Unit III

3 Advanced State Modeling and Interaction Modeling

Syllabus

Advanced State Modeling : Nested state diagrams; Nested states; Signal generalization; Concurrency; A sample state model; Relation of class and state models; Practical tips. Interaction Modeling : Use case models; Sequence models; Activity models. Use case relationships; Procedural sequence models; Special constructs for activity models.

Contents

Part I : Advanced State Modeling

3.1 Nested States

The nested states are the states that occur within the one super state. For example in the following state machine diagram, the **Active** state is a super state in which the sub states such as **Display, Reading Card Validating PIN** and so on are present. When the Perform shutdown event occurs then the system is restored to the **Idle** state from the **Active state.**

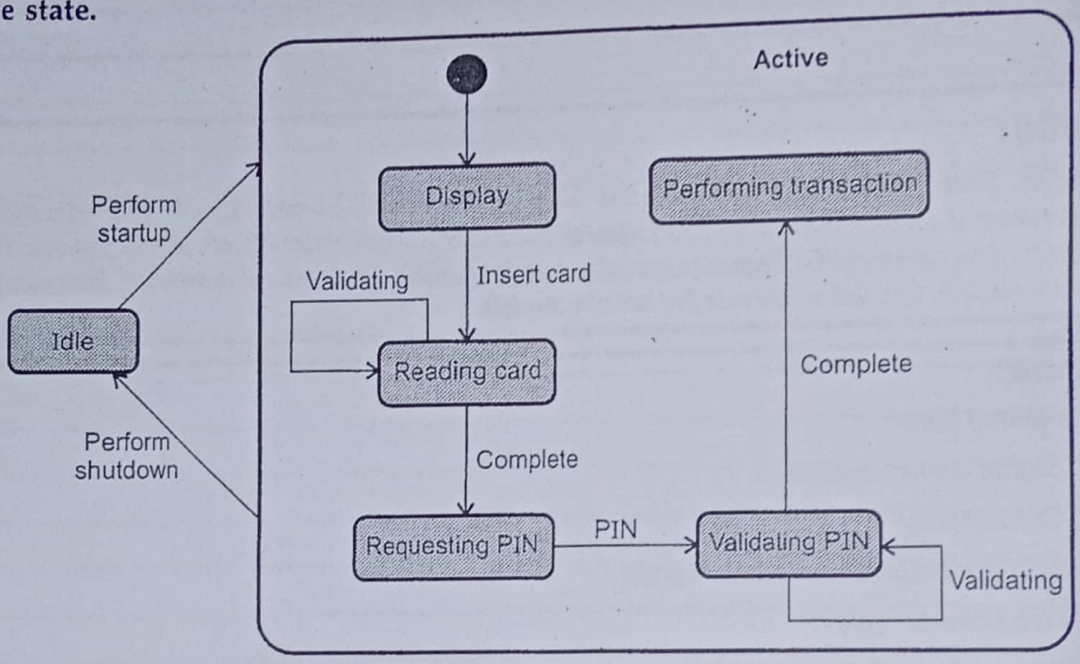


Fig. 3.1.1 ATM machine

Review Question

1. *Explain nested states with appropriate diagram.*

3.2 Signal Generalization

In the event driven systems signal events are hierarchical. Following are the steps that are followed for modeling the family of signals -

- Consider various types of signals to which the active objects may respond.
- Find out the common types of signals and arrange them in generalization specialization hierarchy. The general type of signals will be arranged at the high level of the hierarchy and the specialized type of signals will be arranged at the lower level of the hierarchy.
- Identify the situations for representing the polymorphism relationship and adjust the hierarchy accordingly.

- Example - Following is a hierarchy of signals that represent the signals that occur when computer gives the beep sound as a signal on occurrence of some problem.

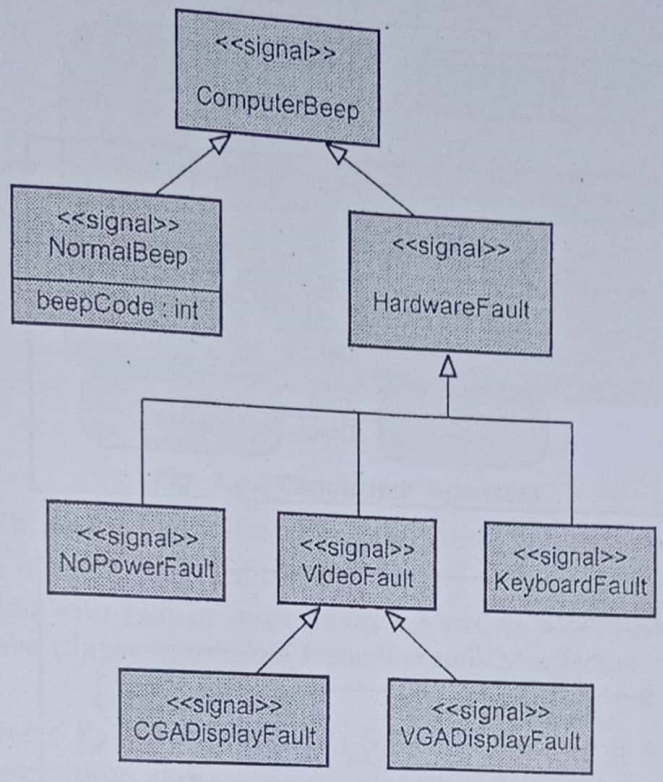


Fig. 3.2.1 Modeling families of signal

Review Question

1. *How signal can be organized into generalization hierarchy with inheritance of signal attributes ?*

3.3 Concurrency

The concurrency among the objects can be represented by the state diagram. Although the objects work concurrently they are not completely independent. In this section we will discuss various types of concurrency.

3.3.1 Aggregation Concurrency

- Aggregation concurrency occurs when the state diagram is a collection of state diagrams one for each part.
- Aggregation represents **and relationship.**
- The objects within the aggregation concurrency can change the state independently.
- Due to aggregation concurrency the modularity can be brought in the design.

- For example -

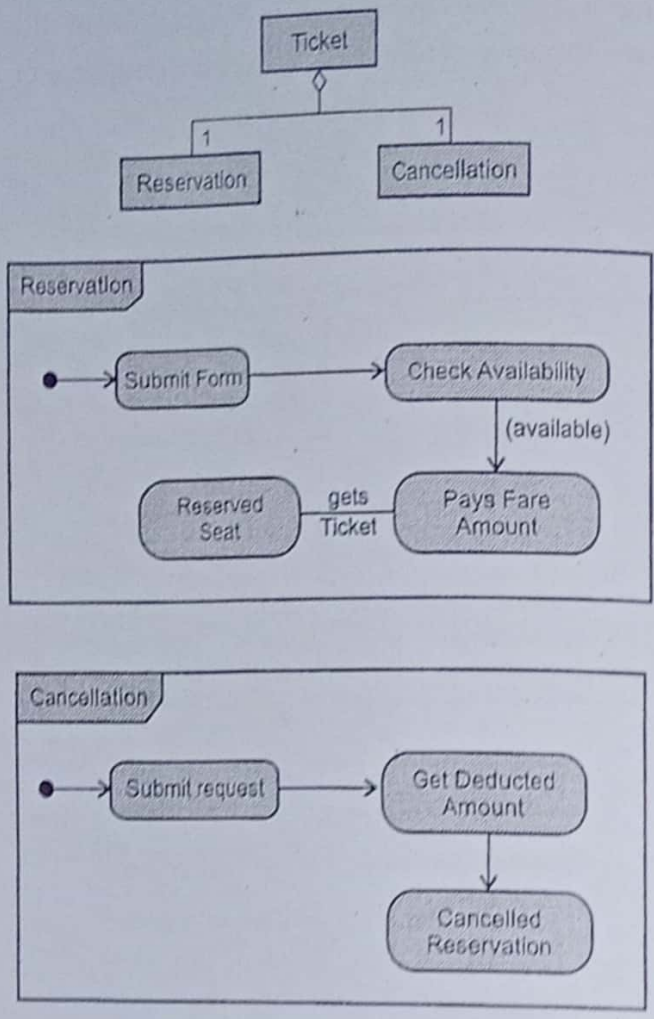


Fig. 3.3.1 Aggregation concurrency

3.3.2 Concurrency within an Object

- The concurrency can be represented by partitioning the composite state into regio with dotted line.
- These regions execute in parallel in context of enclosing object.
- For example - In Document Editing Software, the printing and displaying document on an output device can be two parallel activities that can execu simultaneously. (Refer Fig. 3.3.2 on next page)

3.3.3 Synchronization(Concurrent Behavior of Objects)

When objects perform activities concurrently, they execute in parallel and completion of all the concurrent activities the object enters in a new state. That mea there exist two controls for the execution of these activities - **splitting control** a **merging control.**

- The transitions that are transiting from single state to several orthogonal states called fork.

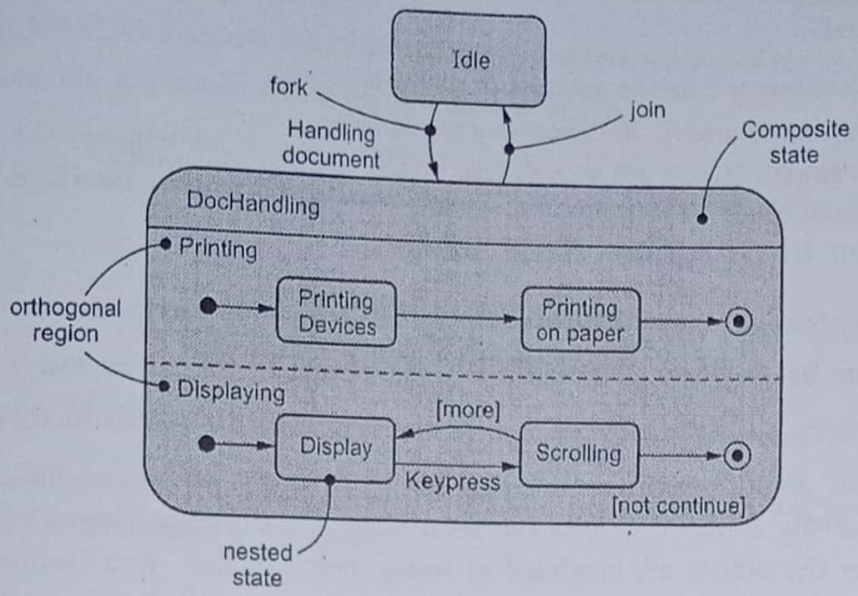


Fig. 3.3.2 Concurrent substates

It is shown with heavy black line with one incoming arrow and several outgoing arrows each to one of the orthogonal states.

- Join is a transition with two or more incoming arrows and only one outgoing arrow. The join may have trigger event. Join transition will be effective only if all the sources are active.
- For example - For a Railway Reservation System following is a state diagram which illustrates the Reservation state.

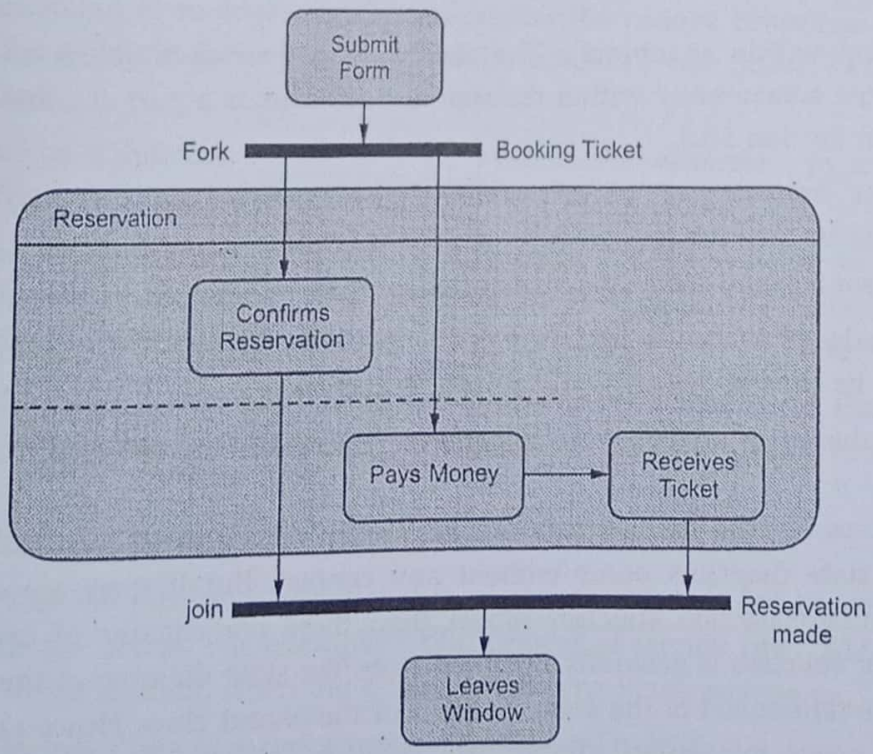


Fig. 3.3.3 Fork and join transaction

Review Questions

1. *What is concurrency ? Explain following concept with example.*
   i) Aggregation concurrency ii) Concurrency within object.
2. *Explain with state diagram how an object can perform activities concurrently ?*

3.4 Relation of Class and State Models

- The state diagram describes the behavior of the objects of a given class.
- The states can be compared with classes as the objects having links and values.
- There is a strong co-relation between state structure and the class structure.
- Normally an object has different states but it does not belong to different classes. Hence extensively different objects can be modeled as different classes but temporary differences of the objects are modeled as states. For instance - In a Online reservation system Ticket and Customer can be modeled as separate classes (Both denote extensive difference between objects) but Booking a Ticket or Cancellation of Ticket are modeled as separate states(both denote temporary difference of object).
- **Concurrency within states model and class model :** A composite state is a collection of concurrent substates. Whereas, there are three sources of concurrency within the class model.
  1. **Aggregation of object :** Each part of aggregation has its own state model. Refer section 3.3.1.
  2. **Aggregation within an object :** The objects have their own values and link and they execute concurrently within the state model. This is a part of composite object state. Refer section 3.3.2.
  3. **Concurrent behavior of objects :** The concurrent behavior of the objects is represented in the state model. Refer section 3.3.3.
- These three sources are used interchangeably.
- The state model of a class is inherited using the subclasses of that class model. Each subclass can be represented as a separate state diagram.
- **How does the state diagram of superclass interact with the state diagram of subclass ?**

If the attributes of both the subclass and superclass are different then the interaction between these state diagrams occur without any conflict. But if there are same set of attributes for superclass and subclass model then there are chances of conflicts. Any state diagram of subclass is generally obtained from the state diagram of the superclass. In short it is the refinement of the state diagram of the parent class. Hence new states or transitions are never introduced in the state diagram of the superclass, instead, this introduction of new states or transitions is done at the subclass level.

- The **states** in the state diagram are obtained from the interaction of the objects.
- The **transitions** are implemented as the operations of objects.
- The **signals** can be defined across different classes. These are more expressive than the operations. However the signals are dependant upon both the classes as well as states.

Part II : Interaction Modeling

3.5 Use Case Models

Use case model focus on the behavior of the system. It exposes the functionalities that can be provided by the system to its user.

3.5.1 Actor

- **Actor** represents the role when users interact with the use cases.
- It is direct external user of the system. It is object or set of objects that communicates directly with the system but it is not the part of the system.
- Each object defines the manner in which the behavior of the system is expected. For example the actor **Student** will enroll for the course in a **course registration system**.
- An object can bound to multiple actors to represent the various behavioral aspects of the system. For example the object **Book** can be bound to two actors - **Librarian** as well as **Student**.
- The actor can be a human, device or other system. For example - in a Company Information System, an Employee is an actor.
- An actor has **well defined purpose**. That means the actor helps to represent particular behavior of the system. For example the **Passenger** will interact with the **Reservation System** in order to make the reservations or to cancel the reservation.
- Actors will also help to represent the identity of different objects and the scope of these objects. The actors can be directly or indirectly connected to the system.

3.5.2 Use Cases

- Use **case** specifies the particular **functionality** of the system that can be obtained by interacting with the actors. For example - Withdrawal of money from ATM system, Issueing book to the student, reserving a seat, purchasing items and so on.
- Each use case includes **one or more actors** or the system itself.

- Use case includes a **sequence of messages** among the system and the actors. For example - For Course Reservation System the use case for Reservation of Course will include following sequence of messages.
  - Student selects the option "Reservation".
  - Student scrolls through the available courses and then selects the desired course.
  - Student Fills up the Form.
  - Student Submits the Form.
- In use cases, there can be fixed sequence of messages or the messages might have variations. For example depending upon the availability of course the student either might select or does not select the course.
- In use cases the **error conditions** can also be represented.
- Use case represents the **behavior of the system** using Main Flow, variations in normal flow, exceptions and error conditions and so on.
- The use cases can be written using following template -

Use Case template	
Use Case	Write the name of the Scenario.
Actor	Specify the role of the entity who interacts with the system.
Summary	Specify the purpose or goal of the system.
Preconditions	This is the condition which has to be satisfied before the use case starts.
Description	Sequence of steps that describe the main scenario of the use case.
Exceptions	Describe the exceptional situations that might occur during the execution of the use case.
Post conditions	This is a condition that should might occur after completion if the use case.

- For example : Following is an example of informal description or text story for withdrawal of money from ATM.
- Customer wants to operate the ATM machine for performing some transactions. For instance customer wants to withdraw the money. He enters the ATM card and then types in the PIN. The system validates the customer. If the customer is the valid customer then the customer is allowed to do further transactions. Otherwise the card will be rejected. The valid customer enters the amount to be withdrawn. It is then checked if withdrawal amount < balance amount. If it is so, then the machine dispenses the desired amount. If there is no further transactions then the card is ejected. Customer collects the cash, statement and card."

Use Case template	
Use Case	ATM system
Primary Actor	Customer
Summary	All the functionalities of the system will be monitored. The user will select the Withdrawal of money operation. The required amount of money will be withdrawn by the Customer.
Preconditions	ATM system has to be programmed and as to recognise and validate the PIN.
Description	1. Customer observes the control panel. 2. Customer enters the ATM card 3. Customer enters the PIN. 4. Customer selects the operation - withdrawal 5. Customer collects the cash, statement, card etc.
Exceptions	1. The control panel is not ready. 2. PIN is incorrect. 3. Card is not recognized. 4. Insufficient balance 5. Limit for the transaction exceeds. 6. Total number of allowed transactions per day.
Post Condition	The machine will wait for another transaction.

Key Point *Use cases are not diagrams they are text.*

Example 3.5.1 *Write use case summaries for a vending machine.*

Solution :

Use case Summaries for Vending Machine	
Use Case	Vending Machine
Primary Actor	Customer
Summary	All the functionalities of the system will be monitored. The user will get the selected the desired beverage. Customer pays for it and if some change needs to be returned then machine well return it back.
Preconditions	Vending machine must be loaded with popular beverages.
Description	**Buy a Cold-drink :** The customer selects the desired cold-drink, pays for it and then the vending machine delivers the cold-drink. **Perform Periodic Maintenance :** For keeping the machine in good condition, the repair technician performs the periodic servicing. **Make Repairs :** If some unexpected situation occurs, then the repair technician performs the repair operation. **Load Items :** A stock maintenance person adds items into the vending machine to keep the machine loaded with the stock.

Example 3.5.2 *Write the use case description for ATM System.*

Solution : An informal description or text story for withdrawal of money from ATM is as follows -

- "Customer wants to operate the ATM machine for performing some transactions. For instance customer wants to withdraw the money. He enters the ATM card and then types in the PIN. The system validates the customer. If the customer is the valid customer then the customer is allowed to do further transactions. Otherwise the card will be rejected. The valid customer enters the amount to be withdrawn. It is then checked if withdrawal amount < balance amount. If it is so, then the machine dispenses the desired amount. If there is no further transactions then the card is ejected. Customer collects the cash, statement and card.".

Use Case Description		
Use case Name :	Withdraw money from ATM System	
Scenario :	To monitor the withdrawal of money from ATM	
Triggering Event :	Customer enters the pin and request for withdrawal of money.	
Brief Description :	Customer requests for withdrawal of money by entering the amount to be withdrawn. If the sufficient balance is present in the account then, the money is dispensed from the ATM machine.	
Actors :	Customer	
Related Use Cases :	Check balance, get_Statement	
Stakeholders :	Administrator	
Preconditions :	ATM system has to be programmed and as to recognize and validate the PIN.	
Postconditions :	The requested amount must be ejected, card must be ejected and balance must be updated. The customer logs off.	
Flow of Events :	**Actor**	**System**
	1. Customer observes the control panel.	1. The system displays the Welcome Screen
	2. Customer enters the ATM card	2. System validates the card.
	3. Customer enters the PIN.	3. System validates the PIN.
	4. Customer selects the operation(withdrawal, deposit, inquiry)	4. System display main menu and allows the use to select one option.

	5. Customer collects the cash, statement, card etc.	5. System displays the payment details screen. 6. System beeps on ejecting money and card.
Exception Conditions :	1. The control panel is not ready. 2. PIN is incorrect. 3. Card is not recognized. 4. Insufficient balance 5. Limit for the transaction exceeds. 6. Total number of allowed transactions per day.	

Another use case description is for **deposit money** operation.

Use Case Description		
Use case Name :	Deposit money through ATM System	
Scenario :	To monitor the deposition of money from ATM	
Triggering Event :	Customer enters the pin and request for deposition of money.	
Brief Description :	Customer requests for deposition of money in his account via ATM machine.	
Actors :	Customer	
Related Use Cases :	Check balance, get_Statement	
Stakeholders :	Administrator	
Preconditions :	ATM system has to be programmed and as to recognize and validate the PIN.	
Postconditions :	The customer deposits money and logs off.	
Flow of Events :	**Actor**	**System**
	1. Customer observes the control panel.	1. The system displays the Welcome Screen
	2. Customer enters the ATM card	2. System validates the card.
	3. Customer enters the PIN.	3. System validates the PIN.

	4. Customer selects the deposit money option.	4. System display main menu and allows the use to select one option.
	5. Customer enters the amount to be deposited.	5. System displays the message to place the depositing amount on the slot.
	6. Customer places money on deposit slot.	6. System accepts the money and checks the exact amount.
	7. Customer collects the card etc.	7. System beeps on ejecting card.
Exception Conditions :	1. The control panel is not ready. 2. PIN is incorrect. 3. Card is not recognized. 4. Limit for the transaction exceeds. 5. Non positive amount is entered to deposit the money. 6. Failure to place the depositing amount on depositing slot. 7. Total number of allowed transactions per day.	

3.5.3 Use Case Diagram

- The system is represented using the set of use cases and the set of actors.
- The **set of use cases** show the complete functionality of the system.
- The **set of actors** represent the objects that interact with the system in order to exploit the behavior.
- Use case diagram is a graphical notation used to represent the set of use cases and set of objects.
- The rectangular box is used to represent the **boundry** of the system.
- The oval is used to represent each use case.
- The stick man is used to denote the actor. The solid lines connect use cases to participating actors. For example -

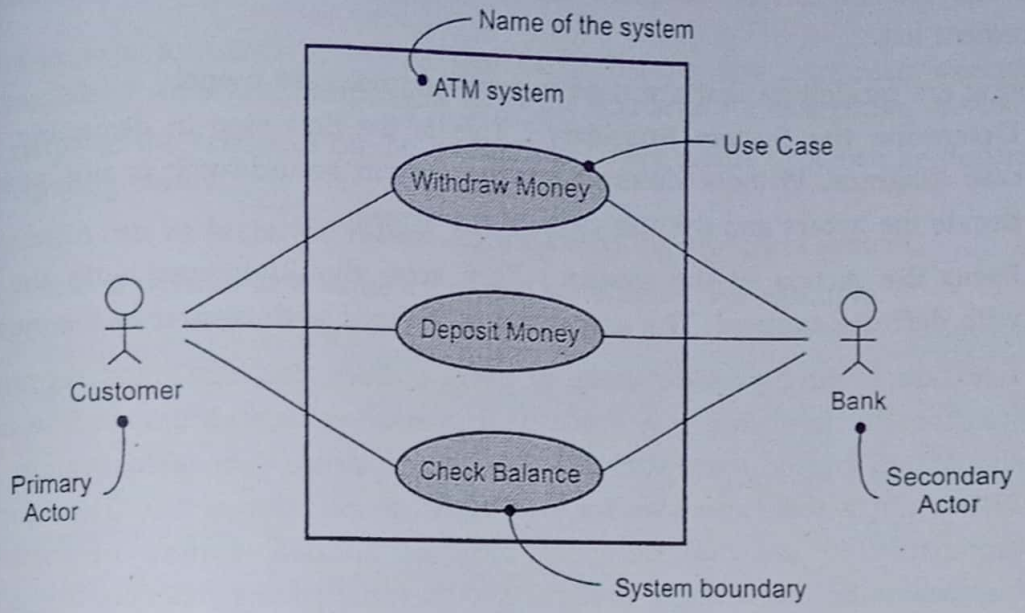


Fig. 3.5.1 Use case diagram for ATM system

Example 3.5.3 *Draw the use case diagram for the vending machine.*

Solution :

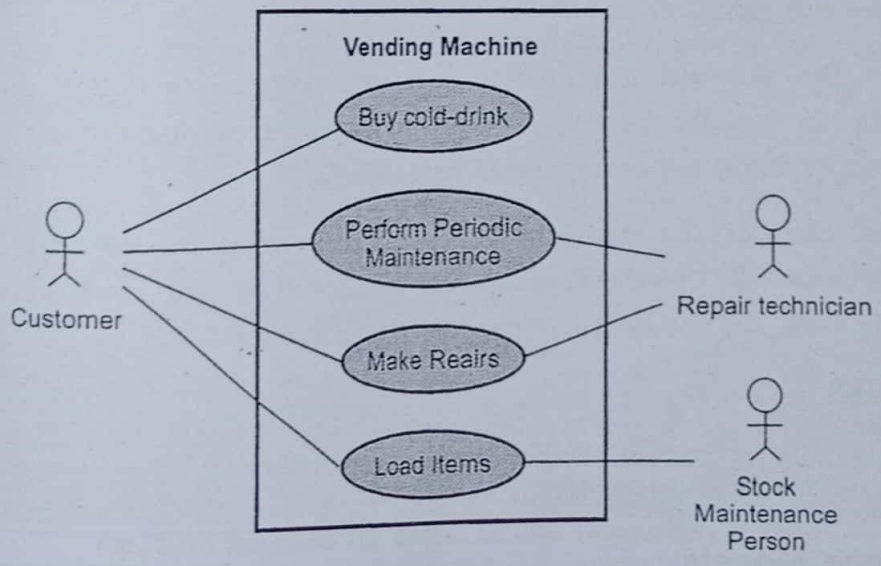


Fig. 3.5.2 Use case diagram for vending machine

3.5.4 Guidelines for Use Case Models

- Use cases are used to exploit the functionality of the system from users point of view whereas the **requirement list** contains the functionality of the system that is vague to the users. However, the traditional requirement list represents the global constraints of the system more effectively.

