VISHNU INSTITUTE OF TECHNOLOGY
Vishnupur, BHIMAVARAM – 534 202

LABORATORY MANUAL
IV B.Tech I Sem CSE

Unified Modeling Language & Design Patterns Lab

DEPARTMENT OF CSE

OUR MISSION

LEARN TO EXCEL

Regd.No : _____

SUMMARY OF WORK-DONE						
Exercise No.	**Date**	**Marks**				**Signature**
		Lab (5)	**Record (5)**	**Viva**	**Total(15)**	
1						
2						
3						
4						
5						
6						
7						
8						
9						
10						
11						
12						
13						
14						
15						
16						
17						
18						
19						
20						
21						
22						
23						
24						
25						

Total Marks Awarded: _____

Signature of the Teacher

Signature of HOD

List of Experiments

S.No	Experiment
1	To create a UML diagram of ATM APPLICATION
2	To create a UML diagram of LIBRARY MANAGEMENT SYSTEM
3	To create a UML diagram of ONLINE BOOK SHOP
4	To create a UML diagram of RAILWAY RESERVATION SYSTEM
5	To create a UML diagram of BANKING SYSTEM
6	To design a Document Editor
7	Using UML, design Abstract Factory design pattern
8	Using UML, design Builder design pattern
9	Using UML, design Façade design pattern
10	Using UML, design Bridge design pattern
11	Using UML, design Decorator design pattern
12	User gives a print command from a word document. Design to represent this chain of responsibility design pattern

UML

What is UML?

"The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems".— OMG UML Specification

"UML is a graphical notation for modeling various aspects of software systems." — whm

Why use UML?

Two questions, really:

1) Why use a graphical notation of any sort?
Facilitates construction of models that in turn can be used to:
Reason about system behavior.
Present proposed designs to others.
Document key elements of design for future understanding.

2) Which graphical notation should be used?
UML has become the de-facto standard for modeling object oriented systems.
UML is extensible and method-independent.
UML is not perfect, but it's good enough.

The Origins of UML

Object-oriented programming reached the mainstream of programming in the late 1980's and early 1990's. The rise in popularity of object-oriented programming was accompanied by a profusion of object-oriented analysis and design methods, each with its own graphical notation.

Three OOA/D gurus, and their methods, rose to prominence Grady Booch — The Booch Method, James Rumbaugh, et al. — Object Modeling Technique, Ivar Jacsobson — Objectory In 1994, Booch and Rumbaugh, then both at Rational, started working on a unification of their methods. A first draft of their Unified Method was released in October 1995. In 1996, (+/-) Jacobson joined Booch and Rumbaugh at Rational; the name UML was coined. In 1997 the Object Management Group (OMG) accepted UML as an open and industry standard visual modeling language for object oriented systems. Current version of UML is 2.0.

UML Diagram Types

There are several types of UML diagrams:
Use-case Diagram
Shows actors, use-cases, and the relationships between them.
Class Diagram
Shows relationships between classes and pertinent information about classes themselves.
Object Diagram
Shows a configuration of objects at an instant in time.
Interaction Diagrams

Show an interaction between a group of collaborating objects.
Two types: Collaboration diagram and sequence diagram

Package Diagram

Shows system structure at the library/package level.

State Diagram

Describes behavior of instances of a class in terms of states, stimuli, and transitions.

Activity Diagram

Very similar to a flowchart—shows actions and decision points, but with the ability to accommodate concurrency.

Deployment Diagram

Shows configuration of hardware and software in a distributed system.

UML Modeling Types

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

Structural modeling:

Structural modeling captures the static features of a system. They consist of the followings:

- Classes diagrams
- Objects diagrams
- Deployment diagrams
- Package diagrams
- Component diagrams

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

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

Behavioral Modeling

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

- Activity diagrams
- Interaction diagrams
- Use case diagrams

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

Architectural Modeling

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

UML Basic Notations

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

UML notations are the most important elements in modelling. 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.

So 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.

Structural Things

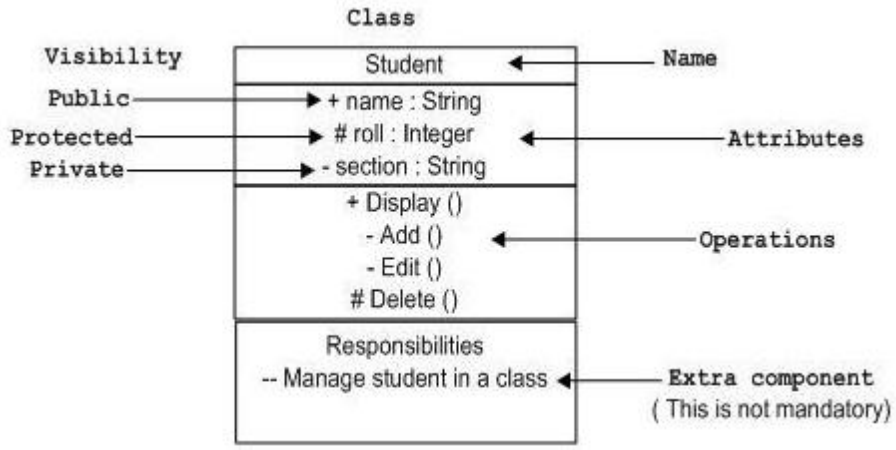
Graphical notations used in structural things are the most widely used in UML. These are considered as the nouns of UML models. Following are the list of structural things.

- Classes
- Interface
- Collaboration
- Use case
- Active classes
- Components
- Nodes

Class Notation:

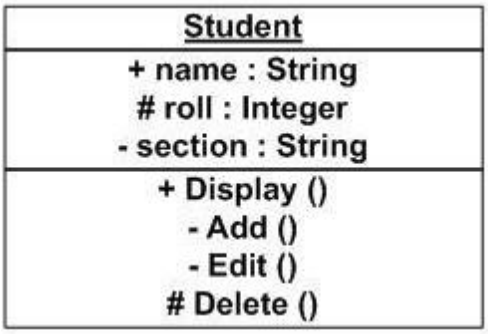
UML class is represented by the diagram shown below. The diagram is divided into four parts.

- The top section is used to name the class.
- The second one is used to show the attributes of the class.
- The third section is used to describe the operations performed by the class.
- The fourth section is optional to show any additional components.



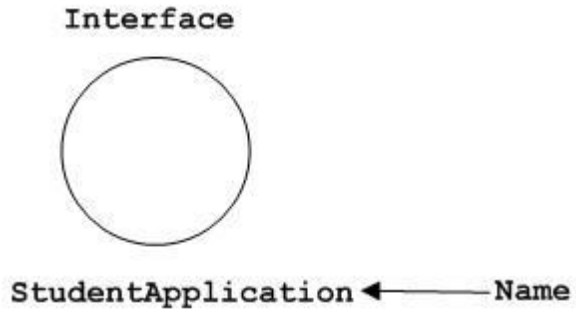
Classes are used to represent objects. Objects can be anything having properties and responsibility.

Object Notation:
The object is represented in the same way as the class. The only difference is the name which is underlined as shown below:



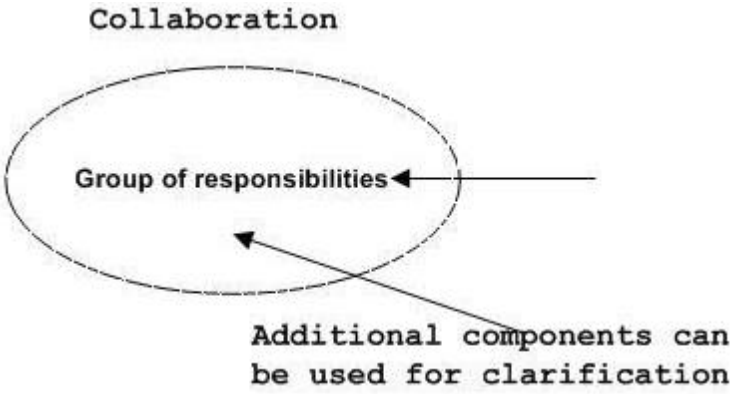
As object is the actual implementation of a class which is known as the instance of a class. So it has the same usage as the class.

Interface Notation:
Interface is represented by a circle as shown below. It has a name which is generally written below the circle.



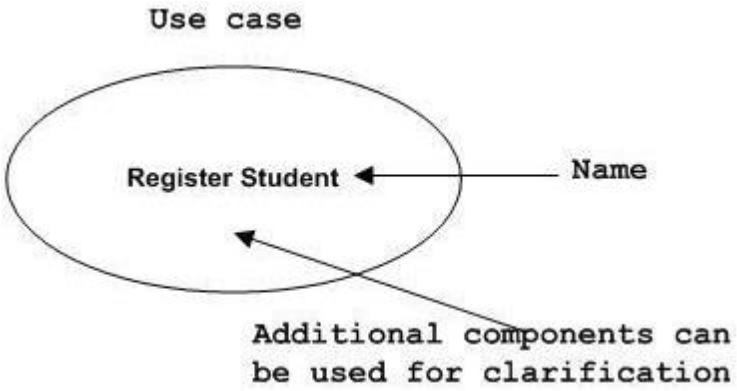
Interface is used to describe functionality without implementation. Interface is the just like a template where you define different functions not the implementation. When a class implements the interface it also implements the functionality as per the requirement.

Collaboration Notation:
Collaboration is represented by a dotted eclipse as shown below. It has a name written inside the eclipse.



Collaboration represents responsibilities. Generally responsibilities are in a group.

Use case Notation:
Use case is represented as an eclipse with a name inside it. It may contain additional responsibilities.

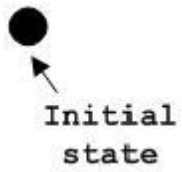


Use case is used to capture high level functionalities of a system.

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.

Initial State Notation:
Initial state is defined show the start of a process. This notation is used in almost all diagrams.

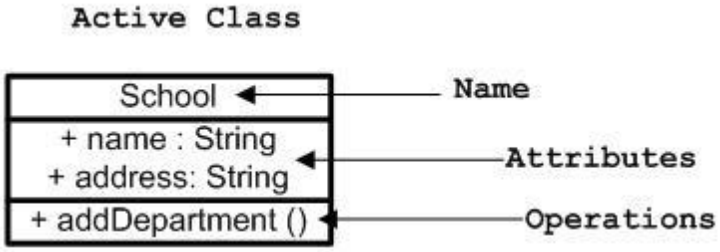


The usage of Initial State Notation is to show the starting point of a process.

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.

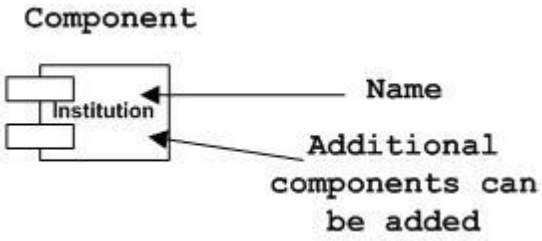
The usage of Final State Notation is to show the termination point of a process.

Active class Notation:
Active class looks similar to a class with a solid border. Active class is generally used to describe concurrent behavior of a system.



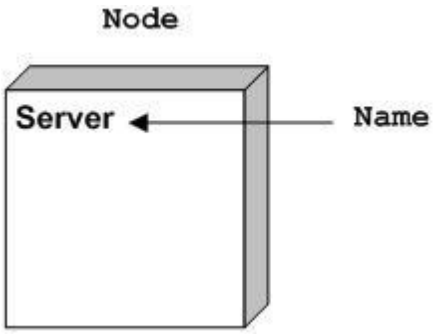
Active class is used to represent concurrency in a system.

Component Notation:
A component in UML is shown as below 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.

Node Notation:
A node in UML is represented by a square box as shown below with a name. A node represents a physical component of the system.



Node is used to represent physical part of a system like server, network etc.

Behavioural Things:

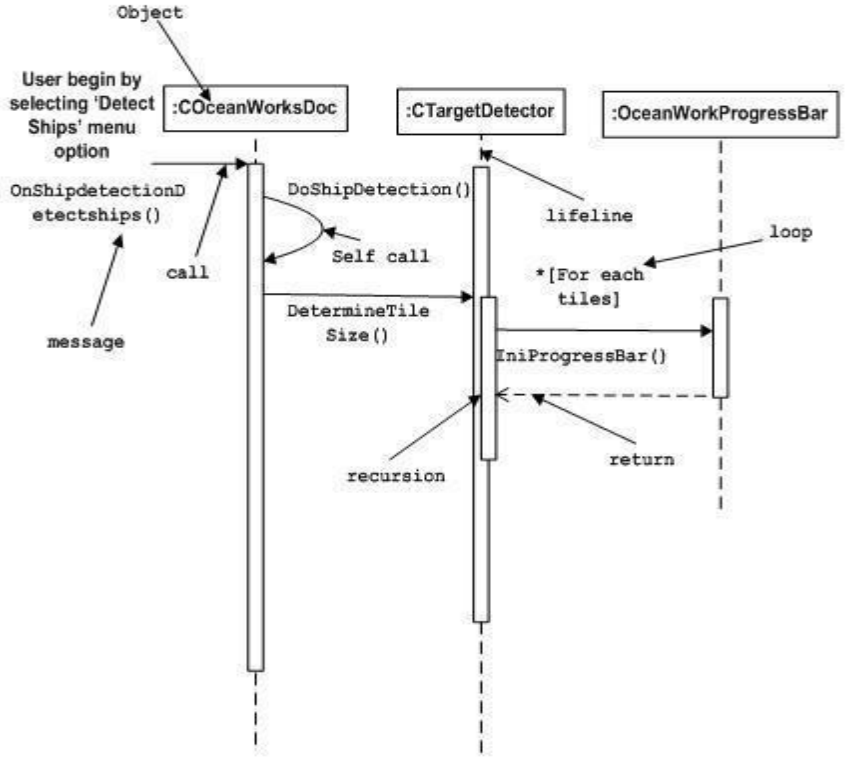
Dynamic parts are one of the most important elements in UML. UML has a set of powerful features to represent the dynamic part of software and non software systems. These features include interactions and state machines.

Interactions can be of two types:

- Sequential (Represented by sequence diagram)
- Collaborative (Represented by collaboration diagram)

Interaction Notation:

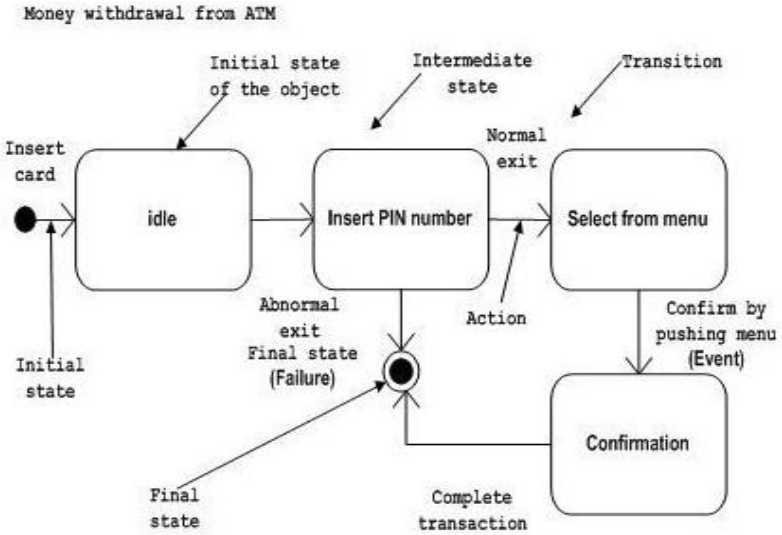
Interaction is basically message exchange between two UML components. The following diagram represents different notations used in an interaction.



Interaction is used to represent communication among the components of a system.

State machine Notation:

State machine describes the different states of a component in its life cycle. The notations are described in the following diagram.



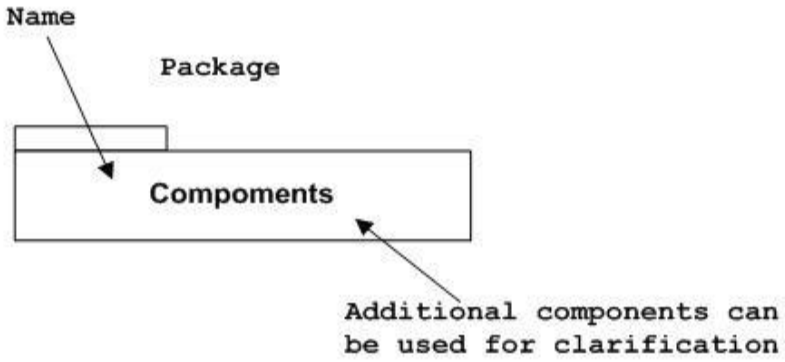
State machine is used to describe different states of a system component. The state can be active, idle or any other depending upon the situation.

Grouping Things:

Organizing the UML models are one of the most important aspects of the design. In UML there is only one element available for grouping and that is package.

Package Notation:

Package notation is shown below and this is used to wrap the components of a system.

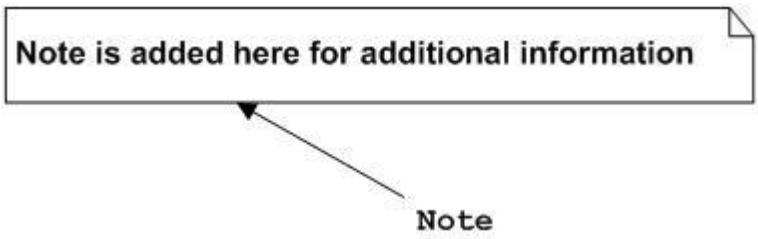


Annotational Things:

In any diagram explanation of different elements and their functionalities are very important. So UML has notes notation to support this requirement.

Note Notation:

This notation is shown below and they are used to provide necessary information of a system.

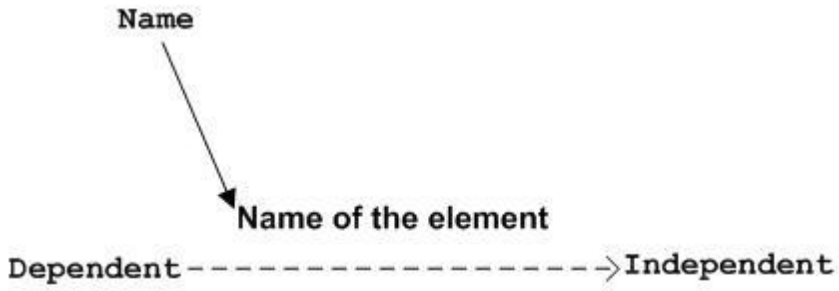


Relationships

A model is not complete unless the relationships between elements are described properly. The Relationship gives a proper meaning to an UML model. Following are the different types of relationships available in UML.

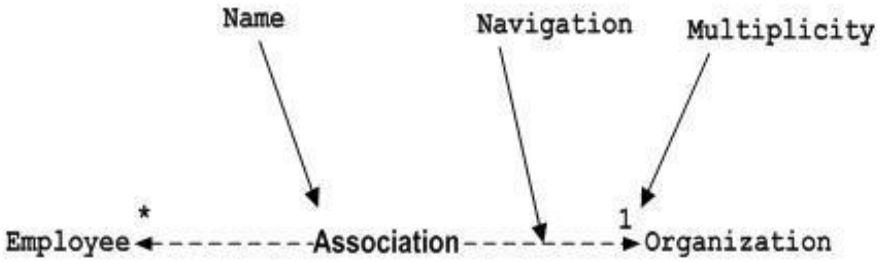
- Dependency
- Association
- Generalization
- Extensibility

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.



Dependency is used to represent dependency between two elements of a system.

Association Notation:
Association describes how the elements in an UML diagram are associated. In simple word 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.



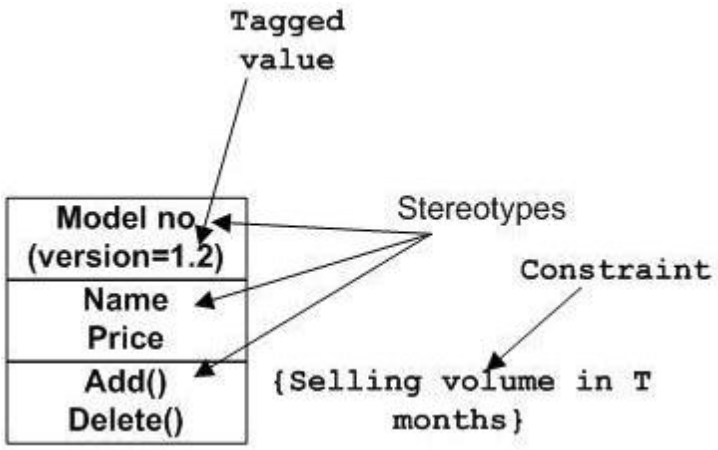
Association is used to represent the relationship between two elements of a system.

Generalization Notation:
Generalization describes the inheritance relationship of the object oriented world. It is parent and child relationship.
Generalization is represented by an arrow with hollow arrow head as shown below. One end represents the parent element and the other end child element.

Generalization is used to describe parent-child relationship of two elements of a system.

Extensibility Notation:
All the languages (programming or modelling) have some mechanism to extend its capabilities like syntax, semantics etc. UML is also having the following mechanisms to provide extensibility features.

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



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

UML Class Diagram

The class diagram is a static diagram. It represents the static view of an application. Class diagram is not only used for visualizing, describing and documenting different aspects of a system but also for constructing executable code of the software application.

The class diagram describes the attributes and operations of a class and also the constraints imposed on the system. The class diagrams are widely used in the modelling of object oriented systems because they are the only UML diagrams which can be mapped directly with object oriented languages.

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

Purpose:

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.

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

So 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 Class Diagram?

Class diagrams are the most popular UML diagrams used for construction of software applications. So it is very important to learn the drawing procedure of class diagram.

Class diagrams have 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. So a collection of class diagrams represent the whole system.

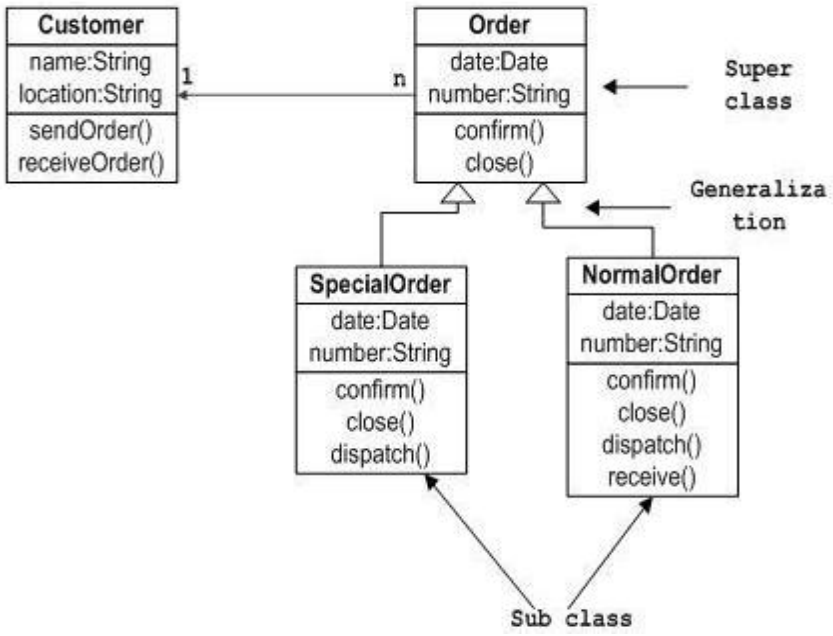
The following points should be remembered while drawing a class diagram:

- 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. Because unnecessary properties will make the diagram complicated.
- Use notes when ever required to describe some aspect of the diagram. Because 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 rework as many times as possible to make it correct.

Now the following diagram is an example of an Order System of an application. So it describes a particular aspect of the entire application.

