oriented design phase, emphasis is given on the requirements and their fulfilment. In this stage the relationship between objects are identified to implement required functionalities easily. After those relationships or associations are identified and defined completely, this phase is over.
- The last phase is object oriented implementation. In this phase the design is implemented using programming languages like Java, C++, Python, C *etc.*

Even a procedural language like C can be used to implement object oriented design by following various conventions. For example, the Gnome framework has object oriented design, but it is implemented in C.

11.6 Role of UML in OO design

UML is a modeling language used to model software and non software systems. Although UML is used for non software systems, the emphasis is on modeling object oriented software applications. The UML diagrams are used to model different aspects of software systems. Some of those diagrams are class diagram, object diagram, collaboration diagram, interaction diagrams *etc.* All such diagrams are designed based on the objects.

It is very important to understand the relationship between OO design and UML. The OO design is transformed into UML diagrams according to the requirements. Before understanding the UML in details, the OO concepts should be learned properly. Once the OO analysis and design is done, the next step is comparatively easy. The output from the OO analysis and design is the input for the UML diagrams.

11.7 *UML Building Blocks*

To work with UML, first of all a conceptual model of the system is to be prepared. From there onwards, one should proceed to work on each of those models separately. Conceptual model of UML can be mastered by learning the following three major elements:

1. UML building blocks
2. Rules to connect the building blocks
3. Common mechanisms of UML

The building blocks of UML can be defined as:

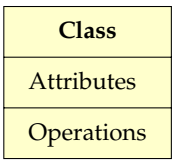
1. Things
2. Relationships
3. Diagrams

11.7.1 *Things*

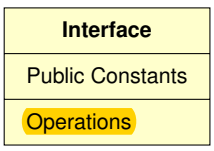
Things are most important building block of UML. They can be:

1. **Structural:** The Structural things define the static part of the model. They represent physical and conceptual elements. Examples are Class, Interface, Collaboration *etc.* Short descriptions of each of the structural things are given below.

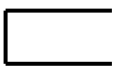
   (a) **Class:** Class represents set of object having same responsibilities.

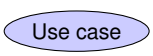
   (b) **Interface:** Interface defines a set of operations which specify the responsibility of a class.

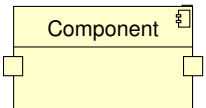
   (c) **Collaboration:** Collaboration defines interaction between elements.

   (d) **Use Case:** Use case represents a set of actions performed by a system for a specific goal.

   (e) **Component:** Component describes physical part of a system.

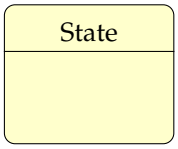
   

2. **Behavioral:** A behavioral thing consists of the dynamic parts of UML models. Following are the behavioral things:

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

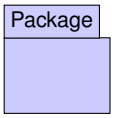
   

   (b) **State Machine:** State machine is useful when the state of an object in its life cycle is important. It defines the sequence of states an object goes through in response to events. Events are external factors responsible for state change.

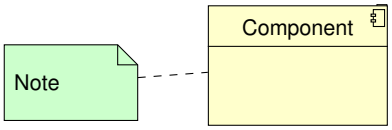
3. **Grouping:** Grouping things can be defined as a mechanism to group elements of a UML model together.

   (a) **Package:** Package is a grouping thing available for gathering structural and behavioral things.

4. **Annotational:** Annotational things can be defined as a mechanism to capture remarks, descriptions, and comments of UML model elements. Note is an Annotational thing.

   (a) **Note:** A note is used to render comments, constraints etc of an UML element.

11.7.2 *Relationships*

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

There are four kinds of relationships defined in UML.

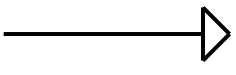
1. **Dependency:** Dependency is a relationship between two things in which change in one element also affects the other one.

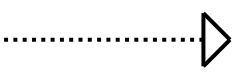
2. **Association:** Association is basically a set of links that connects elements of an UML model. It also describes how many objects are taking part in that relationship.

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

4. **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.

11.8 UML Diagrams

Any complex system is best understood by making some kind of diagrams or pictures. These diagrams have a better impact on our understanding of that system. We prepare UML diagrams to understand a system in a simple and better way. A single diagram is not enough to cover all aspects of the system. So UML defines various kinds of diagrams to cover most of the aspects of a static as well as a dynamic system. We can also create our own set of diagrams to meet our requirements.

Diagrams are generally made in an incremental and iterative way. There are two broad categories of diagrams.

1. Structural Diagrams
2. Behavioral Diagrams

Each elements – things as well as relationships – described in previous section is used to build various kinds of UML diagrams. A UML diagram represents a system. Nine different kinds of UML diagrams are possible.

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

11.9 UML 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 finalized with different perspectives in mind. The most important part is to visualize the system from different viewer's perspective. The better we understand these perspectives, the better we make the system.

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

- Design
- Implementation

- Process
- Deployment

And at 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.

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

11.10 UML Modeling Types

It is very important to distinguish between various UML models. Different diagrams are used for different types of UML modeling. There are three important type of UML modelings in use. The details follow.

11.10.1 Structural Modeling

Structural modeling captures the static features of a system. Hence structural diagrams represent the static aspect of the system. These static aspects represent those parts of a diagram which forms the main structure or skeleton of the system.

They consist of following diagrams.

1. Class diagrams
2. Object diagrams
3. Deployment diagrams
4. Package diagrams
5. Composite structure diagrams
6. Component diagrams

Structural model represents the framework for the system and this framework is the place where all other components exist. So the class, component, and deployment diagrams are considered as part of structural modeling. They all represent the elements and the mechanism to assemble them. But it should be remembered that the structural model never describes the dynamic behavior of the system. Class diagram is one of the most widely used structural diagram.

11.10.2 *Behavioral Modeling*

Behavioral model describes the interaction in the system. It represents the interaction among the structural diagrams. Behavioral modeling shows the dynamic nature of the system. UML has a set of powerful features to represent the dynamic part of software and non-software systems. These features include interactions and state machines.

Behavioral modeling consists of the following diagrams:

1. Activity diagrams
2. Interaction diagrams
3. Use case diagrams
4. Sequence diagrams
5. Collaboration diagrams

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

11.10.3 *Architectural Modeling*

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

11.11 *UML Basic Notations*

UML is popular for its diagrammatic notations. It is used for visualizing, specifying, constructing and documenting the components of software and non software systems. Here the Visualization is the most important part which needs to be understood well.