- Hence the system can be designed using the use cases along with the traditional requirement list.
- Following are guidelines that are used to construct use case model -
  1. **Determine the System Boundary :** This is the first step in designing the use case diagram. Without determining the system boundary it is not possible to decide the actors and the use cases of the system.
  2. **Focus the Actors of the system :** Each actor should interact with the system with **definite purpose**. The actor must be defined with respect to the system.
  3. **Use case should provide value to users :** Each use case must represent the functionality in such a way that it will provide value to the user. The use case should not be too narrow or too wide. For instance - In ATM system, type a **PIN** is not a valid use case for **Withdraw** money transaction. Thus too much granularity in use case designing must be avoided. Instead of focusing on implementation details the designer should focus on the functionality provided by the system.
  4. **Co-relate Actors and use cases :** Every actor must be associated with at least one use case and one use case must be co-related with at least one actor. There can be multiple actors for the single use case and single actor can interact with multiple use cases.
  5. **Design the informal use cases :** It is not expected to design the use cases formally. In fact, the use cases must be designed from users' point of view and the functionalities can be organized accordingly.
  6. **The use cases can be organized gradually :** For designing the large systems the use cases can be co-related using the relationships. For small systems at the primary level the use cases are distinct. (Refer section 3.5.3)

Review Question

1. Describe guidelines for use case models.

3.6 Sequence Models

SPPU : April-16, Marks 5

Use cases describe how external actors **interact** with the system, whereas the **sequence models** represent the **flow of system events**, actions and messages between the object over the time period.

There are two kinds of sequence models - **Scenarios** and **sequence diagram**

Let us discuss them one by one with the help of examples.

3.6.1 Scenario

- **Scenario** is a sequence of events that occur during one particular execution of a system. Many software practitioners write the scenarios for describing the use cases.
- The **scope** of the scenarios can be limited to certain events or it can be defined by the participating objects.
- A scenario can be **historical record** for the executing an actual system.
- A scenario is written as a sequence of **text statements**. For example -

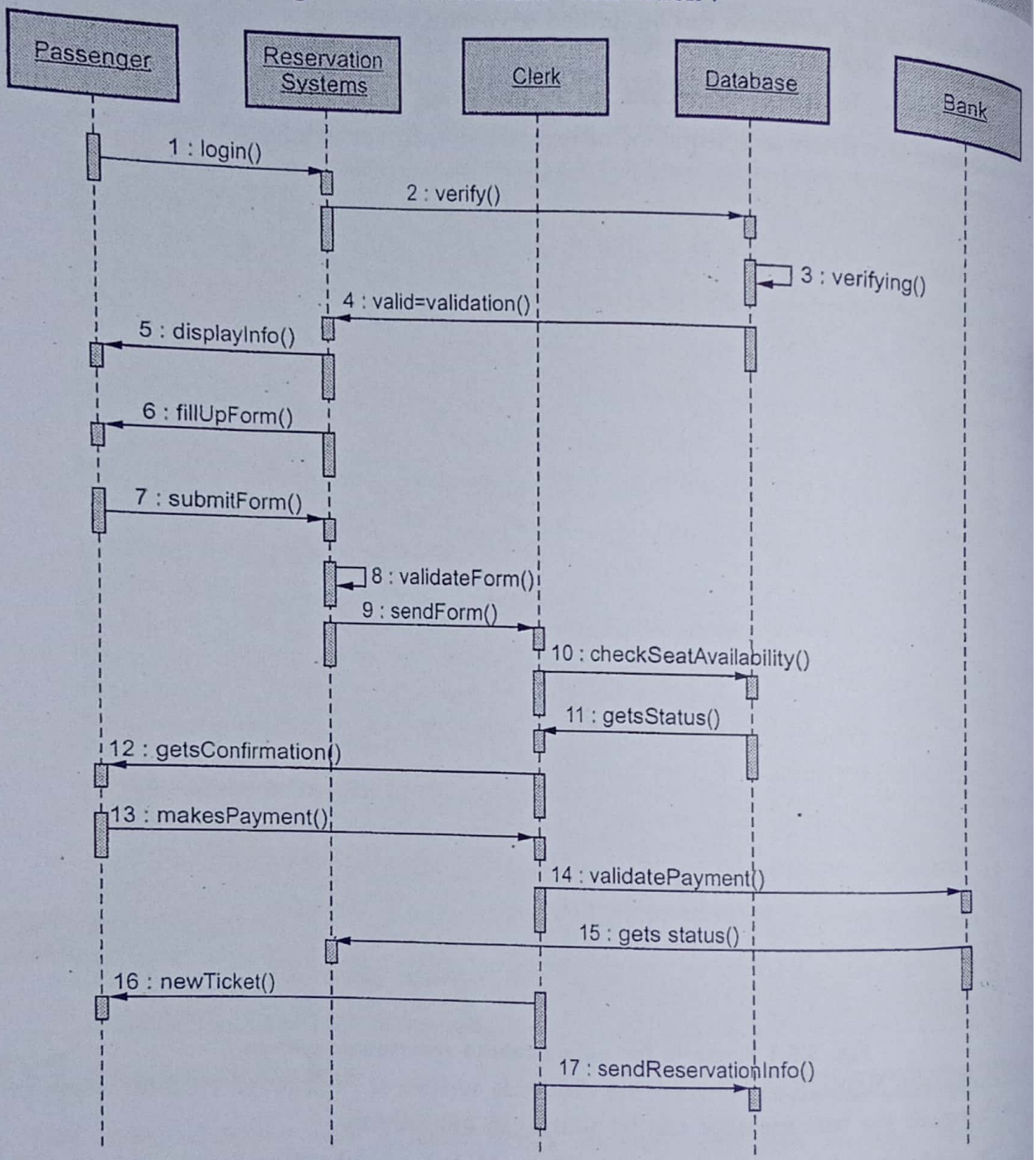
Student logs in.
System validates the student.
System displays the Home page for Course Reservation System.
Student scrolls through the available course.
Student selects the desired available course.
The system displays the form to be filled up.
The student fills up the form and submits it.
The system validates the eligibility of form.
System displays the form acceptance message.
The student receives the eligibility message and proceeds for payment.
Student selects the mode of payment.
Student fills up the required payment details.
Student confirms the reservation.
System validates the payment.
System generates the receipt.
Student selects the print receipt option.
Student gets the printout of the receipt.
Student logs out.

Fig. 3.6.1 Scenario for online course reservation system

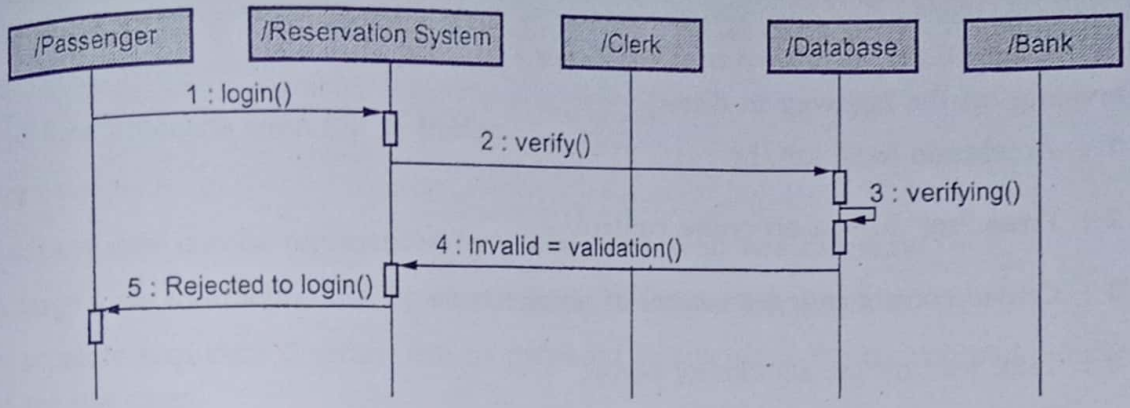
- Normally the interaction between the objects is written at high level. At later stage of development the text message can be written in more detail.
- In the scenario, the messages that are passed between the interacting objects are written. Each message transmit the information from one object to another.
- While writing the scenario - first identify the objects that interact with each other. Then determine the sender and receiver of each message. And finally decide the sequence of messages. Finally the internal activities can be added due to which the scenarios can be transformed into the implementation code.

Example 3.6.1 *Prepare sequence diagram for booking a train ticket on line. Also prepare sequence diagram for booking a train ticket on line that fails.*

Solution : Sequence diagram for online ticket reservation :



Sequence diagram for online ticket booking that fails :



Example 3.6.2 *Write scenario and sequence diagram for getting ready to take a trip in your car. Assume an automatic transmission. Don't forget your seat belt and emergency brake.*

Solution :

1) Open door.
2) Get in car and sit down.
3) Close door.
4) Put key in ignition.
5) Put on seat belt.
6) Check that transmission is in park.
7) Depress and release accelerator pedal.
8) Turn key.
9) Engine starts.
10) Release key.
11) Depress brake pedal.
12) Release emergency brake.
13) Move transmission lever to drive.
14) Check for traffic in rear view mirror.
15) Turn on left directional light.
16) Move foot to accelerator pedal.
17) Depress pedal slowly and begin to drive.

Example 3.6.3 *Write scenario and sequence diagram for operation of a car cruise control include an encounter with slow-moving traffic that requires you to disengage and then resume control.*

Solution : Conditions are shown in parentheses -
(Traveling on the highway in drive.)

Step 1 : Accelerate to 90 km/hr.

Step 2 : Press "set" button on cruise control.

Step 3 : Cruise control engages control of accelerator.

Step 4 : Take foot off the accelerator pedal.
(Car operates accelerator under cruise control.)

Step 5 : Encounter slow moving car that you want to pass.

Step 6 : Depress accelerator and pass car.

Step 7 : Remove foot from accelerator.
(Cruise control continues to maintain speed.)

Step 8 : Encounter slow moving traffic that you cannot pass.

Step 9 : Depress brake pedal.

Step 10 : Cruise control disengages control of accelerator.

Step 11 : Maneuver past slow moving traffic.

Step 12 : Press "resume" button on cruise control.

Step 13 : Cruise control re-engages control of accelerator.

Step 14 : Take foot off accelerator.

Step 15 : Cruise control accelerates to preset speed.
(Car is operating under cruise control)

3.6.2 Sequence Diagram

- The scenarios are simple to write but they does not clearly specify the sender and receiver of each message.
- The sequence diagram is a graphical representation that shows the sender and receiver objects and sequence of messages. Thus interaction between the objects is represented by the sequence diagram.

- The object are represented at the top within the rectangle.
- The vertical line represents the **lifeline**. The messages are represented on the horizontal rows.
- The time proceeds from top to bottom.
- For example -
- Each use case can be represented by one or more sequence diagrams.
- A large scale interaction can be shown using the sequence diagram.
- A separate sequence diagram can be prepared to represent the exceptional conditions of the use cases.
- The sequence diagram is normally created to cover the basic behavior of the system.

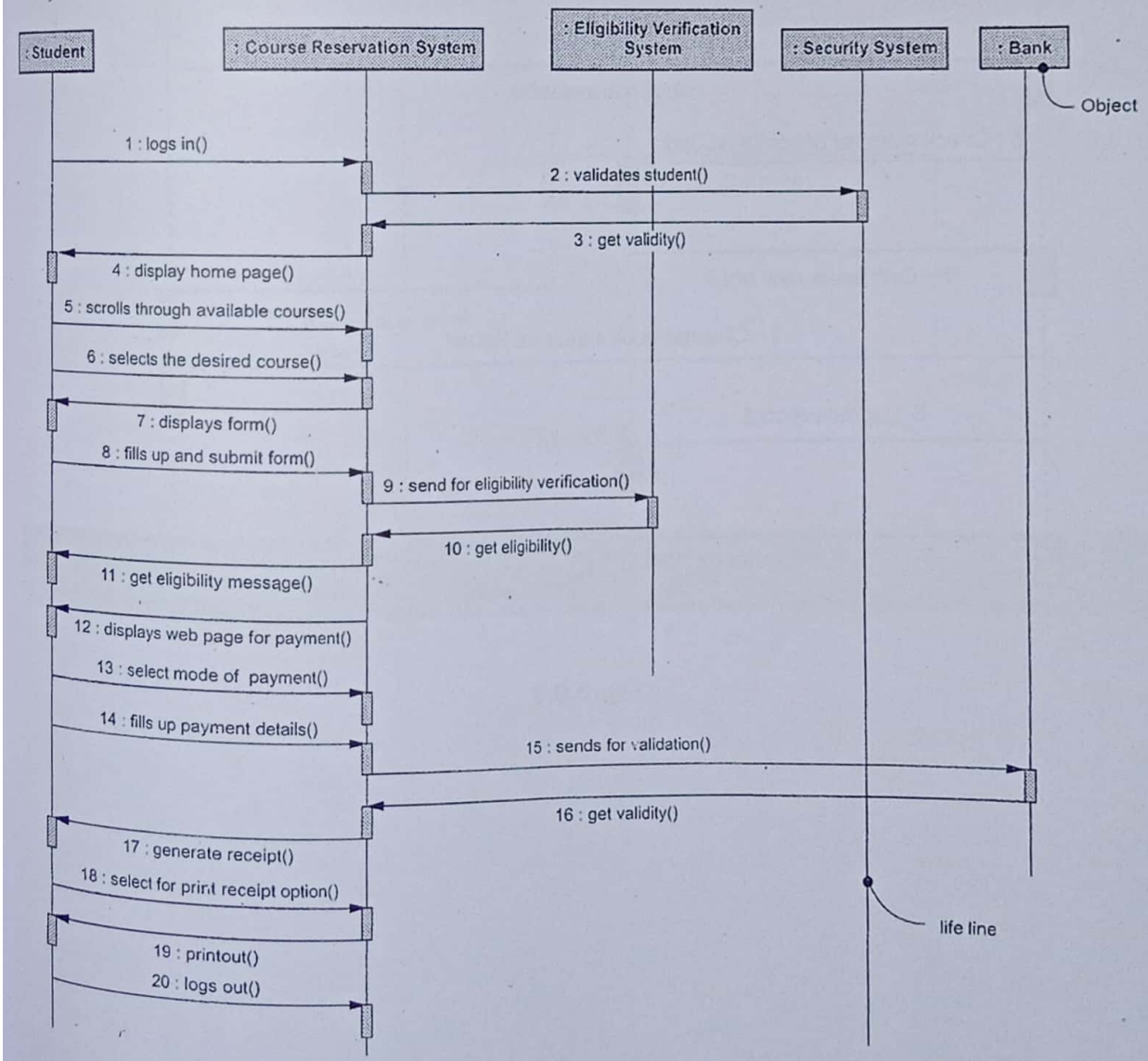


Fig. 3.6.2 Sequence diagram for online course reservation system

Example 3.6.4 *Draw a sequence diagram for issuing a book and renewing a book in online library management system.*

Solution : i) Sequence diagram for issue of a book

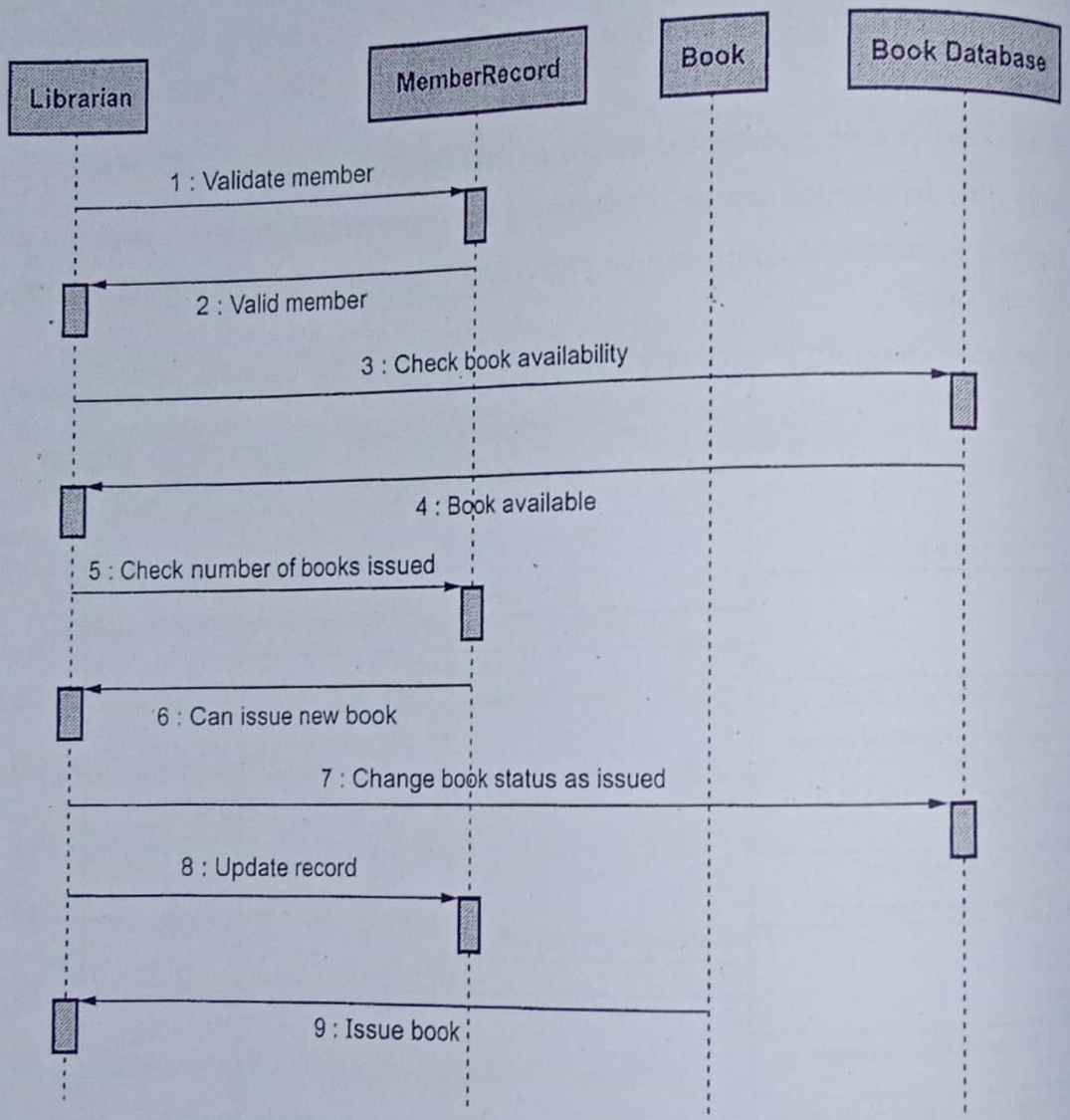


Fig. 3.6.3

ii) Renewal of a book

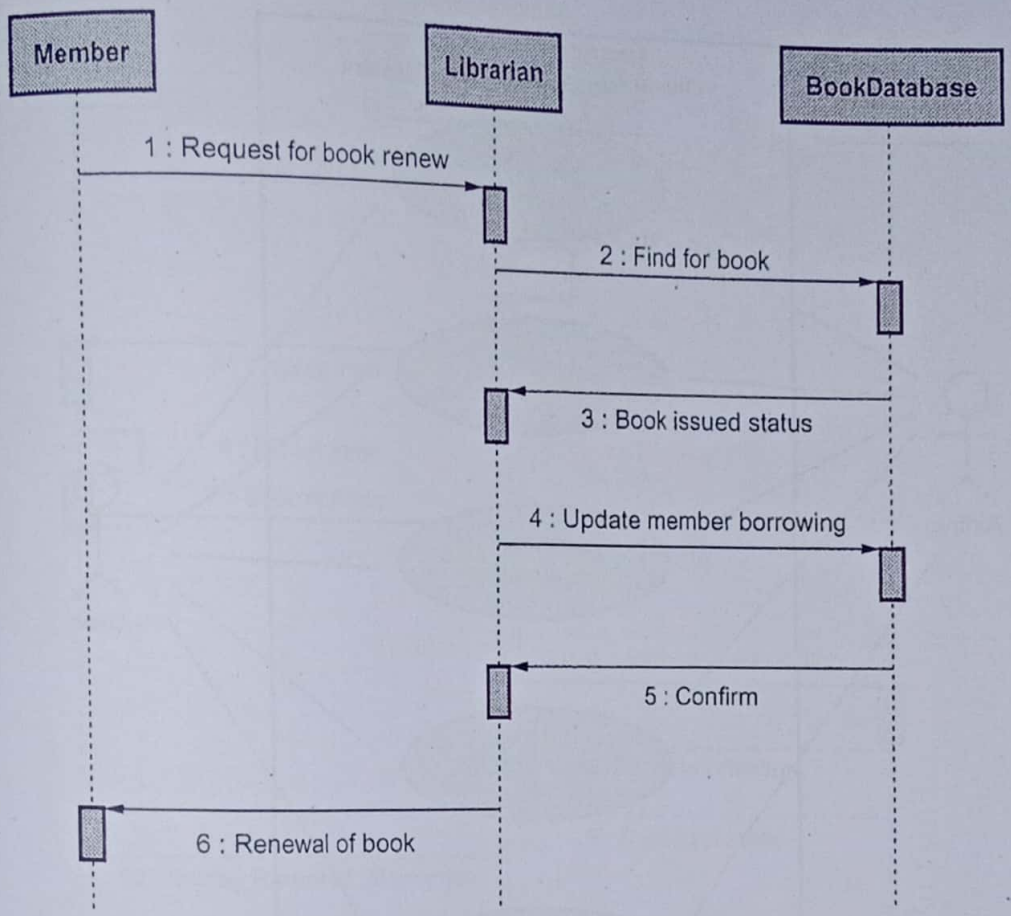


Fig. 3.6.4

Example 3.6.5 *Prepare a use case diagram and sequence diagram for an online airline reservation system.*