- First of all Order and Customer are identified as the two elements of the system and they have a one to many relationship because a customer can have multiple orders.
- We would keep Order class is an abstract class and it has two concrete classes (inheritance relationship) SpecialOrder and NormalOrder.
- The two inherited classes have all the properties as the Order class. In addition they have additional functions like dispatch () and receive ().

So the following class diagram has been drawn considering all the points mentioned above:



UML Object Diagram

Object diagrams are derived from class diagrams so object diagrams are dependent upon class diagrams.

Object diagrams represent an instance of a class diagram. The basic concepts are similar for class diagrams and object diagrams. Object diagrams also represent the static view of a system but this static view is a snapshot of the system at a particular moment.

Object diagrams are used to render a set of objects and their relationships as an instance.

Purpose:

The purpose of a diagram should be understood clearly to implement it practically. The purposes of object diagrams are similar to class diagrams.

The difference is that a class diagram represents an abstract model consists of classes and their relationships. But an object diagram represents an instance at a particular moment which is concrete in nature.

It means the object diagram is more close to the actual system behaviour. The purpose is to capture the static view of a system at a particular moment.
So the purpose of the object diagram can be summarized as:

- Forward and reverse engineering.
- Object relationships of a system .
- Static view of an interaction.
- Understand object behaviour and their relationship from practical perspective.

How to draw Object Diagram?

We have already discussed that an object diagram is an instance of a class diagram. It implies that an object diagram consists of instances of things used in a class diagram.

So both diagrams are made of same basic elements but in different form. In class diagram elements are in abstract form to represent the blue print and in object diagram the elements are in concrete form to represent the real world object.

To capture a particular system, numbers of class diagrams are limited. But if we consider object diagrams then we can have unlimited number of instances which are unique in nature. So only those instances are considered which are having impact on the system.

From the above discussion it is clear that a single object diagram cannot capture all the necessary instances or rather cannot specify all objects of a system. So the solution is:

- First, analyze the system and decide which instances are having important data and association.
- Second, consider only those instances which will cover the functionality.
- Third, make some optimization as the numbers of instances are unlimited.

Before drawing an object diagrams the following things should be remembered and understood clearly:

- Object diagrams are consist of objects.
- The link in object diagram is used to connect objects.
- Objects and links are the two elements used to construct an object diagram.

Now after this the following things are to be decided before starting the construction of the diagram:

- The object diagram should have a meaningful name to indicate its purpose.
- The most important elements are to be identified.
- The association among objects should be clarified.
- Values of different elements need to be captured to include in the object diagram.
- Add proper notes at points where more clarity is required.

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

- Customer
- Order
- SpecialOrder
- NormalOrder

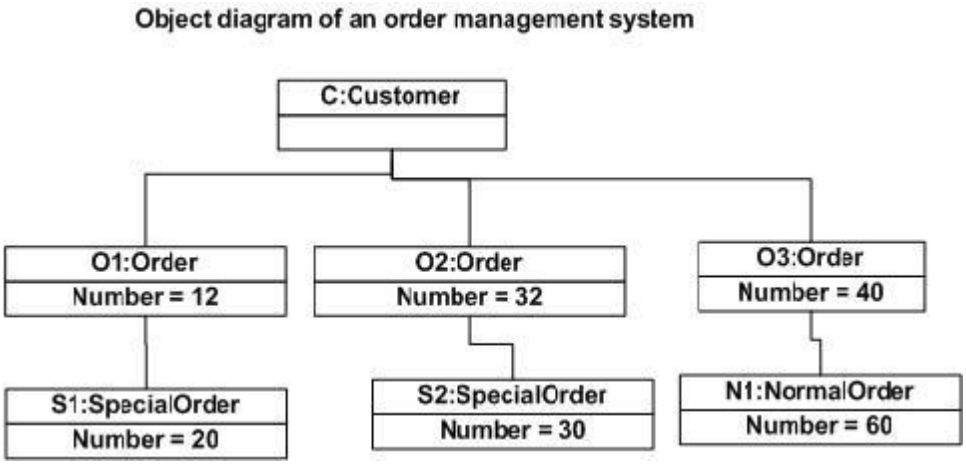
Now the customer object (C) is associated with three order objects (O1, O2 and O3). These order objects are associated with special order and normal order objects (S1, S2 and N1). The

customer is having the following three orders with different numbers (12, 32 and 40) for the particular time considered.

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

For orders the values are 12, 32, and 40 which implies that the objects are having these values for the particular moment (here the particular time when the purchase is made is considered as the moment) when the instance is captured.

The same is for special order and normal order objects which are having number of orders as 20, 30 and 60. If a different time of purchase is considered then these values will change accordingly. So the following object diagram has been drawn considering all the points mentioned above:



UML Component Diagram

Component diagrams are different in terms of nature and behaviour. Component diagrams are used to model physical aspects of a system.

Now the question is what are these physical aspects? Physical aspects are the elements like executables, libraries, files, documents etc which resides in a node.

So component diagrams are used to visualize the organization and relationships among components in a system. These diagrams are also used to make executable systems.

Purpose:

Component diagram is a special kind of diagram in UML. The purpose is also different from all other diagrams discussed so far. It does not describe the functionality of the system but it describes the components used to make those functionalities.

So from that point component diagrams are used to visualize the physical components in a system. These components are libraries, packages, files etc.

Component diagrams can also be described as a static implementation view of a system. Static implementation represents the organization of the components at a particular moment.

A single component diagram cannot represent the entire system but a collection of diagrams are used to represent the whole.

So the purpose of the component diagram can be summarized as:

- Visualize the components of a system.
- Construct executables by using forward and reverse engineering.
- Describe the organization and relationships of the components.

How to draw Component Diagram?

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

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

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

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

So before drawing a component diagram the following artifacts are to be identified clearly:

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

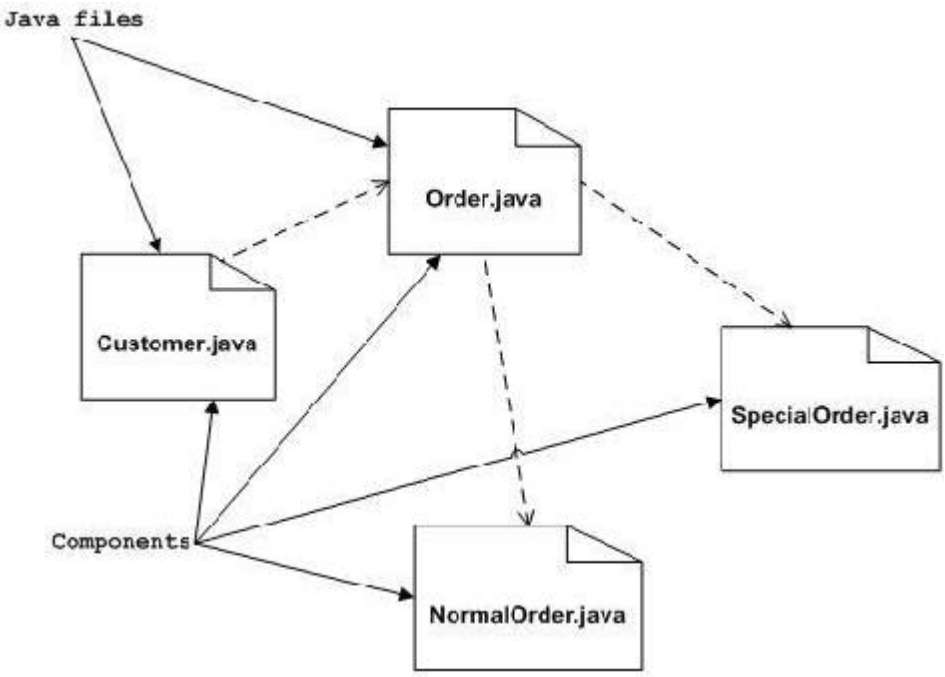
Now after identifying the artifacts the following points needs to be followed:

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

The following is a component diagram for order management system. Here the artifacts are files. So the diagram shows the files in the application and their relationships. In actual the component diagram also contains dlls, libraries, folders etc.

In the following diagram four files are identified and their relationships are produced. Component diagram cannot be matched directly with other UML diagrams discussed so far. Because it is drawn for completely different purpose.

So the following component diagram has been drawn considering all the points mentioned above:



UML Deployment Diagram

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

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

Purpose:

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

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

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

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

The purpose of deployment diagrams can be described as:

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

How to draw Deployment Diagram?

Deployment diagram represents the deployment view of a system. It is related to the component diagram. Because the components are deployed using the deployment diagrams. A deployment diagram consists of nodes. Nodes are nothing but physical hardwares used to deploy the application.

Deployment diagrams are useful for system engineers. An efficient deployment diagram is very important because it controls the following parameters

- Performance
- Scalability
- Maintainability
- Portability

So before drawing a deployment diagram the following artifacts should be identified:

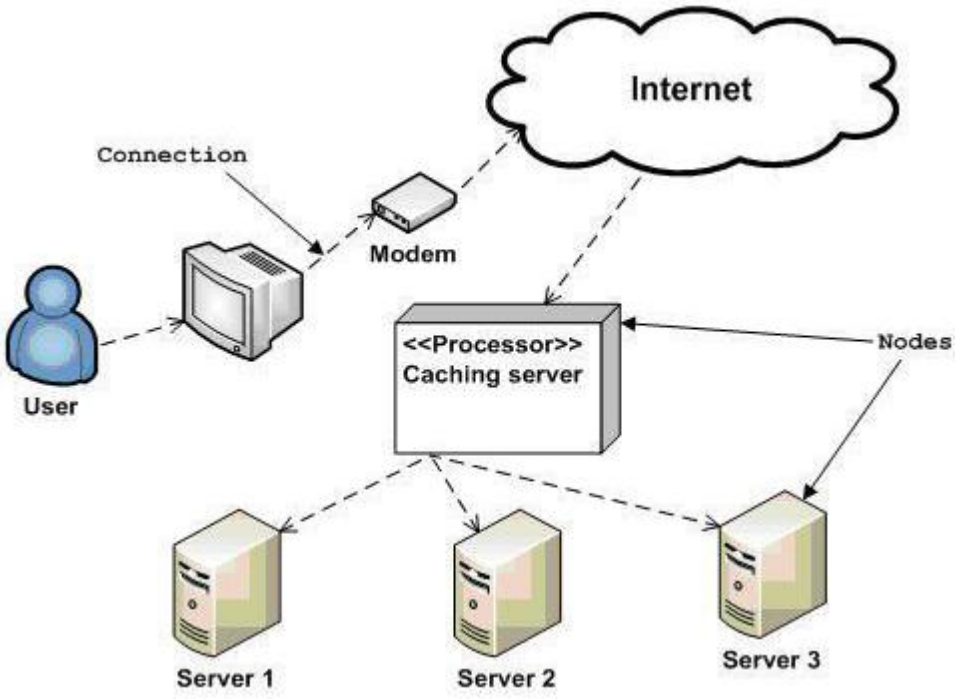
- Nodes
- Relationships among nodes

The following deployment diagram is a sample to give an idea of the deployment view of order management system. Here we have shown nodes as:

- Monitor
- Modem
- Caching server
- Server

The application is assumed to be a web based application which is deployed in a clustered environment using server 1, server 2 and server 3. The user is connecting to the application using internet. The control is flowing from the caching server to the clustered environment.

So the following deployment diagram has been drawn considering all the points mentioned above:



UML Use Case Diagram

To model a system the most important aspect is to capture the dynamic behaviour. To clarify a bit in details, dynamic behaviour means the behaviour of the system when it is running /operating.

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

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

So to model the entire system numbers of use case diagrams are used.

Purpose:

The purpose of use case diagram is to capture the dynamic aspect of a system. But this definition is too generic to describe the purpose.

Because other four diagrams (activity, sequence, collaboration and Statechart) are also having the same purpose. So we will look into some specific purpose which will distinguish it from other four diagrams.

Use case diagrams are used to gather the requirements of a system including internal and external influences. These requirements are mostly design requirements. So when a system is analyzed to gather its functionalities use cases are prepared and actors are identified.

Now when the initial task is complete use case diagrams are modelled to present the outside view. So in brief, the purposes of use case diagrams can be as follows:

- Used to gather requirements of a system.
- Used to get an outside view of a system.
- Identify external and internal factors influencing the system.
- Show the interacting among the requirements are actors.

How to draw Use Case Diagram?

Use case diagrams are considered for high level requirement analysis of a system. So when the requirements of a system are analyzed the functionalities are captured in use cases.

So we can say that uses cases are nothing but the system functionalities written in an organized manner. Now the second things which are relevant to the use cases are the actors. Actors can be defined as something that interacts with the system.

The actors can be human user, some internal applications or may be some external applications. So in a brief when we are planning to draw an use case diagram we should have the following items identified.

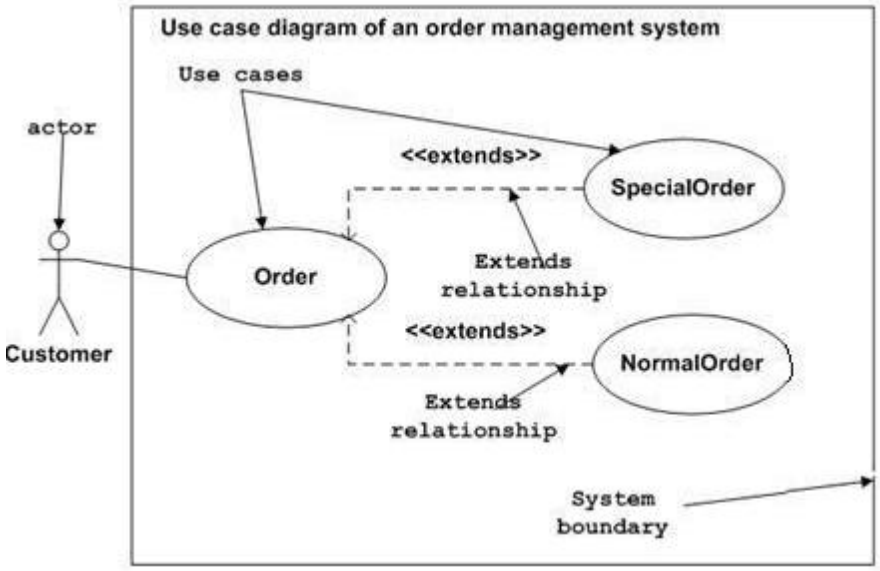
- Functionalities to be represented as an use case
- Actors
- Relationships among the use cases and actors.

Use case diagrams are drawn to capture the functional requirements of a system. So after identifying the above items we have to follow the following guidelines to draw an efficient use case diagram.

- The name of a use case is very important. So the name should be chosen in such a way so that it can identify the functionalities performed.
- Give a suitable name for actors.
- Show relationships and dependencies clearly in the diagram.
- Do not try to include all types of relationships. Because the main purpose of the diagram is to identify requirements.
- Use note when ever required to clarify some important points.

The following is a sample use case diagram representing the order management system. So if we look into the diagram then we will find three use cases (Order, SpecialOrder and NormalOrder) and one actor which is customer.

The SpecialOrder and NormalOrder use cases are extended from Order use case. So they have extends relationship. Another important point is to identify the system boundary which is shown in the picture. The actor Customer lies outside the system as it is an external user of the system.



UML Interaction Diagram

From the name Interaction it is clear that the diagram is used to describe some type of interactions among the different elements in the model. So this interaction is a part of dynamic behaviour of the system.

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

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

Purpose:

The purposes of interaction diagrams are to visualize the interactive behaviour of the system. Now visualizing interaction is a difficult task. So the solution is to use different types of models to capture the different aspects of the interaction.

That is why sequence and collaboration diagrams are used to capture dynamic nature but from a different angle.

So the purposes of interaction diagram can be describes as:

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

How to draw Interaction Diagram?

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

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

So the following things are to identified clearly before drawing the interaction diagram:

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

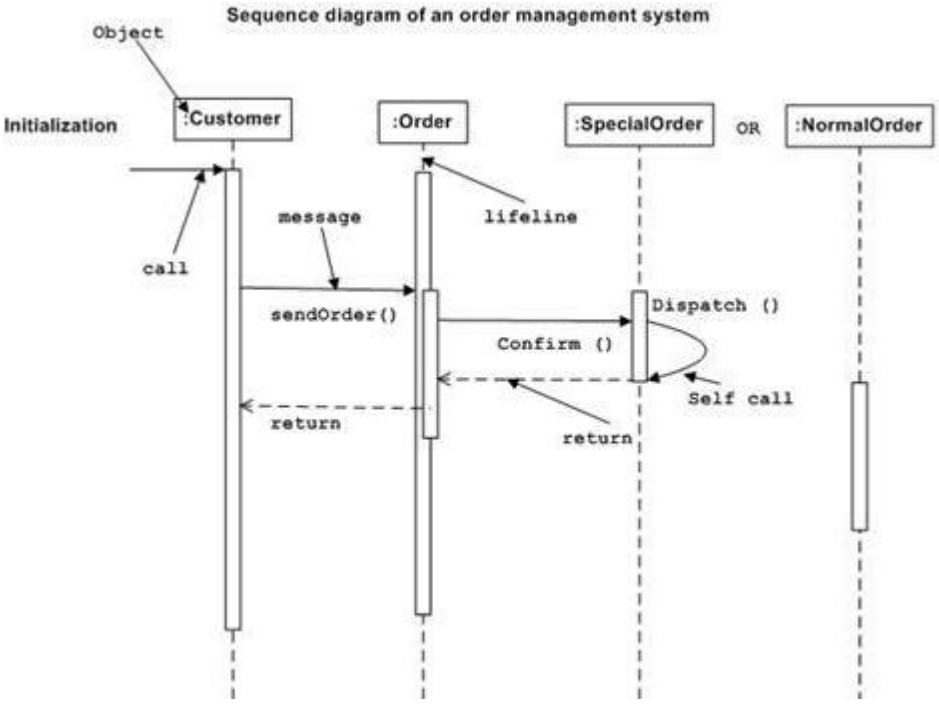
Following are two interaction diagrams modelling order management system. The first diagram is a sequence diagram and the second is a collaboration diagram.

The Sequence Diagram:

The sequence diagram is having four objects (Customer, Order, SpecialOrder and NormalOrder).

The following diagram has shown the message sequence for SpecialOrder object and the same can be used in case of NormalOrder object. Now it is important to understand the time sequence of message flows. The message flow is nothing but a method call of an object.

The first call is sendOrder () which is a method of Order object. The next call is confirm () which is a method of SpecialOrder object and the last call is Dispatch () which is a method of SpecialOrder object. So here the diagram is mainly describing the method calls from one object to another and this is also the actual scenario when the system is running.

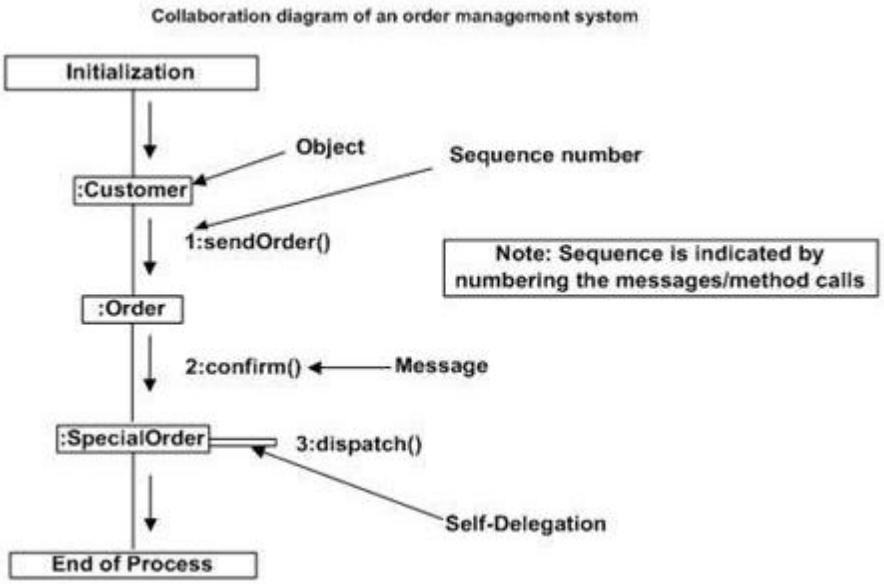


The Collaboration Diagram:

The second interaction diagram is collaboration diagram. It shows the object organization as shown below. Here in collaboration diagram the method call sequence is indicated by some numbering technique as shown below. The number indicates how the methods are called one after another. We have taken the same order management system to describe the collaboration diagram.

The method calls are similar to that of a sequence diagram. But the difference is that the sequence diagram does not describe the object organization where as the collaboration diagram shows the object organization.

Now to choose between these two diagrams the main emphasis is given on the type of requirement. If the time sequence is important then sequence diagram is used and if organization is required then collaboration diagram is used.



UML Statechart Diagram

The name of the diagram itself clarifies the purpose of the diagram and other details. It describes different states of a component in a system. The states are specific to a component/object of a system.

A Statechart diagram describes a state machine. Now to clarify it state machine can be defined as a machine which defines different states of an object and these states are controlled by external or internal events.

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

Purpose:

Statechart diagram is one of the five UML diagrams used to model dynamic nature of a system. They define different states of an object during its lifetime. And these states are changed by events. So Statechart diagrams are useful to model reactive systems. Reactive systems can be defined as a system that responds to external or internal events.

Statechart diagram describes the flow of control from one state to another state. States are defined as a condition in which an object exists and it changes when some event is triggered. So the most important purpose of Statechart diagram is to model life time of an object from creation to termination.

Statechart diagrams are also used for forward and reverse engineering of a system. But the main purpose is to model reactive system.

Following are the main purposes of using Statechart diagrams:

- To model dynamic aspect of a system.
- To model life time of a reactive system.
- To describe different states of an object during its life time.
- Define a state machine to model states of an object.

How to draw Statechart Diagram?

Statechart diagram is used to describe the states of different objects in its life cycle. So the emphasis is given on the state changes upon some internal or external events. These states of objects are important to analyze and implement them accurately.

Statechart diagrams are very important for describing the states. States can be identified as the condition of objects when a particular event occurs.

Before drawing a Statechart diagram we must have clarified the following points:

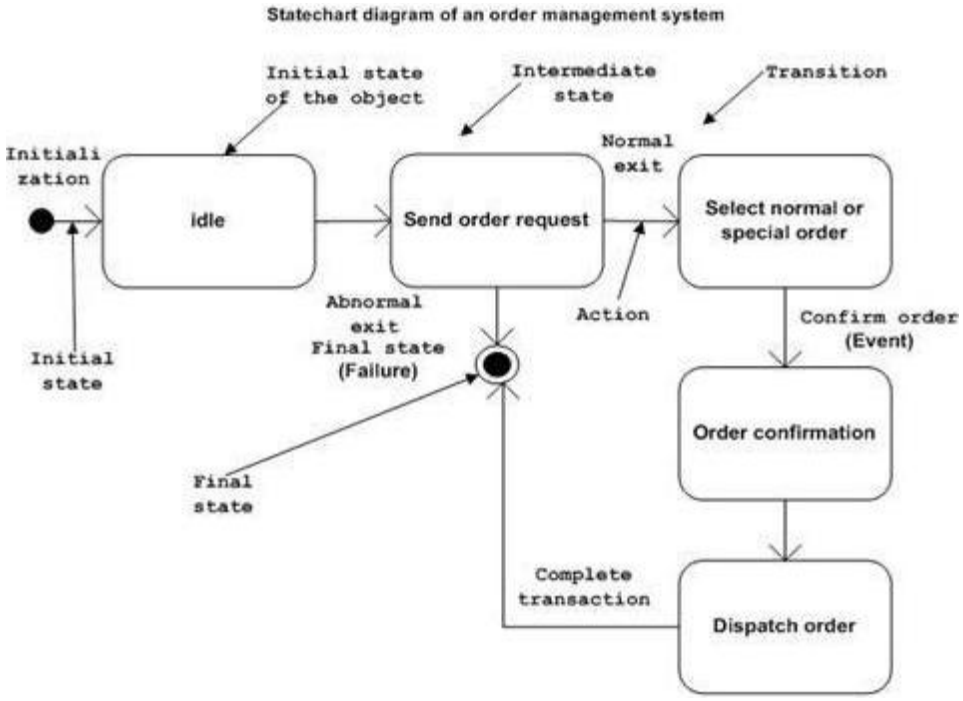
- Identify important objects to be analyzed.
- Identify the states.
- Identify the events.