UML notations are the most important elements in modeling. Efficient and appropriate use of notations is very important for making a complete and meaningful model. The model is useless unless its purpose is depicted properly. Hence learning notations should be emphasized from the very beginning. Different notations are available for things and relationships. And the UML diagrams are made using the notations of things and relationships. Extensibility is another important feature which makes UML more powerful and flexible. The details of each notation is covered in following sections.

11.11.1 *Class Notation*

UML class notation is represented in figure 11.1. This diagram is divided into four parts.

1. The top section is used to name the class.
2. The second one is used to show the attributes of the class.

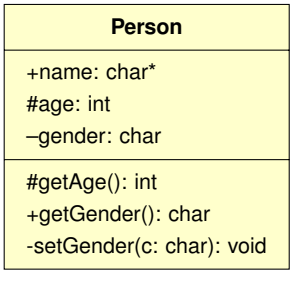


Figure 11.1: UML Class Notation

3. The third section is used to describe the operations performed by the class.

Classes are used as templates to create objects. Objects can be anything having a particular state and some responsibility. The class diagram shows a collection of classes, interfaces, associations, collaborations and constraints.

The purpose of the class diagram is to model the static view of an application. The class diagrams are the only diagrams which can be directly mapped with object oriented languages and thus widely used at the time of construction of the system. The UML diagrams like activity diagram, sequence diagram *etc.* can only give the sequence or flow of the events in an application. But class diagram is a bit different as it provides the static view of the application's components. It is the most popular UML diagram used by software developers.

Following points provides the summary of salient features of class diagrams.

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

Following points should be remembered while constructing class diagrams:

- The name of the class diagram should be meaningful enough to provide the information on the main purpose of that class in the system.
- Each element of the system 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 presence of unnecessary properties will make the diagram complicated.
- Use notes wherever required to describe those aspect of the diagram which are not obvious to the developers.
- Construct the diagram in software or on plain paper and analyze its usage and interaction before incorporating the result in the software.

The following diagram is an example of an Order Processing System of an application. It describes only a specific aspect of the entire application.

- First of all, Order and Customer are identified as the two elements of the system. Since a customer can have multiple orders, they have a one-to-many relationship.
- Since order is an abstract concept which has two concrete forms `SpecialOrder`, and `NormalOrder`; the `Order` class is kept as an abstract class.
- The specialization of classes are implemented via inheritance relationships.
- The two inherited classes have all the properties of the Order class. In addition, they have additional functions like `dispatch()` and `receive()`.

Based on above assumptions, following class diagram can be prepared.

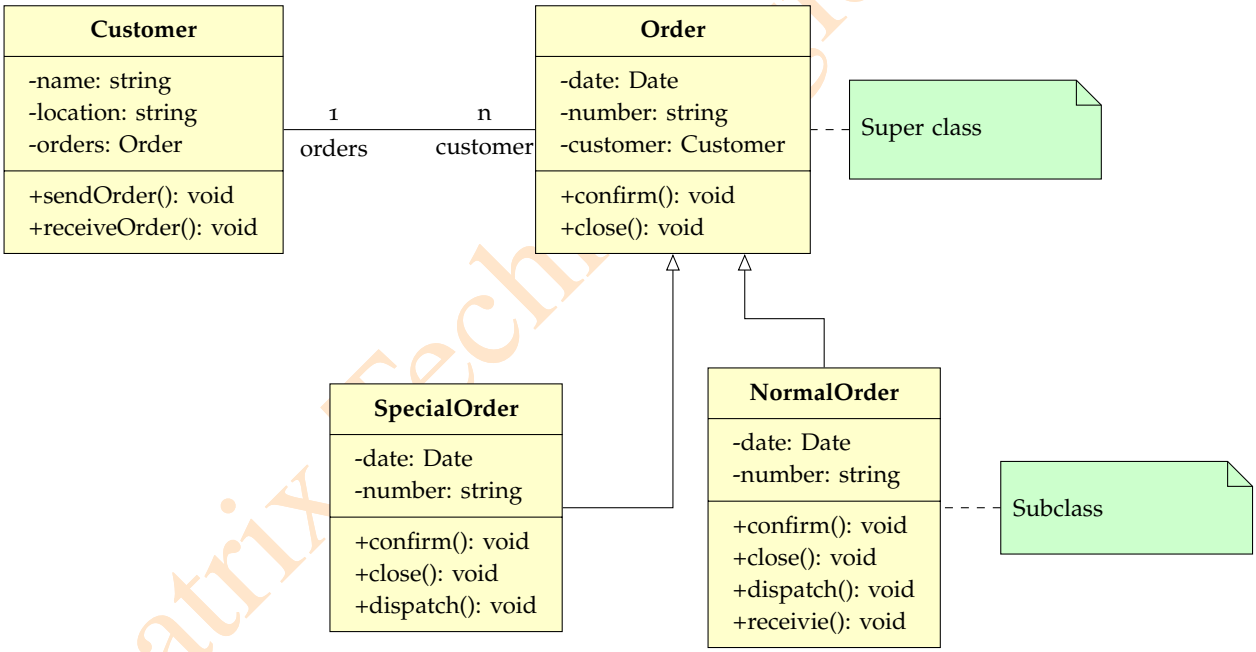


Figure 11.2: A section of class Diagram for an Order Processing System

11.11.2 *Object Notation*

Object diagrams represent an instance of a class diagram. The basic concepts are similar for both of them. Object diagrams also represent the static view of a system but this static view is a snapshot of the system at a particular moment in time. They are used to render a set of objects and their relationships. A class diagram represents an abstract model consisting of classes and their relationships. But an object diagram represents instances of those classes at a particular moment in time. The purpose of existence of object diagrams are listed below.

- for forward and reverse engineering.

- for defining object relationships and their behavior within a system at a given point of time.
- for providing static view of an interaction between two or more objects.

To capture a particular system's behavior, numbers of class diagrams are limited. But if we consider object diagrams, then we can have unlimited number of them which are unique in nature. Hence while constructing object diagrams, only those of them are considered which have major impact on the system. Since a single object diagram cannot capture all the necessary instances of objects at a given moment of time, following points should be remembered while dealing with UML diagrams of objects and classes:

1. Analyze the system and decide which instances are having important data and associations.
2. Choose only those instances which will be covered in the functionality under consideration.
3. Make some optimizations, as the number of object-instances are practically uncountable even for a mid-size system.
4. The object diagram should have a meaningful name to indicate its purpose.
5. The most important elements to be included should be identified next.
6. There should be clear associations among objects.
7. Important attributes of different elements need to be captured to include in the object diagram.
8. Wherever required, add proper notes for more clarity.