Solution : Use case diagram

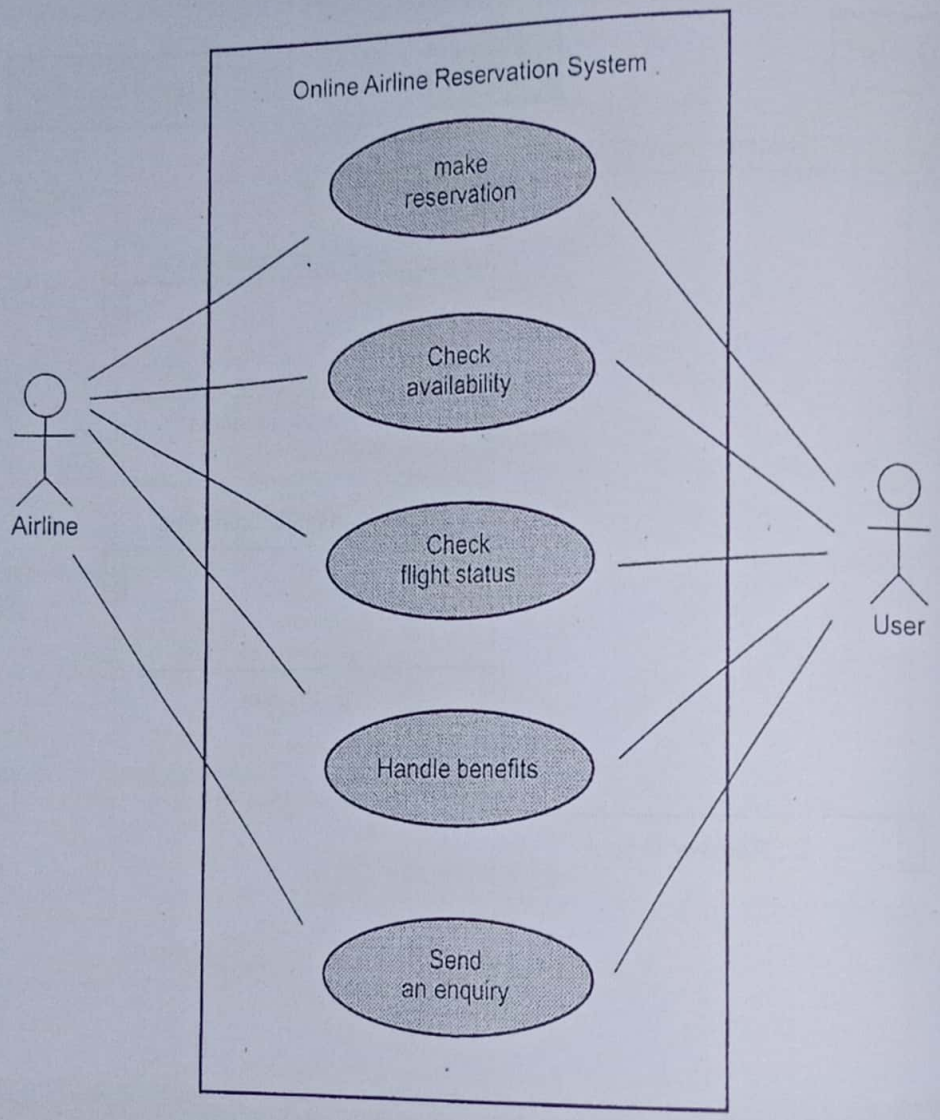


Fig. 3.6.5

Sequence diagram

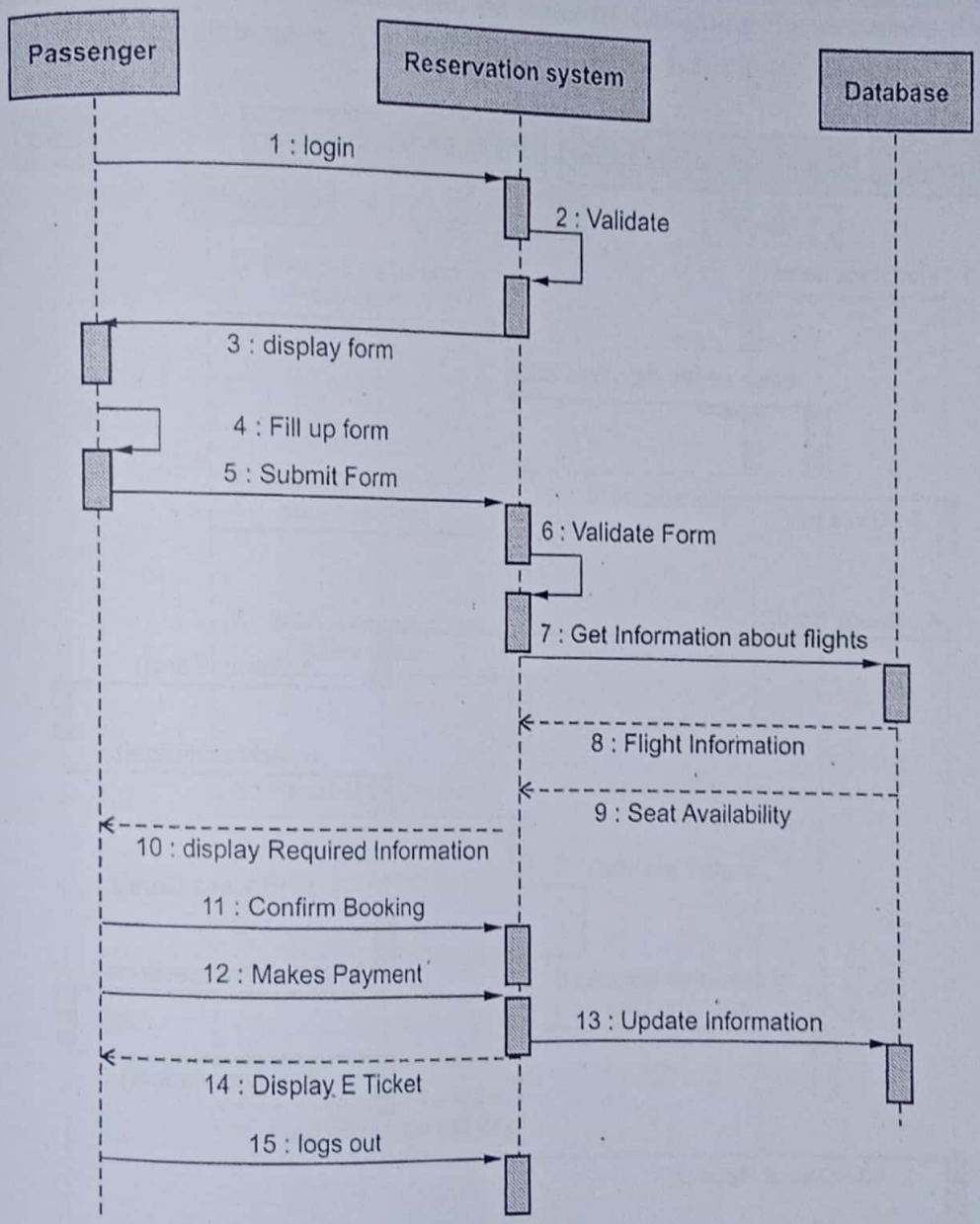


Fig. 3.6.6

Example 3.6.6 *Draw a sequence diagram for online shopping system.* **SPPU : April-16, In Sem, Marks 5**

Solution :

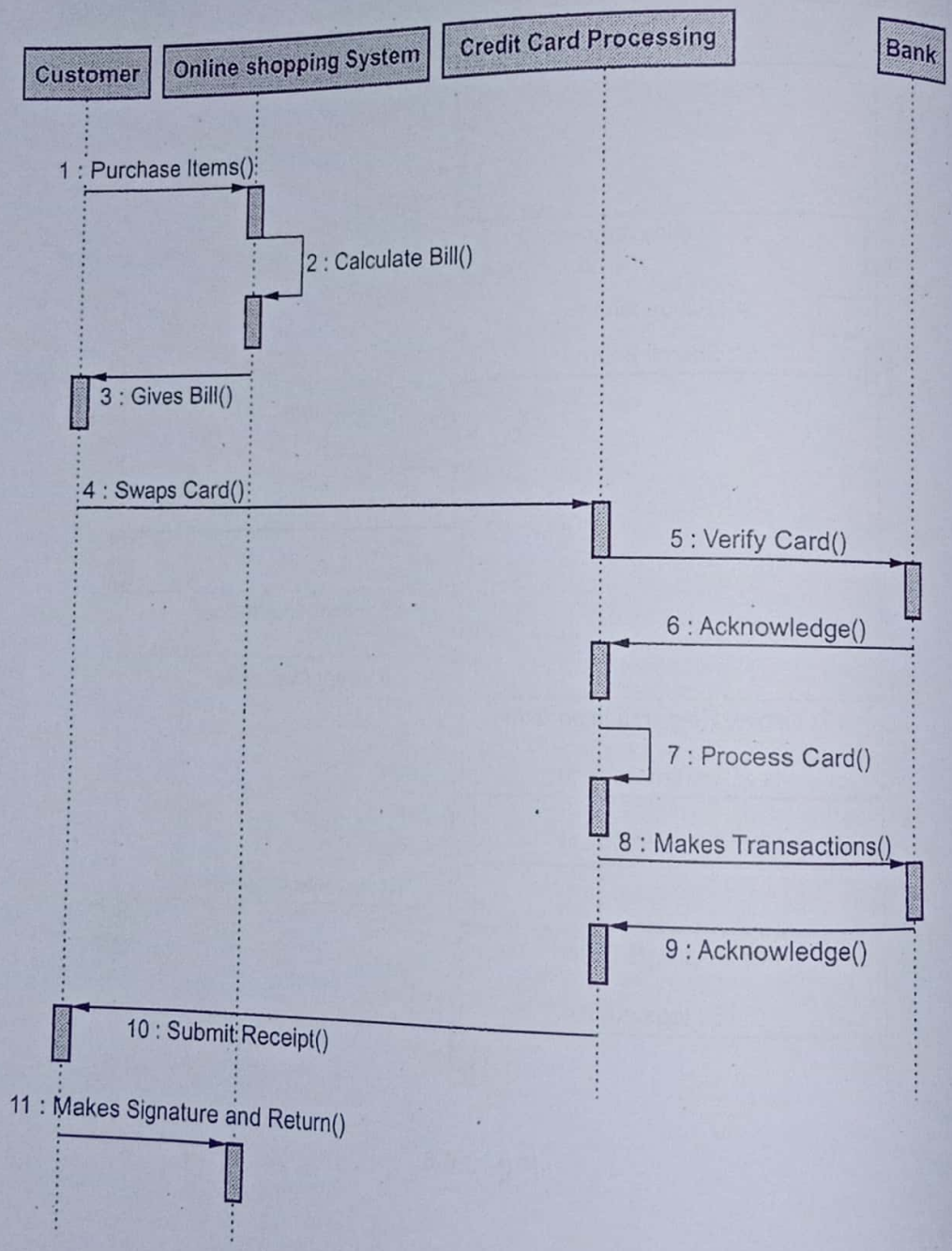


Fig. 3.6.7 Sequence diagram for online shopping system

Example 3.6.7 *Apply interactive modeling for a payroll system in UML.*

Solution : The interaction modeling can be done by designing the sequence diagram for the given problem statement. The sequence diagram for payroll system is as shown below

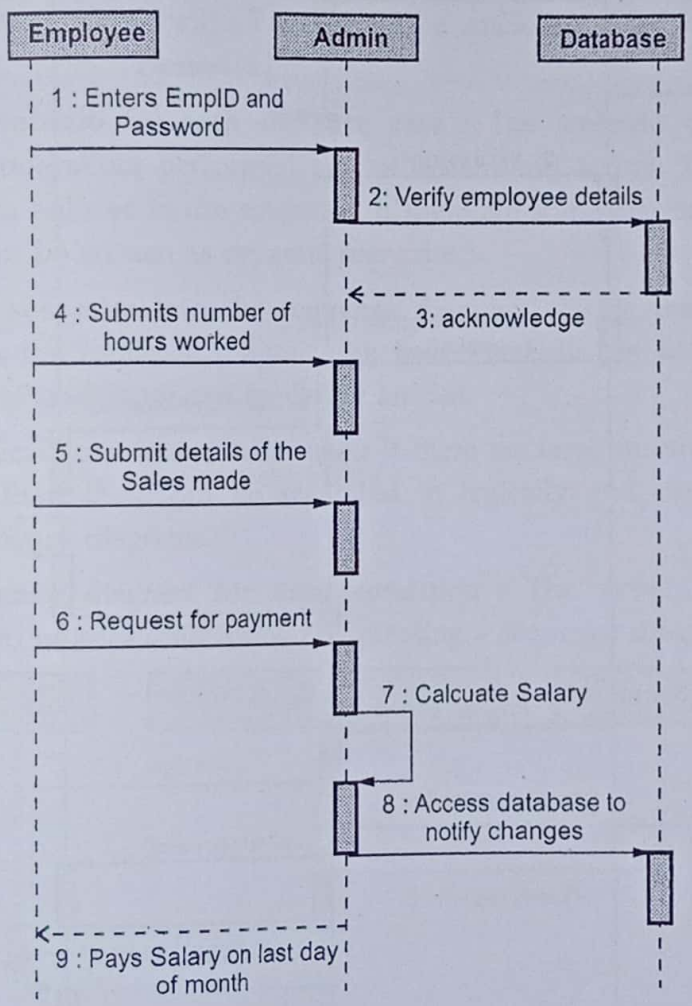


Fig. 3.6.8

Example 3.6.8 *Explain about interaction diagram notation for inventory management system.*

Solution :

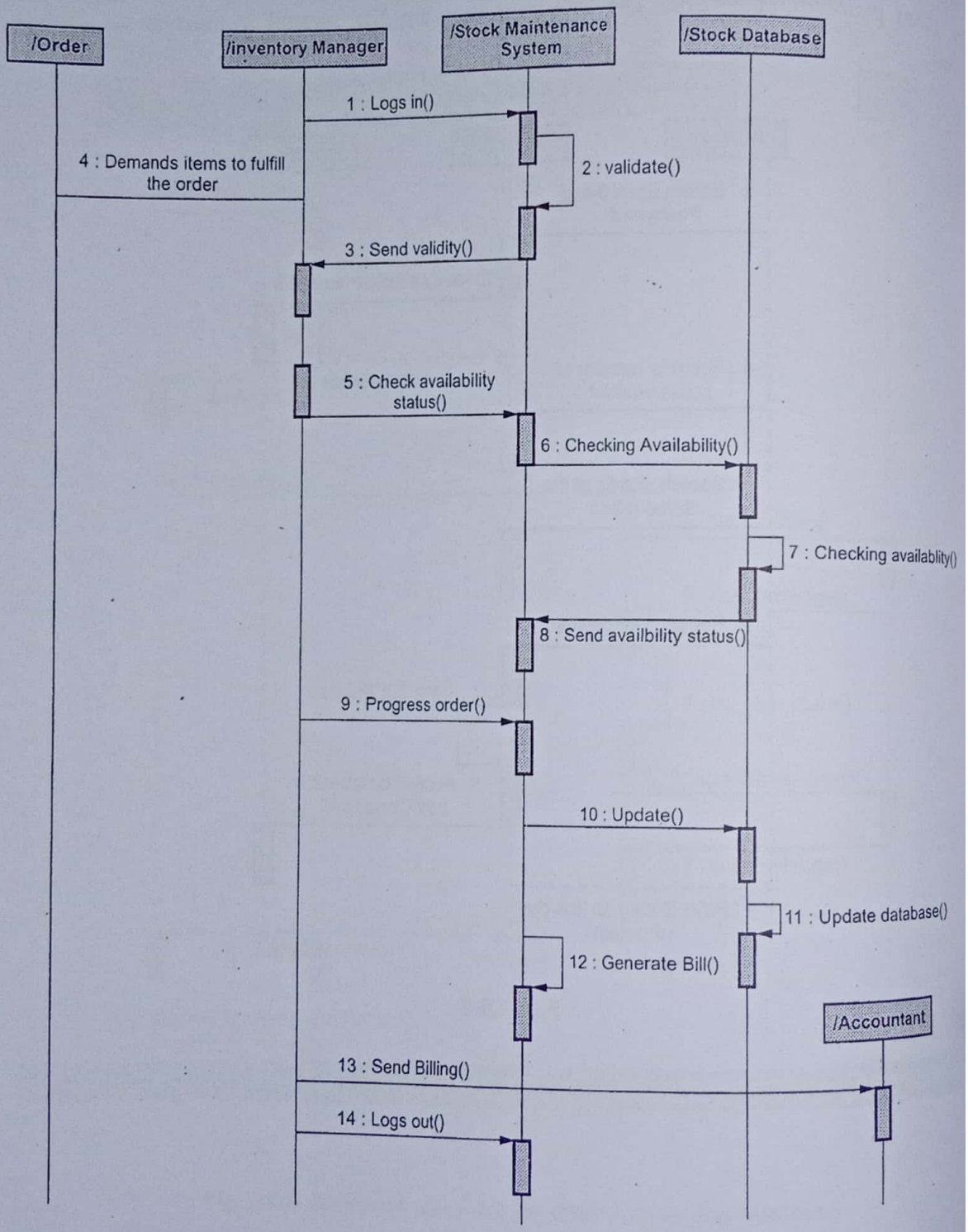


Fig. 3.6.9

3.6.3 Guidelines for Sequence Models

Sequence models are used to represent the informal themes of use cases. These can be modeled to add up the details of the system.

Normally there are two kinds of sequence models - **scenarios** and **sequence diagram.** The scenarios are the textual collection of events and sequence diagram represents the sequence of messages graphically. Following is a guideline used while modeling the sequence models -

1. **Write one scenario for each one use case :** The scenario contains the logical sequence of operations performed by the system and actors. The simple mainline actions can be enlisted in the scenarios. If there are some alternate mainline actions then those can be written as separate scenarios.
2. **Abstract the scenarios into the sequence diagram :** Using the actions enlisted in the scenarios the sequence diagram can be created. In the sequence diagram the contribution of each actor can be clearly shown.
3. **Complex interactions into smaller one :** If there are large number of interactions in the system, then those can be separated in logically and can be modeled as a separate sequence diagrams.
4. **Create sequence diagram for error condition :** The system response to error conditions can be separately shown by creating a sequence diagram. For example -

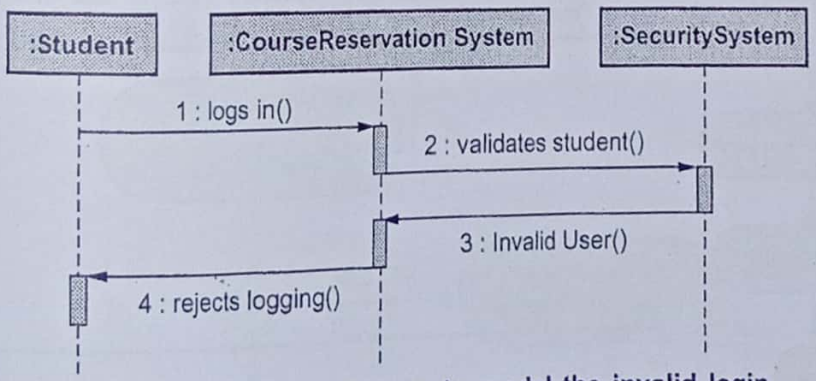


Fig. 3.6.10 Sequence diagram to model the invalid login

3.7 Activity Models

- An activity diagram is just similar to the flow chart in which flow of control from activity to activity is shown. Using algorithms or workflow the activity diagram can be drawn.
- In activity diagram the focus is on operations instead of objects.
- Activity diagrams are drawn during the early stages of designing algorithms or workflow.
- Fig. 3.7.1 represents the activity diagram for **Online reservation system.**

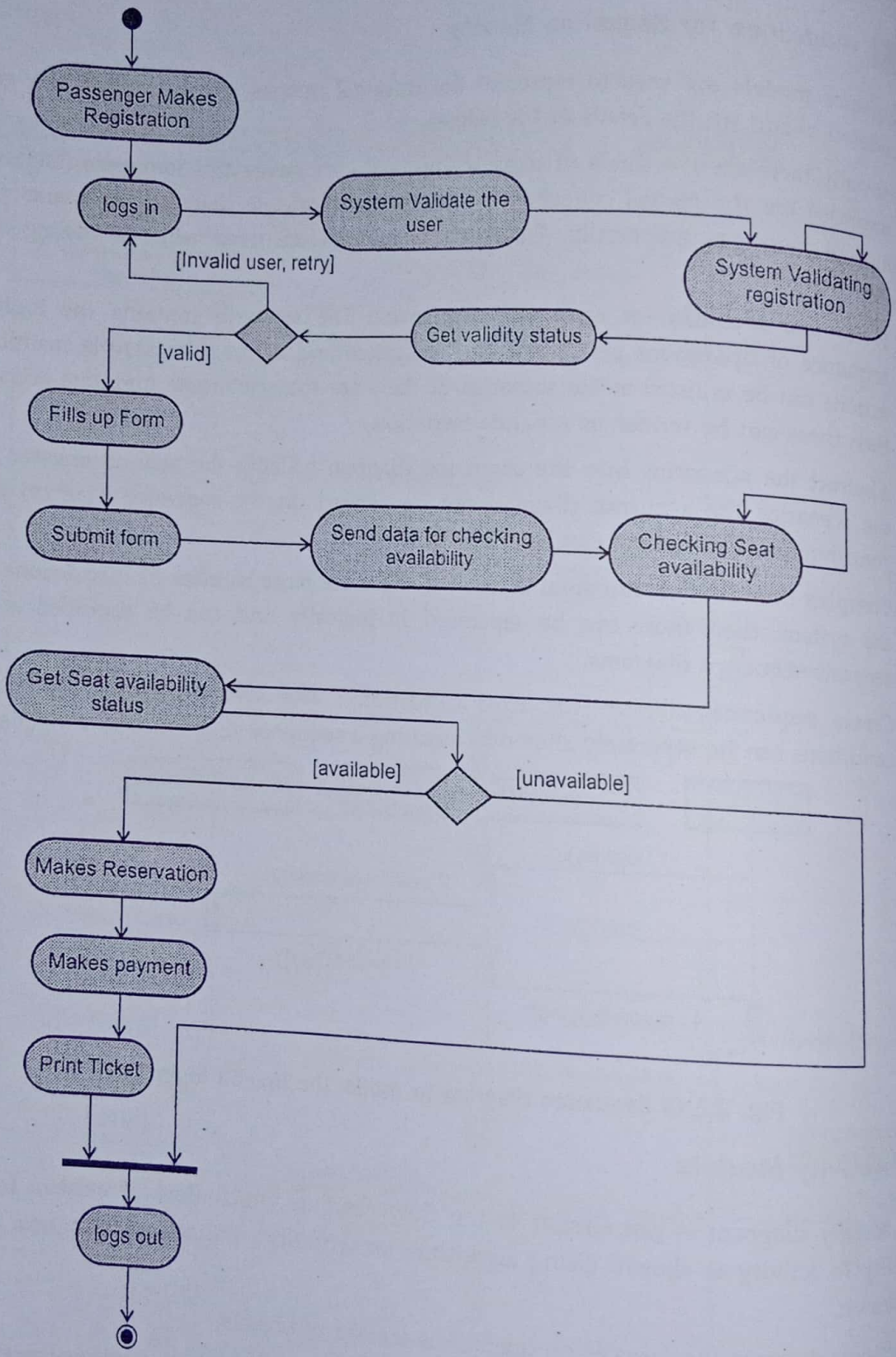


Fig. 3.7.1 Activity model

3.7.1 Activities

- The activities are nothing but the operations. These operations can be identified from the state diagram.
- The activity diagram shows the complex processes that occur in specific sequence The constraints imposed on these processes are also illustrated in activity diagram.
- Normally the activities complete their execution and terminate by themselves. But there are some activities that continue their execution until some external event interrupt them.
- When one activity finishes another activity starts. The two activities are connected by the **transitions**. These transitions are unlabelled.
- One complex activity can be decomposed into finer activities. For example the **Makes Reservation** this activity can be composed into smaller activities such as **Get Reservation Status, Scrolls through Status** and **Click Confirm button**.

3.7.2 Branches

- In activity diagram the branching is used to specify the alternate path based on some boolean expression.
- Each arrow of the alternative path is labeled by some condition. This condition is specified within the square bracket. For example In Fig. 3.7.2 the conditions **Not OK** and **OK** are specified within the square brackets.

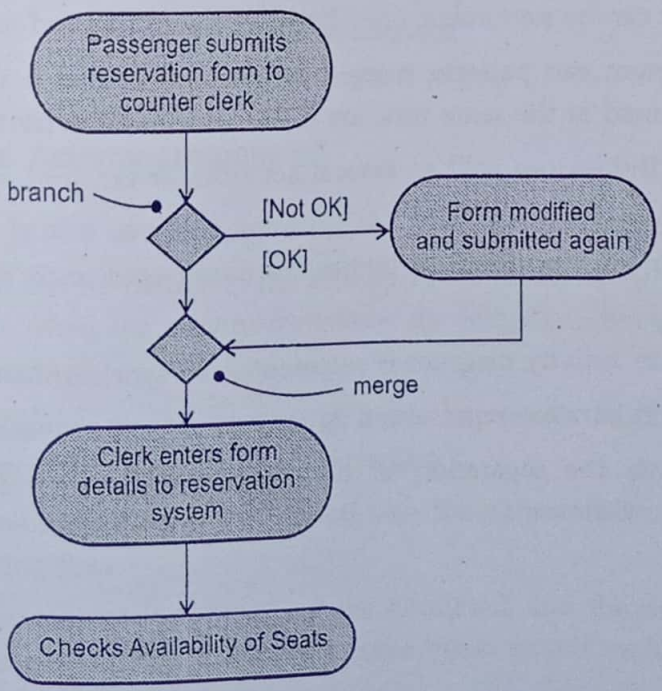


Fig. 3.7.2 Representing branches

- Graphically branch is specified as **diamond**.
- The branch generally have one incoming and two or more outgoing flows and on each outgoing path the conditions are specified.
- When two paths merge together, they merge in **diamond**.