The following is an example of a Statechart diagram where the state of Order object is analyzed.

The first state is an idle state from where the process starts. The next states are arrived for events like send request, confirm request, and dispatch order. These events are responsible for state changes of order object.

During the life cycle of an object (here order object) it goes through the following states and there may be some abnormal exists also. This abnormal exit may occur due to some problem in the system. When the entire life cycle is complete it is considered as the complete transaction as mentioned below.

The initial and final state of an object is also shown below:



UML Activity Diagram

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

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

So the control flow is drawn from one operation to another. This flow can be sequential, branched or concurrent. Activity diagrams deals with all type of flow control by using different elements like fork, join etc.

Purpose:

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

Activity is a particular operation of the system. Activity diagrams are not only used for visualizing dynamic nature of a system but they are also used to construct the executable system by using forward and reverse engineering techniques. The only missing thing in activity diagram is the message part.

It does not show any message flow from one activity to another. Activity diagram is some time considered as the flow chart. Although the diagrams looks like a flow chart but it is not. It shows different flow like parallel, branched, concurrent and single.

So the purposes can be described as:

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

How to draw Activity Diagram?

Activity diagrams are mainly used as a flow chart consists of activities performed by the system. But activity diagram are not exactly a flow chart as they have some additional capabilities. These additional capabilities include branching, parallel flow, swimlane etc.

Before drawing an activity diagram we must have a clear understanding about the elements used in activity diagram. The main element of an activity diagram is the activity itself. An activity is a function performed by the system. After identifying the activities we need to understand how they are associated with constraints and conditions.

So before drawing an activity diagram we should identify the following elements:

- Activities
- Association
- Conditions
- Constraints

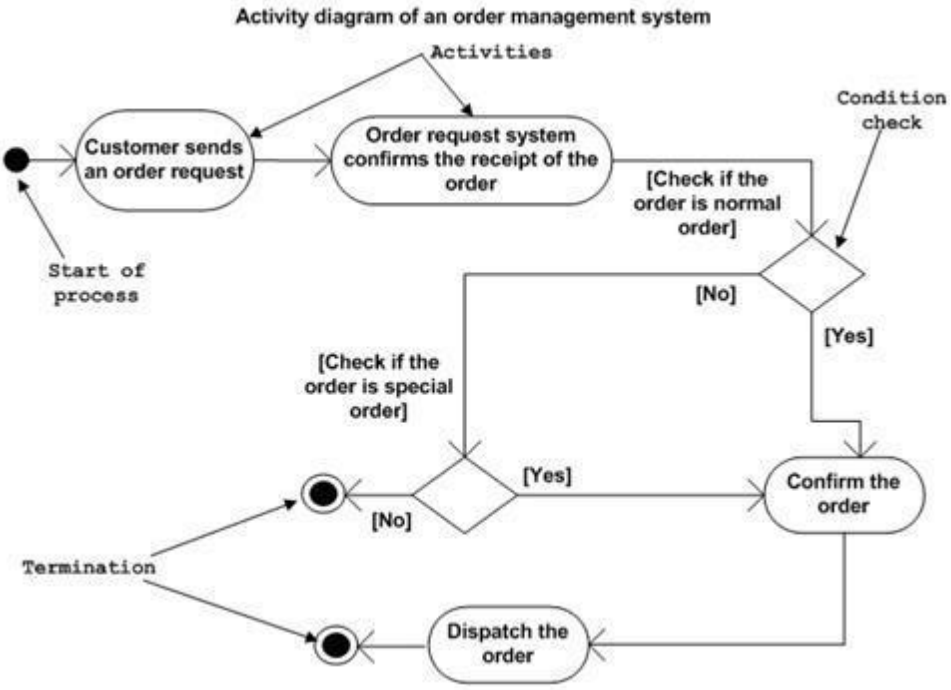
Once the above mentioned parameters are identified we need to make a mental layout of the entire flow. This mental layout is then transformed into an activity diagram.

The following is an example of an activity diagram for order management system. In the diagram four activities are identified which are associated with conditions. One important point should be clearly understood that an activity diagram cannot be exactly matched with the code. The activity diagram is made to understand the flow of activities and mainly used by the business users.

The following diagram is drawn with the four main activities:

- Send order by the customer
- Receipt of the order
- Confirm order
- Dispatch order

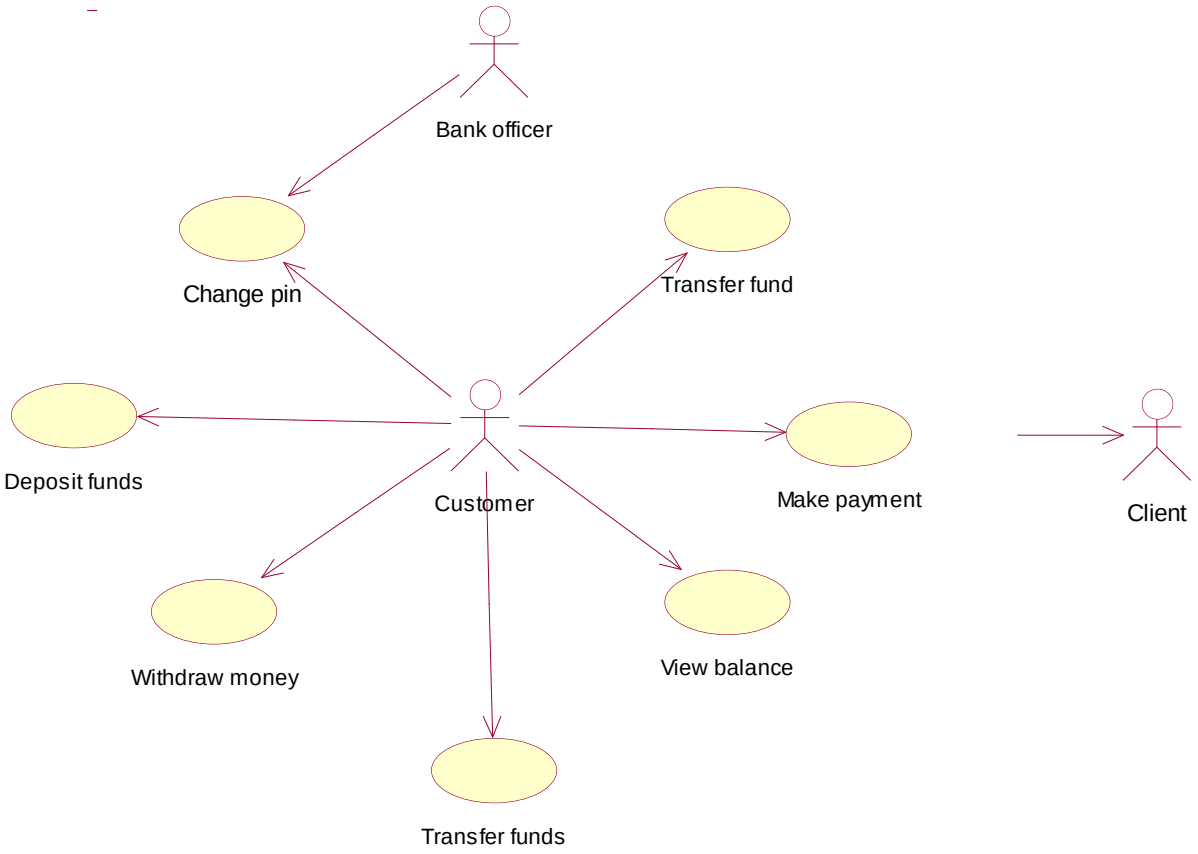
After receiving the order request condition checks are performed to check if it is normal or special order. After the type of order is identified dispatch activity is performed and that is marked as the termination of the process.


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

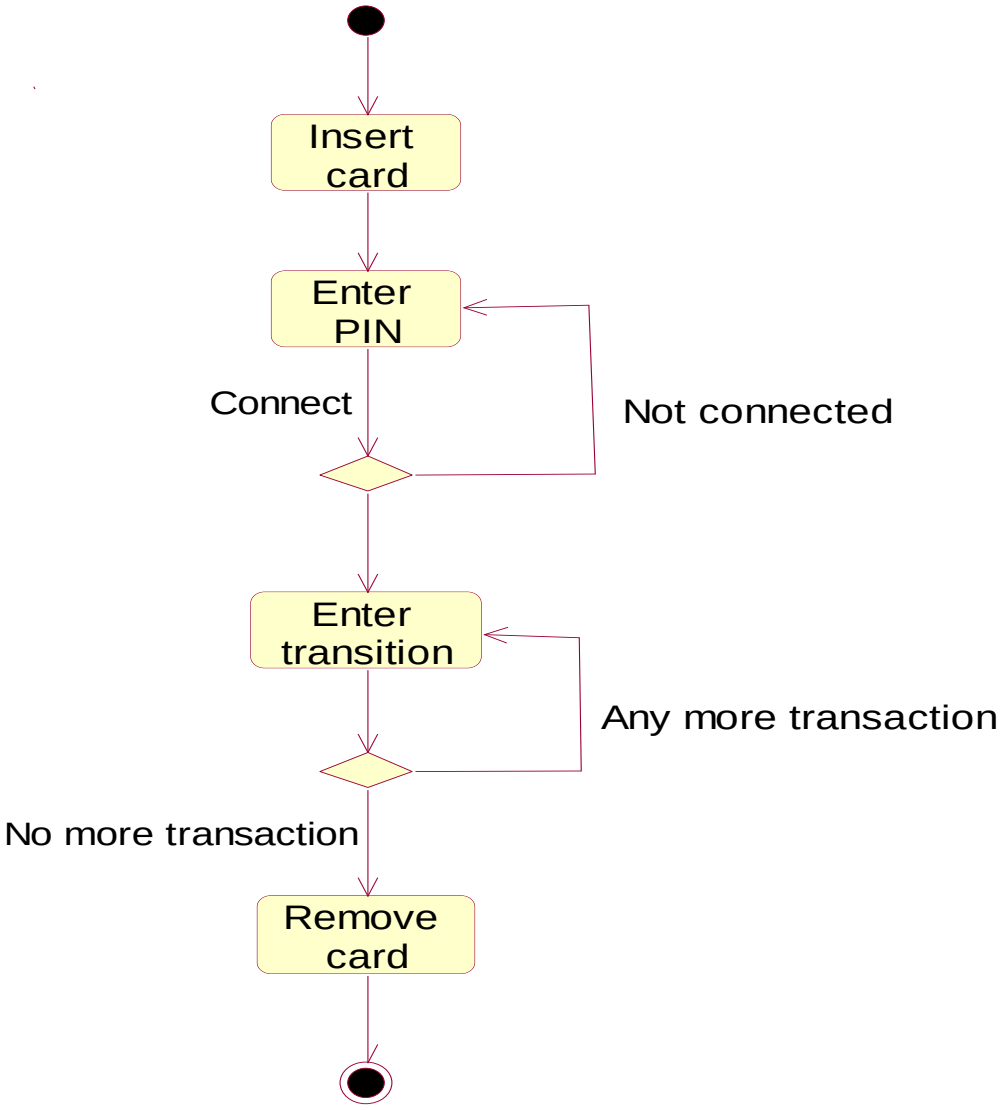
Experiments

ATM Application (ATM)

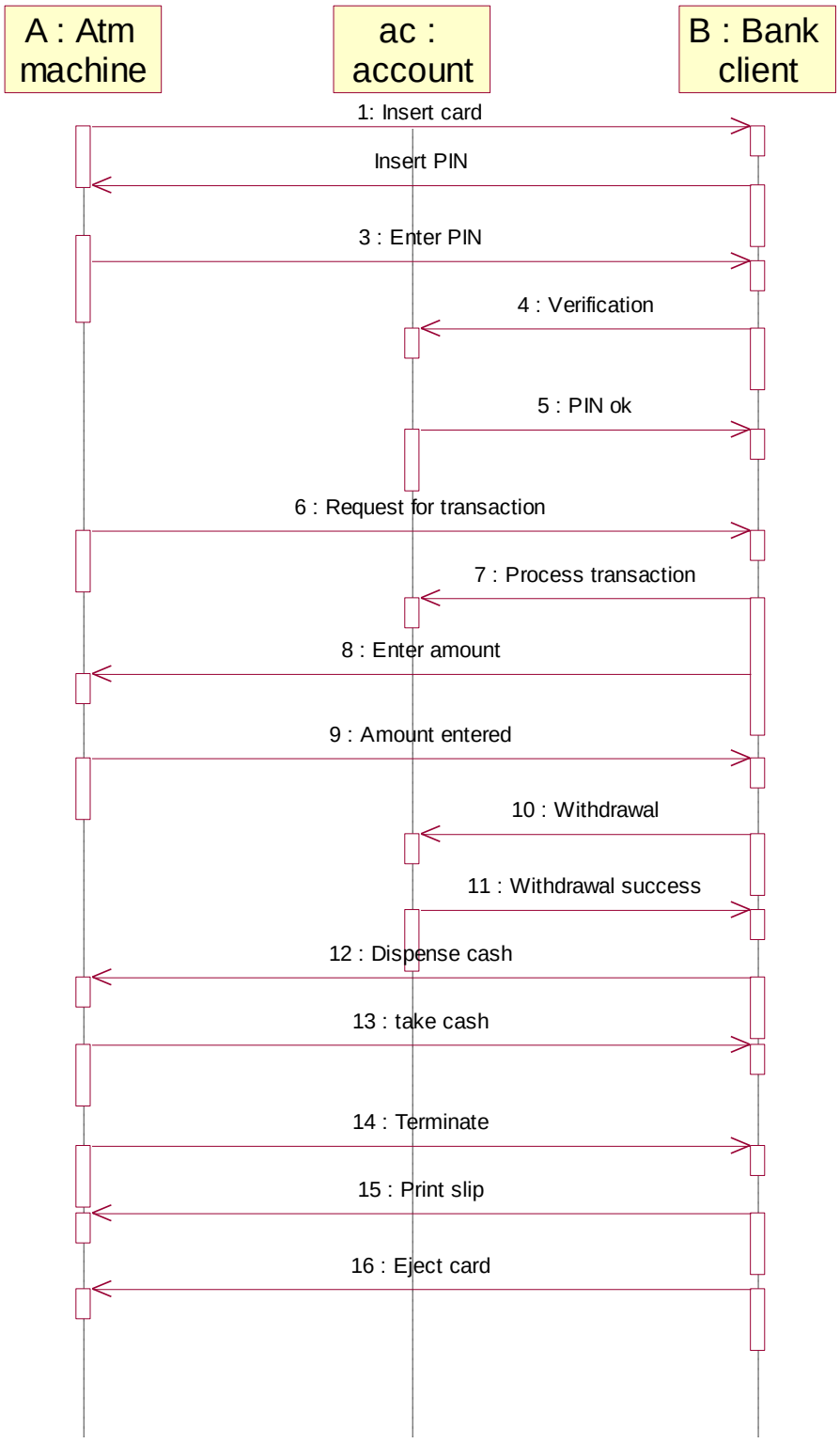
ATM Scenario Use Case Diagram:



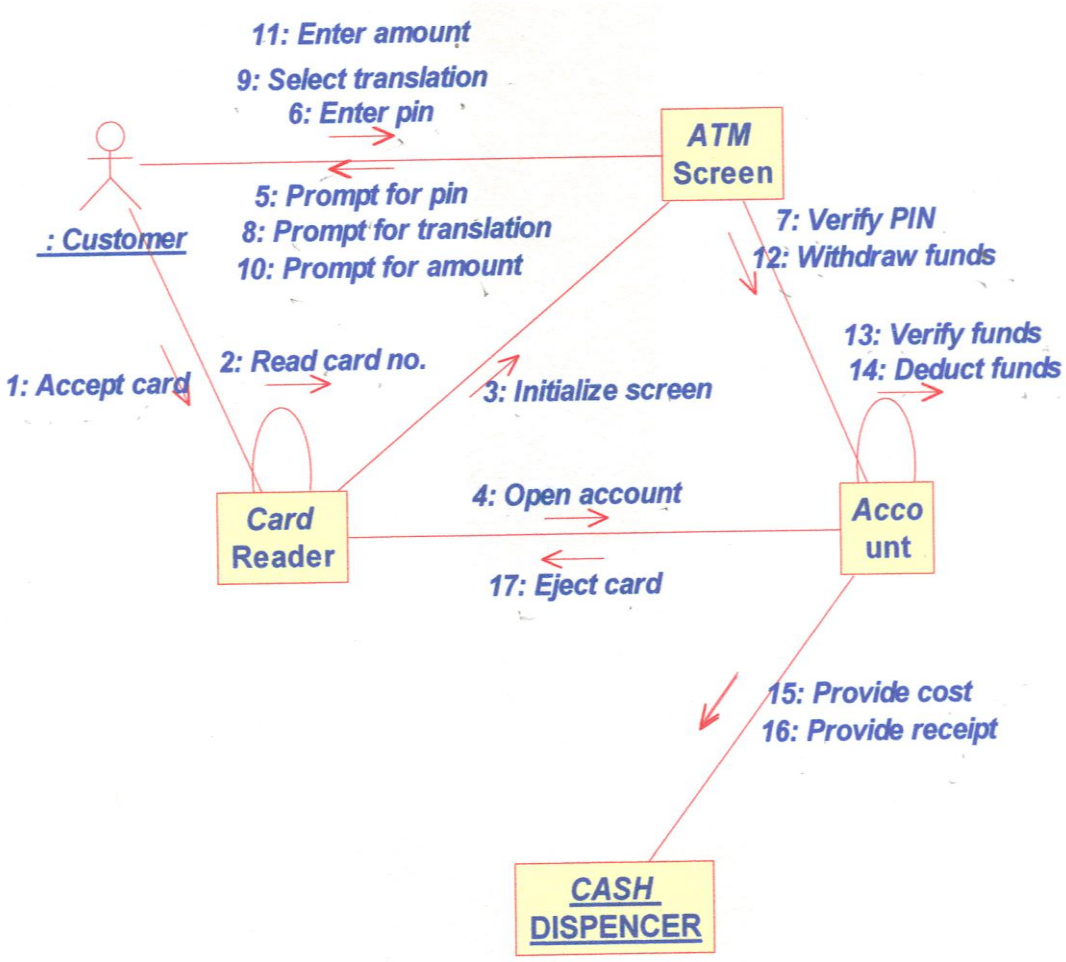
ATM Scenario Activity Diagram:



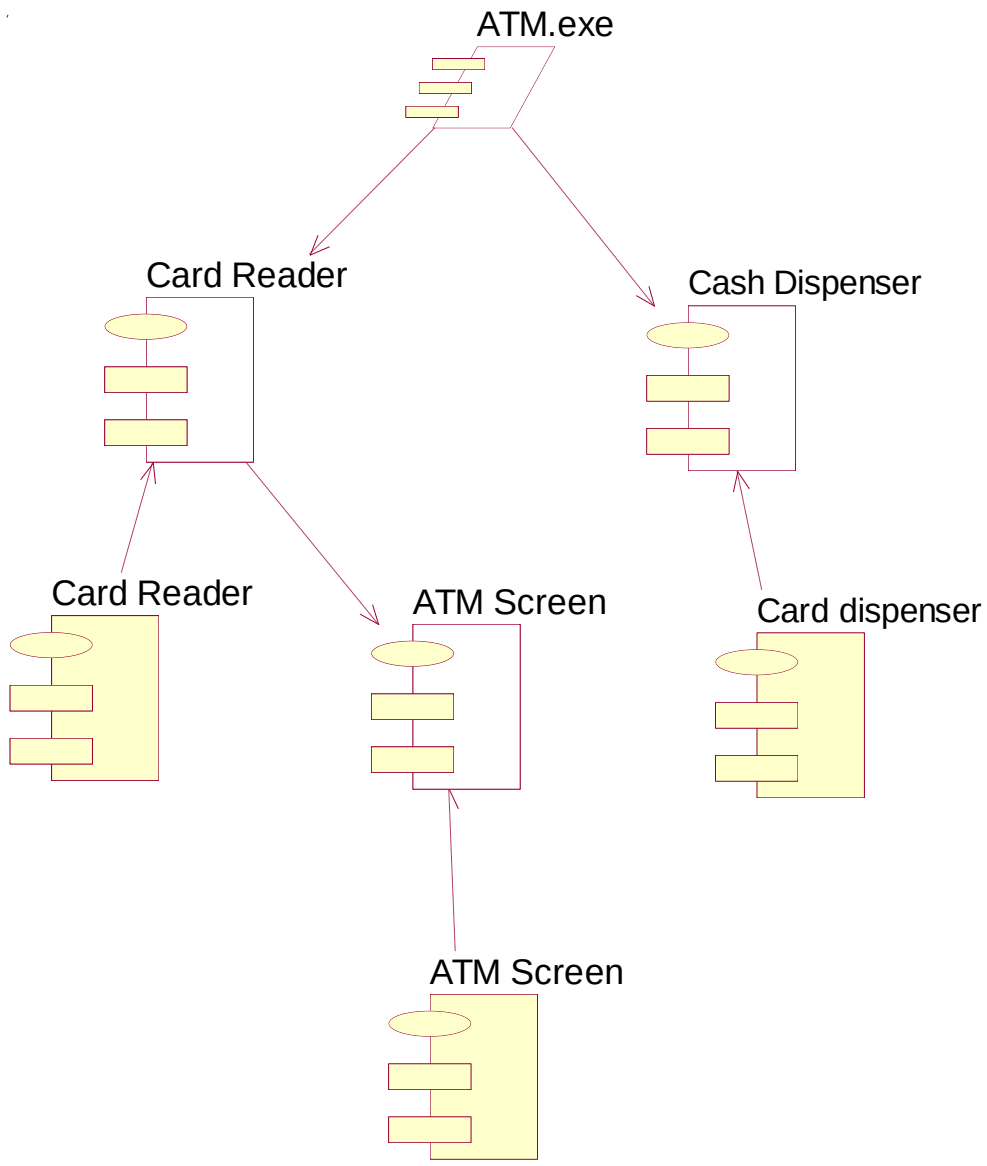
ATM Scenario Sequence Diagram:



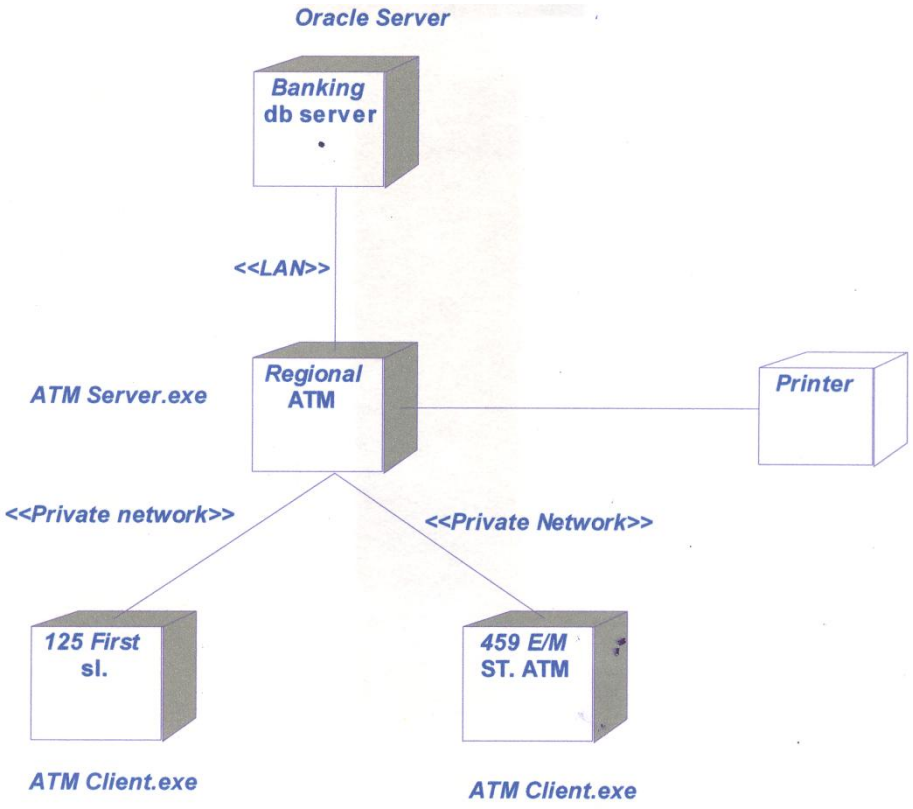
ATM Scenario Collaboration Diagram:



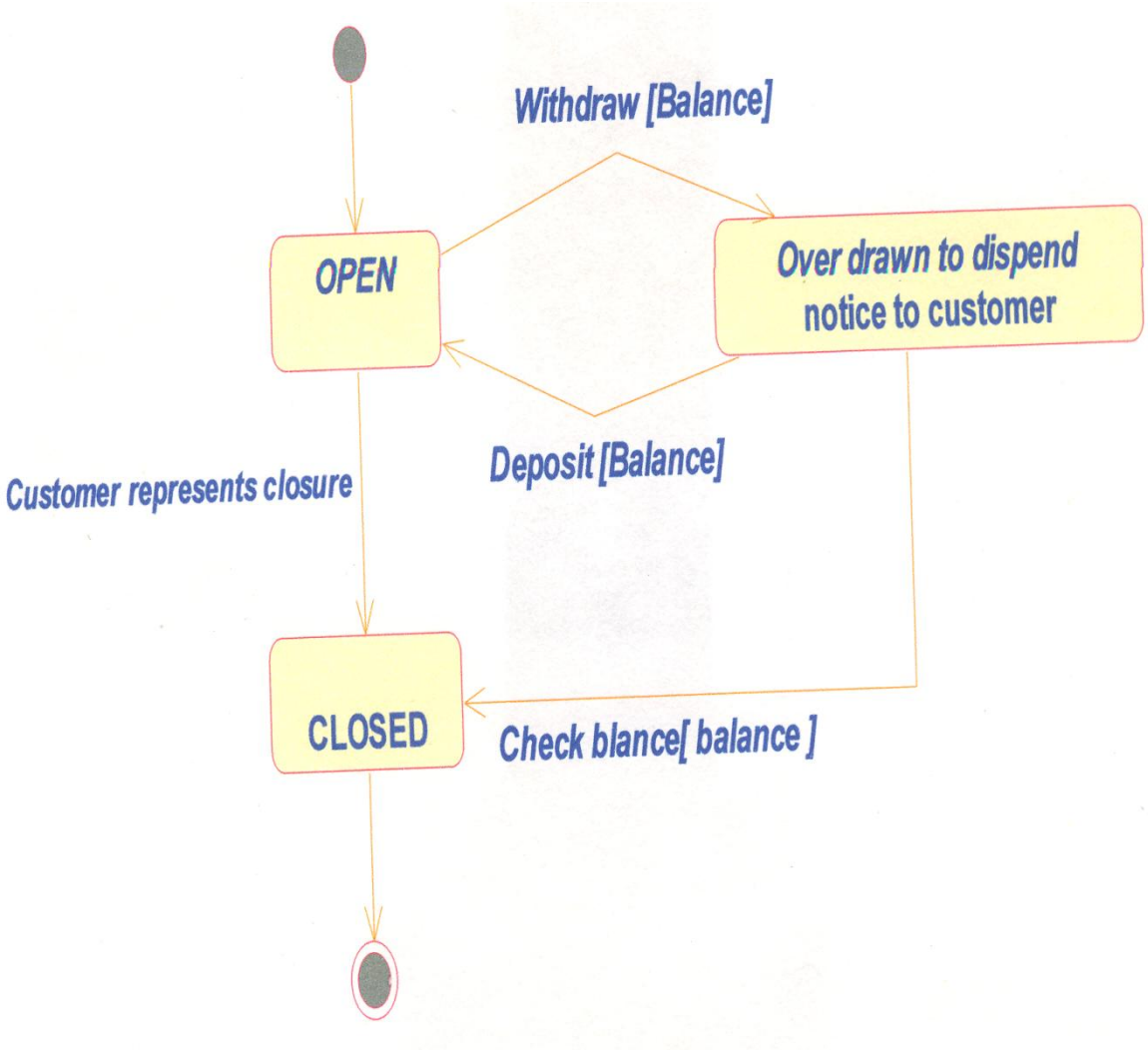
ATM Scenario Component Diagram:



ATM Scenario Deployment Diagram:

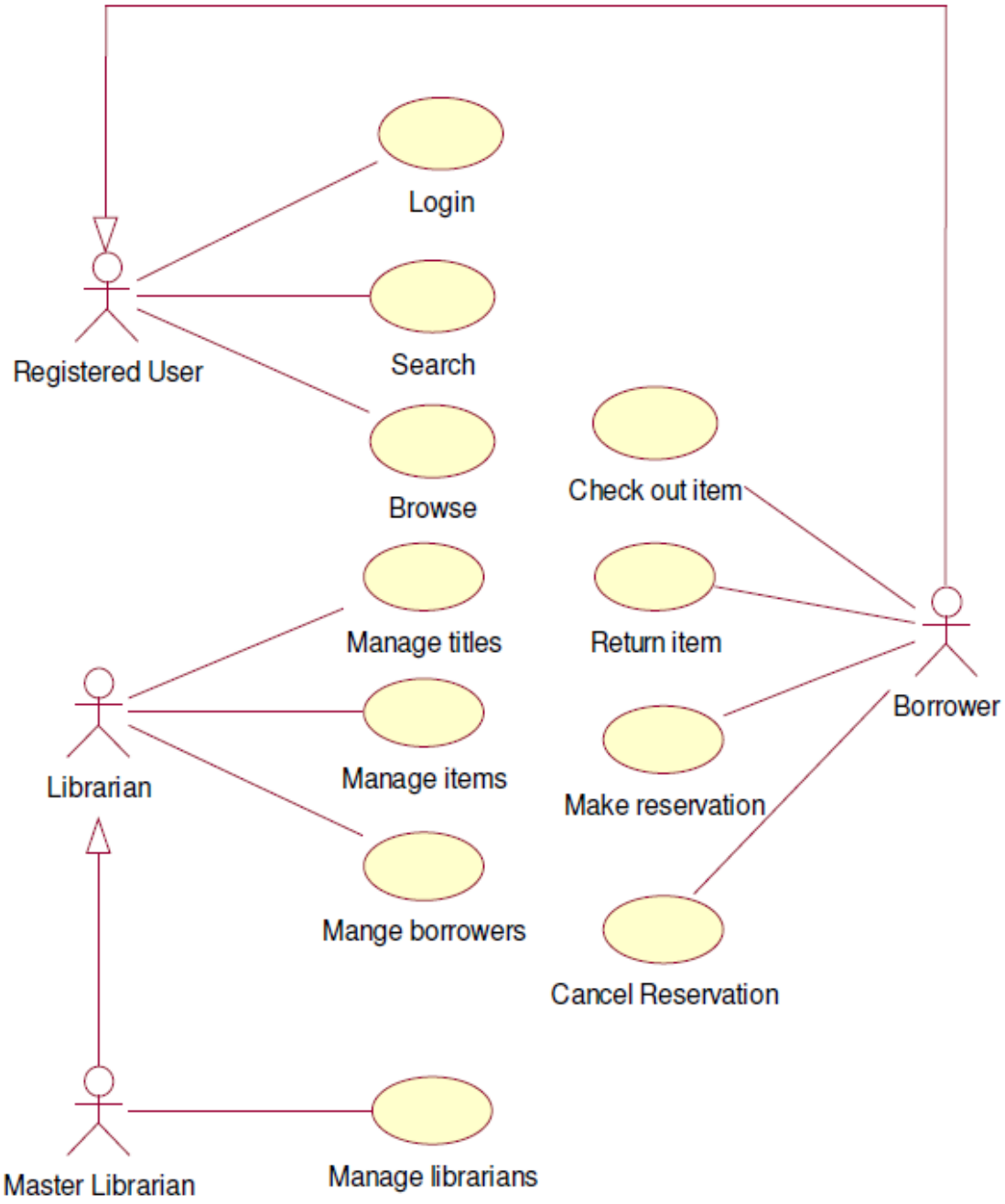


ATM Scenario State Chart Diagram

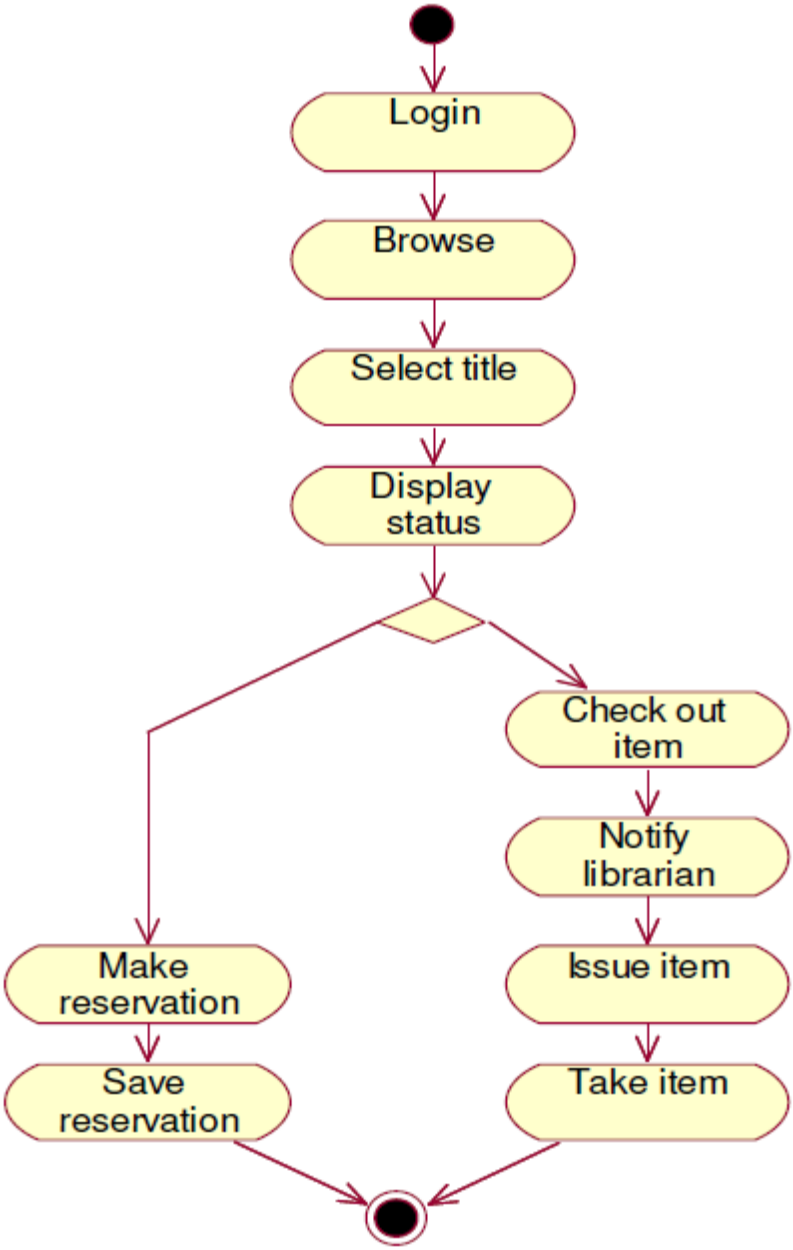


Library Management System (LMS)

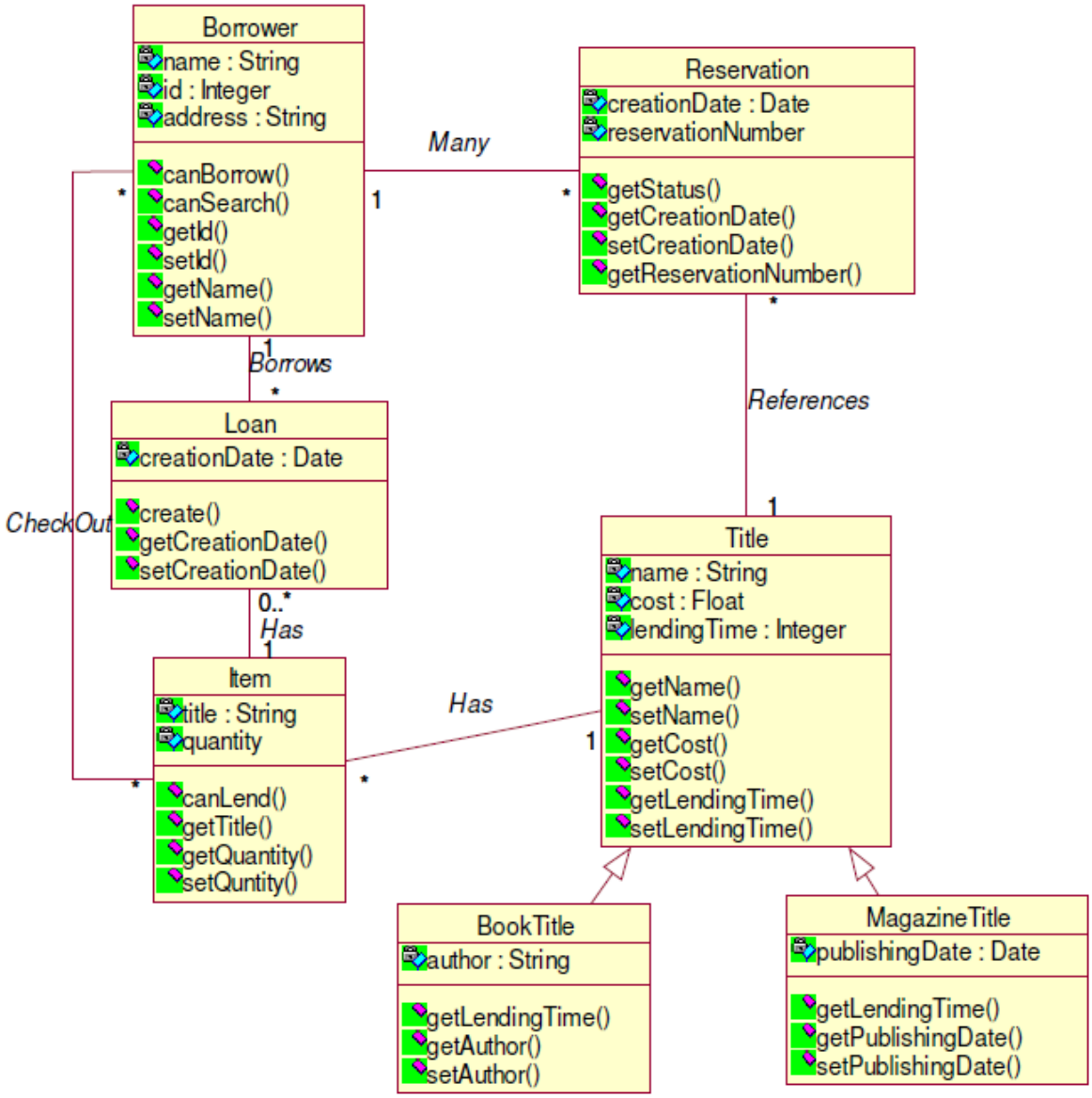
LMS Login Use Case Diagram:



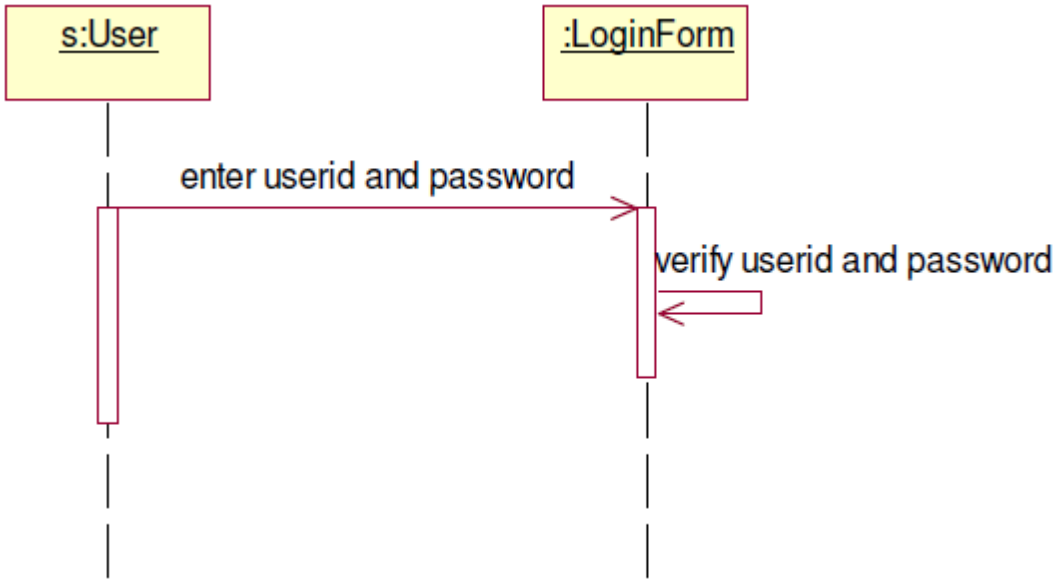
LMS Activity Diagram:



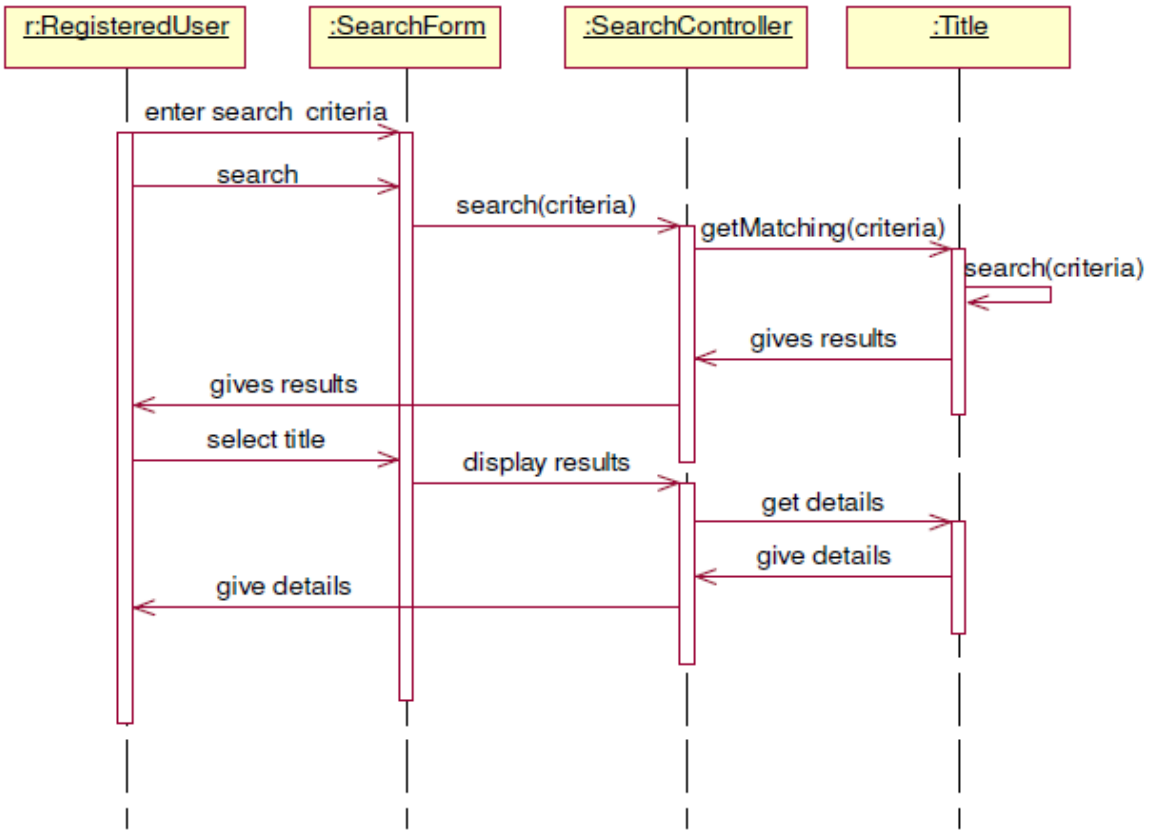
LMS Class Diagram:



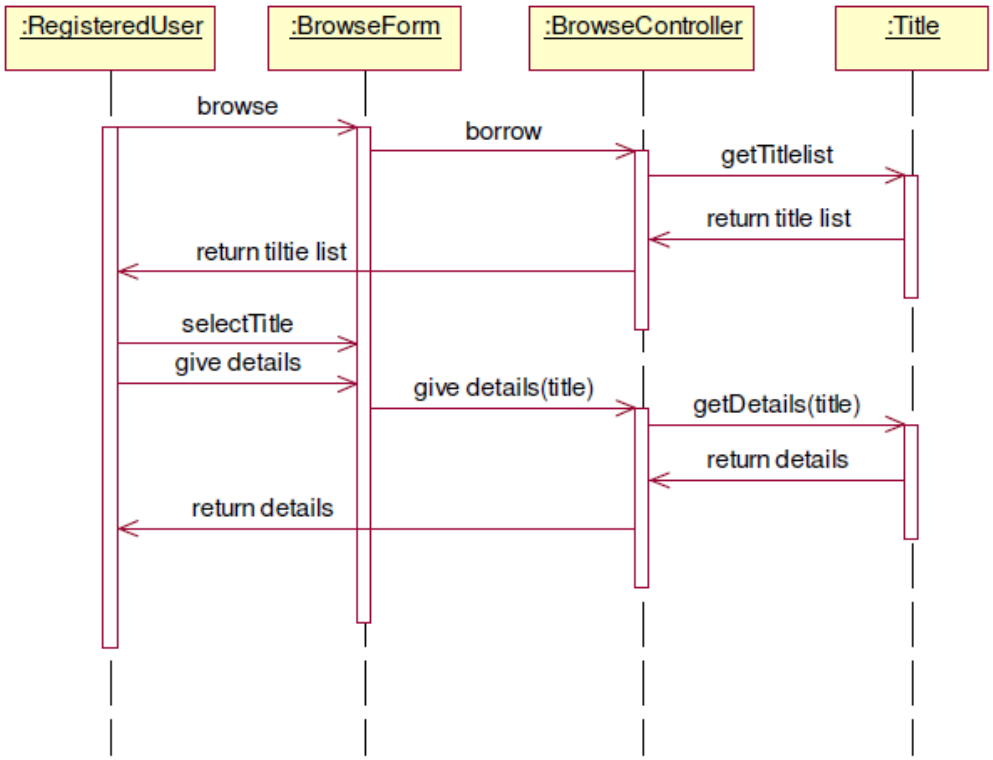
LMS Login Sequence Diagram:



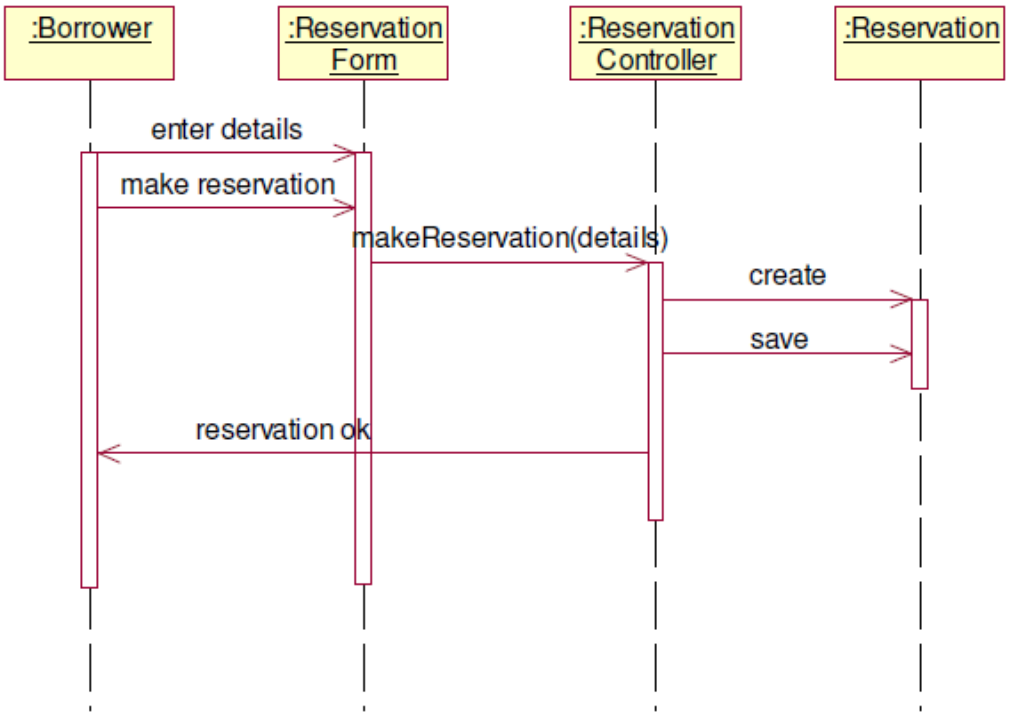
LMS Search Sequence Diagram:



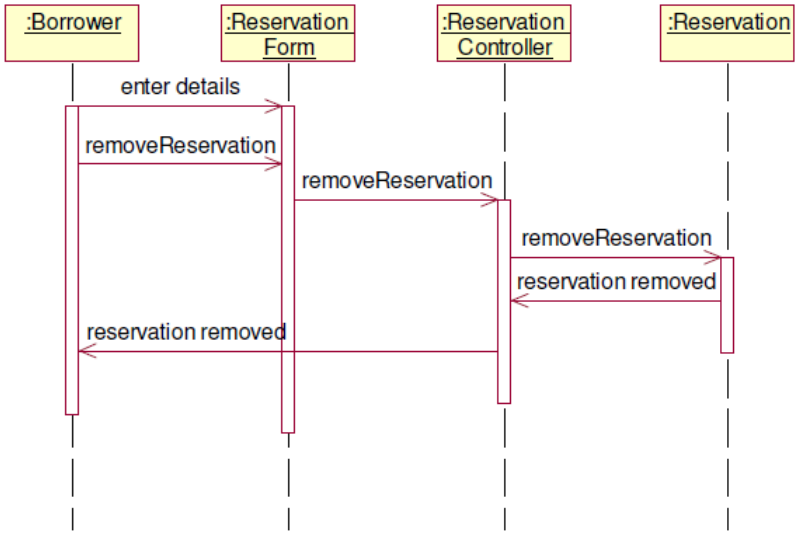
LMS Browse Sequence Diagram:



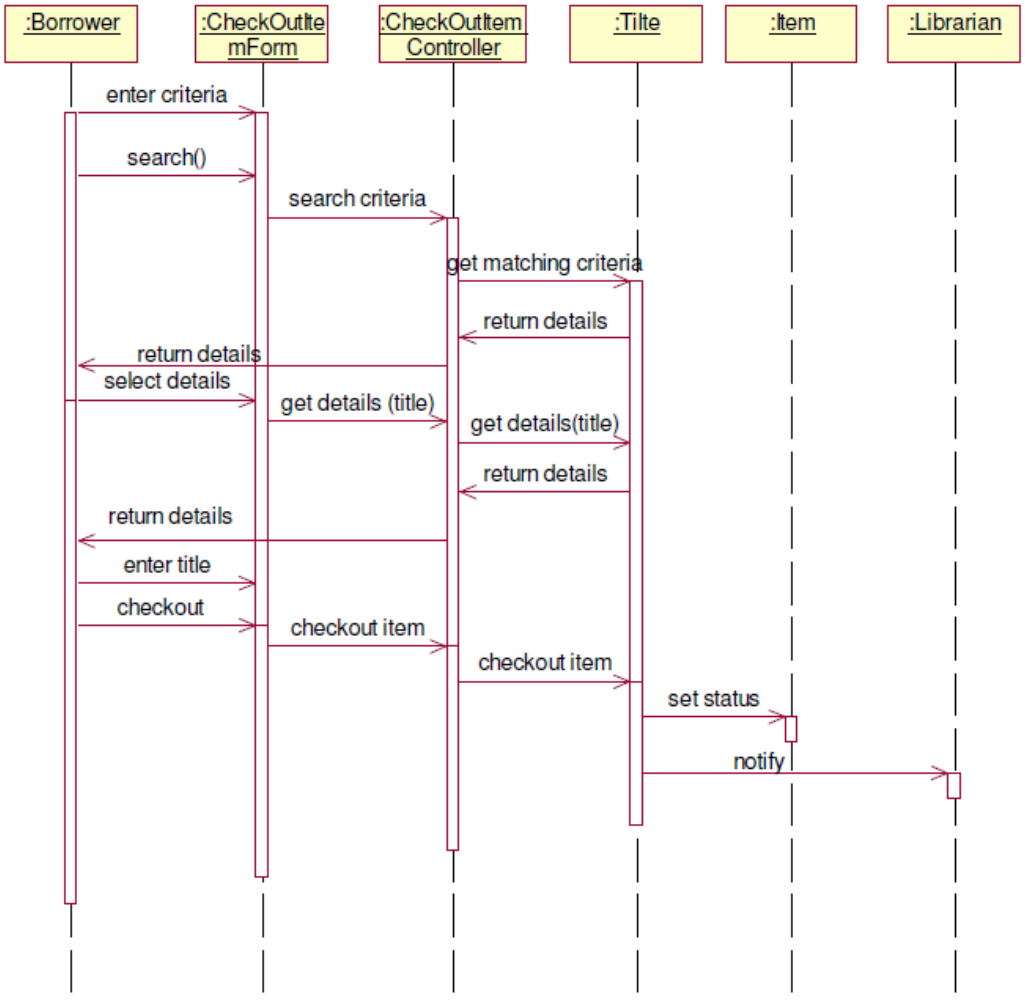
LMS Reservation Sequence Diagram:



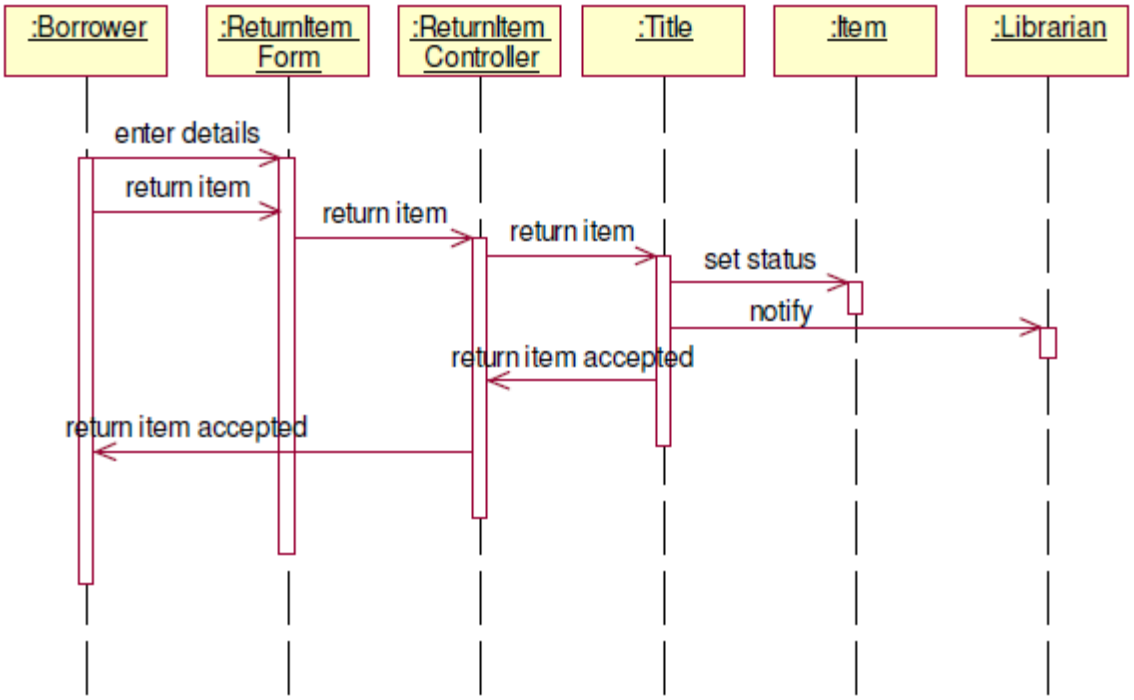
LMS Remove Reservation Sequence Diagram:



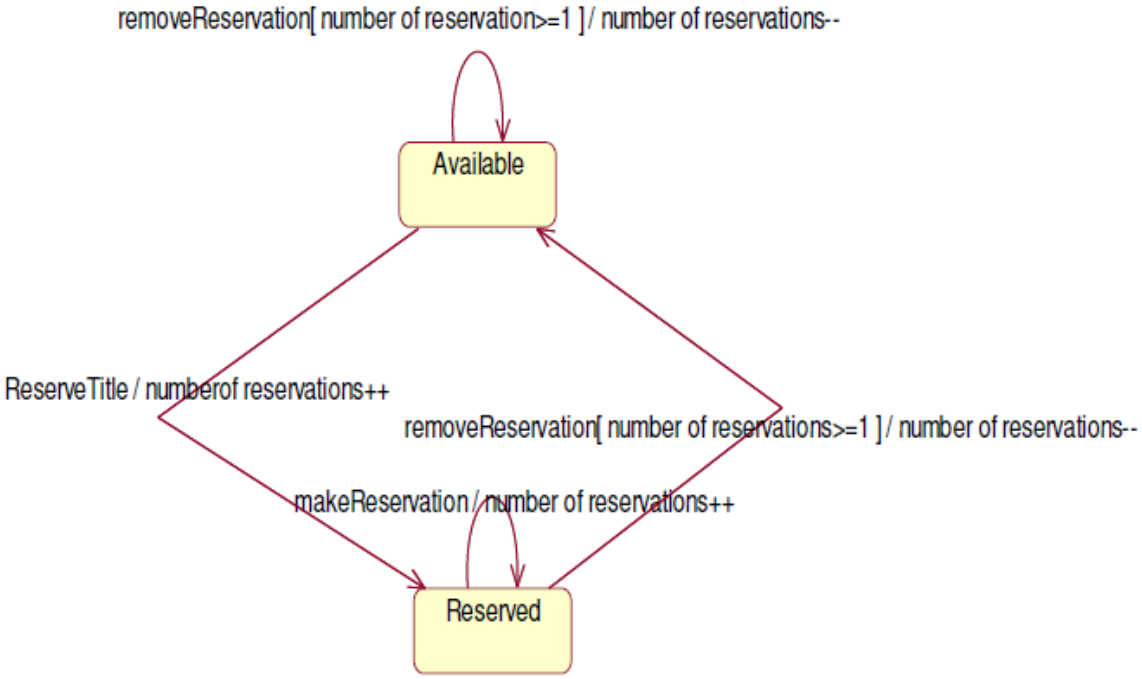
LMS Check Out Item Sequence Diagram:



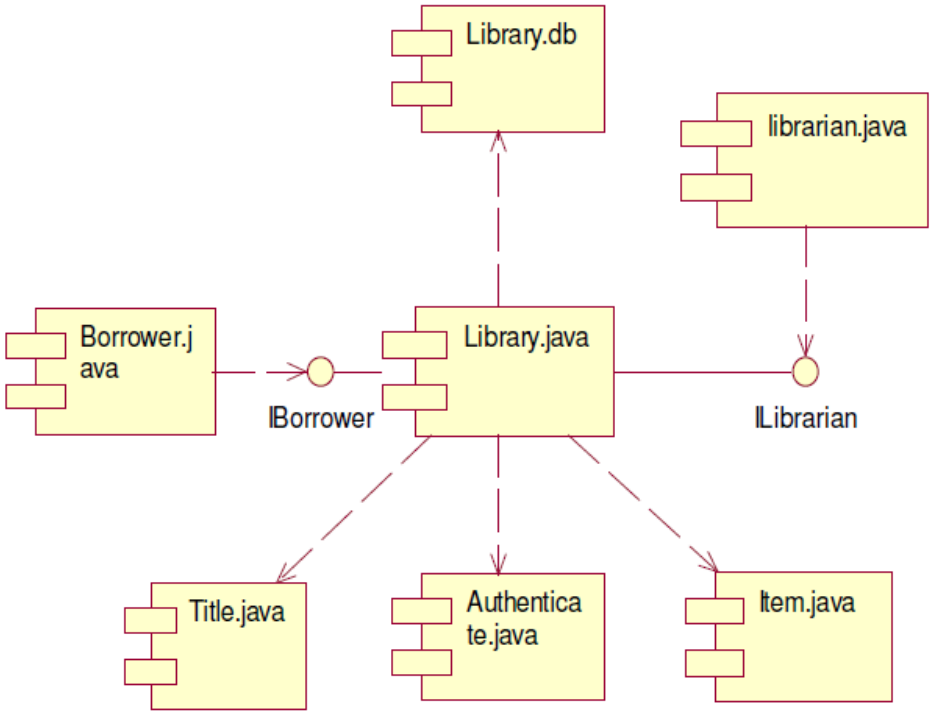
LMS Return Item Sequence Diagram:



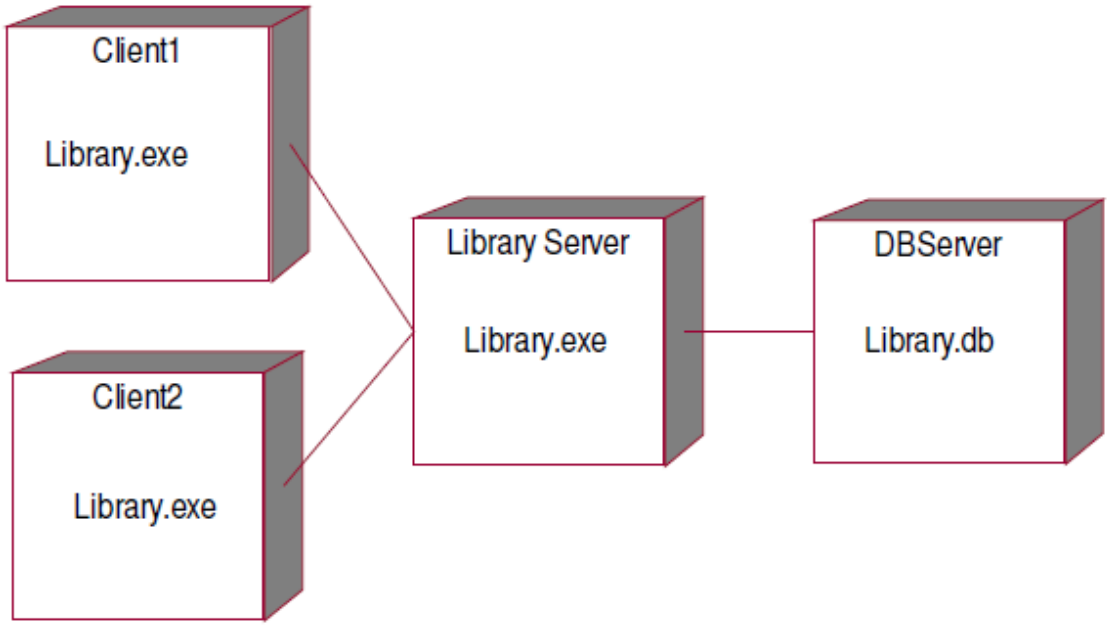
LMS State Chart Diagram for Title Class:



LMS Component Diagram:

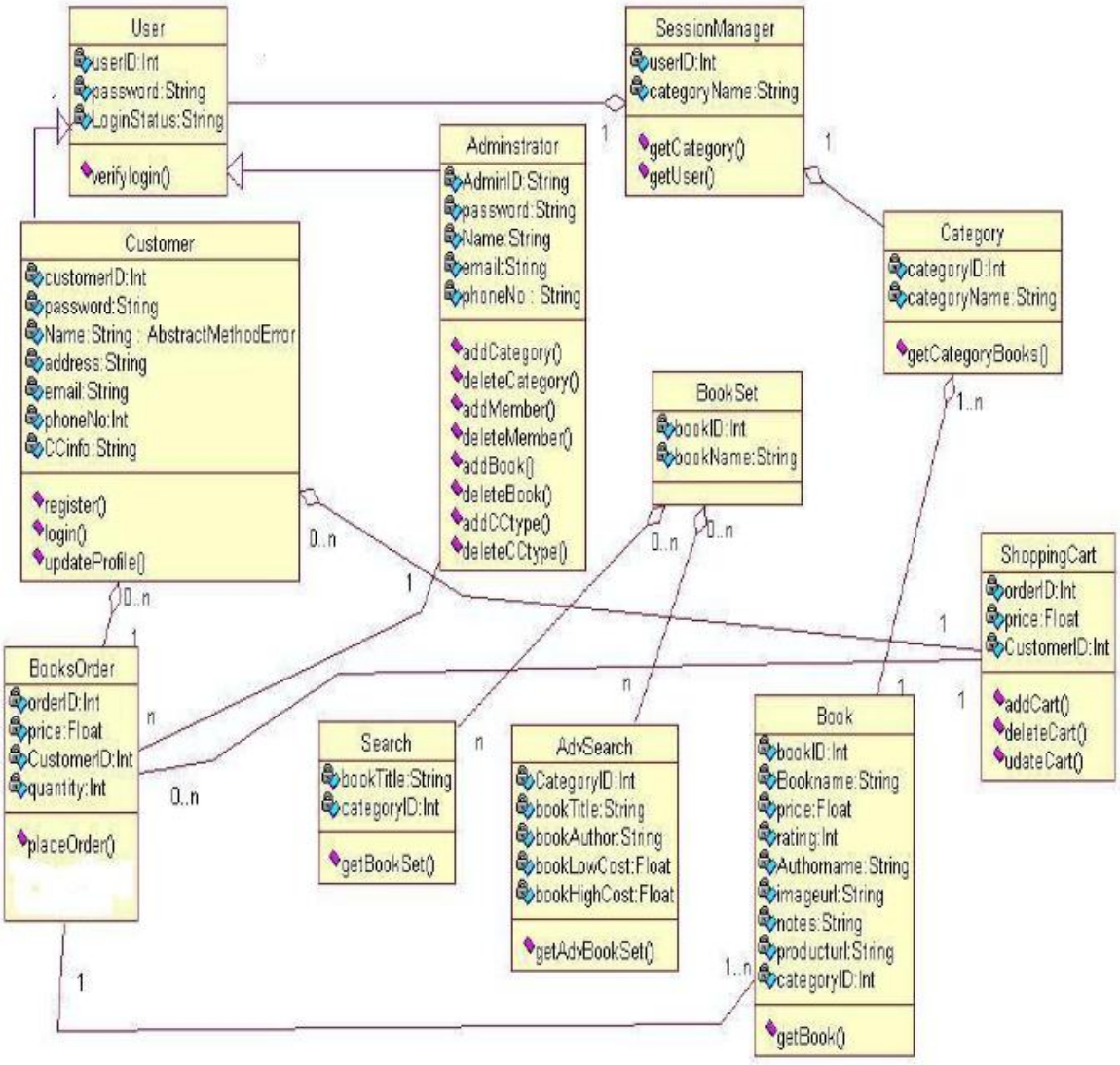


LMS Deployment Diagram:

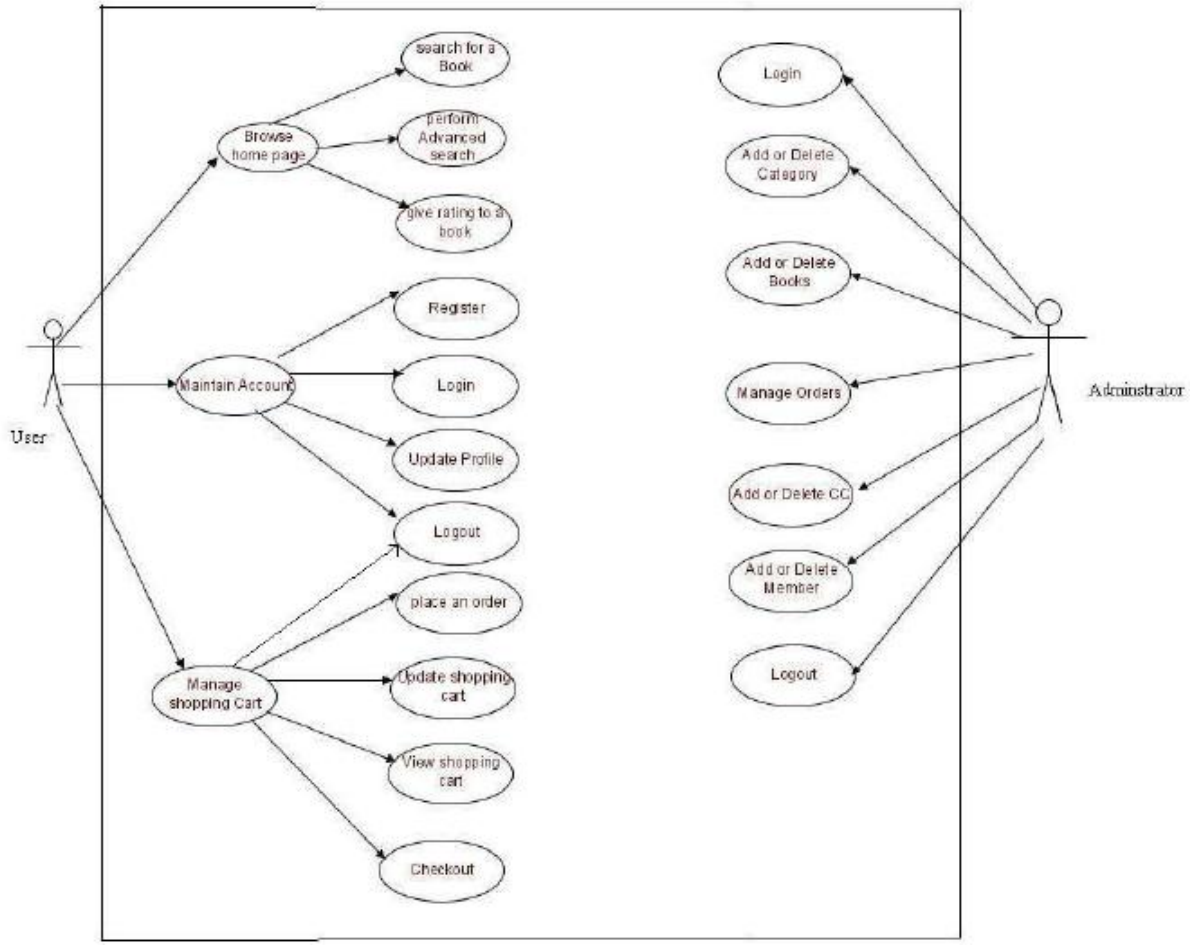


Online Book Shop (OBS)