The object is represented in the same way as class in a class diagram. The only difference is in the way its name and various states of the objects are displayed.

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

This diagrams consists of following objects:

- Customer
- Order
- SpecialOrder
- NormalOrder

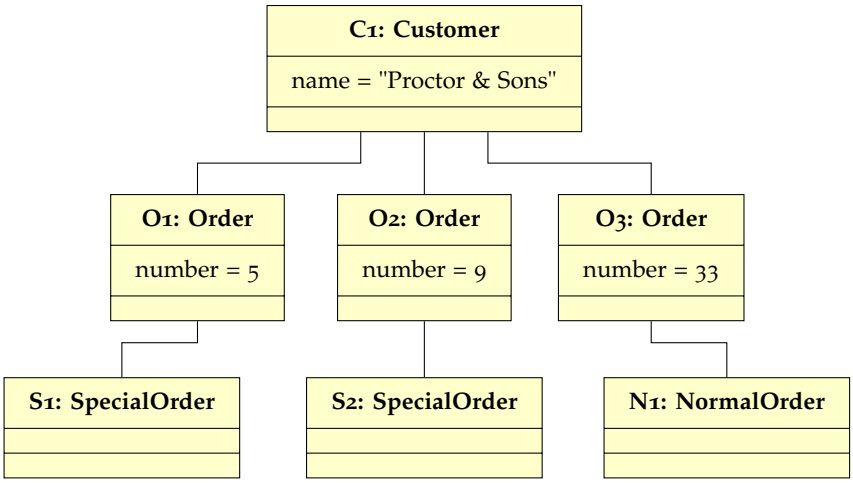


Figure 11.3: A section of Object Diagram at a given moment in time for an Order Processing System

Here the customer object(C1) is associated with three order objects, namely O1, O2 and O3. These order objects are associated with special order and normal order objects, namely S1, S2 and N1 respectively. The customer has three orders with different numbers 5, 9 and 33 at that moment of time. The customer can increase number of orders in future, and in that scenario, the object diagram will reflect those additional objects too. Each object in the object diagram has its own states represented in it. Since representing the states of objects require displaying a lot many attributes and their values, in practice, only significant attributes and their values are displayed. For example, in the diagram shown above, only the order numbers are displayed.

11.11.3 *Interface Notation*

Interface is represented by a circle as shown in figure 11.4. It has a name which is generally written below the circle.

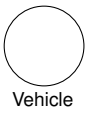


Figure 11.4: UML Interface Notation

Interface is used to describe functionality without implementation. Interface is like a template describing different functionalities expected from its implementations. Here various functions are declared sans any implementation details. When a class implements the interface, it implements all the functionalities declared in it. A class implementing an interface is shown in figure 11.5. An interface can also declare one or more constants.

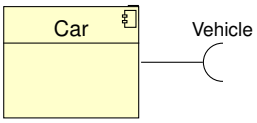


Figure 11.5: A class implementing an Interface

11.11.4 *Use Case Notation*

Use case notations consist of a set of use cases, actors, and their relationships. They represent the use case view of a system. A use case represents a particular functionality of the system. Hence use case diagrams are used to describe the relationships among the functionalities and their internal as well as external controllers. These controllers are known as actors.

Use case is represented as an eclipse with a name inside it. It may contain additional responsibilities. It is used to capture high level functionalities of a system.

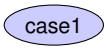


Figure 11.6: Use Case Notation

11.11.5 *Actor Notation*

An actor can be defined as some internal or external entity that interacts with the system. Actor is used in a use case diagram to describe the internal or external entities.

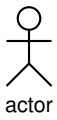


Figure 11.7: UML Actor Notation

11.11.6 *Initial State Notation*

Initial state of a process represents the start of a process. This notation is used in almost all diagrams. The usage of Initial State Notation is to show the starting point of a process in various UML diagrams.

Figure 11.8: Initial State Notation

11.11.7 *Final State Notation*

Final state is used to show the end of a process. This notation is also used in almost all diagrams to describe the end. The usage of Final State Notation is to show the termination point of a process.

Figure 11.9: Final State Notation

11.11.8 *Component Notation*

A component is part of a whole system. In UML, it is shown as with a name inside. Additional elements can be added wherever required. Component is used to represent any part of a system for which UML diagrams are made. In figure 11.10, component A encloses components B and C.

UML component diagram shows components and dependencies between them. This type of diagrams is used in Component-Based Development(CBD) to describe systems with Service-Oriented Architecture(SOA). Component-based development is based on the assumptions that previously constructed components could be reused and those components could be replaced by some other "equivalent" or "conformant" components, if required. The artifacts that implement component are intended to be capable of being deployed and re-deployed independently.[3]

[3]

Components in UML could represent logical components (*e.g.*, business components, process components), and physical components (*e.g.*, CORBA components, EJB components, COM+ and .NET components, WSDL components, *etc.*.), along with the artifacts that

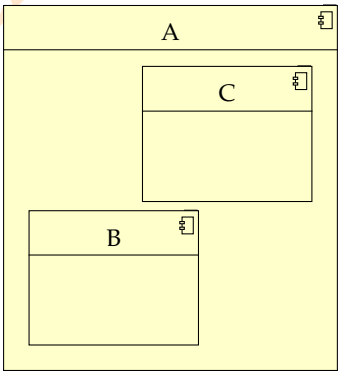


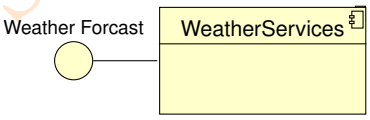
Figure 11.10: Components in Component

implement them and the nodes on which they are deployed and executed. It is anticipated that profiles based around components will be developed for specific component technologies and associated hardware and software environments.

The following nodes and edges are typically drawn in a component diagram:

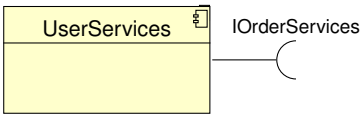
- **Component:** A component can consist of one or more other components. See figure 11.10 for an example of this case.
- **Interface:** One or more interfaces can be included within a component. An interface defines the contract of interaction of external components with the entity implementing that interface.
- **Provided Interface:** A provided interface is the one that is either realized directly by the component itself, or realized by one of the classifiers realizing component, or is provided by a public port of the component.