3.7.3 Initiation and Termination

Initial State : In activity diagram, the initial state is represented by a solid circle. From this circle the transition to the first activity starts. Thus the initial state represents that all the activities start their execution from this point.

Termination : In activity diagram, the termination is represented by **bull's eye** i.e. a solid circle surrounded by a hollow circle. This state represents the termination of all the activities at this point. Fig. 3.7.3 represents these states.

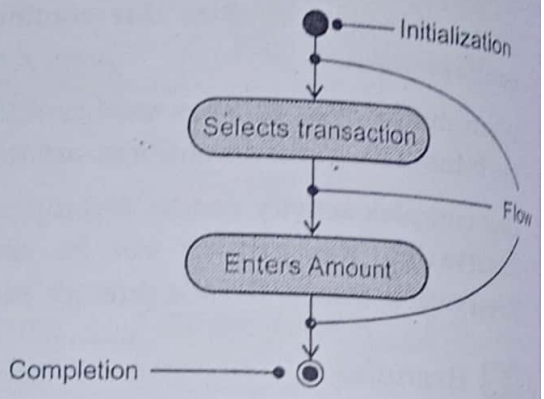


Fig. 3.7.3 Initiation and termination

3.7.4 Concurrent Activities

- There are two types of activities - sequential activities and concurrent activities.
- The activities that can be performed one after the other are called sequential activities.
- The computer system can perform more than one activities at a time. The activities that can be performed at the same time are called concurrent activities.
- The concurrent activities can split in several activities or can join and merge in some action.
- In activity diagram the concurrent activities are represented with the help of **forking and joining**.
- The fork and join in activity diagram is represented by **synchronization bars**.
- The synchronization bars are represented by thick horizontal or vertical bars.
- The **fork** represents the separation of two control flows whereas **join** represents joining of two control flows.

Following Fig. 3.7.4 represents the concurrent activities by using fork and join.

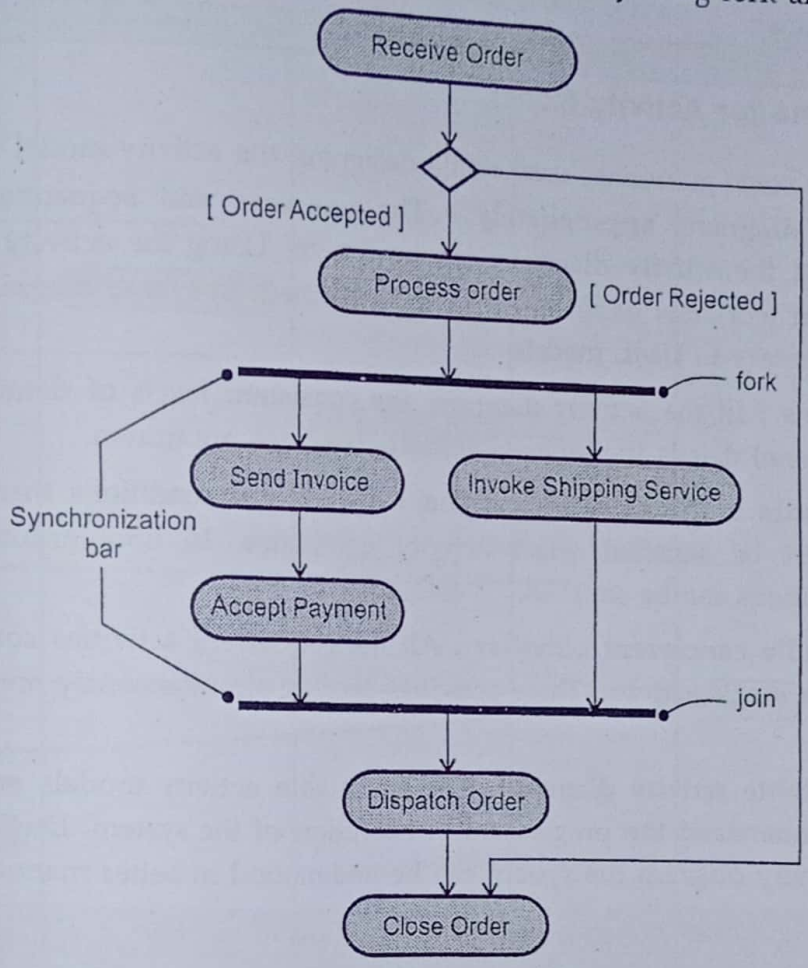


Fig. 3.7.4 Forking and joining

3.7.5 Executable Activity Diagram

- Activity diagram is also useful to represent the flow of execution of the process.
- An **activity token** can be placed on the activity nodes to represent that certain activity is executing. And when the activity completes, the token can be removed and placed on the outgoing arrow.
- In the sequential flow of the control the token moves to the next activity which indicates the progress of execution of the activities.
- When there are multiple flows of execution then the **activity token** is placed only on one of the executing flow.
- The conditions in multiple control flow are examined and the successor activity is determined. Only one successor activity receive token evenif multiple conditions are true.

- During the concurrent flow multiple **activity tokens** can be present. When multiple concurrent flows need to be followed then the **activity tokens** can be placed on concurrent flows.

3.7.6 Guideline for Activity Model

Following are some guidelines used while designing the activity model -

- **Use activity diagrams appropriately :** The use case and sequential models are elaborated and the activity diagrams can be drawn. Using the activity diagrams the algorithms and workflow of the system can be studied. But these diagrams are drawn as a supplementary to UML models.
- **Level diagrams :** In the activity diagram, the consistent levels of details are drawn. For the additional details the separate activity diagrams are drawn.
- **Carefully handle branches and condition :** If there are conditions then at least one condition must be satisfied when activity completes. In undeterministic models, multiple conditions can be satisfied.
- **Carefully handle concurrent activities :** All the concurrent activities complete before merging into a single activity. These activities execute simultaneously and perform the required task.
- **Design executable activity diagram :** The executable activity models are helpful for the users to understand the progression of execution of the system. Due to executable models of activity diagram the system can be understood in better manner.

Example 3.7.1 *A customer decides to upgrade her PC and purchase a DVD player. She begins by calling the sales department of the PC vendor and they tell her to talk to customer support. She then calls customer support and they put her on hold while talking to engineering. Finally the customer support tells the customer about several supported DVD options. The customer chooses a DVD and it is shipped by the mail department. The customer receives the DVD, installs it satisfactorily and then mails her payment to accounting.*

Construct an activity diagram for this process. Use swimlanes to show the various interactions.

Solution :

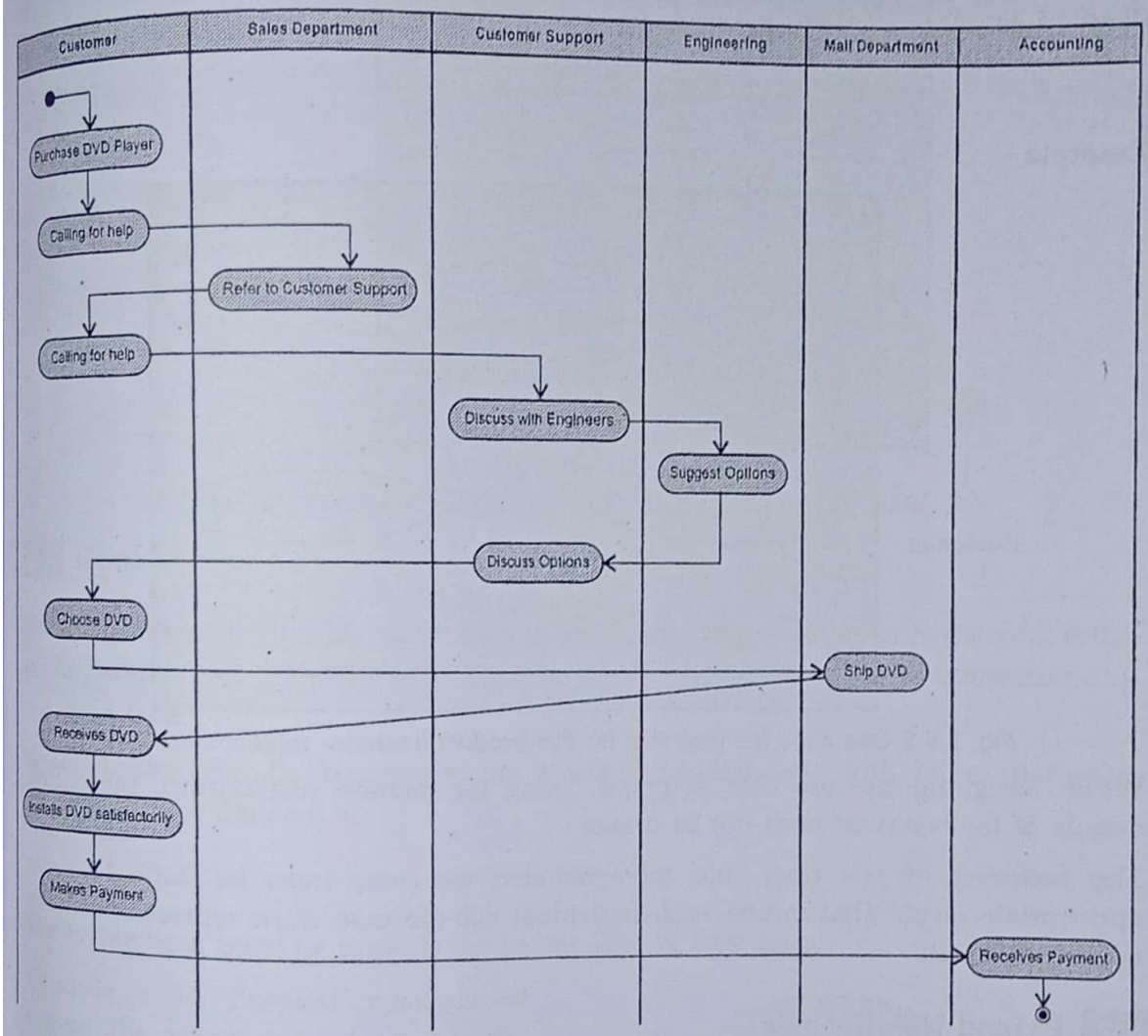


Fig. 3.7.5 Swimlane diagram

3.8 Use Case Relationships

SPPU : May-17, April-18, Marks 5

As we know, the use cases are designed in order to expose the functionalities of the system. The complex use cases can be built using various relationships such as include, extend and generalization.

3.8.1 Include Relationship

- The include relationship is used to show that the base case is obtained by the behavior of other use cases.
- Using the stereotype **<<include>>** the include relationship can be represented.

- When there are multiple steps to carry out a single task then that task is divided into the sub-functions and these sub-functions can be denoted by the use cases. It is denoted as -

<<include>>

- **Example -**

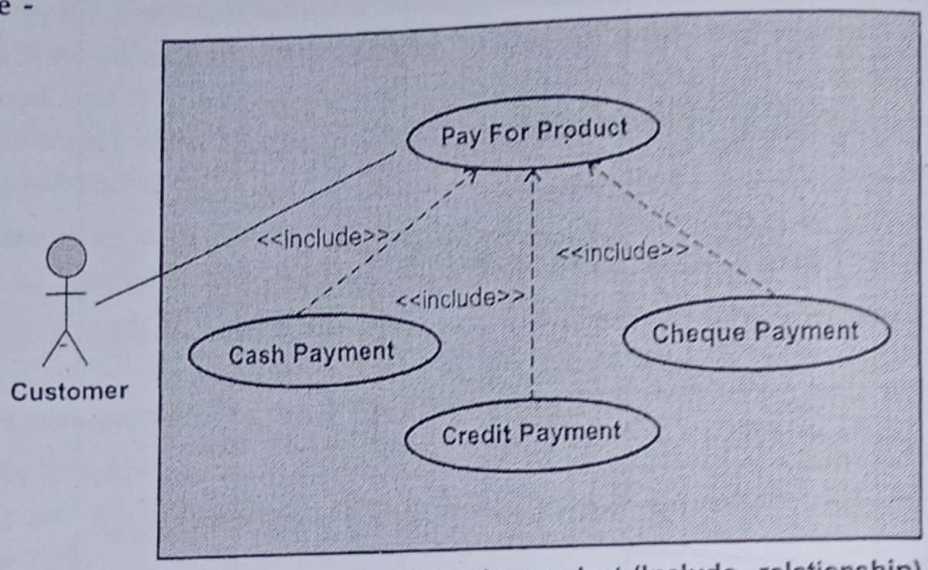


Fig. 3.8.1 Use case for payment for the product (Include- relationship)

- While designing the use case diagrams, using the **include** relationship, much fine details of the behavior must not be drawn.
- The factoring of use cases into more detailed use cases must be done to some appropriate level. That means each individual sub-use case must represent at least one functionality.

3.8.2 Extend Relationship

- The extend relationship is used when an extended use case is connected to the base use case.
- The extend relationship adds incremental behavior to use case. It represents frequent situations in which some initial conditions occurs and when new features are added in the module.
- The extension relationship is often a fragment that means it cannot occur alone; rater it will occur with the base use case.
- It can be denoted as -

<<extend>>

- "The base use case can specify an insert location within the base class. These extension are called extension points. In most of the cases, an extend behavior occurs only when condition is true. Following Fig. 3.8.2 represents the extend relationship -

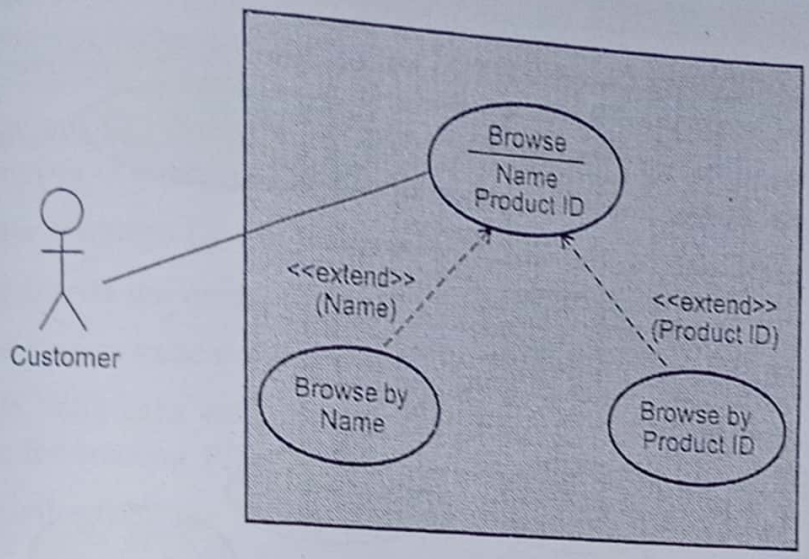
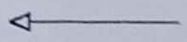


Fig. 3.8.2 Use case for browsing product (Extend - relationship)

3.8.3 Generalization

- Generalization is a kind of relationship in which the child class inherits the properties of its parent class. Sometimes the behavior of the parent class can be overridden by the child class.
- Generalization can be represented by a solod directed line with large triangular arrowhead. It is denoted as



- The arrow head must be towards the parent class or base class.
- **Example** : In hospital management system, the patient gets admitted to the hospital. He can be OPD patient or IPD patient. The admission process for these both types can be different. Hence it can be represented using the generalization relationship.

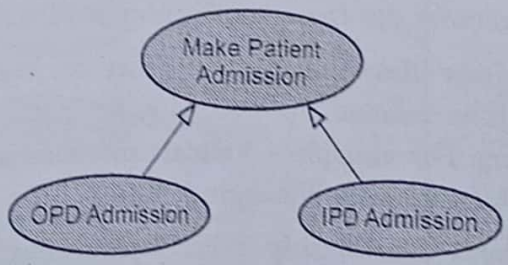


Fig. 3.8.3 Example for generalization

3.8.4 Combination of Use Case Relationships

A single use case diagram might contain various relationships such as include, exclude and generalization. Here is an example -

Example 3.8.1 *Draw the use case diagram for demonstrating various relationships in it.*

Solution :

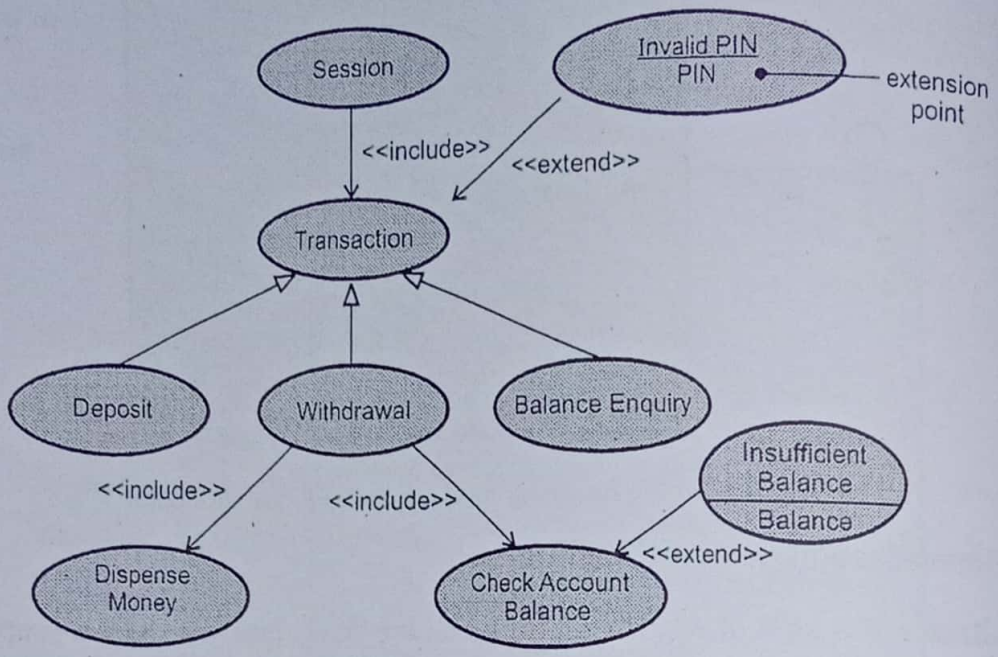


Fig. 3.8.4 Withdrawal of money from ATM

3.8.5 Guideline for Use Case Relationship

Following are some guidelines that can be used while designing the use case relationships -

- **Generalization Relationship :** If the use case has several variations then make the most general use case as parent or base use case and the specialized use cases as the child use cases. Do not use the generalization relationship to share the behavior of particular use case. For this purpose the include relationship can be used.
- **Include Relationship :** In order to fragment the particular behavior make use of include relationship. Secondly, the name of the use case must be meaningful to the users. For example - **Validate process** can be the base use case which can be further elaborated using the include relationship.
- **Extend Relationship :** For representing the optional features of the modeling the extended use cases are used. The extended use case can also be used when the system might be deployed in different configurations.
- **Include Relationship Vs. Extend Relationship :** The include and extend relationships are used to represent the division of single functionality into smaller functionalities. But the **include relationship** is used to represent what are the necessary functionalities that need to include to accomplish the main functionality. On the other hand the **extend relationship** is used to represent the optional behavior of the main functionality.

3.8.6 Examples

Example 3.8.2 *Describe use case "Validate User" in modeling an ATM system.*

Solution : There are two actors for the Bank ATM system. The **Customer** and the **Bank**. There are two types of customers - current account holder and savings account holder.

Following are the steps by which the customer interacts with the ATM system -

1. Customer inserts the card.
2. The ATM system validates the card.
3. If there is valid card entered by the customer then the ATM system requests the customer for entering PIN, The customer enters the PIN.
4. The PIN entered by the customer is validated.
5. If the valid PIN is entered then only the customer is prompted for performing the transactions.
6. Customer can select the desired transaction such as Withdrawal of money, Deposition of fund or simply enquires for the available balance amount.

The use case for Validate User is as given below -

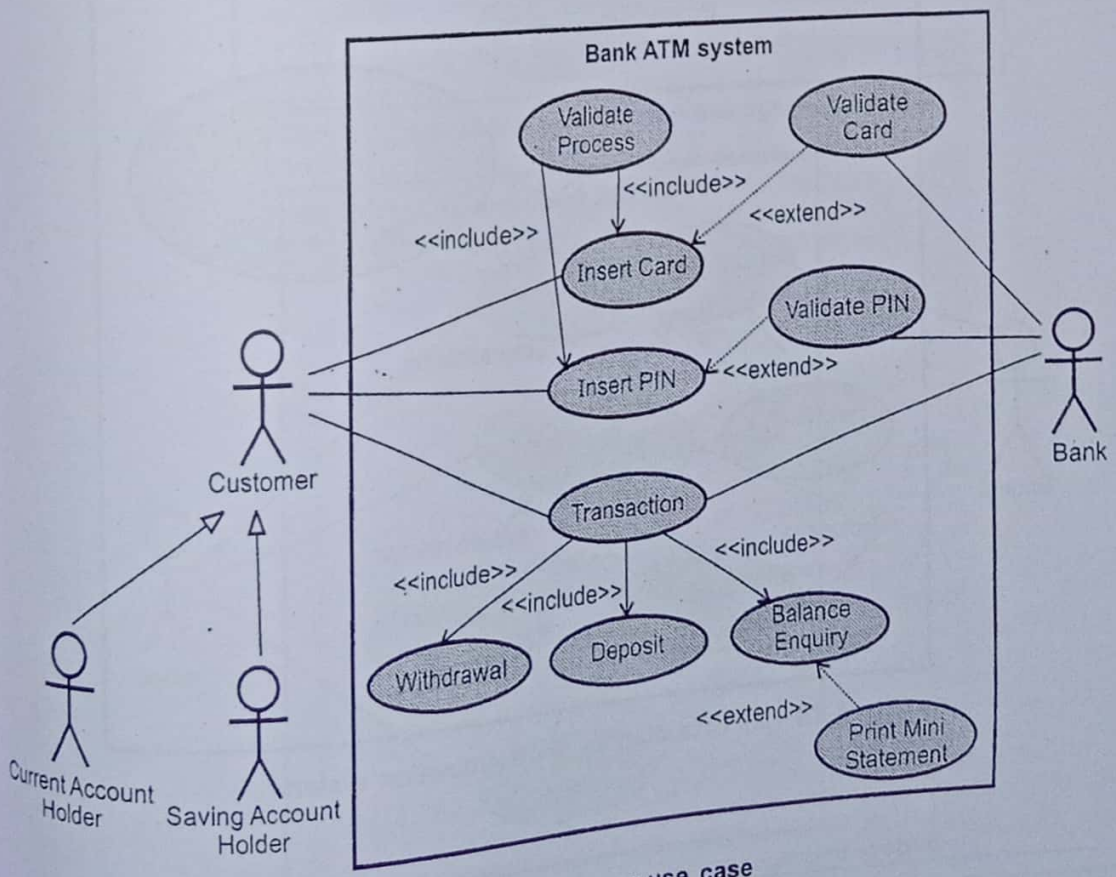


Fig. 3.8.5 Validate user use case

Use case Name : Validate Process

This use case is for validating the user interacting with the system. It consists of two use cases -

1. Insert Card
2. Enter PIN

For the card insertion the extended use case will be activated. By which the card is validated.

For the Enter PIN use case, the extended use case Validate PIN will be activated.

These validations can be performed by the Bank or by the authentication system which is associated with the bank.