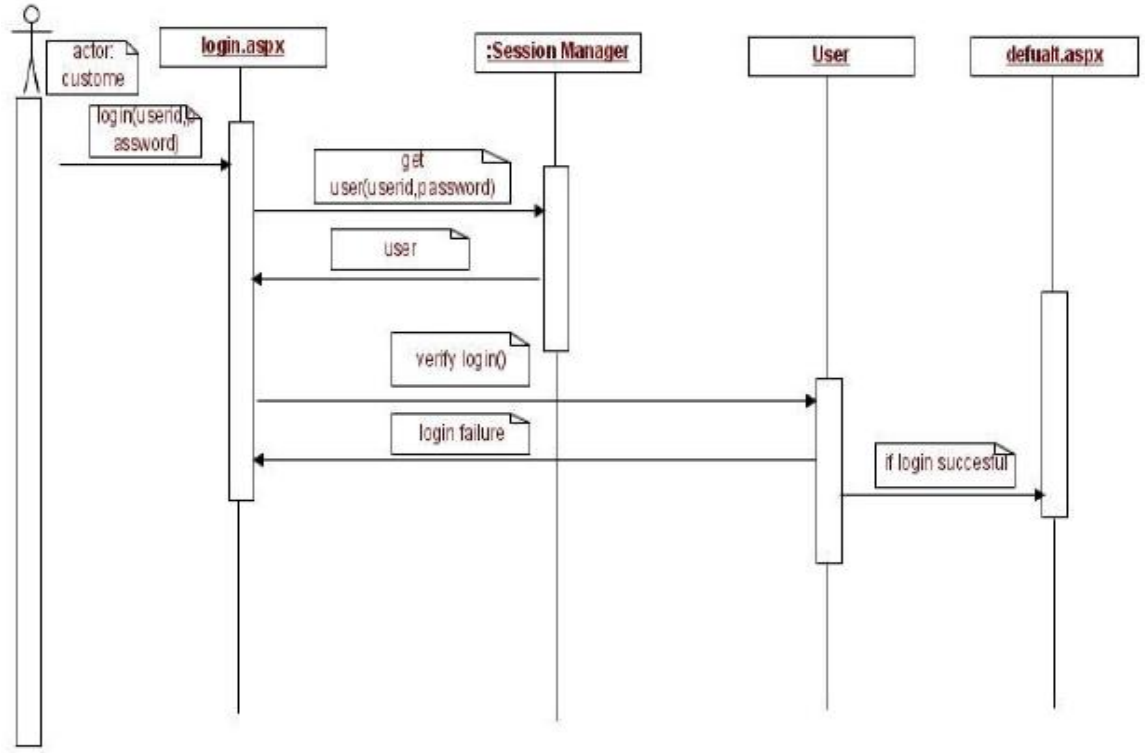
OBS Class Diagram:



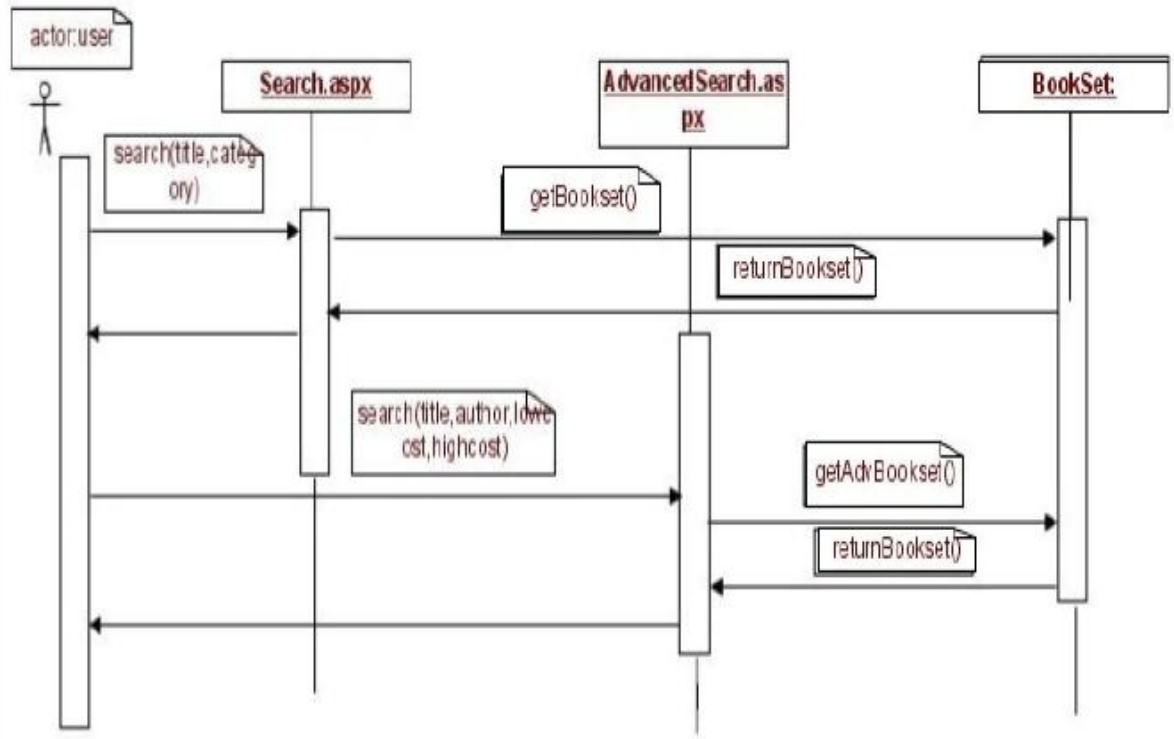
OBS Use Case Diagram:



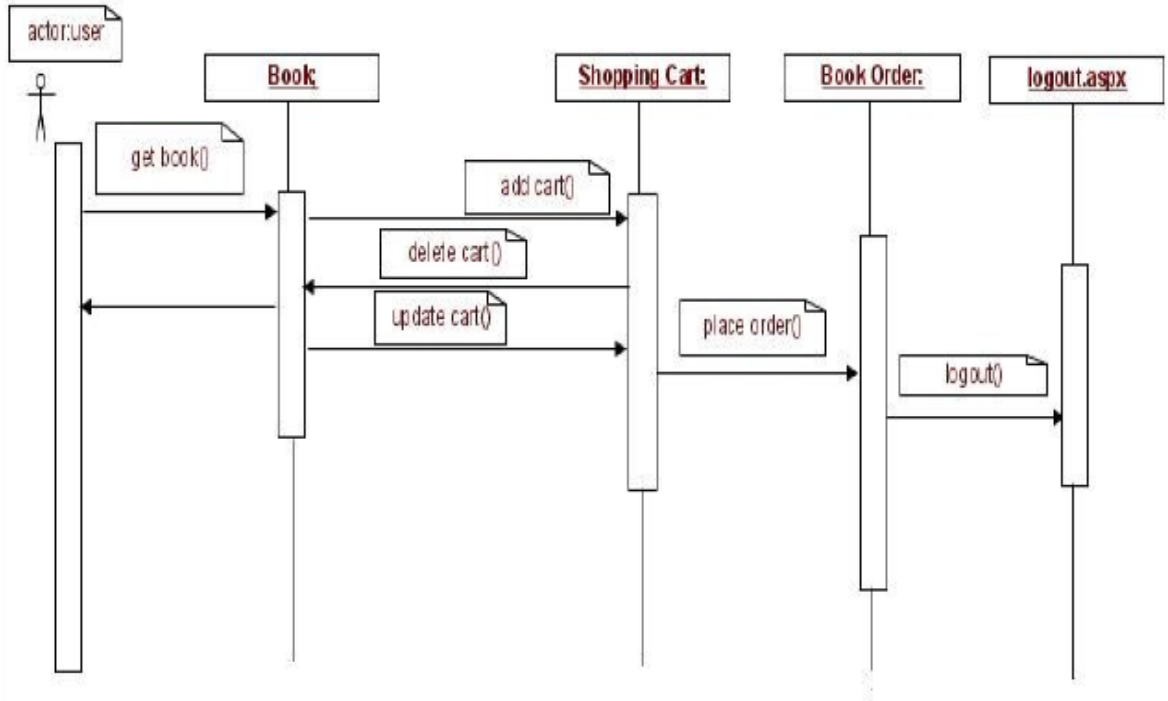
OBS User Login Sequence Diagram:



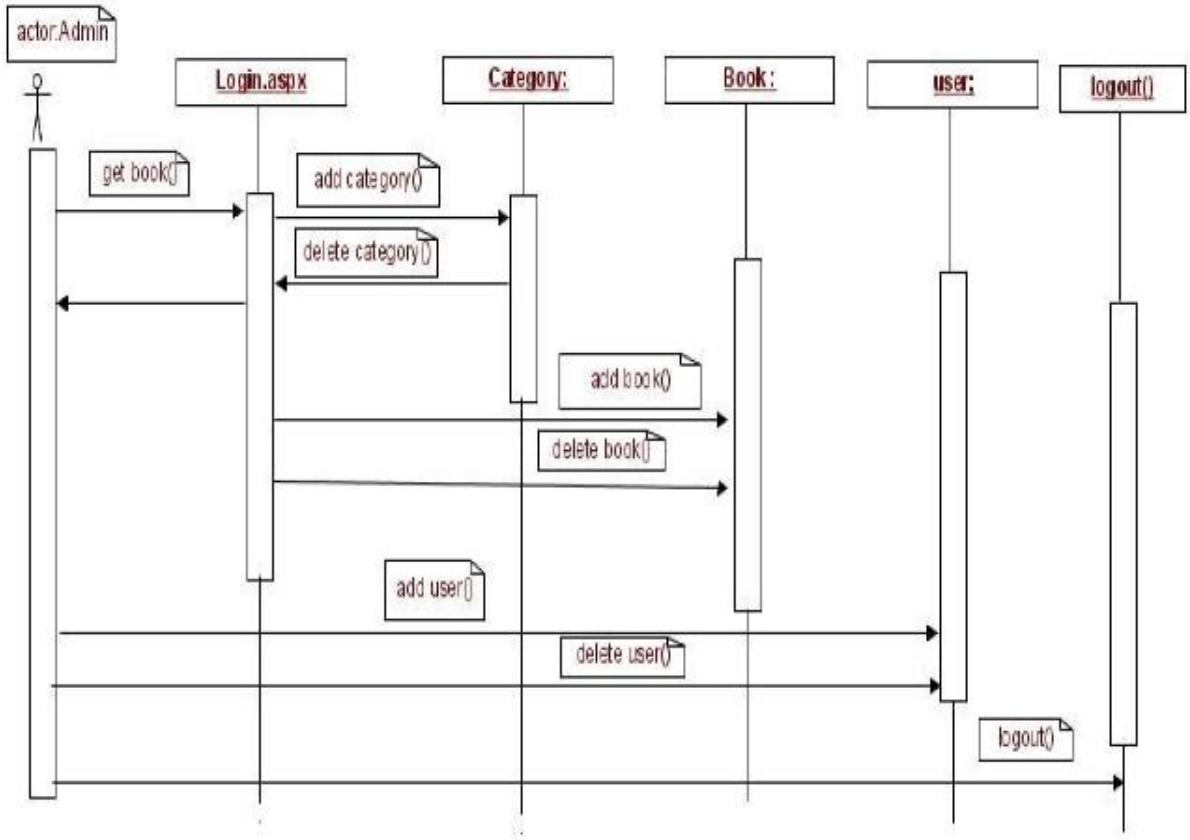
OBS Book Search Sequence Diagram:



OBS Add To Cart Sequence Diagram:

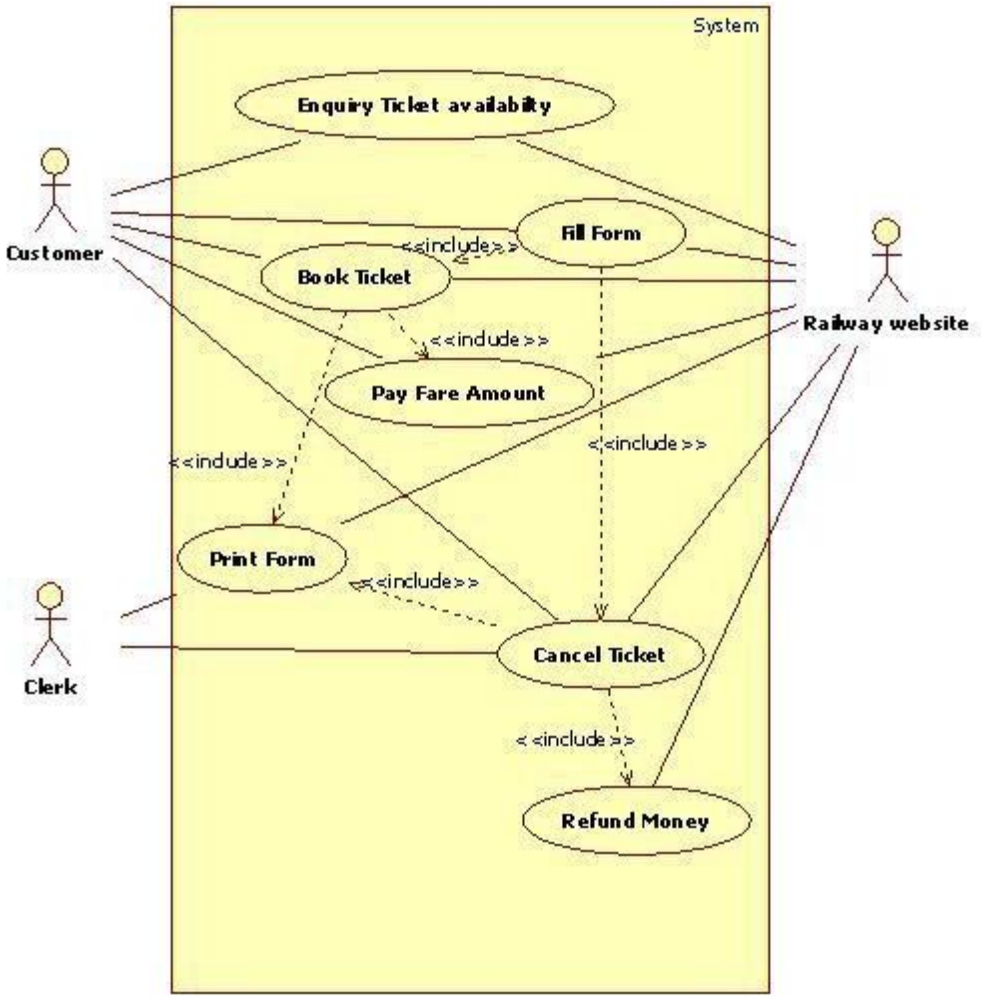


OBS Administration Sequence Diagram:

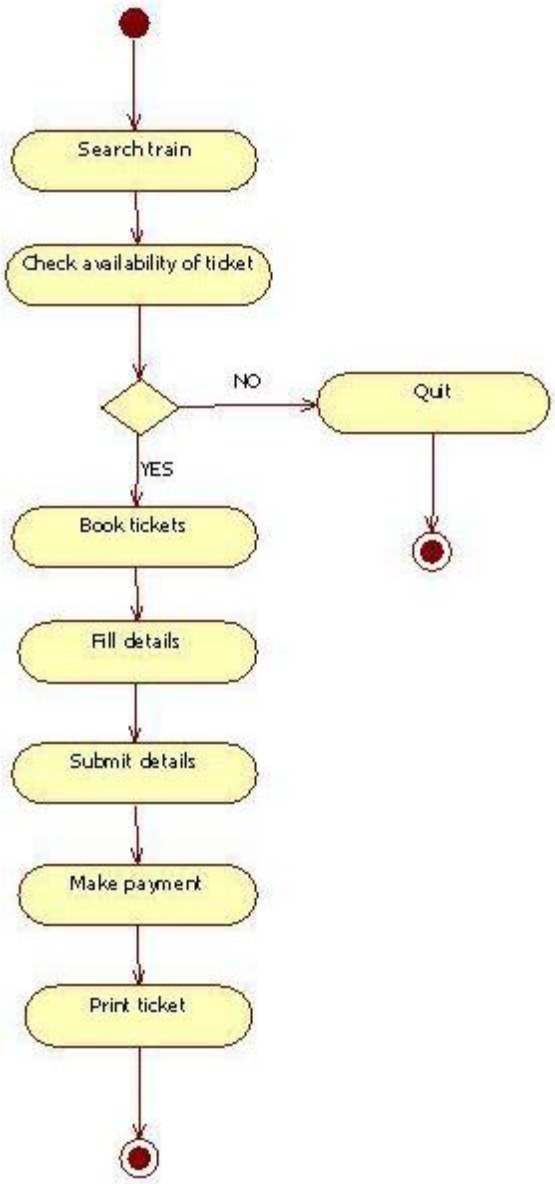


Railway Reservation System (RRS)

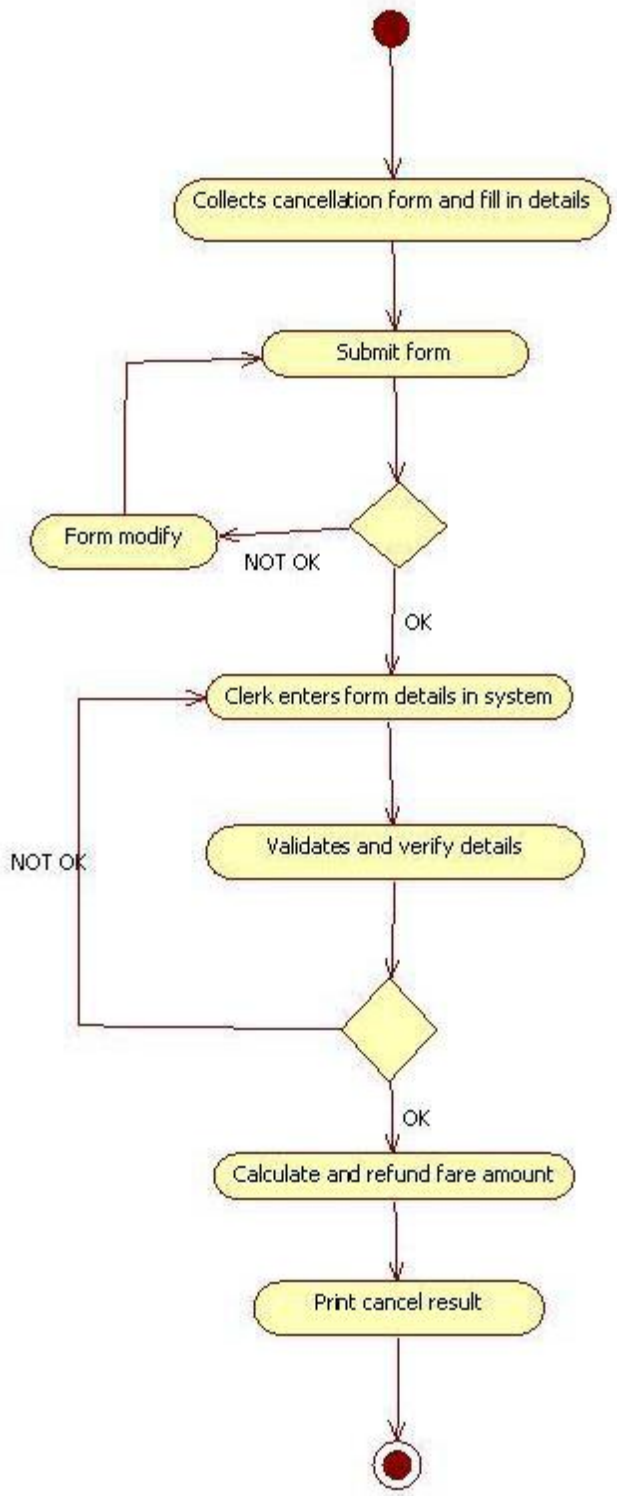
RRS Use Case Diagram:



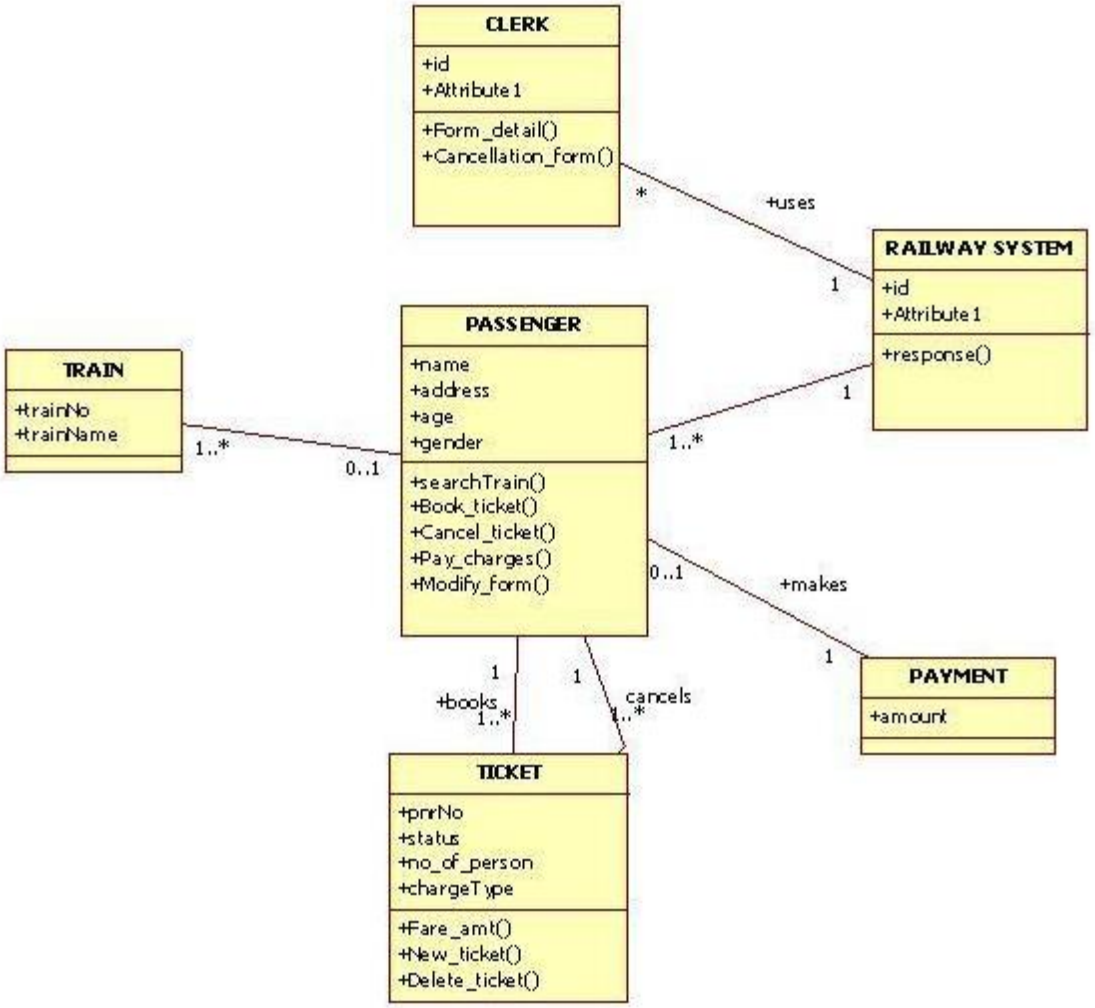
RRS Activity Diagram for Booking Ticket:



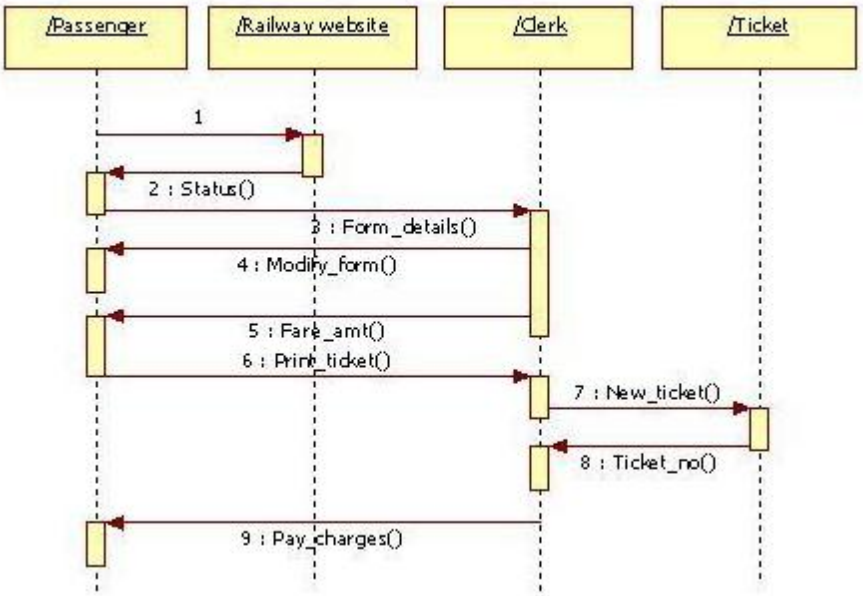
RRS Activity Diagram for Cancelling Ticket:



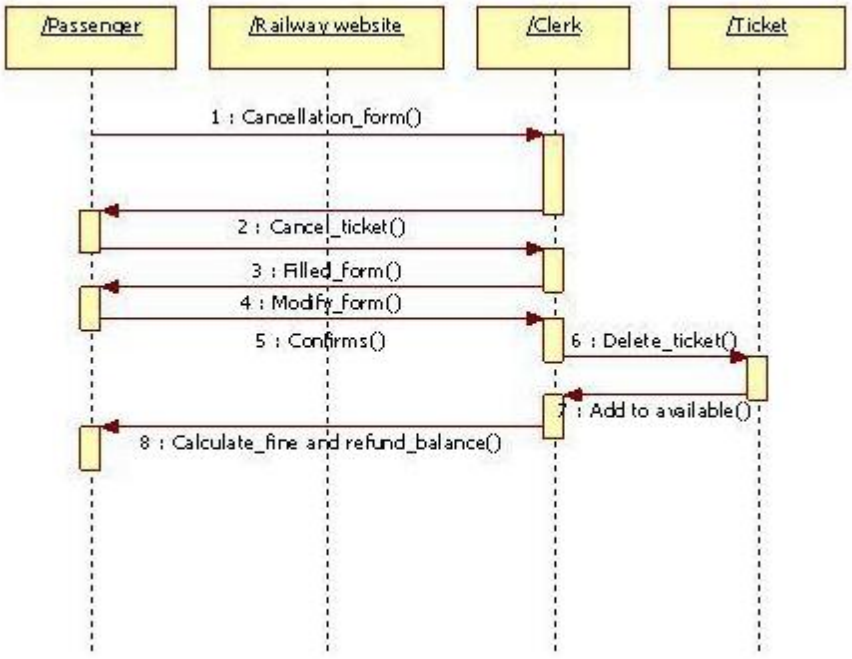
RRS Class Diagram:



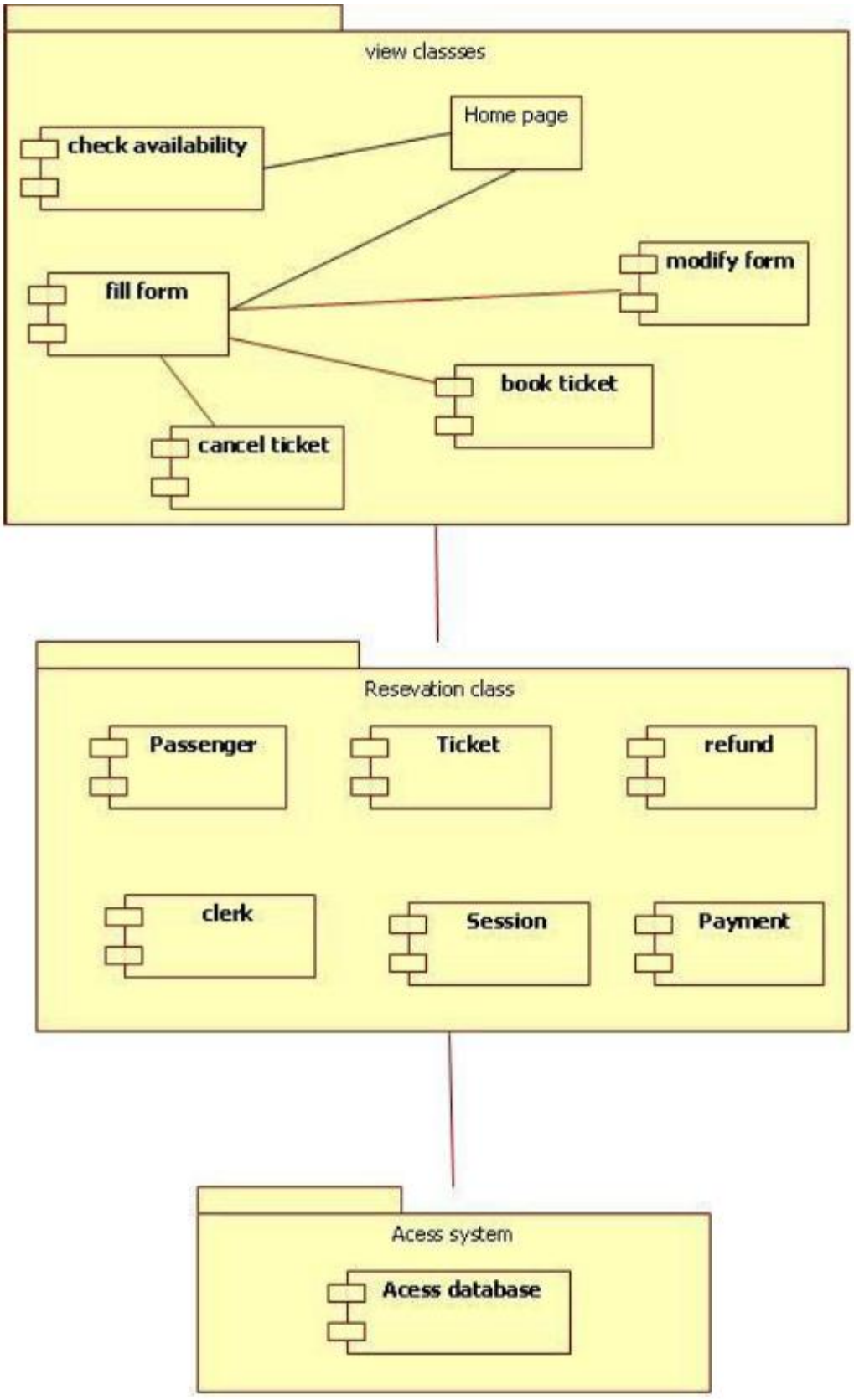
RRS Sequence Diagram for Booking Ticket:



RRS Sequence Diagram for Cancelling Ticket:

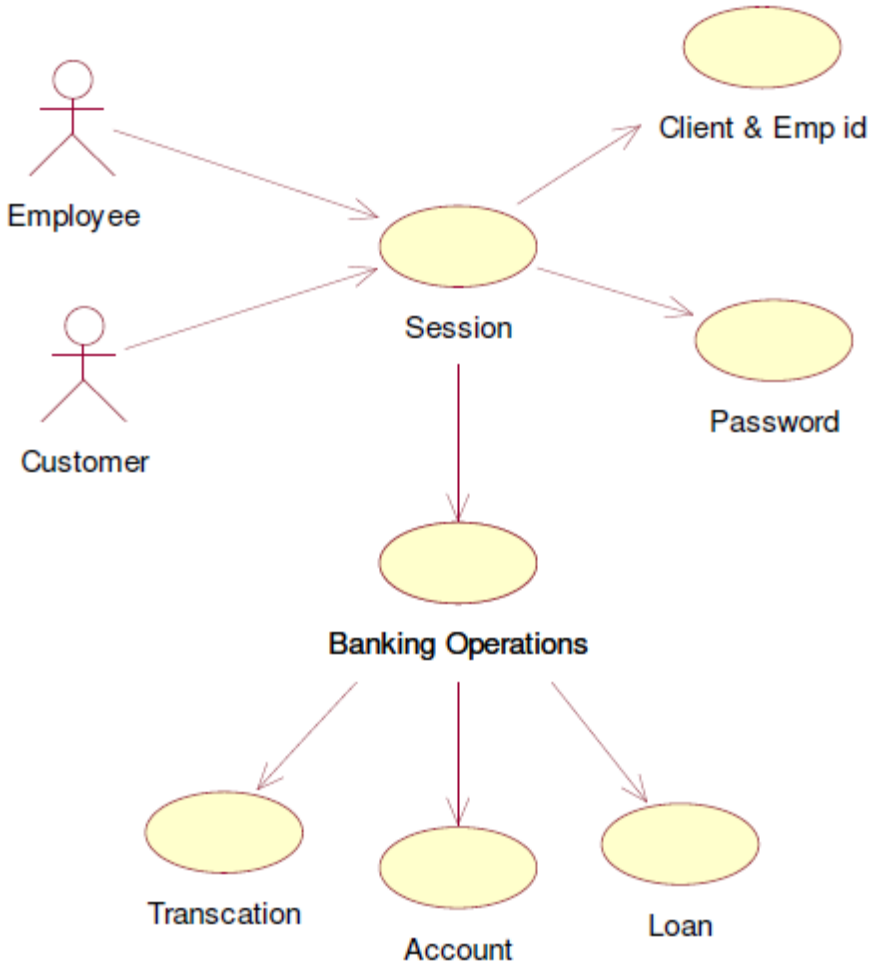


RRS Component Diagram:

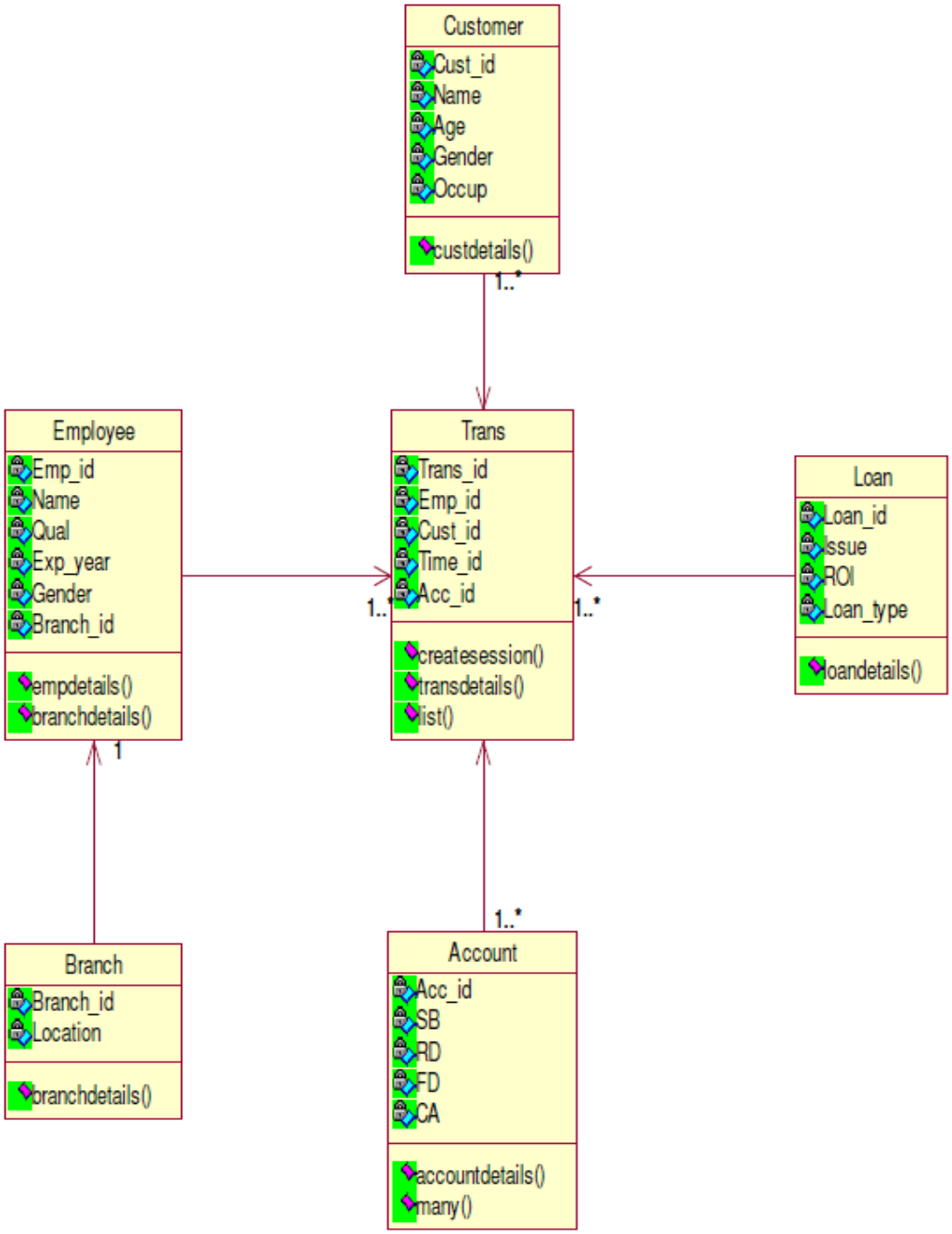


Banking System

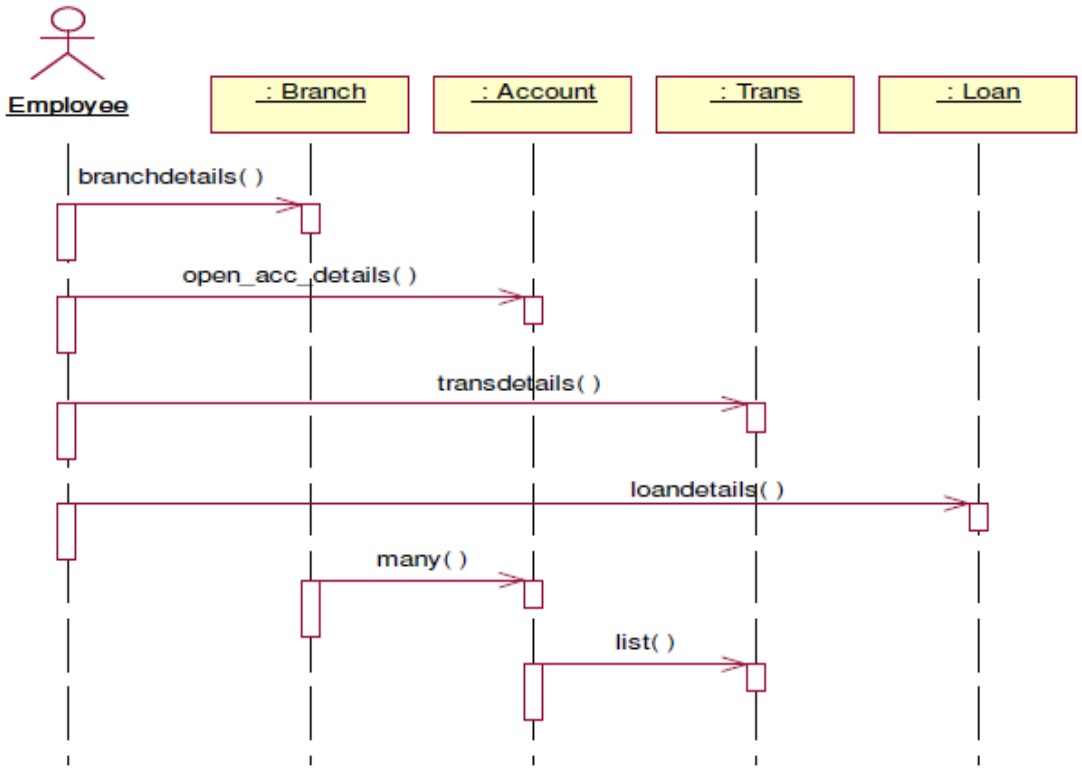
Banking System Use Case Diagram:



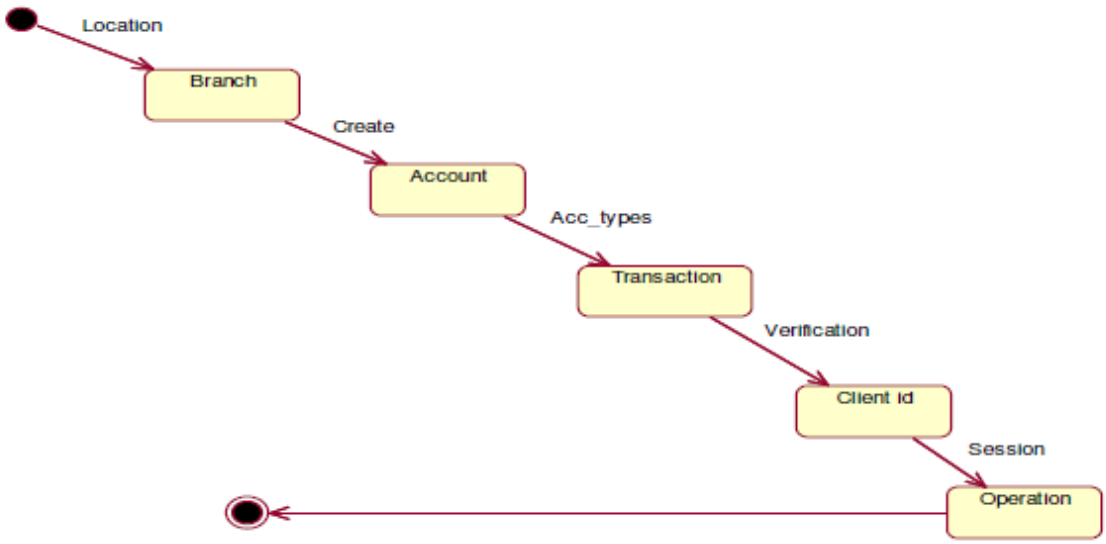
Banking System Class Diagram:



Banking System Sequence Diagram:



Banking System State Chart Diagram:



Design Patterns

Introduction

A design pattern is:

- a standard solution to a common programming problem
- a technique for making code more flexible by making it meet certain criteria
- a design or implementation structure that achieves a particular purpose
- a high-level programming idiom
- shorthand for describing certain aspects of program organization
- connections among program components
- the shape of an object diagram or object model

When (not) to use design patterns

The first rule of design patterns is the same as the first rule of optimization: delay. Just as you shouldn't optimize prematurely, don't use design patterns prematurely. It may be best to first implement something and ensure that it works, then use the design pattern to improve weaknesses; this is especially true if you do not yet grasp all the details of the design. (If you fully understand the domain and problem, it may make sense to use design patterns from the start, just as it makes sense to use a more efficient rather than a less efficient algorithm from the very beginning in some applications.)

Design patterns may increase or decrease the understandability of a design or implementation. They can decrease understandability by adding indirection or increasing the amount of code. They can increase understandability by improving modularity, better separating concerns, and easing description. Once you learn the vocabulary of design patterns, you will be able to communicate more precisely and rapidly with other people who know the vocabulary. It's much better to say, "This is an instance of the visitor pattern" than "This is some code that traverses a structure and makes callbacks, and some certain methods must be present, and they are called in this particular way and in this particular order."

Most people use design patterns when they notice a problem with their design — something that ought to be easy isn't — or their implementation — such as performance. Examine the offending design or code. What are its problems, and what compromises does it make? What would you like to do that is presently too hard? Then, check a design pattern reference. Look for patterns that address the issues you are concerned with.

Examples

Here are some examples of design patterns which you have already seen. For each design pattern, this list notes the problem it is trying to solve, the solution that the design pattern supplies, and any disadvantages associated with the design pattern. A software designer must trade off the advantages against the disadvantages when deciding whether to use a design pattern. Tradeoffs between flexibility and performance are common, as you will often discover in computer science (and other fields).

Encapsulation (data hiding)

Problem: Exposed fields can be directly manipulated from outside, leading to violations of the representation invariant or undesirable dependences that prevent changing the implementation.

Solution: Hide some components, permitting only stylized access to the object.

Disadvantages: The interface may not (efficiently) provide all desired operations. Indirection may reduce performance.

Subclassing (inheritance)

Problem: Similar abstractions have similar members (fields and methods). Repeating these is tedious, error-prone, and a maintenance headache.

Solution: Inherit default members from a superclass; select the correct implementation via run-time dispatching.

Disadvantages: Code for a class is spread out, potentially reducing understandability. Run-time dispatching introduces overhead.

Iteration

Problem: Clients that wish to access all members of a collection must perform a specialized traversal for each data structure. This introduces undesirable dependences and does not extend to other collections.

Solution: Implementations, which have knowledge of the representation, perform traversals and do bookkeeping. The results are communicated to clients via a standard interface.

Disadvantages: Iteration order is fixed by the implementation and not under the control of the client.

Exceptions

Problem: Errors occurring in one part of the code should often be handled elsewhere. Code should not be cluttered with error-handling code, nor return values preempted by error codes.

Solution: Introduce language structures for throwing and catching exceptions.

Disadvantages: Code may still be cluttered. It can be hard to know where an exception will be handled. Programmers may be tempted to use exceptions for normal control flow, which is confusing and usually inefficient.

These particular design patterns are so important that they are built into Java. Other design patterns are so important that they are built into other languages. Some design patterns may never be built into languages, but are still useful in their place.

Classification of Patterns

Patterns are classified by *purpose* and *scope*:

•Creational Patterns:

Creational patterns deal with the creation of objects and help to make a system independent of howobjects are created, composed and represented. They also enable flexibility in what gets created,who creates it, how it gets created and when it gets created.

• Structural Patterns:

Structural patterns deal with how objects are arranged to form larger structures

• Behavioral Patterns:

Behavioural patterns deal with how objects interact, the ownership of responsibility and factoring code in variant and non-variant components.

The scope is defined as :

• class - static relationships through class inheritance (white-box reuse)

• object - dynamic relationships through object composition (black-box reuse) or collaboration

Patterns Summary

There are 5 creational patterns, 7 structural patterns and 11 behavioural patterns :

		Purpose		
		Creational	***Structural***	***Behavioural***
Scope	***Class***	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Pros and Cons of using Design Patterns

Pros:

- Quality, flexibility and re-use
  Design Patterns capture solutions to common computing problems and represent the time, effort and experience gained from applying these solutions over numerous domains/iterations. Generally systems that use Design Patterns are elegant, flexible and have more potential for reuse
- Provide a common frame of reference for discussion of designs
- Patterns can be combined to solve one or more common computing problems
- Provide a common format for pattern specification
  Intent, Motivation, Applicability, Structure, Participants, Collaborations, Consequences

Cons:

- Complexity
  Design Patterns require a reasonable degree of study and can be difficult for some designers to grasp. Junior designers/developers may not have encountered Design Patterns and have to learn them before they can be productive on a project.