  In the diagram shown here, Weather Services component provides (implements) Weather Forecast interface.



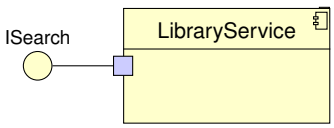
- **Required Interface:** A required interface is either designated by usage dependency from the component itself, or designated by usage dependency from one of the classifiers realizing component, or is required by a public port of the component.

  In the diagram shown here, User Services component requires IOrderServices interface.



- **Class:** Classes are templates used to construct objects. The details are provided in subsection 11.11.1.
- **Port:** Port is a property of (encapsulated) classifier that specifies an interaction point between the classifier and its environment - called *service port*, or between the classifier and its internal parts - called *behavior port*. By default, the port has public visibility.

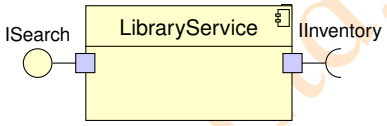
  Port is shown as a small square symbol (colored blue in the given diagrams). The name of the port is placed near the square symbol. The port symbol may be placed either overlapping the boundary of the rectangle symbol denoting that classifier or it may be shown inside the rectangle symbol. Multiplicity of the port - if any - is shown after the port name in square brackets. Both name and multiplicity of port are optional in a UML diagram.



  Service Port is used to denote classifier as provider of services with published functionality. Ports are service ports by default. Service port specifies the services a classifier provides or offers to its environment as well as the services that a classifier expects or requires from its environment.

  A provided interface may be shown using the *lollipop* notation attached to the port. A required interface may be shown using the *socket* notation attached to the port.The required interfaces of

a port characterize the requests that may be made from the classifier to its environment through this port. The provided interfaces of a port characterize requests to the classifier that other classifiers may make through this port. In the diagram shown here, `LibraryService` provides `ISearch` while requires `IInventory` provided by `Inventory` component.



Behavior Port is a port such that requests arriving at this port are sent to the behavior of the classifier owning the port, rather than being forwarded to some contained instances. If there is no behavior defined for this classifier, any communication arriving at the behavior port is lost. By default, ports are not behavior ports.

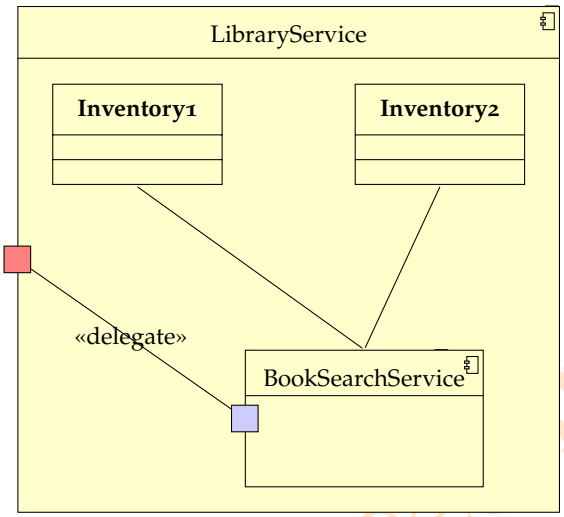


Figure 11.11: Behavior port in in a component diagram

A behavior port is rendered as a port connected by a solid line to an internal component of the classifier containing the port. The behavior of the port is defined by that internal component. In the diagram shown above, the behavior port is marked in red color.

- **Connector:** Connector is feature which specifies a link that enables communication between two or more instances. This link may be an instance of an association, or it may represent the possibility of the instances being able to communicate because their identities are known. The identities can be known due to any of the reasons given below.
  - It is passed in as parameters
  - It is held in variables or slots
  - The communicating instances are the same instance

  The link may be realized by something as simple as a pointer or by something as complex as a network connection. In contrast to associations, which specify links between any instance of the associated classifiers, connectors specify links between instances playing the connected parts only.

  A connector is drawn using the notation for association. The optional name of the connector follows the following syntax:

```
connector-name ::= [ name ] ':'  classname | name
```

Here name is the name of the connector, and classname is the name of the association that is its type. A stereotype keyword within guillemets(« ») may be placed above or in front of the connector name. A property string may be placed after or below the connector name.

- **Artifact:** An artifact is a classifier that represents some physical entity, a piece of information that is used or is produced by a software development process, or by deployment and operation of a system. Artifact is a source of a deployment to a node. A particular instance (or *copy*) of an artifact is deployed to a node instance.

  Artifacts may have properties that represent features of the artifact, and operations that can be performed on its instances. Artifacts have fileName attribute - a concrete name that is used to refer to the artifact in a physical context - *e.g.* file name or URI. Artifact could have nested artifacts too.

  Some real life examples of artifacts are:

  - text document
  - source file
  - script
  - binary executable file
  - archive file
  - table in a database

  The UML Standard Profile defines some standard stereotypes that apply to artifacts. «file» is one of them. It represents a physical file in the context of the system developed.

  Following are some standard stereotypes which are subclasses of «file»:

«document»	A generic file that is not a «source» file or «executable».
«source»	A source file that can be compiled into an executable file.
«library»	A static or dynamic library file.
«executable»	A program file that can be executed on a computer system.
«script»	A script file that can be interpreted by a computer system.

  An artifact is presented using an ordinary class rectangle with the keyword «artifact» or similar stereotypes. Examples in UML specification also show document icon in upper right corner.

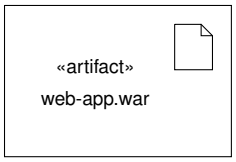


Figure 11.12: Artifact notation

Dependency between artifacts is notated in the same way as general dependency, i.e. as a general dashed line with an open arrow head directed from client artifact to supplier artifact.

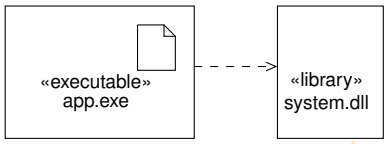


Figure 11.13: Dependency amongst artifacts

- **Component Realization:** Component Realization is specialized realization dependency used to (optionally) define classifiers that realize the contract offered by a component in terms of its provided interfaces and required interfaces.

  A componentâĂŹs behavior may typically be realized (or implemented) by a number of Classifiers. In effect, it forms an abstraction for a collection of model elements. In that case, a component owns a set of Component Realization Dependencies to these Classifiers. In effect, it forms an abstraction for a collection of model elements. In that case, a component owns a set of Realization Dependencies to these Classifiers.