Example 3.8.3 *Consider a software system like 'examination system'. Assume that there are use cases defined like 'view marks', 'input exam id','view practical marks', optionally send/forward my mark sheet as an Email'. Show how use case relationships like includes, generalizations and extends can be used to appropriately model above use cases and their relationships in context of use case diagram.*

Solution :

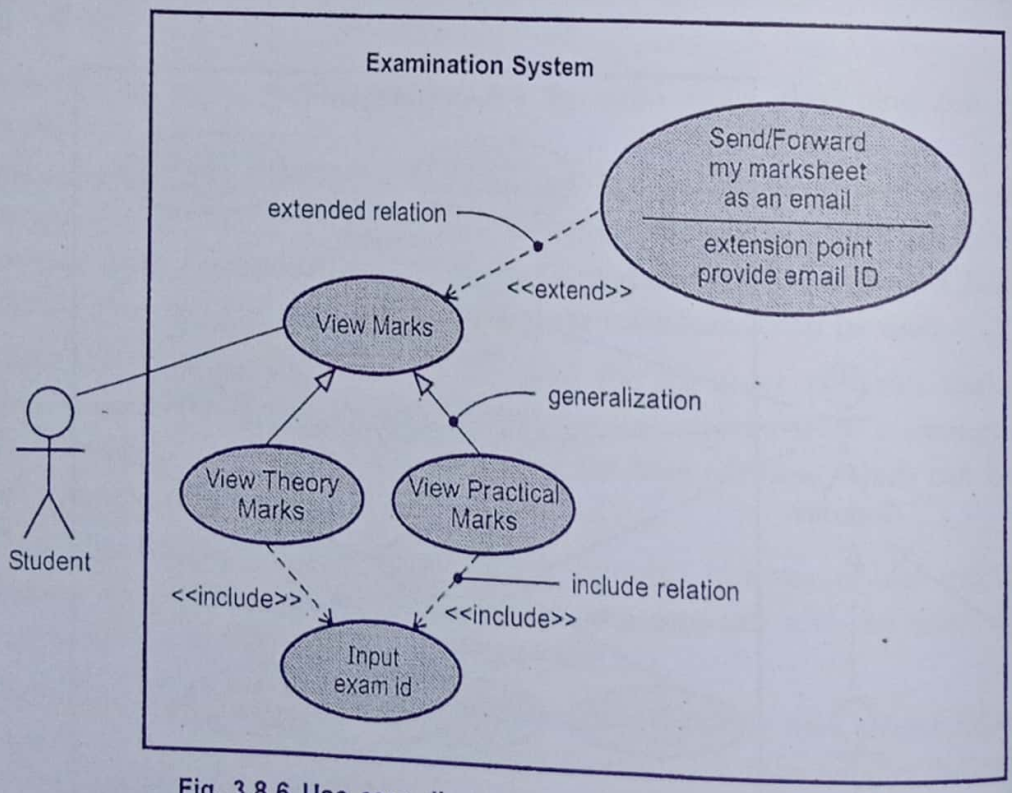


Fig. 3.8.6 Use case diagram for examination system

Example 3.8.4 *Give use case diagram for hospital management system.*

Solution :

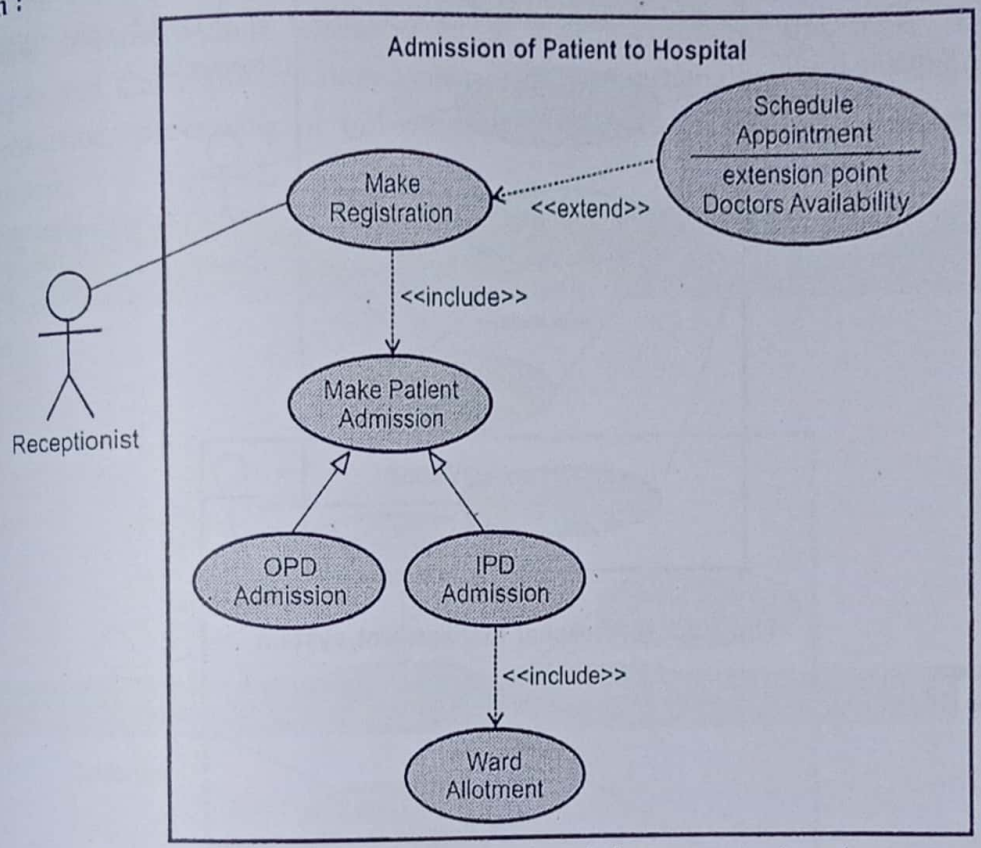


Fig. 3.8.7 (a) Use case diagram for hospital management system

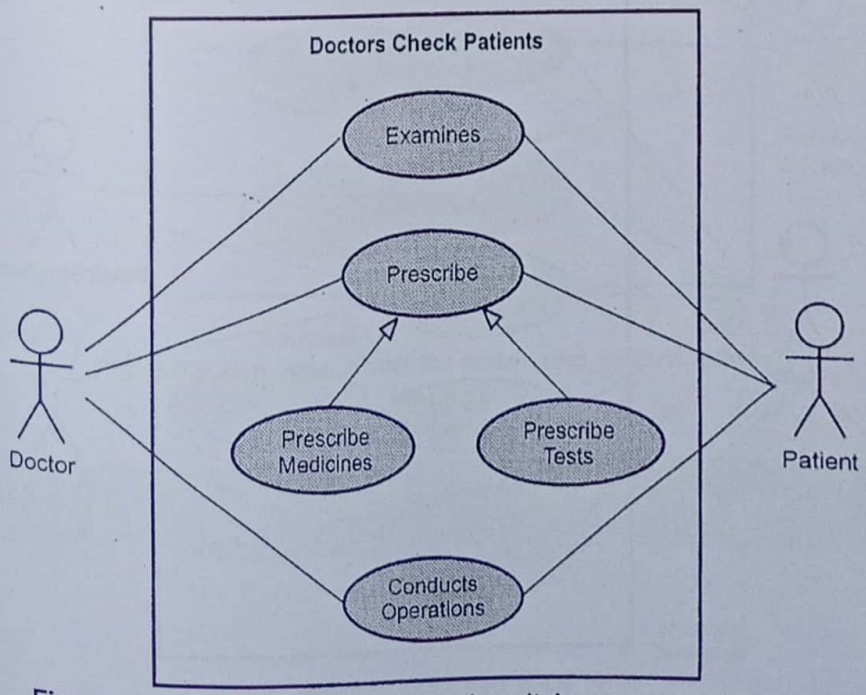
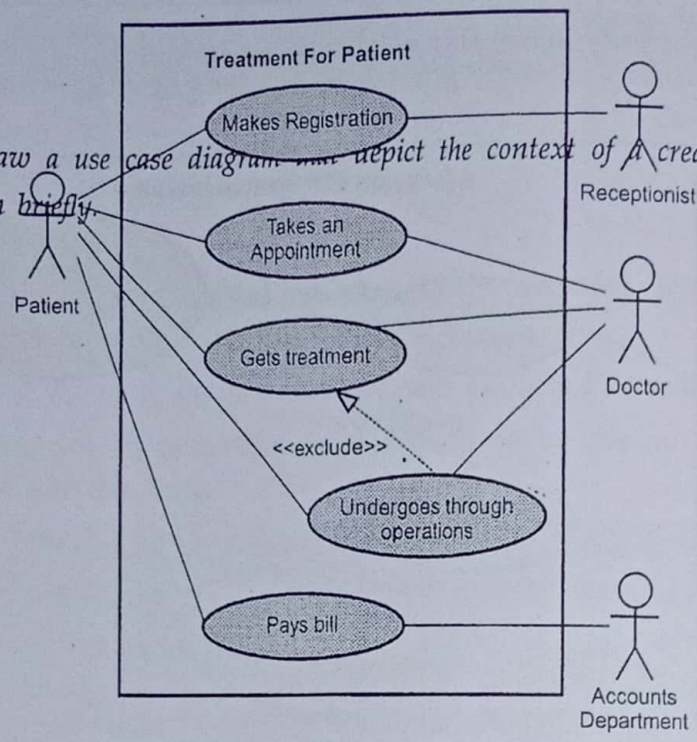


Fig. 3.8.7 (b) Use case diagram for hospital management system



Draw a use case diagram that depict the context of a credit card validation system. Explain briefly.

Fig. 3.8.7 (c) Hospital management system

Example 3.8.5

Solution :

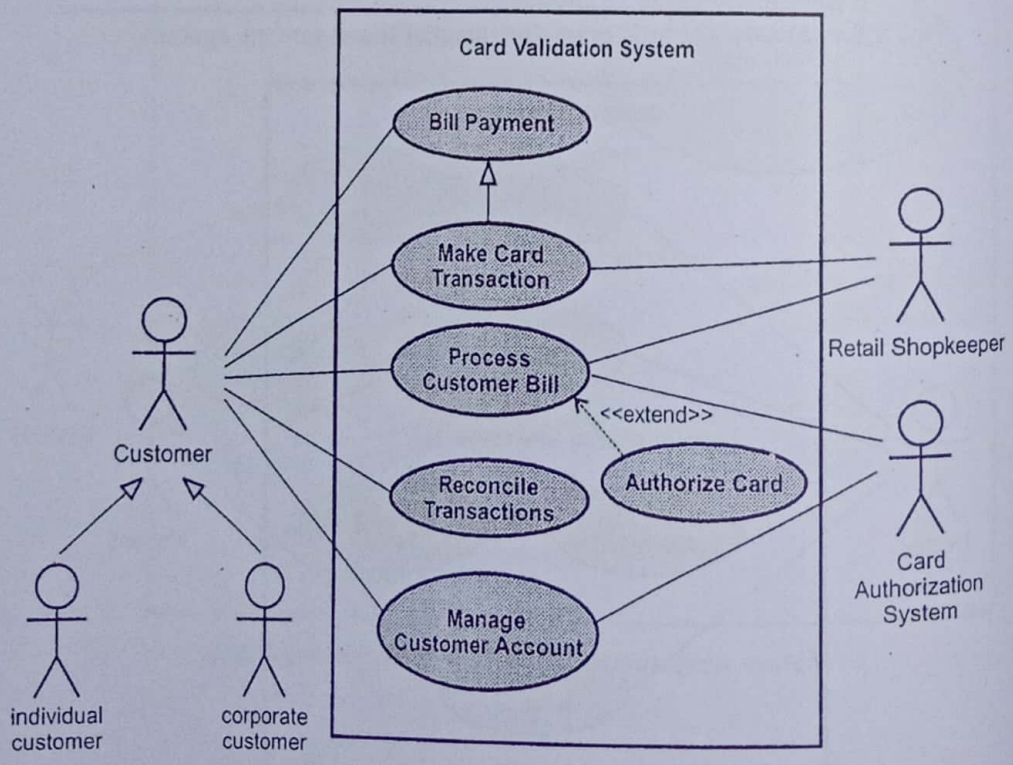


Fig. 3.8.8 Use case diagram

In the credit card validation system, there are various actors that surrounds the system. The Customer can be of Individual customer and Corporate Customer. There two more customers that are included in this transactions and those are Retail Shopkeeper and Card authorization system. Various functionalities involved are making card transaction, processing of bill, reconciling of transactions and customer account management.

Example 3.8.6 *Consider an automated soda machine that gives cool drinks. Draw use case model of the soda machine.*

Solution :

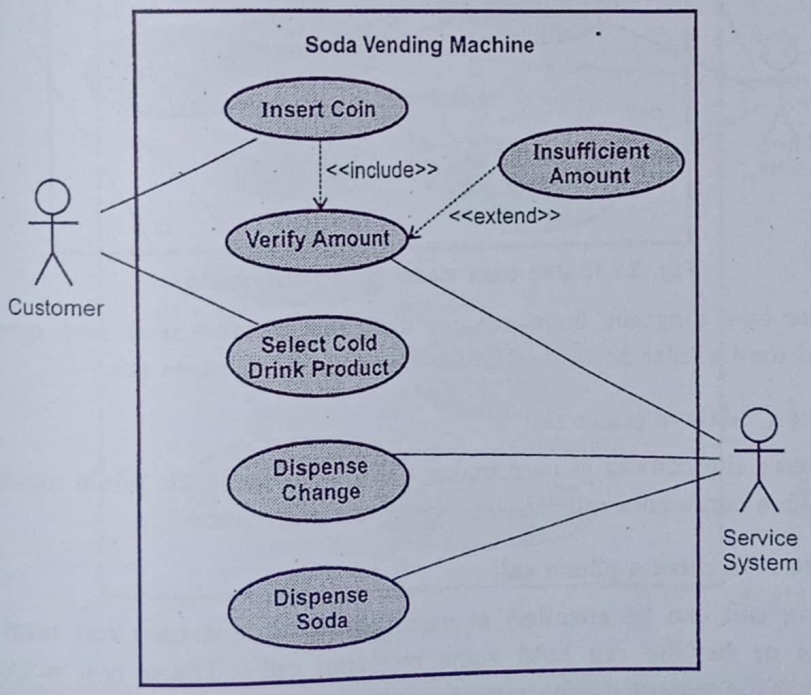


Fig. 3.8.9 Use case model for automated soda machine

Example 3.8.7 *Draw use case diagram to model the behavior of a cellular phone. Explain briefly.*

Solution :

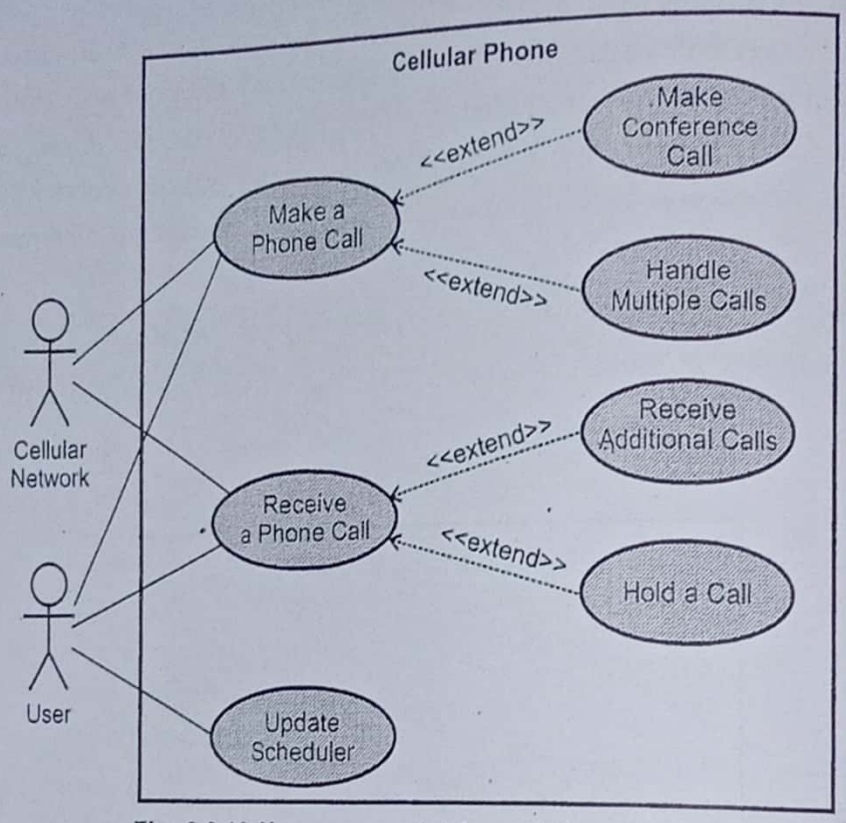


Fig. 3.8.10 Use case model for cellular phone

In above use case diagram, there are two important actors - user who operates the Cell phone and the Cellular network. The three important use cases are -

Use case Name : Make a phone call

In this use case the activity of user makes a call is represented. While making a call user might make a conference call for some group communication.

Use case Name : Receive a phone call

The receiving call can be attended in normal scenario. But user can receive some additional calls or he/she can hold some receiving call. These two activities are represented using the extended relationship.

Use case Name : Update scheduler

The user can update his/her schedule using calendar.

Example 3.8.8 *What is the importance of use case diagram ? Explain the relationships between use cases with suitable example and proper UML notations. Draw use case diagram for 'online railway ticket reservation system'.*

Solution : Importance of Use case diagram - Refer section 3.5.3.

Relationships in use cases : Refer section 3.8.

Use case diagram for online railway ticket reservation system

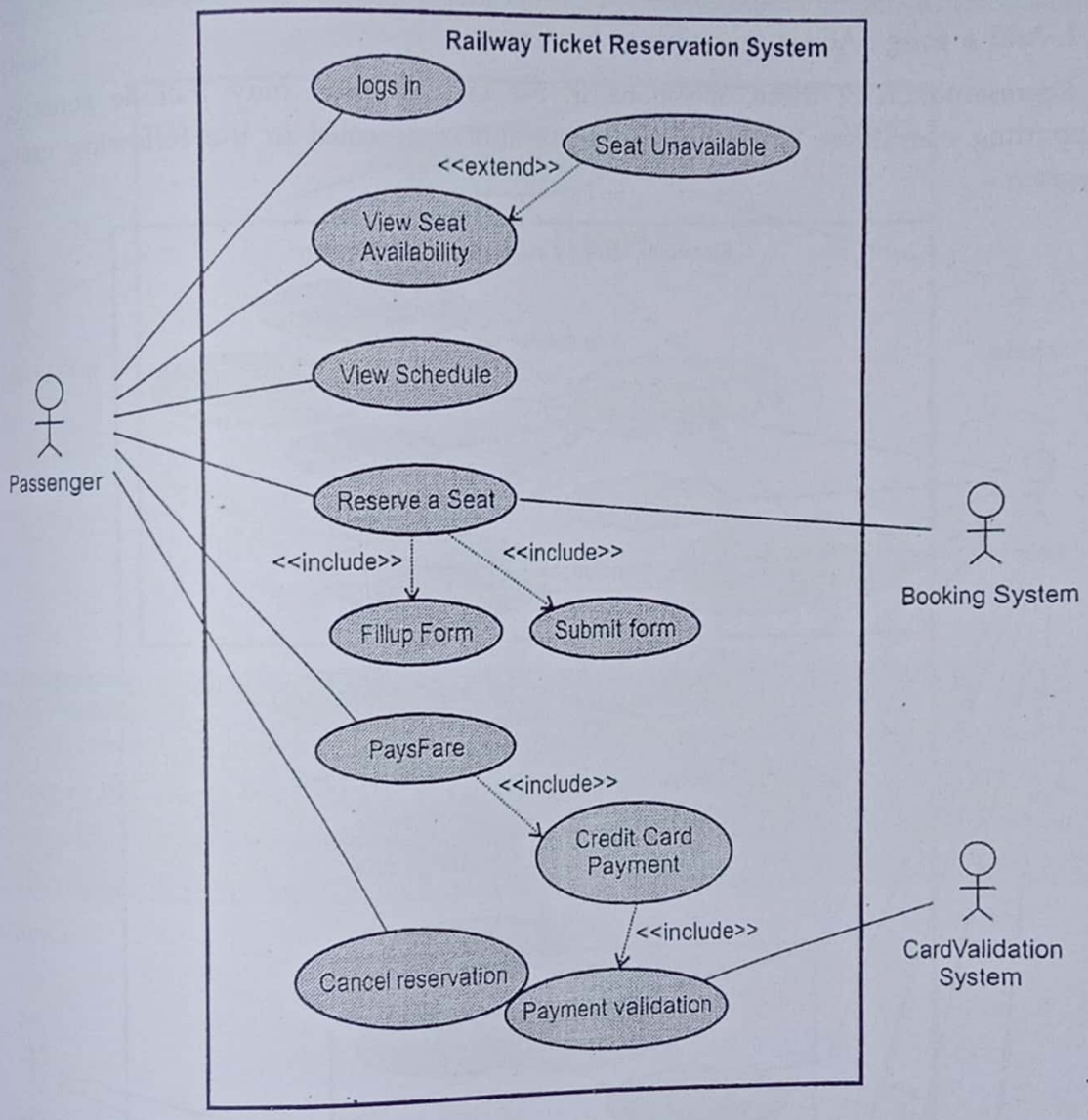


Fig. 3.8.11

Example 3.8.9 *Consider software that manages electronic music files. Prepare a use case diagram and include appropriate relationships for use cases.*
a. Play a song b. Play a library c. Randomized order d. Delete a song e. Destroy a song f. Add a song.

Solution : We will first elaborate each use case. Then the use case diagram can be prepared by adding the appropriate relationship.

a. **Play a song :** Add the desired song to the end of the play list.

b. **Play a library :** Add songs to the library for playing it.

c. **Randomized order :** Randomly reorder the songs in the play list.

d. Delete a song : Delete a song from library.

e. Destroy a song : Delete a song from all the libraries and destroy all the file.

f. Add a song : Add a music file to the library.

Representation of these operations in use case diagram may include some other supporting operations. These operations are also represented in the following use case diagram -

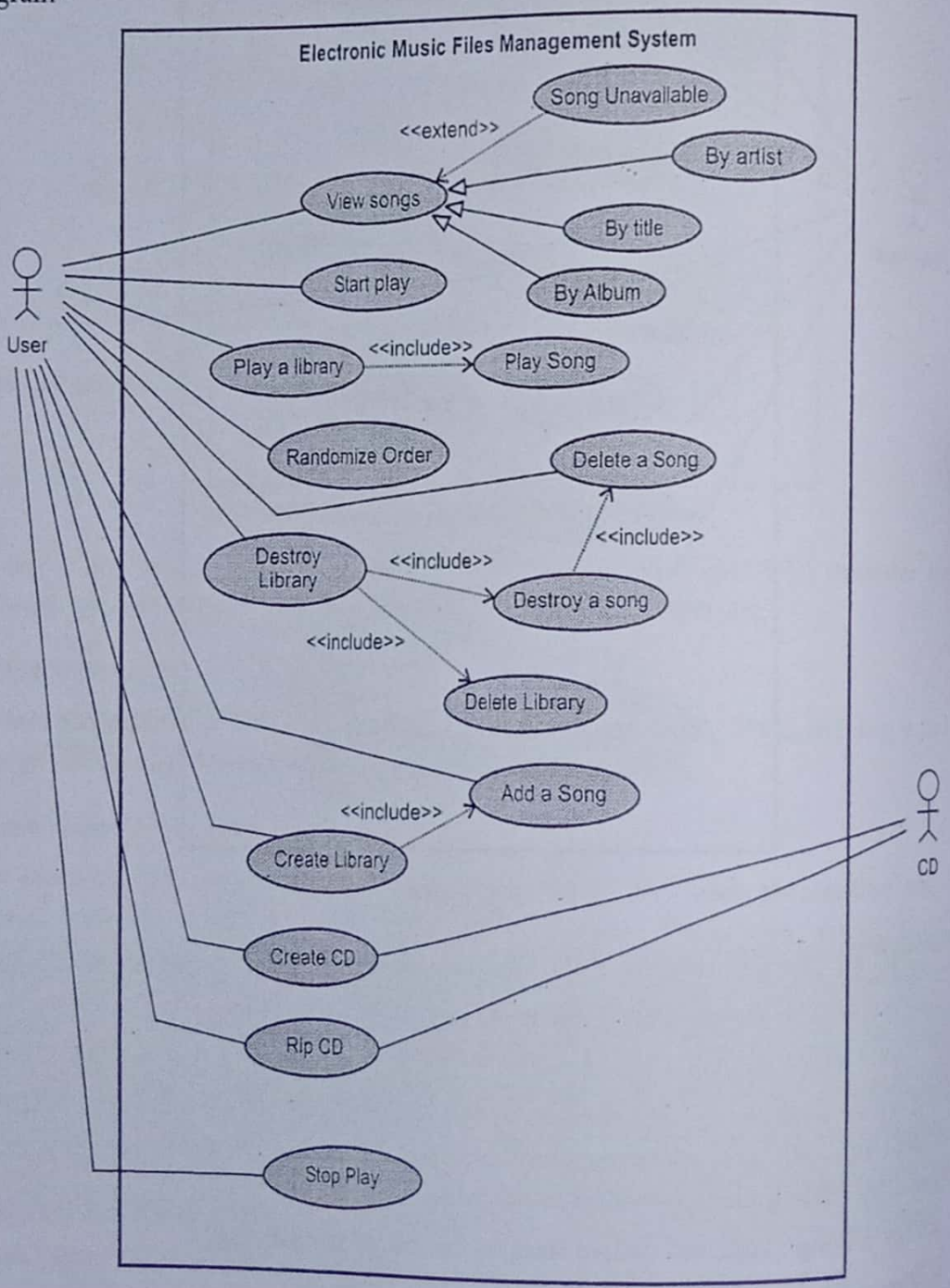
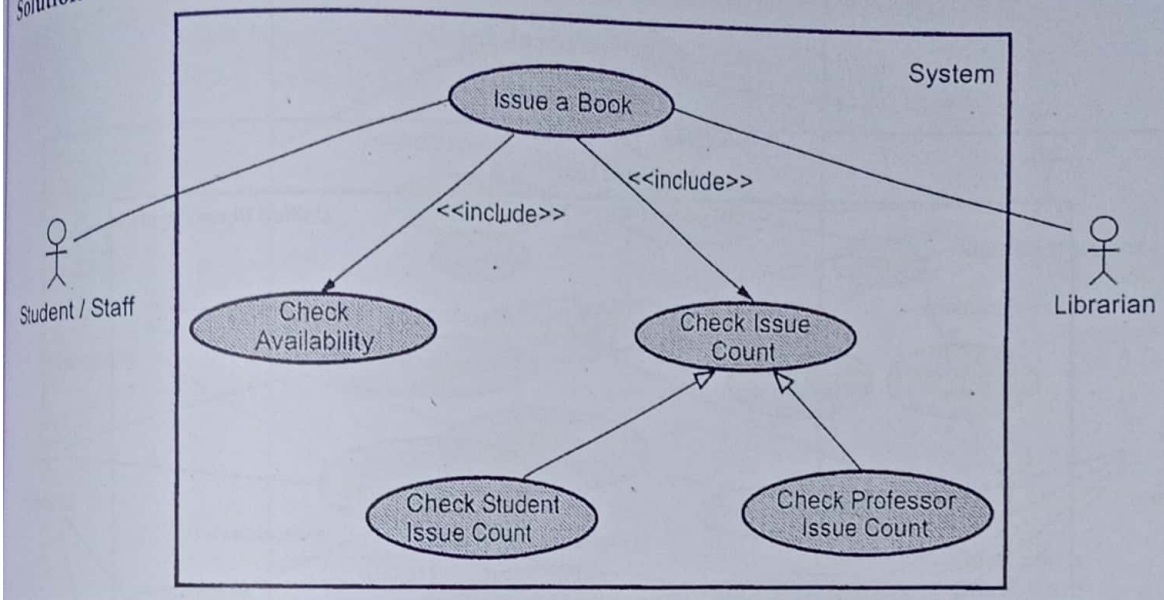