Creational Patterns Summary

Pattern	*Description*	*Pros/Cons*
Abstract Factory *"Provide an interface for creating families of related or dependent objects without specifying their concrete classes"*	A "family" of abstract create methods (each of which returns a different *AbstractProduct*) are grouped in an *AbstractFactory* interface. *ConcreteFactory* implementations implement the abstract create methods to produce *ConcreteProducts*.	Pros. • shields clients from concrete classes • easy to switch product family at runtime – just change concrete factory • "keep it in the family" – enforces product family grouping Cons. • adding a new product means changing factory interface + all concrete factories
Builder *"Separate the construction of a complex object from it's representation so that the same construction process can create different representations"*	An appropriate *ConcreteBuilder* (implements *Builder)* is constructed and associated with a *Director*. The *Director* traverses an object graph and passes each object to the Builder. The Builder uses each object to build-up a complex *Product* over time. When the object graph has been fully traversed, the final Product can be retrieved from the Builder.	Pros. • separates complex construction from (re)presentation • shields the Director from the algorithm and internal structure used to build the Product • enables a consolidated Product to be built up over time – e.g. the Product requires info from multiple sources, info available at different times
Factory Method *"Define an interface for creating an object but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses"*	An abstract *Creator* class defines an abstract create method (or provides a default create method) which returns an abstract *Product*. A *ConcreteCreator* class implements the abstract create method to return a *ConcreteProduct*. This enables the Creator to defer Product creation to a subclass. *N.B. Factory Method is often used in the Abstract Factory pattern to implement the create methods*	Pros. • shields clients from concrete classes • if a framework uses the Factory Method pattern, it enables third-party developers to plug-in new Products • the Creator create method can be coded to return a default Product Cons. • coding a new Product means writing *two* classes – one for the concrete Product and one for the concrete Creator • static – inheritance based
Prototype *"Specify the kinds of objects to create using a prototypical instance and create new objects by copying this prototype"*	Objects implement the *clone ()* method of the *Prototype* interface by returning a copy of self. A client maintains a registry of Prototype instances. When a new instance is required, the client invokes clone ()	Pros. • shields clients from concrete classes • the object is the factory - i.e. Product and Creator combined (saves coding a Creator for every Product) • pre-configured object instances – instead of create/set member vars every time Cons. • every Prototype instance has to implement clone () which may not be easy – e.g. circular references, contained elements don't support copying, large number of classes to be retrofitted, etc.
Singleton *"Ensure a class has only one instance and provide a global point of access to it"*	A Singleton is defined with a static getInstance () method, a protected constructor and any other required instance methods. As the constructor is protected, the only way to obtain an instance is through the static getInstance () method. This serves to control the number of instances created/in use by clients.	Pros. • controls access to the instance(s) • controls the number of instances • more flexible than a static class - the instance(s) constructed in getInstance () can be a subclass so method overrides are allowed (can't use method overrides with a static class).

Structural Patterns Summary

Pattern	*Description*	*Pros/Cons*
Adapter *"Convert the interface of one class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces"*	A concrete *Adapter* class implements methods defined in a *Target* interface by wrapping calls to methods in a concrete *Adaptee* (and also provides equivalent functionality for required Target methods if they don't exist in Adaptee). A *Class Adapter* uses multiple inheritance of Target and Adaptee. With an *Object Adapter*, Adapter contains an Adaptee and forwards requests.	Pros. • enables interoperability – especially useful when using one or more third-party class libraries in your code • highlights the Target "contract" – e.g. if shipping reusable components, include a default adapter for use by clients • *object adapter* – a single Adapter can adapt many Adaptees (including subclasses) • *class adapter* – automatically inherit Adaptee methods; inherited methods can be overridden Cons. • type adapted – to the outside world, the Adaptee looks like an Adapter (can't pass to Adaptee methods unless a two-way adapter is implemented) • *object adapter* – need to write tedious method mapping/delegation code • *class adapter* – need to provide an adapter for each subclass • *class adapter* – multiple-inheritance of potentially similar interfaces (risk of method name collisions)
Bridge *"Decouple an abstraction from its implementation so that the two can vary independently"*	*Abstraction* (an abstract base class) provides core functionality for it's subclasses by aggregating *primitive* methods from an *Implementor* (an abstract class/interface) into high-level methods. *ConcreteImplementor* classes provide specific implementations of the primitive methods. This facilitates a clean separation between elements that are common (e.g. Window.draw ()) and elements that are specific (e.g. XWindow.draw ())	Pros. • decouples abstraction from implementation – clean separation between common aspects and specific differences • extensible – abstraction and implementation can evolve independently • shields clients from concrete classes – a change in the implementation doesn't require the client to be updated • implementation can be swapped at runtime
Composite *"Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly"*	*Component* (an abstract base class) is sub-classed into either a *Leaf* or a *Composite*. A Composite contains one or more Components – i.e. a Leaf or another Composite. This enables a client to view a single item or a group of items as one type – a Component.	Pros. • facilitates uniform view - clients are shielded from details of whether a Component is a Leaf or Composite • easy to add new components – everything referenced by Component Cons. • referring to either as Component makes it too general – can't control what Components make up a Composite without explicitly checking

Pattern	*Description*	*Pros/Cons*
Decorator "Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality"	*ConcreteDecorator* (subclass of *Decorator*) classes wrap *ConcreteComponent* (subclass of *Component*) classes to transparently extend their functionality. This is achieved by added functionality before/after dispatching method calls to the Component. Transparency is achieved as the Decorator interface matches the Component interface	Pros. • more flexible than inheritance - functionality can be extended on an instance basis, at runtime, etc. • promotes reuse – a Decorator can enhance anything that implements *Component* • enables recursive composition – can construct a chain of Decorators Cons. • too transparent – a Decorator looks just like the original Component • difficult to conceptualise – lots of fine-grained Decorators connected in lots of different ways
Facade *"Provided a unified interface to a set of interfaces in a sub-system. Facade defines a higher-level interface that makes the sub-system easier to use"*	A *Facade* provides a simplified view of a complex object model by aggregating methods from multiple *subsystem classes* into a few high-level methods. Communication is one-way – the Facade knows about the subsystem classes but the subsystem don't have any knowledge of the Facade.	Pros. • shields the client from the complexity of the subsystem • decouples the client from the subsystem – relationship management is externalised to the Facade • performance - batch several method calls into one • control – provides a central point to exercise control
Flyweight *"Use sharing to support large numbers of fine-grained objects efficiently"*	A pool of common objects (*Flyweights)* are shared by splitting the object state into static (*intrinsic*) and instance specific (*extrinsic*) components. When invoking methods on the Flyweight, the client must pass the extrinsic state in the method. The pool is managed by a *FlyweightFactory* which ensures that objects are added to the pool on first request and retrieved from the pool thereafter.	Pros. • support a large number of clients using a relatively small pool Cons. • overhead – have to pass in extrinsic state each time
Proxy *"Provide a surrogate or placeholder for another object to control access to it"*	A common *Subject* interface is defined and implemented by a *RealSubject* class and a *Proxy* class. The Proxy acts as a middle-man between the client and the RealSubject. As far as the client is concerned, the Proxy looks identical to the RealSubject (it's transparent).	Pros. • provides a layer-of-indirection between the client and the RealSubject which can be used to implement a variety of useful features (load-on-demand, location transparency, access control, reference counting) Cons. • too transparent – as the Proxy is transparent, the client isn't aware of how the Proxy should be used (e.g. with location transparency, every method call is a remote call)

Behavioral Patterns Summary

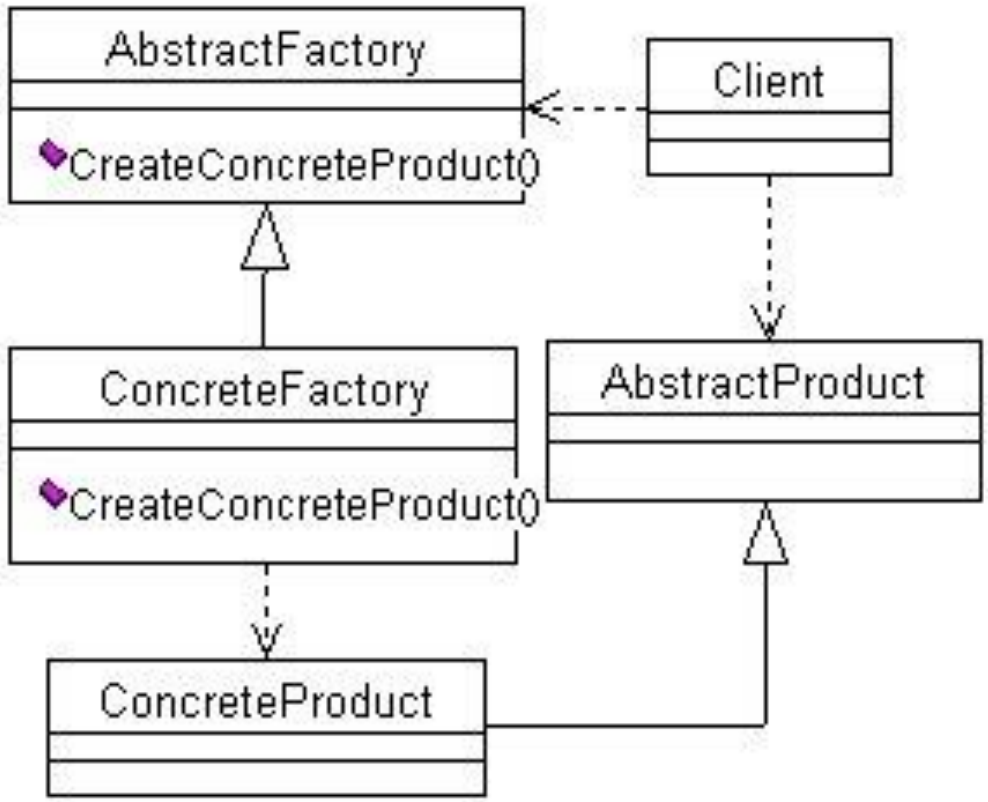
Pattern	*Description*	*Pros/Cons*
Chain of Responsibility *"Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it."*	Decouples the sender of a request from the "ultimate" receiver. The request is passed along a chain of potential *Handler*s until one of them deals with it. If a handler doesn't wish to deal with the request, it passes the request to it's *successor*	Pros. • reduced coupling • flexible responsibility – handling the request is optional Cons. • the request may get handler by the default handler which may not know what to do with it
Command *"Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations."*	The purpose of the Command pattern is to decouple an event generator (the *Invoker*) from the event handler (the *Receiver*). A *ConcreteCommand* class (sub-classed from *Command*) defines an *execute ()* method which calls the appropriate method on the Receiver (the *action* method). The client is responsible for associating the Receiver with the Command and then the Command with an Invoker. *N.B. 1:1:1 mapping between Invoker, Command and Receiver.*	Pros. • decouples Invoker from Receiver – makes Receiver more re-usable as it doesn't manage the relationship with the Invoker • Command encapsulate a request – requests can be stored so they can be undone, processed at a later time, etc. • extensible – easy to add new Commands • macros – commands can be grouped into *macros* so that multiple commands can be run at once • dynamic – e.g. different Commands, multiple Invokers, decide at runtime, etc. Cons. • can't centralise related action methods in one Command class - only one method is used (execute ())
Interpreter *"Given a language, define a representation for its grammar along with an interpreter that use the representation to interpret sentences in the language."*	***Don't care***	

Pattern	*Description*	*Pros/Cons*
Iterator *"Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation."*	A common OO requirement is traversal of an aggregate structure. The implementation of the traversal is factored out of the *Aggregate* class into an *Iterator*. The Factory Method pattern is used by a *ConcreteAggregate* to create a *ConcreteIterator*. The ConcreteIterator keeps track of the "current" position. The same interface is used to iterate regardless of the underlying aggregate structure.	Pros. • shields the client from the aggregrate's internal representation • the aggregate can be iterated in many different ways (i.e. multiple ConcreteIterators) • more than one iterator can be active – the iterator stores the current state so each is self contained • simplifies the ConcreteAggregate code – iterator is in a separate class Cons. • uses Abstract Factory so have to define a ConcreteAggregate in addition to the ConcreteIterator • if the underlying aggregate is updated while using an Iterator, the operation of the Iterator may be undefined.
Mediator *"Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently."*	A collection of related classes called *Colleague*s (sub-classed as *ConcreteColleague*) need to inform each other when an event occurs. Rather than couple every colleague to every one of it's peers, each Colleague publishes the event to a *Mediator* (sub-classed as *ConcreteMediator)*. The Mediator then republishes the event to the other Colleagues. Communication is therefore two-way – the Mediator knows about the Colleagues and vice-versa.	Pros. • promotes a loose coupling between the Colleagues – instead of a Many:Many publish, it's a Many:1 (Colleagues to Mediator) followed by a 1:Many (Mediator to Colleagues) • promotes reuse – Colleagues aren't bogged down with relationship management code so can be reused in other circumstances • centralizes relationship management in the Mediator Cons. • the Mediator can become very complex and difficult to maintain
Memento *"Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later."*	Uses an *Originator* (managed a contained Memento obj), *Memento* (snapshot of originator state, preserves encapsulation) and a *Caretaker* (manages Memento objects)	Pros. • preserves encapsulation • state can be stored and reloaded later on
Observer *"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."*	One or more *Observer*s (sub-classed as *ConcreteObserver)* can be registered with a *Subject* (sub-classed as *ConcreteSubject*). When the state of the Subject changes, all registered Observers are notified. Two notification models are available : *push* (the state change is sent with the notification) and *pull* (the notification is the event only, if the Observer wants to see the state change it requests it from the Subject)	Pros. • abstract coupling of Subject and Observer – Subject doesn't care what an Observer does with the event, just notifies it • supports broadcast – in theory, any number of Observers can be supported (doesn't work in practice) Cons. • the client to the Subject works in isolation – isn't aware that setting the state could cause a cascade of event notifications

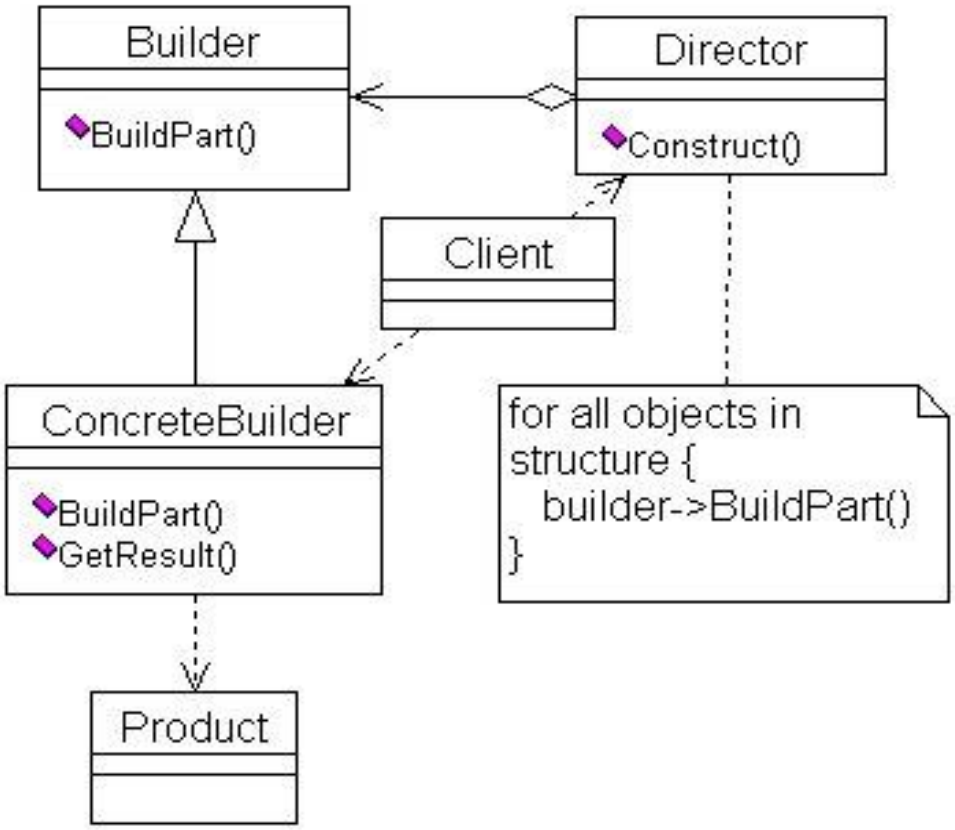
Pattern	*Description*	*Pros/Cons*
State *"Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class."*	A common *State* class is subclassed for all possible states. Each subclass restricts the operation of common methods based on it's state. Current state is stored in a *Context*; next state is return by the current State subclass when *handle ()* is called	Pros. • collects actions + transitions into state specific classes Cons. • doesn't scale – i.e. if large num of states/actions
Strategy *"Define a family of algorithms, encapsulate each one and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it."*	Algorithms are defined as *Strategy* classes. Related algorithms are grouped into a family of Strategy classes. A *StrategyContext* class contains all required info for the algorithm defined in the Strategy and the two classes work in conjunction to execute the algorithm.	Pros. • a family of Strategy classes is available – pick the most suitable or as directed by an external decision making process • simplified/cleaner code – instead of lots of "if" statements or subclasses each implementing an algorithm, one "dispatcher" class can provide all relevant Strategy's on request Cons. • clients must know the classes available in the family - clients instantiate Strategy instances when the StrategyContext is created
Template Method *"Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changes the algorithms structure"*	Capture the *invariant* behaviour of an algorithm in an abstract base class using high-level methods. Define the *variant* behaviour as *abstract primitive methods* so that concrete sub-classes can provide implementations. The high-level methods are defined using a combination of primitive methods and methods defined in the abstract base class. *N.B. basically a behavioural version of the Factory Method*	Pros. • shields the client from the details of the variant behaviour • quality & productivity – only the variant behaviour needs to be implemented
Visitor "Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates"	A client traverses an object graph and for each element invokes *accept (Visitor v)* which in turn calls back on the visitor with itself – i.e. *v.visit (this)*. Consequently the Visitor gets notified when an object is traversed and what type the traversed object is. Each Visitor subclass has to support every type of object that will be traversed	Pros. • cleaner code – factors out type specific event handling from classes and centralises it in a Visitor • easy to add a new "operation" for all *Visitable* classes – an operation is implemented as a Visitor subclass with a handler method for each Visitable object type • can traverse multiple object types in the same traversal – unlike Iterator which can only traverse one type at a time • useful for running a variety of reports – without Visitor every class that you'd want to report on would have to have a custom method per report Cons. • if a new Visitable class is added, all Visitor subclasses have to be extended to support it • might break encapsulation - the Visitor needs access to the elements details

Experiments

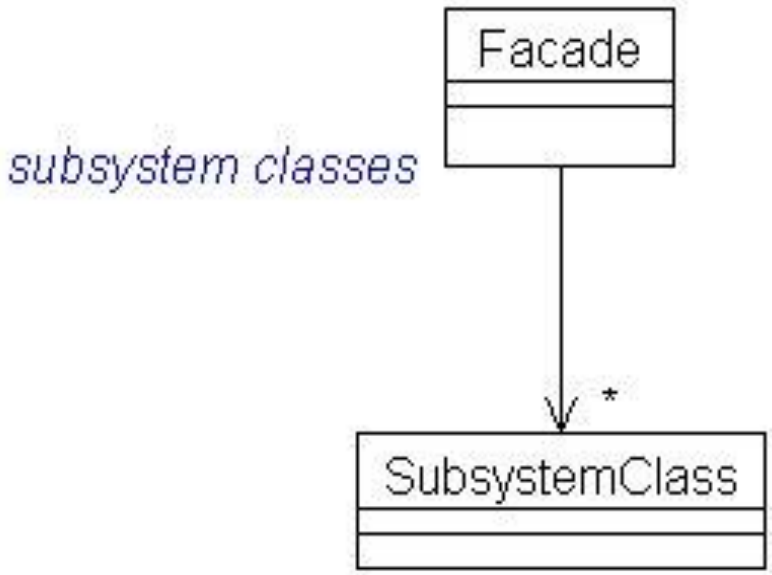
UML design for Abstract Factory design pattern



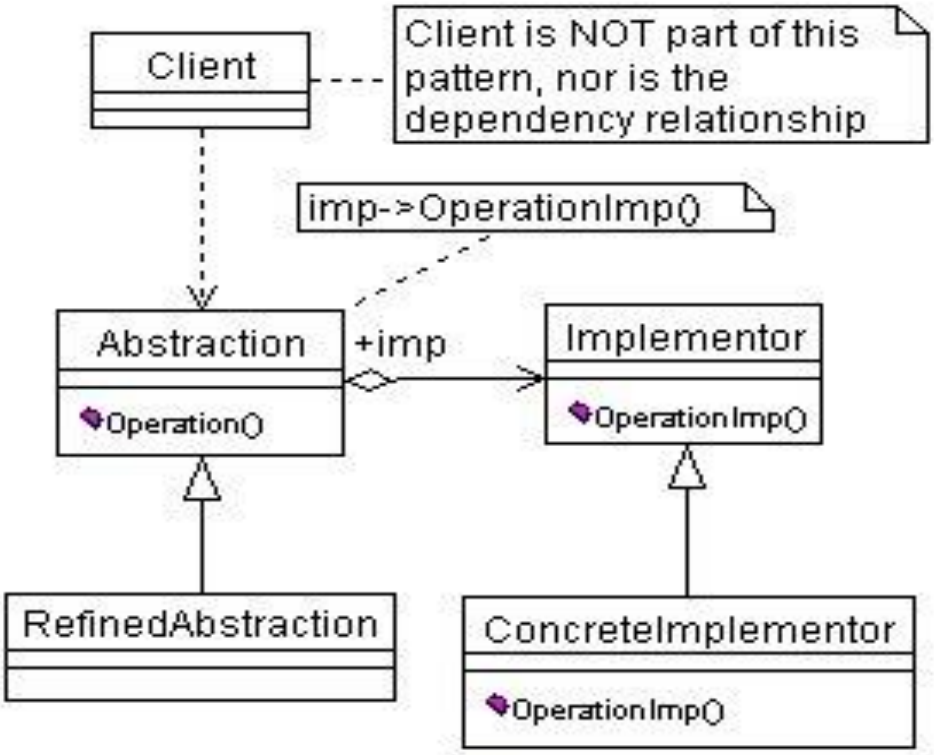
UML design for Builder design pattern



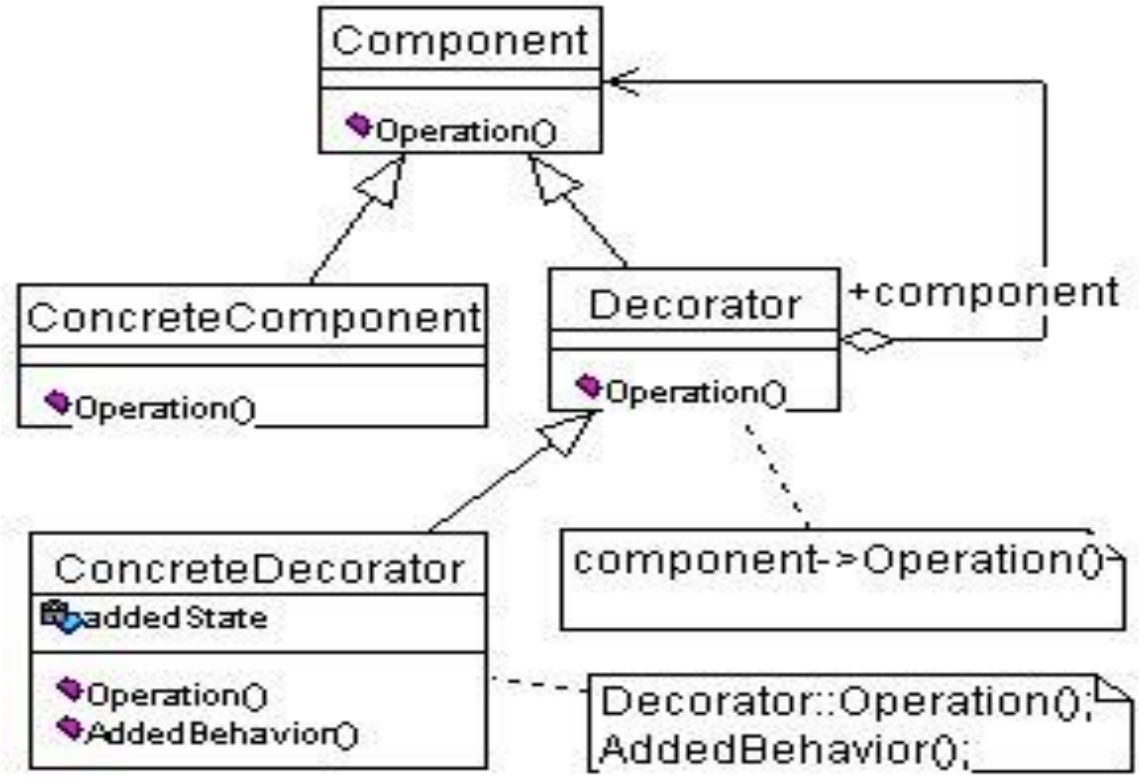
UML design for Façade design pattern



UML design for Bridge design pattern



UML design for Decorator design pattern



Class diagram for Document Editor