  A component realization is notated in the same way as the realization dependency, i.e., as a general dashed line from implementing classifiers to realized component with hollow triangle as an arrowhead.

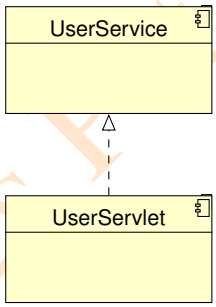


Figure 11.14: Realization of a component

- **Usage:** A usage is a dependency relationship in which one element (client) requires another element (or set of elements) (supplier) for its full implementation or operation.

  The usage dependency does not specify how the client uses the supplier other than the fact that the supplier is used by the definition or implementation of the client. For example, it could mean that some method(s) within a (client) class uses objects (e.g. parameters) of the another (supplier) class.

  A usage dependency is shown as a dependency with a «use» keyword attached to it.

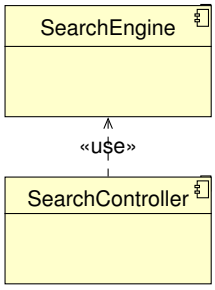


Figure 11.15: Usage of a component

A component is a structured class representing a modular part of a system with encapsulated content and whose manifestation is replaceable within its environment. A component has its behavior defined in terms of provided interfaces and required interfaces (potentially exposed via ports).

Component serves as a type whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics). One component may therefore be substituted by another only if the two are type conformant.

There are several standard UML stereotypes that apply to components:

- **«subsystem»:** Subsystem is a component representing unit of hierarchical decomposition for large systems. It is used to model large scale components. Definitions of subsystems may vary among different domains and software methods. It is expected that domain and method profiles will specialize this element.

  Usually a subsystem is indirectly instantiated. A subsystem may have specification and realization elements.

- **«process»:** UML Standard Profile defines process as a transaction based component.
- **«service»:** Service is a stateless, functional component.
- **«specification»:** Specification is a classifier that specifies a domain of objects without defining the physical implementation of any of them. For example, a component stereotyped by «specification» will only have provided and required interfaces, and is not intended to have any realizing classifiers as part of its definition. This differs from » type«because a «type» can have features such as attributes and methods that are useful to analysts modeling systems.

  «Specification» and «realization» are used to model components with distinct specification and realization definitions, where one specification may have multiple realizations.
- **«realization»:** Realization is a classifier that specifies a domain of objects and that also defines the physical implementation of them.

  For example, a component stereotyped by «realization» will only have realizing classifiers that implement behavior specified by a separate «specification» component. This differs from «implementation class» because an «implementation class» is a realization of a class that can have features such as attributes and methods that are useful to system designers.
- **«implement»:** Implement is a component definition that is not intended to have a specification itself. Rather, it is an implementation for a separate «specification» to which it has a dependency. It seems to duplicate «realization».

While drawing component diagrams, following points should be remembered:

- Always meaningful names should be used to identify components.
- A mental model of the layout of all sub-systems should be prepared for the system under consideration.
- To clarify important points, notes should be used.

Following is a component diagram for order processing system. The diagram shows the components used to generate orders and components representing the customer database.

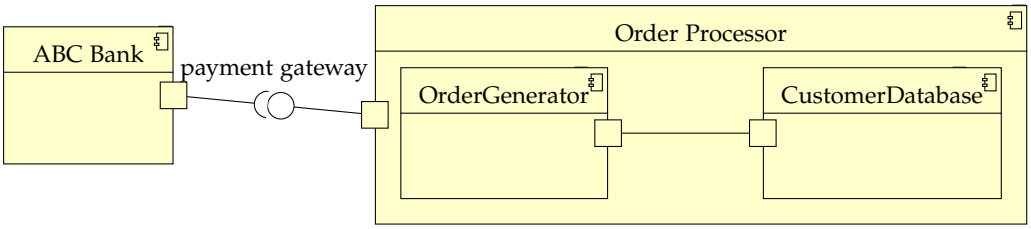


Figure 11.16: Components of Order Processing System

11.11.9 *Interaction Notation*

Interaction is the exchange of messages between two UML components. Figure 11.19 shows different notations used in a particular interaction diagram. It is part of behavioral things. Interaction diagrams can be of many types:

1. Communication Diagrams
2. Sequence Diagrams
3. Activity Diagrams
4. Interaction Overview Diagrams
5. Timing Diagrams

11.11.10 *Communication Notation*

Communication Diagram, which was called Collaboration Diagram in UML 1.x, is another form of interaction diagram. It represents the structural organization of a system and the messages sent/received. Structural organization consists of objects and links. The purpose of communication diagram is similar to sequence diagram. But the specific purpose of it is to visualize the organization of objects and their interactions.

Communication diagram corresponds (*i.e.* could be converted to/from or replaced by) to simple sequence diagram without structuring mechanisms such as Interaction Uses and Combined Fragments. It is also assumed that message overtaking (*i.e.* the order of the receptions are different from the order of sending of a given set of messages) will not take place or is irrelevant.

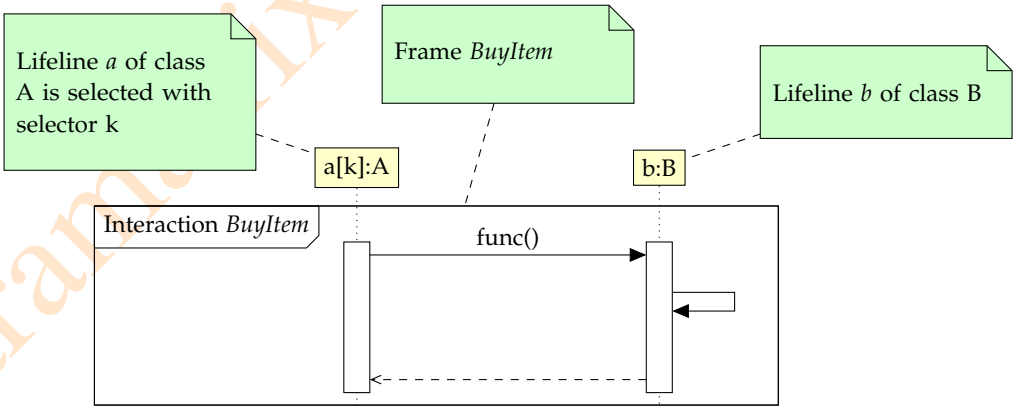


Figure 11.17: Components of Communication Diagram

The following nodes and edges are drawn in a UML communication diagrams:

1. **Frame:** Communication diagrams could be shown within a rectangular frame with the name in a compartment in the upper left corner. There is no specific long form name for communication diagrams heading types. The long form name interaction (used for interaction diagrams in general) could be used.