Fig. 3.8.12

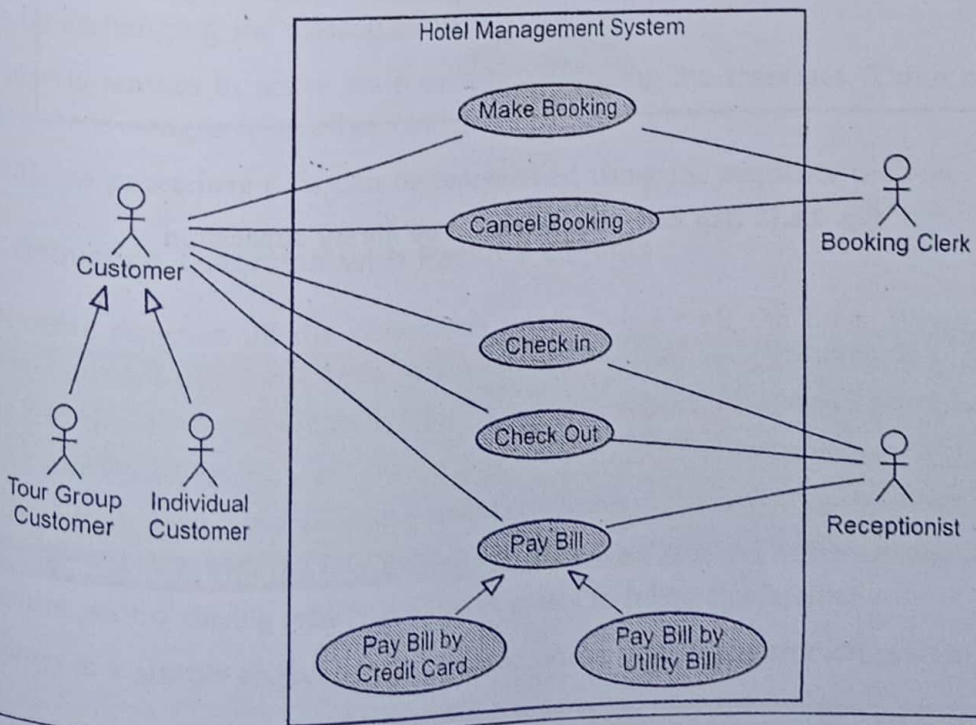
Example 3.8.10 *Prepare a use case description for issue a book from the library.*

Solution :



Example 3.8.11 *Draw the use-case diagram for hotel information system. There are two types of customers : Tour-group customers and individual customers. Both can book, cancel, check-in and check-out of a room by phone or via the internet. There are booking process clerk and reception staff who manages it. A customer can pay his bill by credit card or pay utility bill.*

Solution :



Example 3.8.12 *Draw Use Case diagram for online digital library information system with all advanced notations.* **SPPU : May-17, End Sem, Marks 5**

Solution :

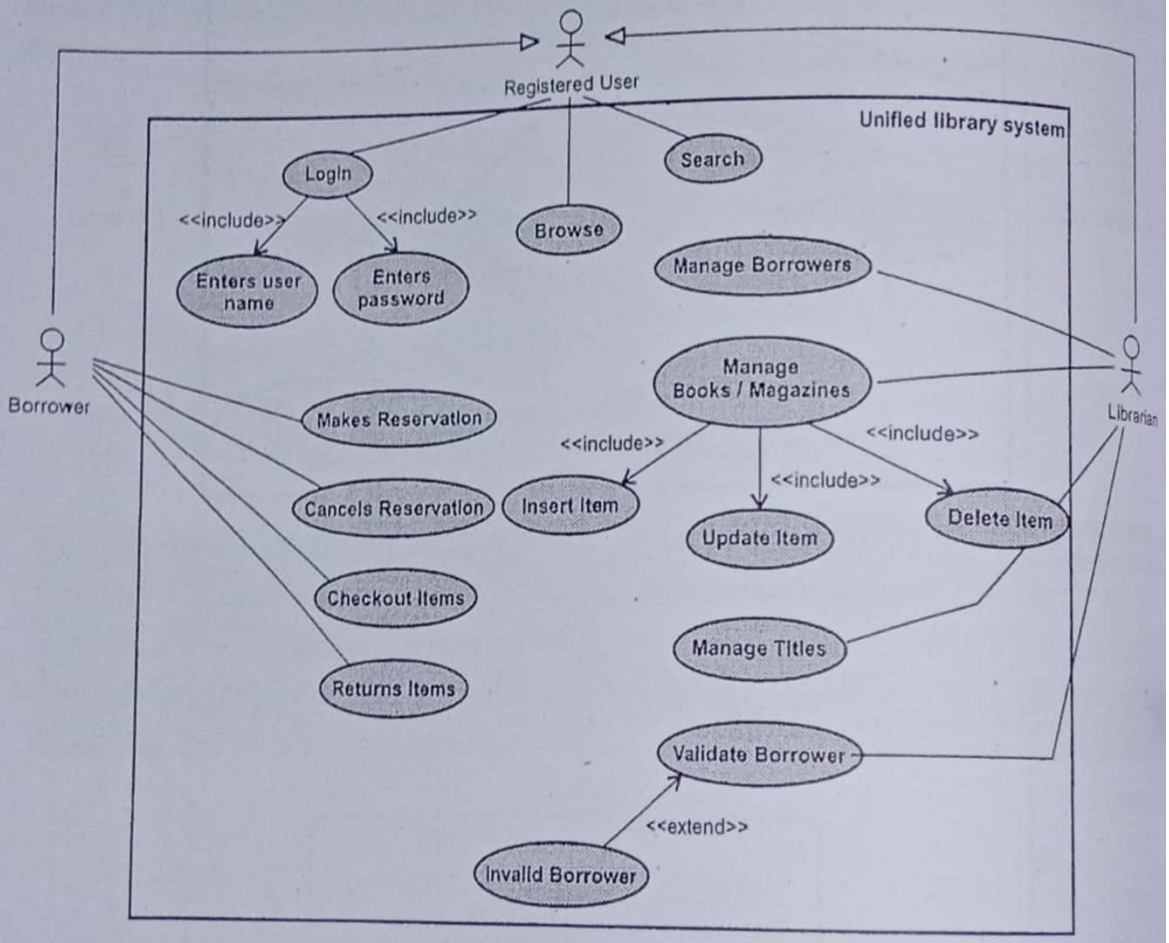
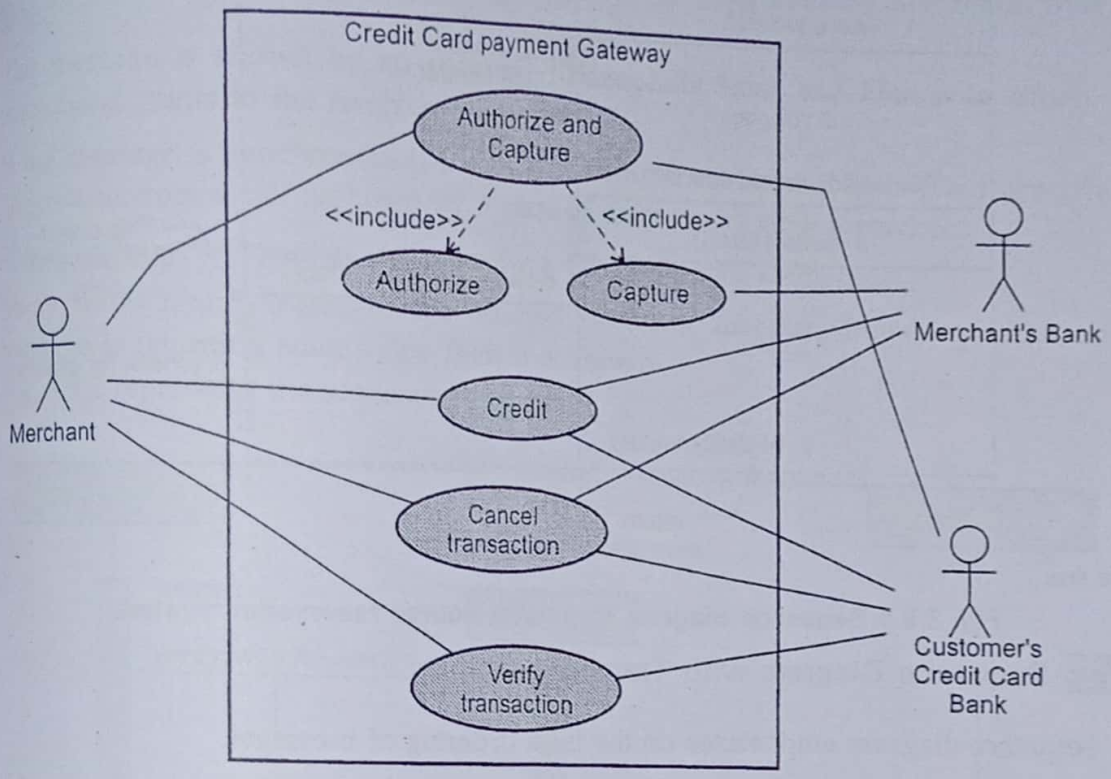


Fig. 3.8.13 Use case model for unified library application

Example 3.8.13 *Draw a use case diagram for the system. Credit card authentication system. The scenario is : The merchant submits a credit card trasnaction request to the credit card payment gateway on behalf of a customer. Bank which issued customer's credit card is actor which could approve or reject the transaction. If transaction is approved, funds will be transferred to merchant's bank account.* **SPPU : April-18, In Sem, Marks 5**

Solution :



3.9 Procedural Sequence Models

- The sequence diagram contains a collection of objects that are interacting with each other by exchanging the messages.
- The objects remain in active state even after sending the messages. These objects can receive the messages from other objects.
- In UML the **procedure calls** can be represented using the sequence models.

3.9.1 Sequence Diagrams with Passive Objects

- In sequence diagram all the objects are not active for all the time. When an object receives the message for invoking its procedure then that object becomes active and when the operation gets completed and control returns back to the caller then the object becomes passive.
- In UML, the period of time for an object's execution is shown as thin rectangle. This is called **focus of control** or **activation**.
- The entire period during which the object exists is called the **lifeline**.
- Following is a simple sequence diagram for online course registration system -

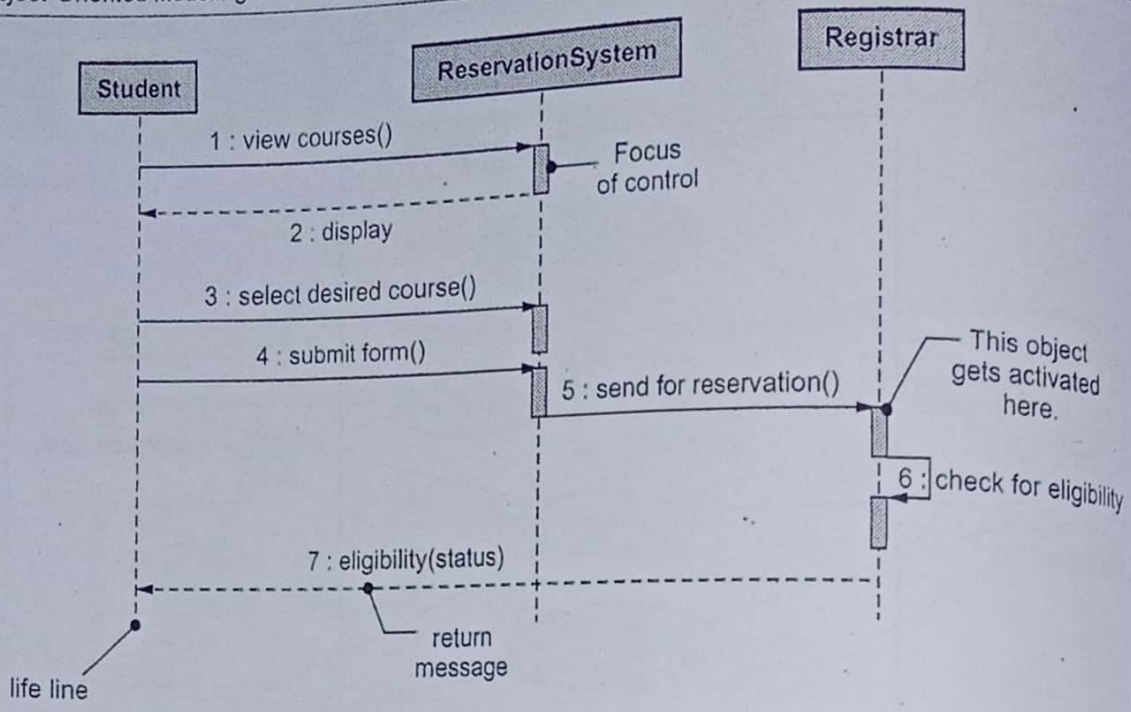


Fig. 3.9.1 Sequence diagram for online course reservation system

3.9.2 Sequence Diagram with Transient Objects

- A sequence diagram emphasizes on the time ordering of messages.
- The objects or the roles that take part in the interaction are placed at the top of the sequence diagram that are placed horizontally.
- The object that initiates the interaction is placed at the left and increasingly the subordinate objects or the roles are placed to the right.
- The messages that are passed among the objects are placed along the vertical axis, in order of increasing time from top to bottom.
- There is a lifeline in the sequence diagram. The lifeline is a vertical dashed line that represents the existence of the object over a period of time.
- Objects exists in the sequence diagram for the complete period of their interaction. Hence they are placed at the top of the diagram with a line that runs vertically from top to bottom.
- Using **create** message the objects are created and using **destroy** message the objects are destroyed.
- If the interaction represents the history of specific object then the objects are underlined. But usually we do not underline the objects.
- A tall thin rectangle represents the focus of control in the interaction diagram. It shows the period of time during which the object is performing its interaction. The top of the rectangle is aligned with the start of the action and the bottom of the rectangle is aligned with the end of the action.

- The nesting focus can also be shown by a self loop or by stacking one action over the other.
- The **message** is shown by an running horizontally from one lifeline to other. The arrowhead points to the receiver.
- If the message is synchronous (i.e. a call) then it is represented by a filled arrowhead. The return from a call is shown by a dashed arrow with stick arrowhead.
- The asynchronous message is shown by a line with stick arrowhead.
- Many times return messages are omitted from the diagram, however if the return message is returning some value then it is shown.

Fig. 3.9.2 represents the sequence diagram.

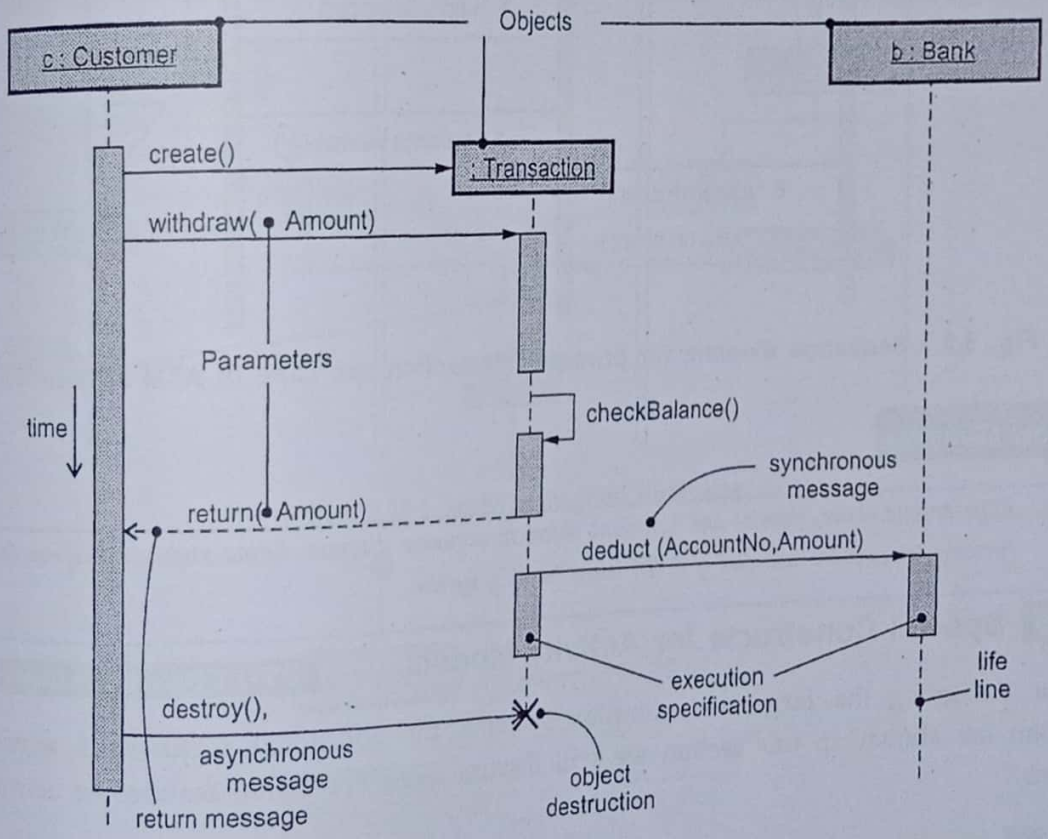


Fig. 3.9.2 Sequence diagram

3.9.3 Guidelines for Procedural Sequence Models

Active Vs Passive Objects : The active objects are always activated and they have their own focus of control. Whereas the passive objects are passive and they do not have their own thread of control.

Advanced Features : By using advanced features in sequence diagram, the implementation becomes easy. Only show implementation details for difficult or in important sequence diagrams.

Example -

Following is a sequence diagram for process transaction use case of ATM based banking system.

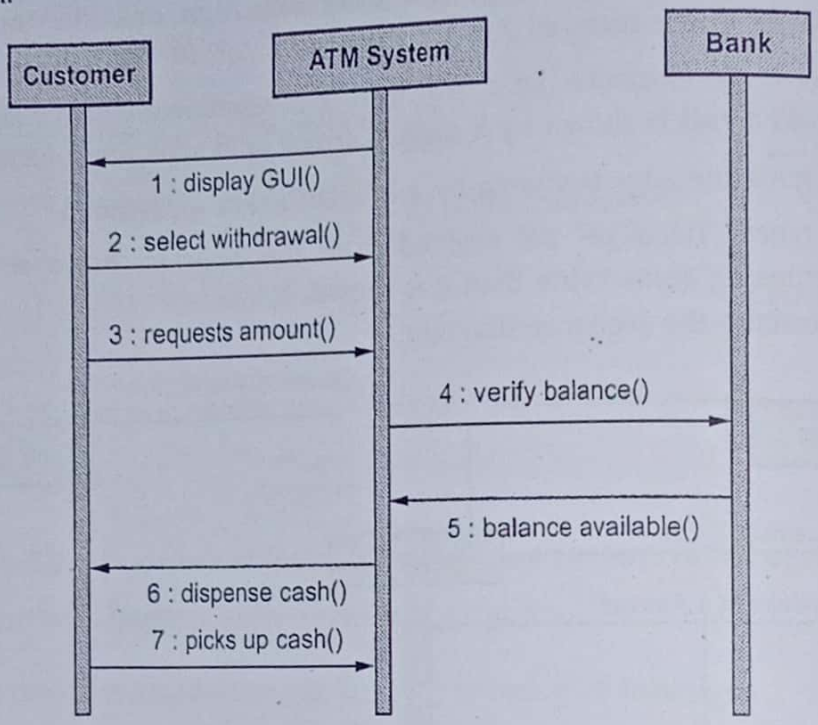


Fig. 3.9.3 Sequence diagram for process transaction use case of ATM system

Review Question

1. *Differentiate active, passive and transient object in sequence diagram. Draw sequence diagram for 'process transaction' use case of ATM based banking system.*

3.10 Special Constructs for Activity Models

SPPU : Dec.-16, Marks 5

For designing the large and complex systems the additional features of activity diagram are shown. In this section we will discuss some additional features of activity models.

3.10.1 Sending and Receiving Signals

In activity diagram, the sending signal activity can be represented as convex pentagon. The receiving signal is represent by concave pentagon.

In the activity diagram, when previous activity gets completed then only the sending signal can be shown and when preceding activity completes the receipt construct waits until the signal is received. Only after receiving the signal the next activity starts.

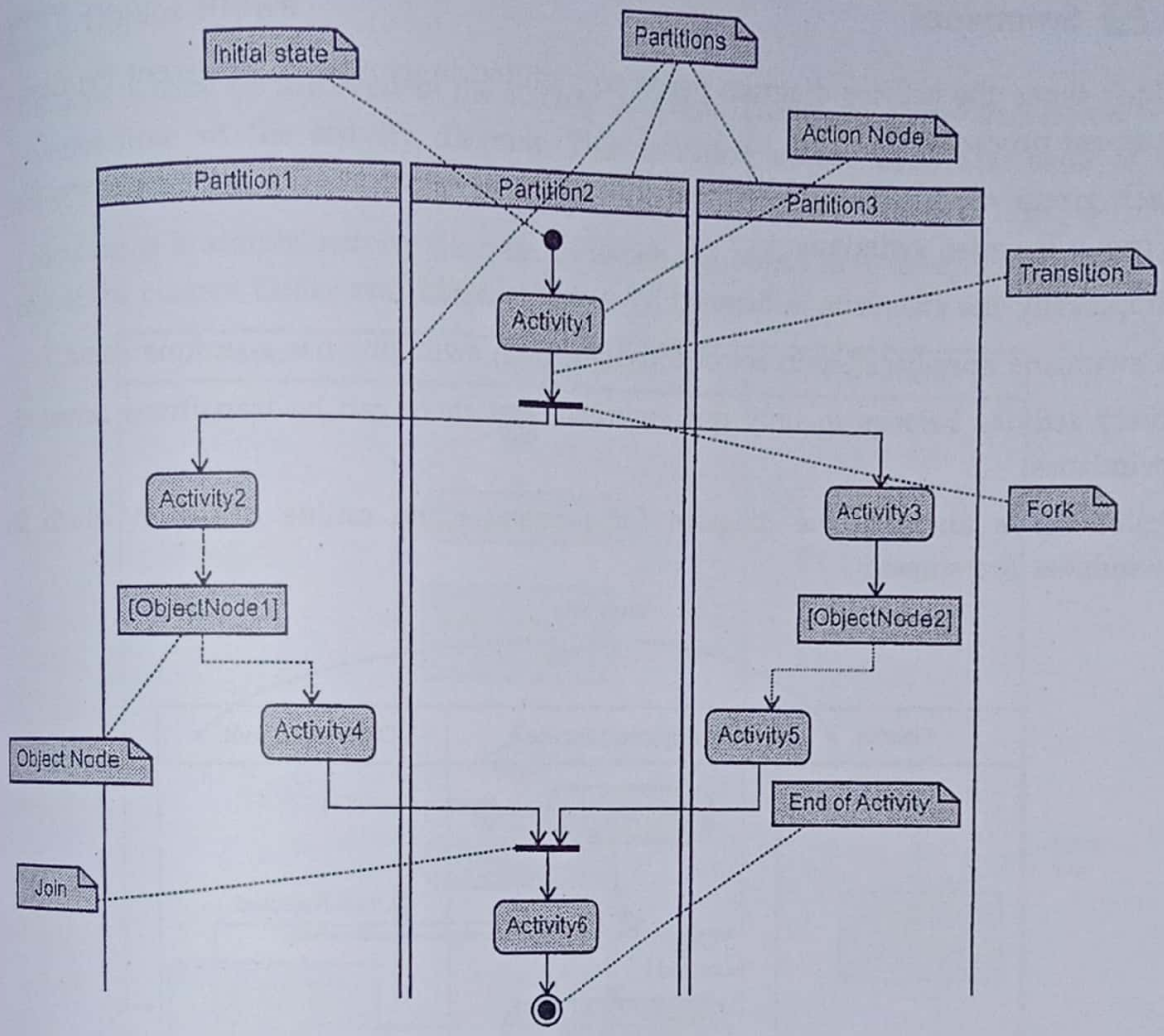


Fig. 3.10.1 Activity diagram notations

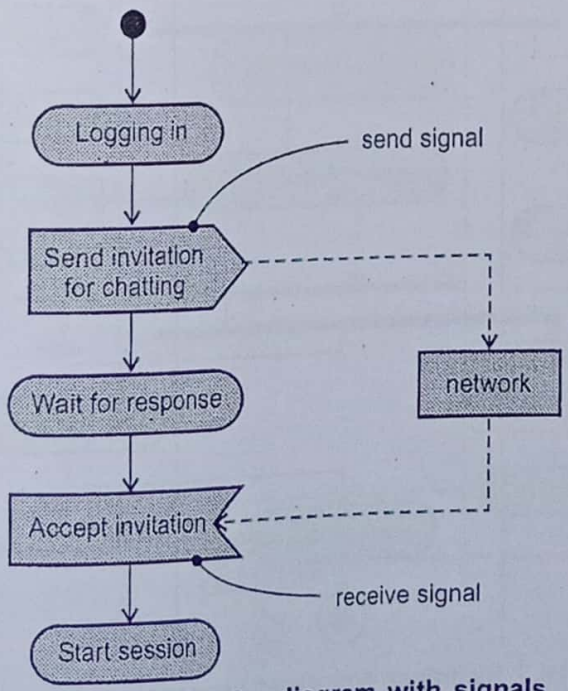


Fig. 3.10.2 Activity diagram with signals

3.10.2 Swimlanes

- Many times the activity diagram needs to partition in groups to represent the flow of business processes.
- Each group represents the particular business flow belonging to some category. Such a group is called **swimlane.**
- Graphically the swimlane is denoted by a vertical solid line.
- A swimlane specifies certain set of activities. Each swimlane has a unique name.
- Every activity belongs to only one swimlane but there can be transitions across the swimlanes.
- Following is an swimlane diagram for processing the **online order** in which the swimlanes are shown.