2. **Lifeline:** Lifeline is a specialization of named element which represents an individual participant in the interaction. While parts and structural features may have multiplicity greater than 1, lifelines represent only one interacting entity. A Lifeline is shown as a rectangle (corresponding to the *head* in sequence diagrams). Lifeline in sequence diagrams does have *tail* representing the line of life whereas *lifeline* in communication diagram has no line, just *head*. The lifeline head has a shape that is based on the classifier for the part that this lifeline represents. Usually the head is a white rectangle containing name of the class after colon.

   If the referenced connectable element is multivalued (*i.e.* has a multiplicity > 1), then the lifeline may have an expression (selector) that specifies which particular part is represented by this lifeline. If the selector is omitted, this means that an arbitrary representative of the multivalued connectable element is chosen.

3. **Message:** Message in communication diagram is usually shown as a line with sequence expression and arrow above that line. The arrow indicates direction of the communication. Sometimes the line itself contains the arrow indicating the direction of message delivery. In the example shown in figure 11.17, the message *func()* is sent to *b* by *a*. Here the sequence expression is *func()*.

11.11.11 *Sequence Notation*

Sequence diagram describes an interaction by focusing on the sequence of messages that are exchanged, along with their corresponding occurrence specifications on the lifelines. Such diagrams deal with sequence of messages flows. The messages flow from one object to another. They usually give us a glimpse of interaction between those components. Such interactions are very important from implementation and execution perspectives. In short, sequence diagrams are made to visualize the sequence of calls in a system to perform a specific functionality. In the diagram shown below, the functionality of comment submission in an application is captured.

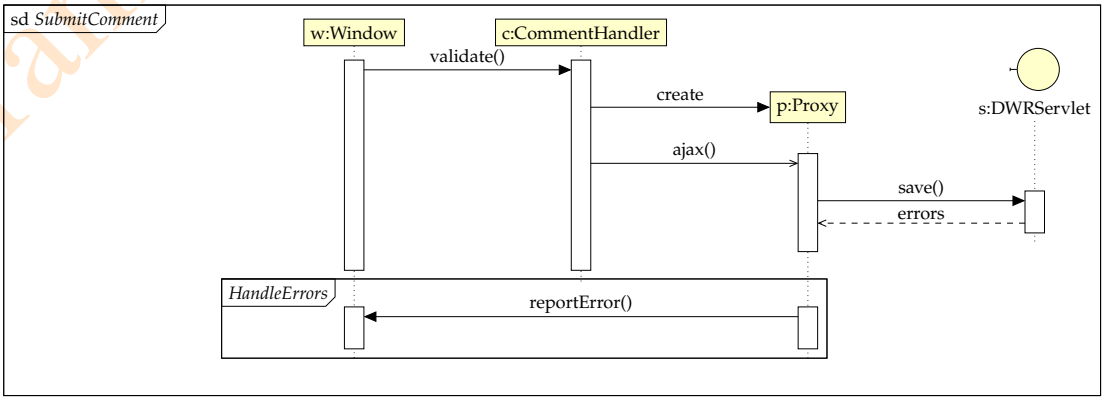


Figure 11.18: Sequence Diagram

In the diagram shown in 11.18, a comment submission sequence is shown. From the application window, a comment submission

is initiated. After validation, it is supplied to comment handling layer, which creates a proxy object. The comment is handed over to this proxy object asynchronously (via ajax()) which is supplied to DWRServlet in the last layer of function call. If no error is generated, then the comment is submitted. In case of error, proxy reports it to the application window using a synchronous call with no returns. This diagram shows the *fire and forget* methodology of comment submission. Unless error is reported in the window, comment is assumed to be submitted successfully.

Following figure is another example of a sequence diagram which is an interaction diagram capturing the interaction between multiple components. Actor A invokes method opA() on database b, which in turn invokes opB() on c. Object c is chosen from a set of similar objects based on a selection mechanism. That's why multiple instances of c is shown in the diagram. Object c invokes obC() on object d which returns 2. The values returned by each methods opB() and opA() are 1 and 0 respectively.

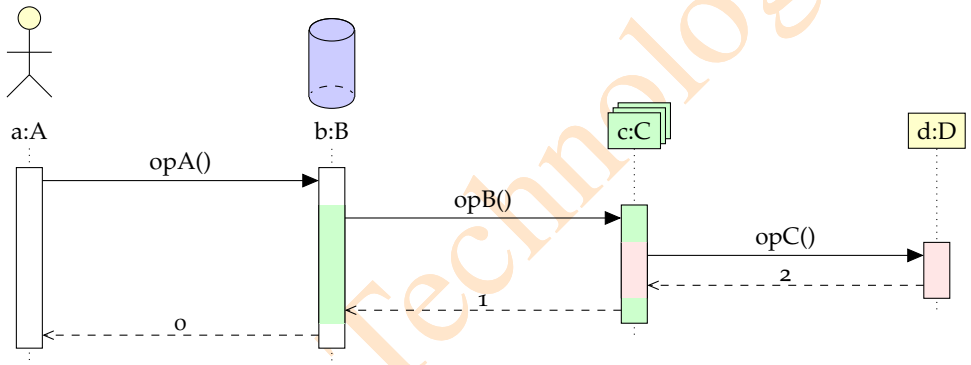


Figure 11.19: Interaction Diagram

Usually the dynamic nature of any system is very difficult to capture in a digram. To ease this task, UML has provided features to capture the dynamics of a system from different angles. Sequence diagrams and collaboration diagrams are isomorphic so they can be converted from one another without losing any information. This is also true for state machine and activity diagrams.

11.11.12 *Activity Notation*

Activity diagram is UML behavior diagram which shows flow of control or object flow with emphasis on the sequence and conditions of the flow. The actions coordinated by activity models can be initiated because other actions finish executing, because objects and data become available, or because some events external to the flow occur.

The activity diagram consists of activities and links. The flow can be sequential, concurrent or branched. Activities are functions of a system. Many activity diagrams are required to capture the entire flow of activities in a system. Activity diagrams are used to visualize the flow of controls in a system. This indicates how the system will work during phase of its execution.