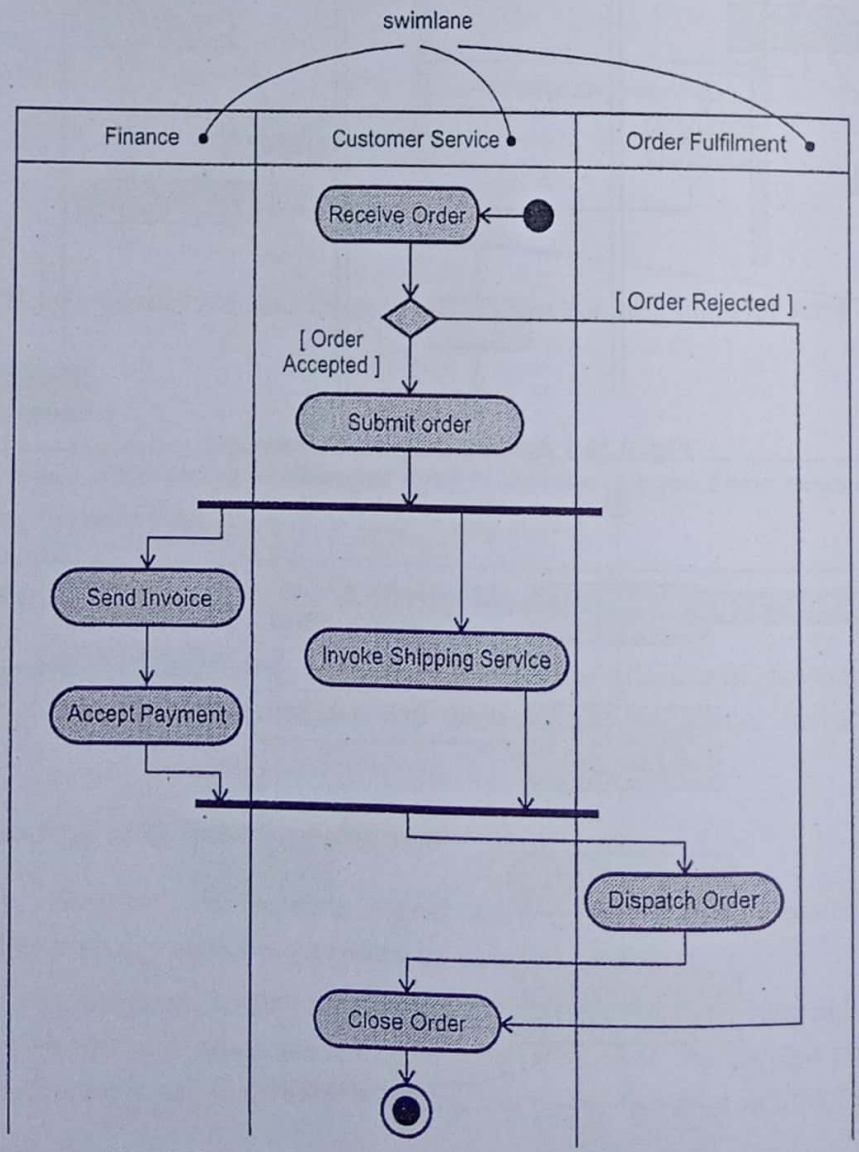


Fig. 3.10.3 Swimlane for order processing system

3.10.3 Object Flows

The objects may be involved in the activity diagram. These objects are associated with the control flow of the activity diagram. This is called **object flow**. The state of the object is specified for each of these objects.

Following is a simple activity diagram in which the object flow is represented for the objects of the classes **Order** and **Invoice**

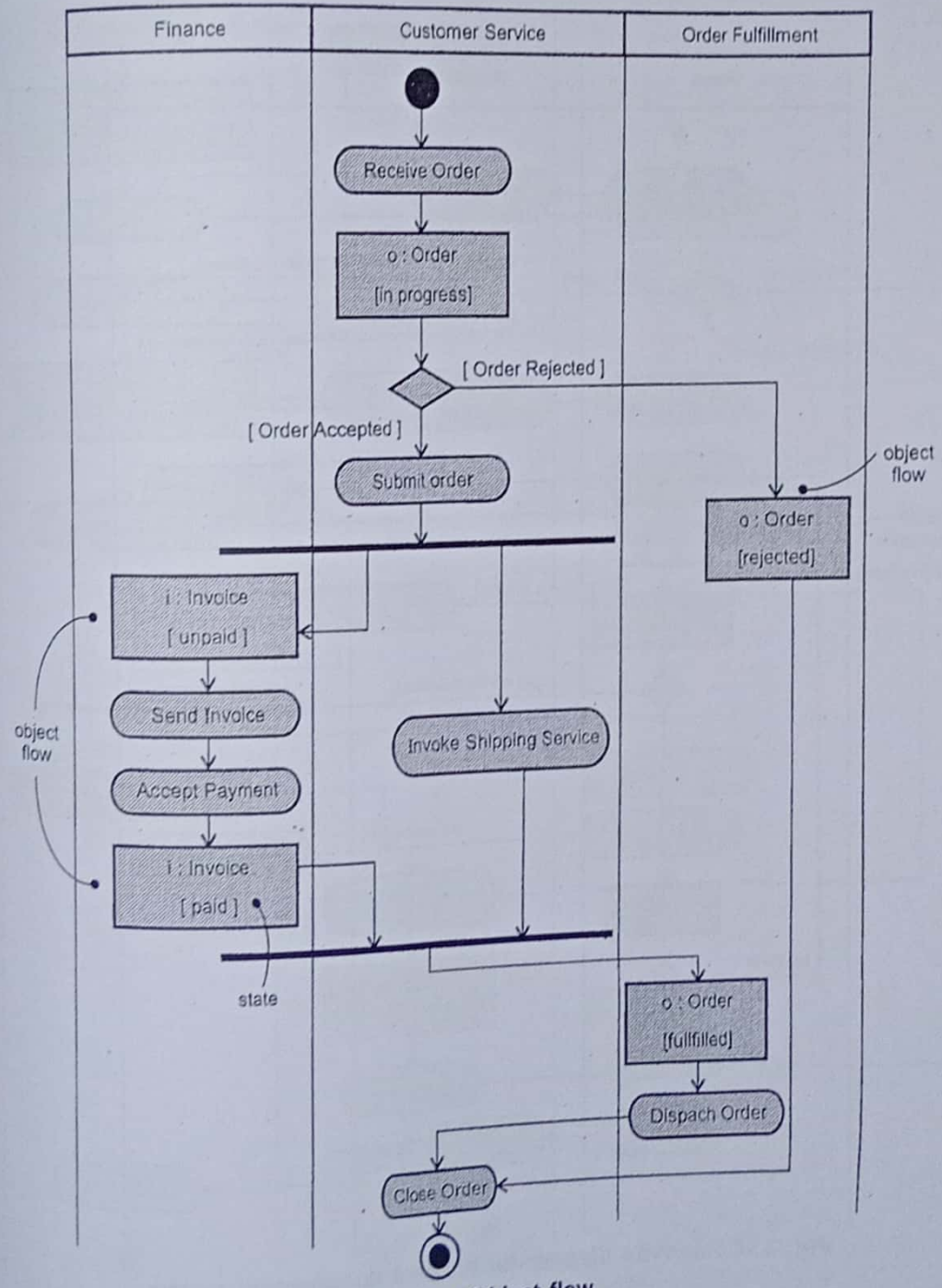


Fig. 3.10.4 Object flow

3.10.4 Synchronization Bars

The synchronization bars are the vertical or horizontal bars in the activity diagram. The fork and join is represented in activity diagram by synchronization bar. (Refer Fig. 3.10.1)

3.10.5 Examples

Example 3.10.1 *Give activity diagram for hospital management system.*

Solution :

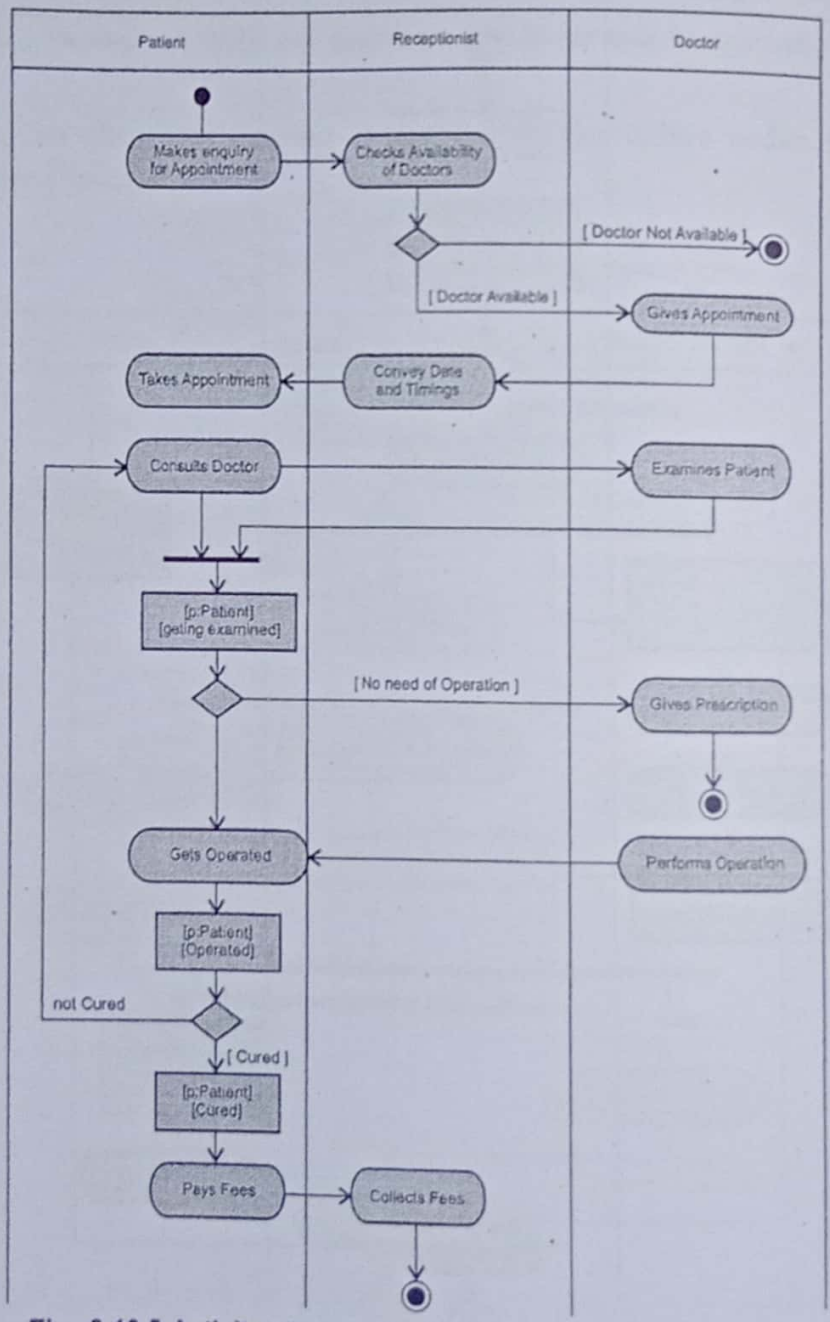


Fig. 3.10.5 Activity diagram for hospital management system

Example 3.10.2 *Draw activity diagram for withdrawal of money from ATM.*

SPPU : Dec.-16, Marks 5

Solution :

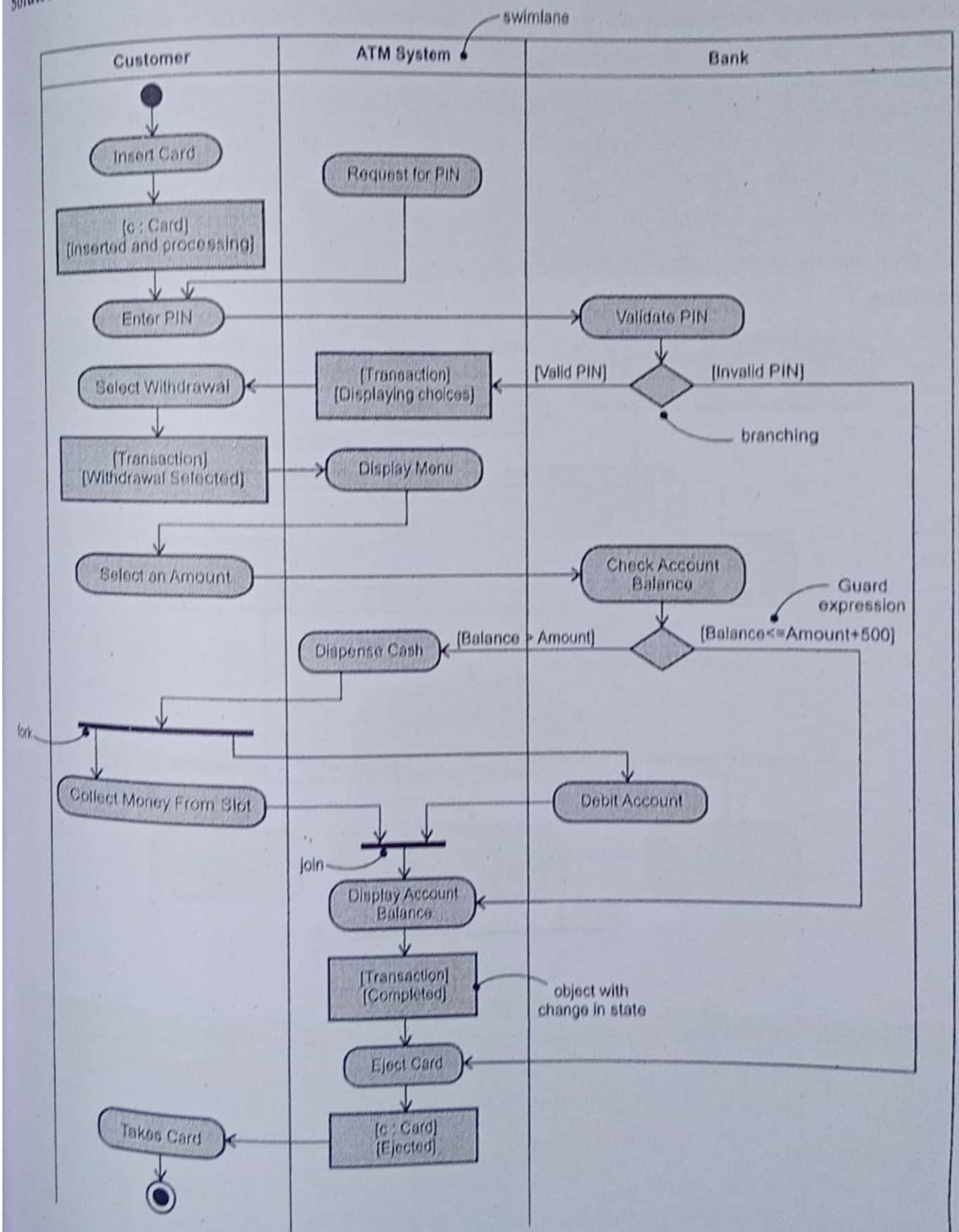
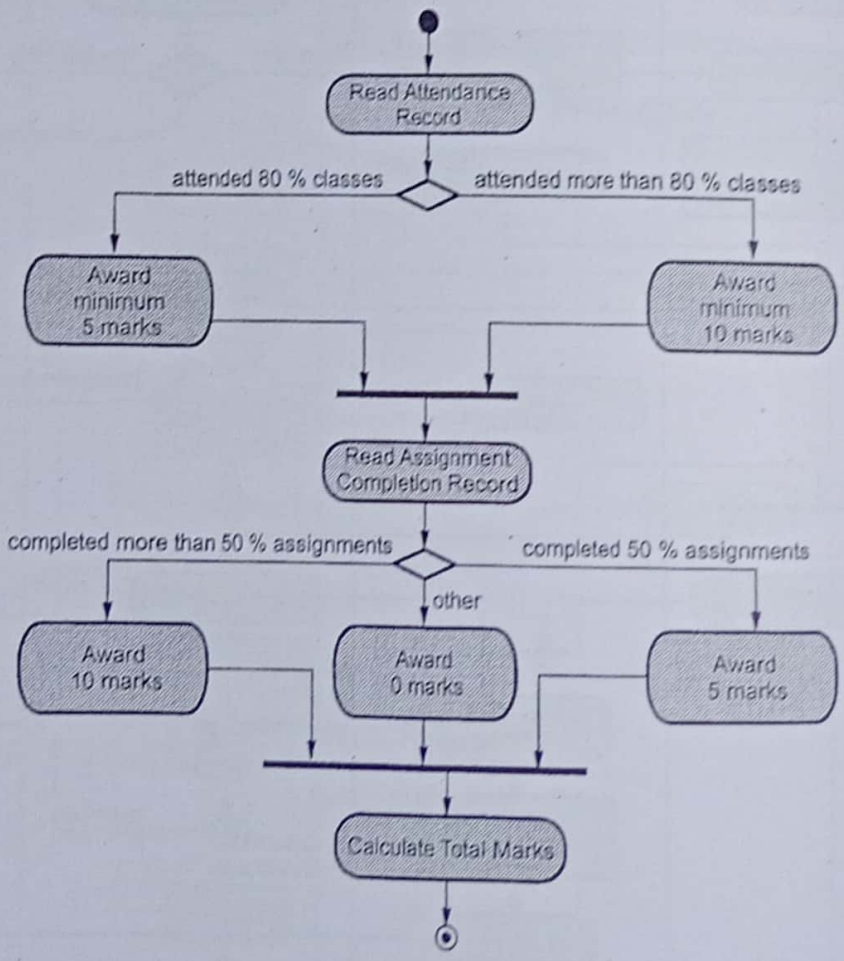


Fig. 3.10.6 Modeling workflow

The activity diagram given in Fig. 3.10.6, is for handling the ATM System. The Customer **withdraws money** from ATM machine. This workflow is illustrated in this figure. There are two objects **Card** and **Transaction** that are shown explicitly which take part in the workflow. Note that the state of these objects gets changed due to the actions that are carried out during the processing. This workflow involves branching, fork and join. It is little bit complex workflow.

Example 3.10.3 *Prepare an activity diagram for awarding marks to regular students. If the student has attended 80 % classes, he is awarded minimum 5 marks. If the student has attended more than 80 % classes, he is awarded minimum 10 marks. The students who have completed assignments are given 10 marks. Those who have completed 50 % are given 5 marks and rests are given 0 marks.*

Solution :



Example 3.10.4 *Prepare an activity diagram for computing a restaurant bill. There should be a charge for each delivered item. The total amount should be subject to tax. There is a service charge of 18 % for groups of six or more and 10 % for smaller groups. Any coupons and gift certificates submitted by the customer should be subtracted.*

Solution :

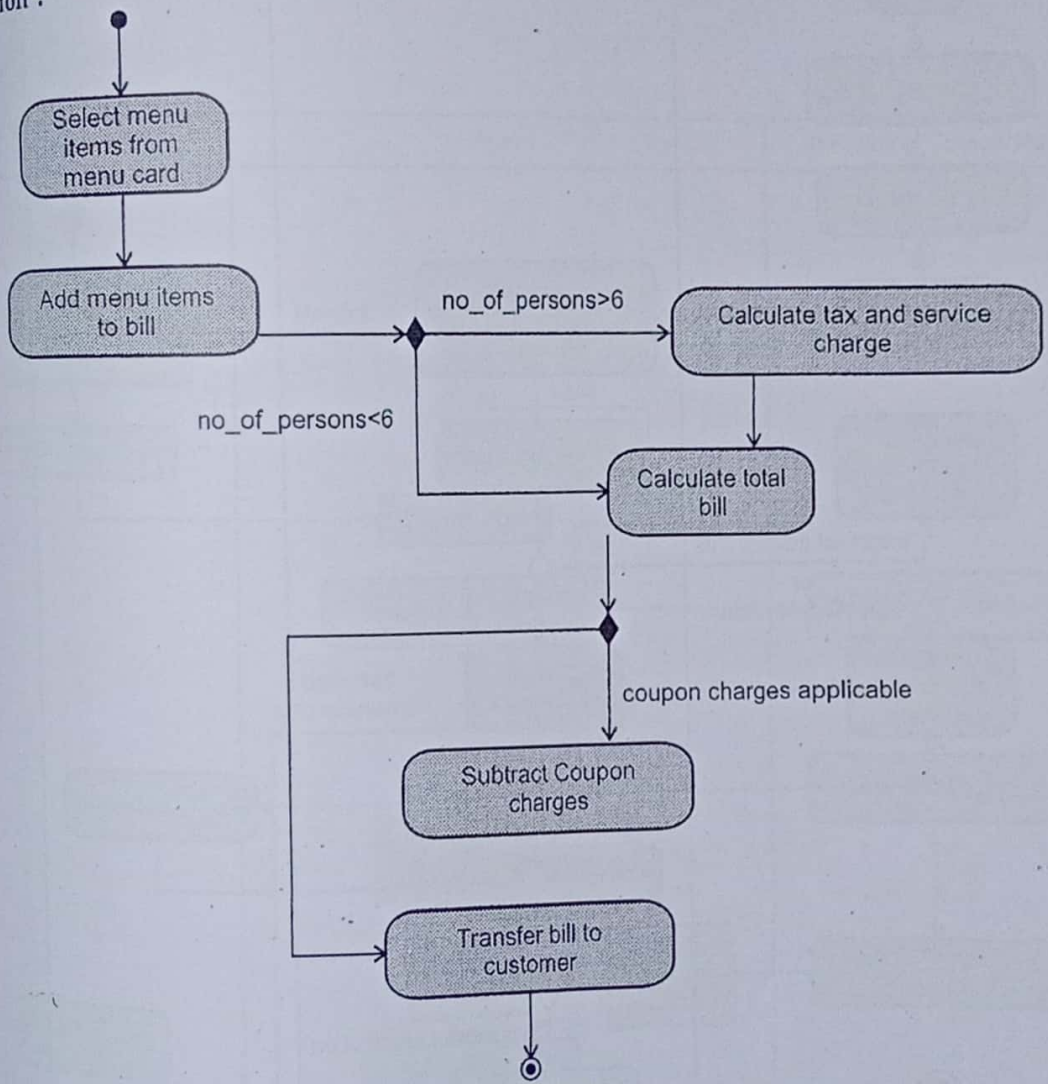


Fig. 3.10.7

Example 3.10.5 *A draw an activity diagram for a passport management system to get a new passport, an applicant has to apply on-line, get the appointment. He has to submit the documents in passport office on the date of appointment. In case of insufficient or incorrect documents, the applicant's has to reapply and get new appointment. After submission of documents, applicant's verification is done by the police. On successful verification, passport is issued to the applicant. If verification is unsucessful, applicant has to reapply for passport.*