An example of activity diagram is shown in figure 11.20 which describes Single Sign-On (SSO) to Google Apps. To interact with partner companies, Google uses single sign-on based on OASIS SAML 2.0 protocol. Google acts as service provider with services such as Gmail or Start Pages. Partner companies act as identity providers and control user names, passwords, and other information used to identify, authenticate and authorize users for web applications that Google hosts. Each partner provides Google with the URL of its SSO service as well as the public key that Google will use to verify SAML responses.

When a user attempts to use some hosted Google application, such as Gmail, Google generates an SAML authentication request and sends redirect request back to the user's browser. Redirect points to the specific identity provider. SAML authentication request contains the encoded URL of the Google application that the user accessed.

The partner identity provider authenticates the user by either asking for valid login credentials or by checking for its own valid authentication cookies. The partner generates an SAML response and digitally signes it. The response is forwarded to Google's Assertion Consumer Service (ACS).

Google's ACS verifies the SAML response using the partner's public key. If the response is valid and user identity was confirmed by identity provider, ACS redirects the user to the destination URL. Otherwise user will see error message.

11.11.13 *State Machine Notation*

Any real time system is expected to be reacted by some kind of internal/external events. These events are responsible for state changes of the system. State machine diagram is used to represent this event driven state changes of a system. It basically describes the state change of a class, interface, system *etc.* Thus state machine diagram is used to visualize the reaction of a system by internal as well as external factors.

State machine describes the different states of a component in its life cycle. The state can be active, idle or any other depending upon the situation. The notations in a UML stated diagram are shown in the figure 11.21.

11.11.14 *Package Notation*

UML models are organized by grouping them together. Package notation groups individual UML elements together. A package is rendered as a tabbed folder - a rectangle with a small tab attached to the left side of the top of the rectangle. If the members of the package are not shown inside the package rectangle, then the name of the package should be placed inside it. In the image shown here (figure 11.22), a package can be used to wrap the components of a system.

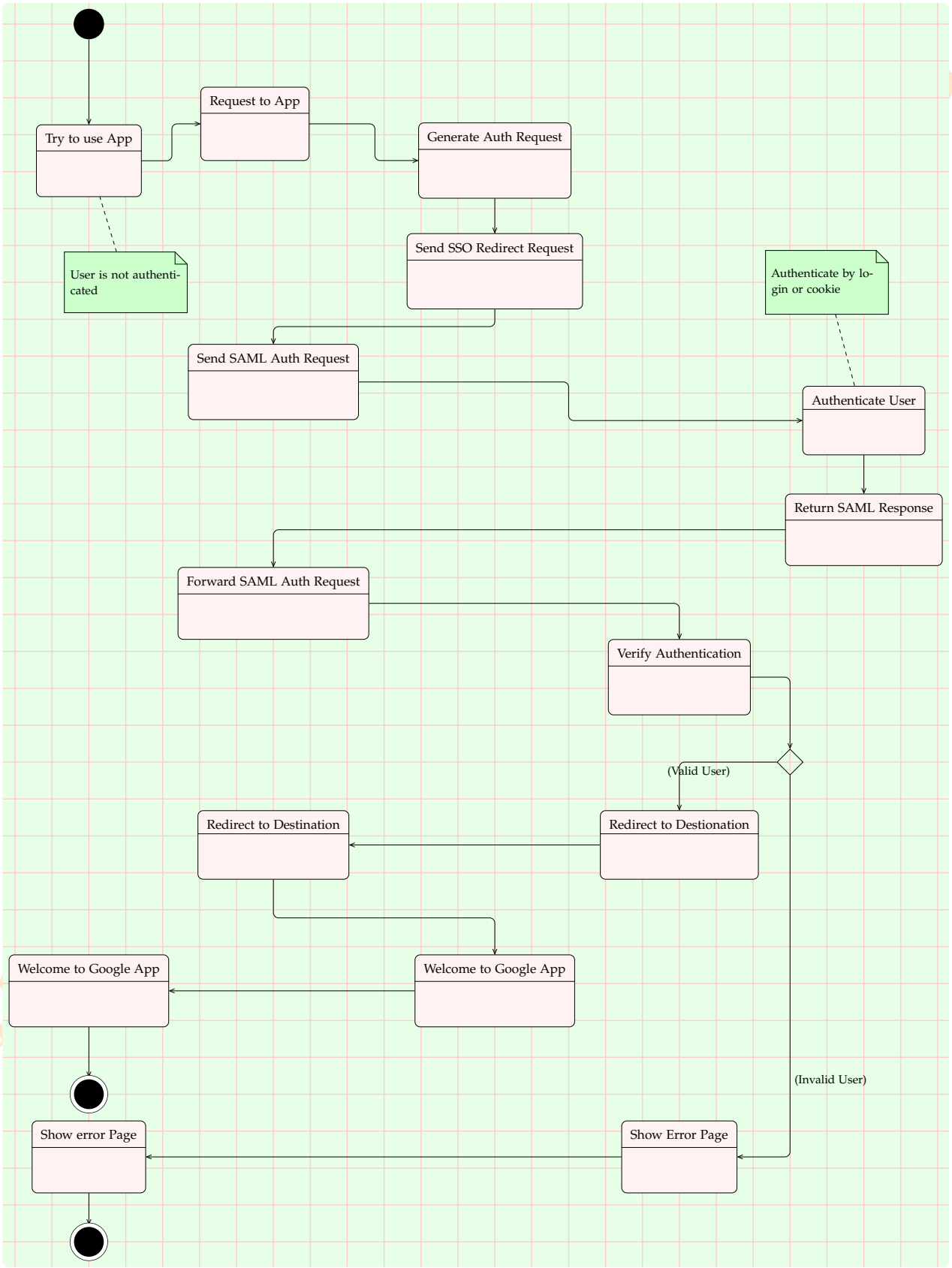


Figure 11.20: Google Apps Sign-in Activities

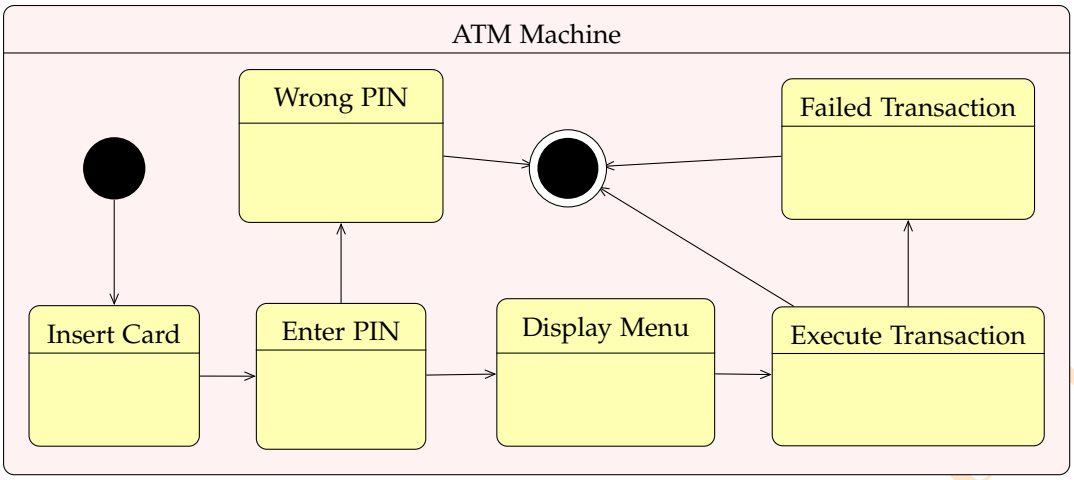


Figure 11.21: State Diagram of an ATM Machine

Owned members of a package should all be packageable elements. If a package is removed from a model, so are all the elements owned by the package. In the diagram shown in figure 11.22, if package P is removed, its content A, B, and C are also removed. By the way, package by itself is a packageable element, so any package could be also a member of other package.

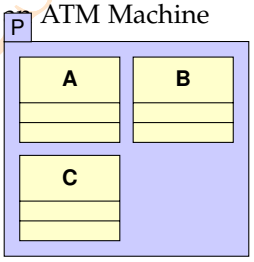


Figure 11.22: Package with Components

11.11.15 *Note Notation*

Annotations of any component are supported in UML via Note notations. Explanations of various components and their functionalities are possible included by creating appropriate notes as shown in the figure 11.23.

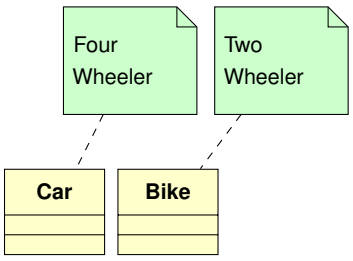


Figure 11.23: Annotations via Notes

11.11.16 *Dependency Notation*

Dependency is an important aspect in UML elements. It describes the dependent elements and the direction of dependency. Dependency is represented by a dotted arrow as shown below. The arrow head represents the independent element and the other end the dependent element.

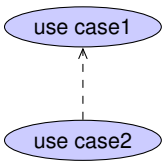


Figure 11.24: Dependency Notation

11.11.17 *Association Notation*

Association describes how the elements in a UML diagram are associated. In simple words, it describes how many elements are taking part in an interaction. Association is represented by a dotted line with (without) arrows on both sides. The two ends represent two associated elements as shown below. The multiplicity is also mentioned at the ends (1, * *etc.*) to show how many objects are associated in that particular association.

The relationship between three classes are shown below. An Employee object is associated with Department as well as Role objects. An Employee can play multiple roles, but is placed in one department

only. The association between these classes are shown in the figure shown below.

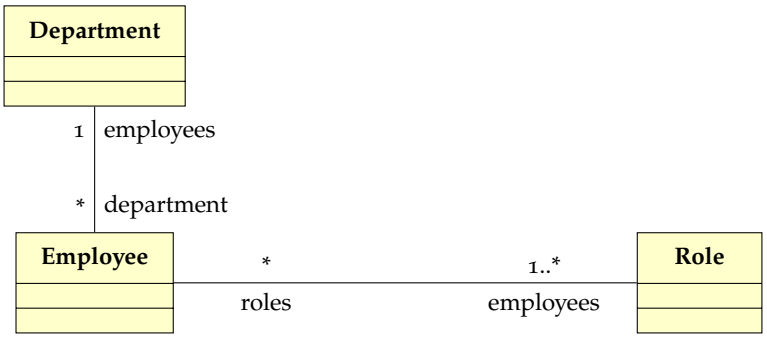


Figure 11.25: Association Notation

11.11.18 *Generalization Notation*

Generalization describes the inheritance relationship of the object oriented world. It is also known as parent-child relationship. Generalization is represented by an arrow with hollow arrow head as shown in figure 11.26. One end of the arrow represents the parent element and the other end the child element.

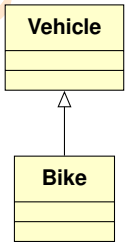


Figure 11.26: Generalization Notation

11.11.19 *Extensibility Notation*

UML has following mechanisms to provide extensibility to its features.

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

The stereotype of a relation is a keyword contained between a pair of guillemot symbols « and ». An example is shown in figure 11.27.

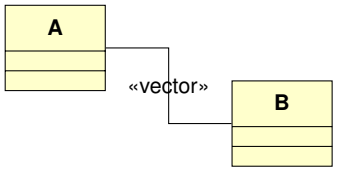


Figure 11.27: Stereotype Notation

A constraint notation is the text describing certain constraints involved in the entities being discussed, which should be enclosed within a pair of angular braces. Example is

```
{Age should be less than 40 years}
```

A tagged value is the value of certain entity enclosed within a pair of round braces. Example is

```
(version=1.0)
```

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

11.12 Conclusion

It is important to distinguish between the UML model and the set of diagrams of a system. A diagram is a partial graphic representation of a system's model. The model also contains documentation that drives the model elements and diagrams (such as written use cases).

UML diagrams represent two different views of a system model:

1. Static (or structural) view which emphasizes the static structure of the system using objects, attributes, operations and relationships. The structural view includes class diagrams and composite structure diagrams.

2. Dynamic (or behavioral) view which emphasizes the dynamic behavior of the system by showing collaborations among objects and changes to the internal states of objects. This view includes sequence diagrams, activity diagrams and state machine diagrams.

UML is not a development method by itself; however, it was designed to be compatible with the leading object-oriented software development methods of its time like OMT, Booch method, Objectory *etc.* Since UML has evolved over time, some of these methods have been recast to take advantage of the new notations (for example OMT), and new methods have been created based on UML, such as IBM Rational Unified Process (RUP). Others include Abstraction Method and Dynamic Systems Development Method. Studying these methods give us more insights on general software modeling process.

★ ★ ★ ★ ★

ïż£