Solution :

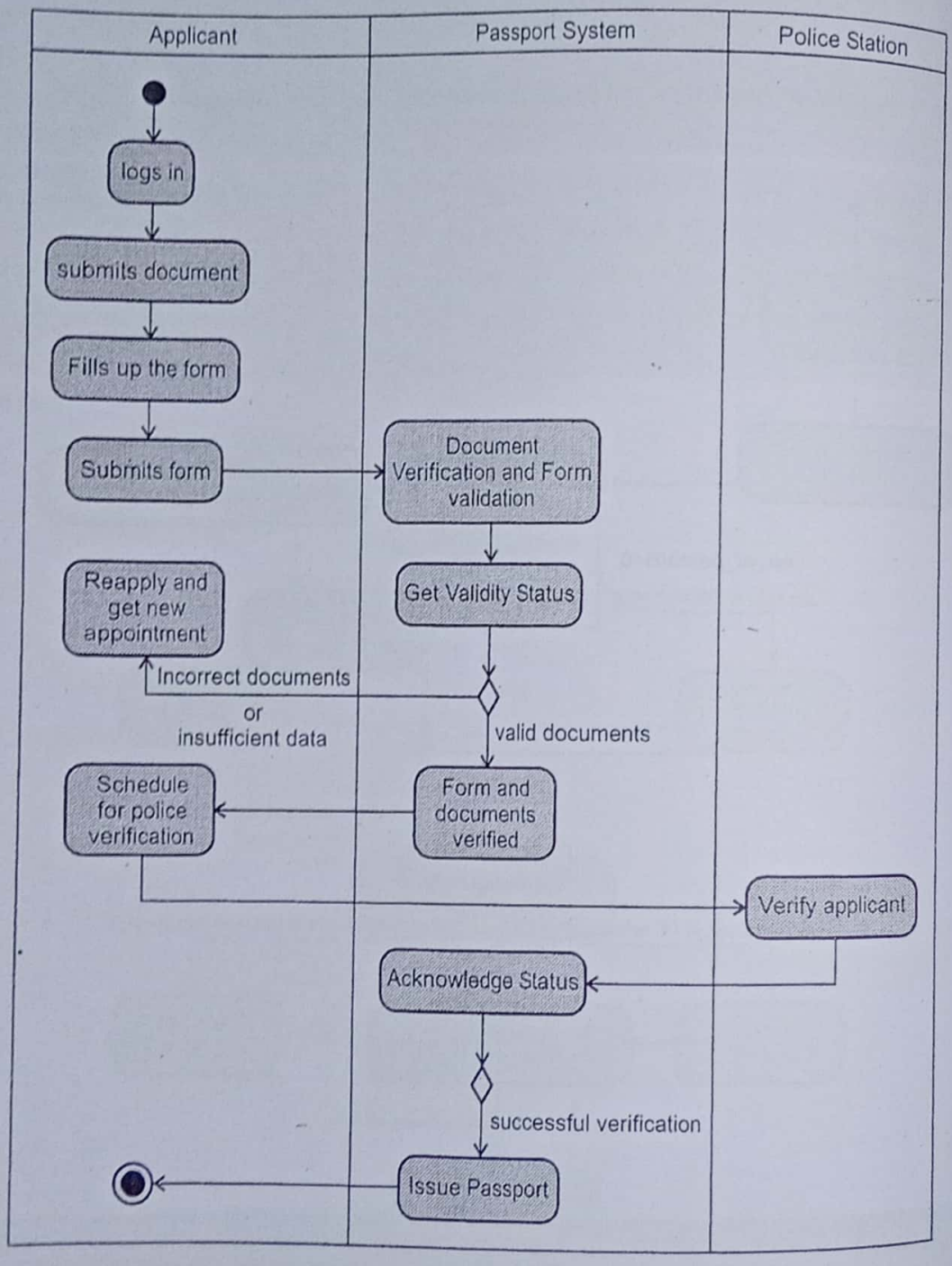


Fig. 3.10.8

Example 3.10.6 *Insurance system provides vehicle insurance to the owners. Initially the customer fill the form which has vehicle details and owner's details. This information is submitted to the agent. The agent sends the information to various insurance companies. The companies quote the insurance. The agent selects the best policy and gives it to the owner. Draw an activity diagram with swimlane and other applicable notations.*

Solution :

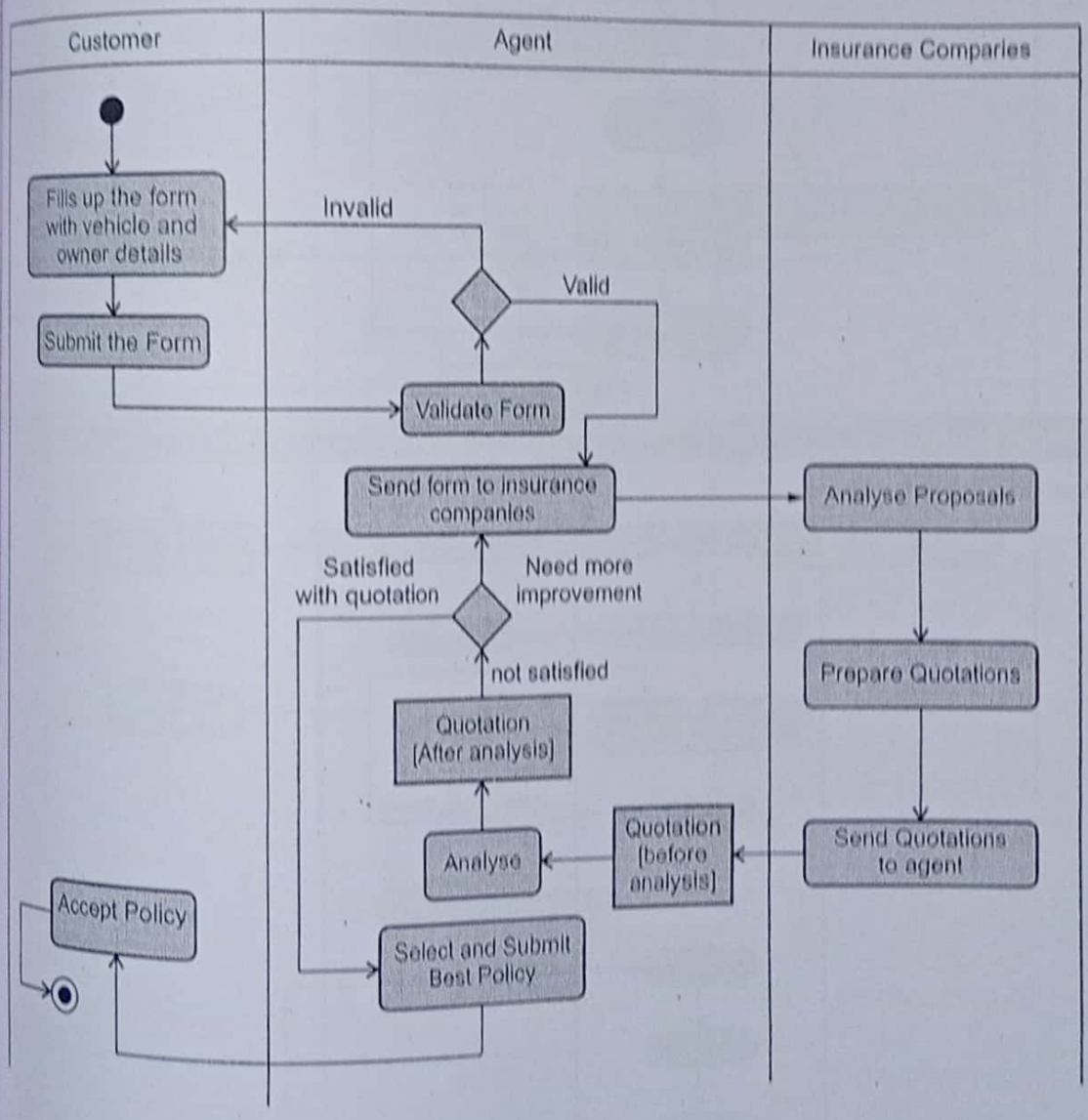


Fig. 3.10.9

Example 3.10.7 *Draw activity diagram for issue of book in library information system.*

Solution :

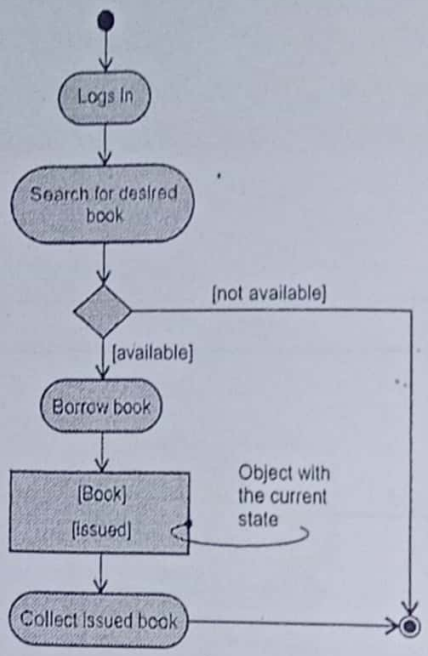


Fig. 3.10.10

Example 3.10.8 *Draw swimlane diagram for exam registration systems.*

Solution :

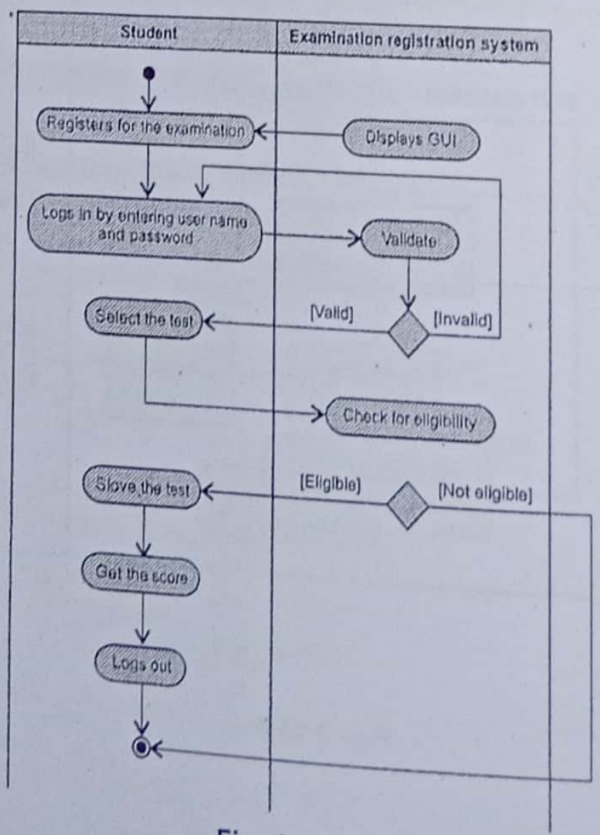


Fig. 3.10.11

Example 3.10.9 *Draw activity/swimlane diagram for Stock maintenance systems.*

Solution :

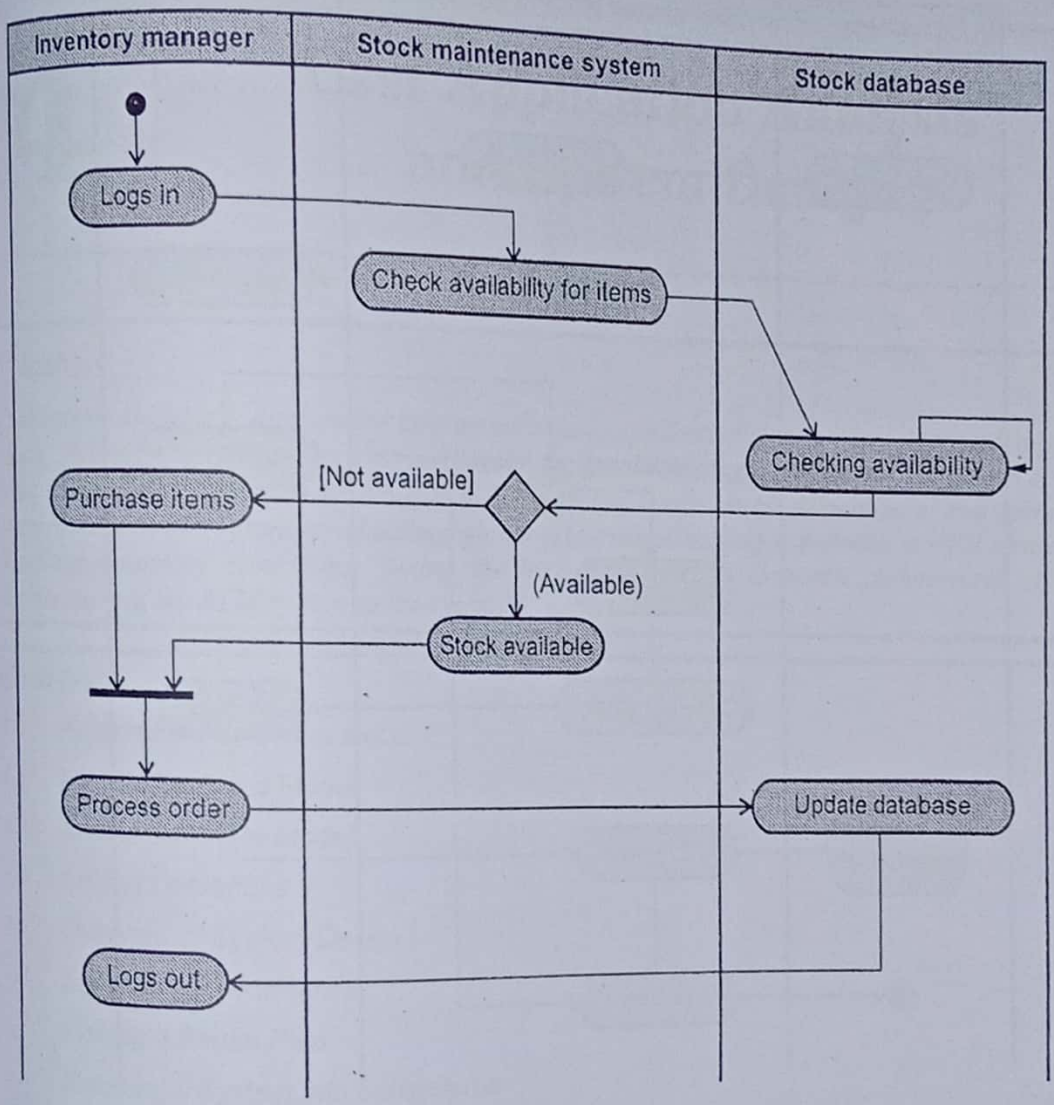


Fig. 3.10.12

Example 3.10.10 *Draw activity/swimlane diagram for credit card processing system.*

Solution :

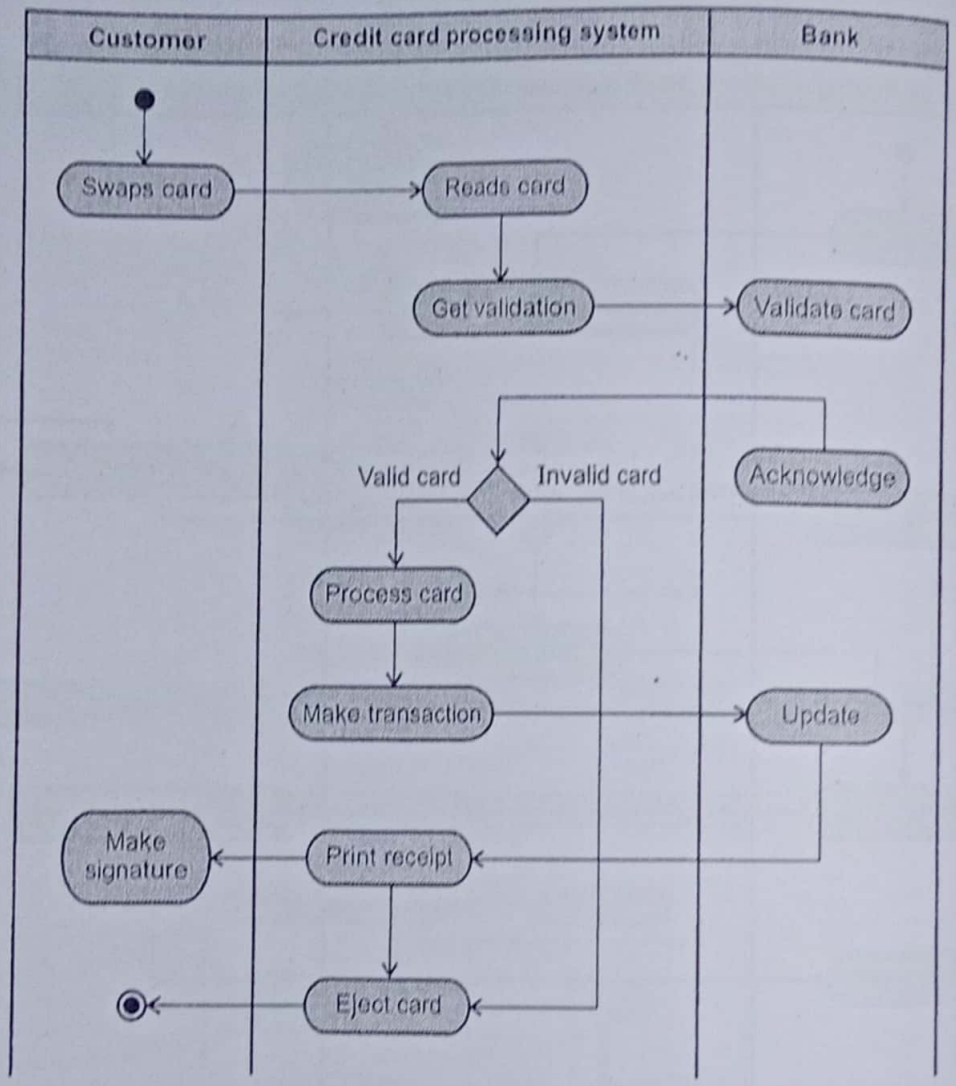


Fig. 3.10.13

Review Question

1. *Explain the purpose of activity diagram ? In which situation activity diagram is not necessary ? Explain the use of following concepts for activity diagram : Synchronization bar, swimlane and sending-receiving signals.*

□□□

Unit IV

4 User Application Analysis and System Design

Syllabus

Application Analysis : Application interaction model; Application class model; Application state model; Adding operations. Overview of system design; Estimating performance; Making a reuse plan; Breaking a system in to sub-systems; Identifying concurrency; Allocation of sub-systems; Management of data storage; Handling global resources; Choosing a software control strategy; Handling boundary conditions; Setting the trade-off priorities; Common architectural styles; Architecture of the ATM system as the example

Contents

Part I : User Application Analysis

4.1 Application Interaction Model

- The domain model creates a static model and the operations represented in this model are not of much importance because the focus of domain modeling is merely on intrinsic concepts of the application.
- Hence after completing the domain model, the focus should be to build the application model.
- First of all the application interaction model is to be built because in this model the boundary of the system is determined. Then identify the use cases, prepare the scenarios and design the sequence diagram. For the complex use cases the activity diagram can also be prepared. Finally check the interaction model against the domain class model.

  Let us discuss these steps in more detail.

4.1.1 Determining System Boundary

The system boundary means the scope of the application. This helps in specifying the **functionality**.

The system boundary determines what the system should include and what it should omit.

The system boundary can be viewed as a **black box** in its interaction with outside world.

4.1.2 Finding Actors

- Actors are the external objects that interact directly with the system.
- After determining the system boundary it becomes very continent to find out the system boundary.
- The actor can be humans, external devices and software systems. But actors are not under the control of the application.
- There can be more than one actors that interact with the system from the external world.
- **For example :** For **online Course Registration system, Student** and **Accountant, Registrar** are the actors that interact with the system

4.1.3 Finding Use Cases

- Every use case represent the fundamental unit of **functionality**.
- The use cases are drawn in order to **represent the behavior of the system.**
- Each use case represents the kind of service that the system provides. It provides certain value to the actor.
- **For example :** For the Online Course Reservation System, the use case can be as shown below -

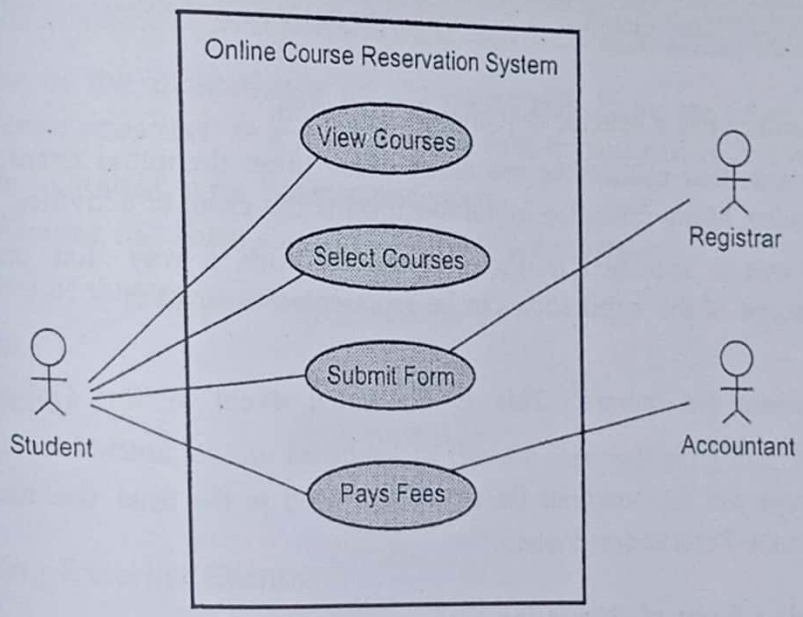


Fig. 4.1.1 Use case for online course reservation system

- **View Courses :** The student logs in to the system. He scrolls through various available courses and the eligibility criteria.
- **Select Courses :** Student select the desired appropriate course.
- **Submit Form :** Student fills up personal details and submits the form. This form is viewed by the Registrar for verification.
- **Pays Fees :** If the form is verified, then the accountant calculates the fees. Student pays the fees to the accountant and confirms the admission.

4.1.4 Finding Initial and Final Events

- In use cases the functionalities are partitioned into discrete pieces and actors are involved in these functionalities.

View Courses	Student logs in to the system. The system displays the page containing the list of available courses along with some eligibility criteria(if any). The student scrolls through the courses.
Select Course	The student selects the desired course by clicking the course
Submit Form	The system displays a registration form. Student fills up the form by entering the personal details. Student clicks the submit button to submit the form. The form is validated by the Registrar. If the form is valid then system displays the form acceptance message.

- Each functionality has a specific sequence of execution.
- There are events that initiate the use cases. Many times the initial event is - request for some service. Many times the initial event starts the chain of activities.
- The final event is specified in the use case in such a way that one complete operational scope of the application can be represented completely.
- For example :

  "Student views the courses". This is the initial event in the Online Course Reservation system.

  "Student pays fees and confirms the admission". This is the final use case event in Online Course Reservation System.

4.1.5 Preparing Normal Scenarios

- **Scenario** is the sequence of events among the set of interacting objects.
- The logical correctness of the scenario depends in the sequence of interactions.
- For normal case prepare the scenarios without any unusual inputs or error conditions. The information values exchanged during the events become the event parameters.
- For example : In the **Select Course** use case the name of the course becomes the event parameter.
- The normal scenario for each use case is as follows -

Submit Form	The system displays a registration form. Student fills up the form by entering the personal details. Student clicks the submit button to submit the form. The form is validated by the Registrar. If the form is valid then system displays the form acceptance message.

Pays Fees	The system displays the options for payment. System displays the amount of fees to be paid. Student selects the payment mode. Students fill up the required information for payment. Student pays the fees. The system verifies the payment. The payment is accepted by the system. The admission confirmation message will be displayed by the system.

4.1.6 Adding Variations and Exception Scenarios

The behavior of the application is not always normal. There are some situations in which special cases arise such as inputting wrong data, invalid entries and so on.

For example - Online Course Reservation System

- Selection of course for which eligibility criteria is not satisfied.
- Unavailability of desired course
- Invalid form
- Invalid payment

These additional scenarios can be added to the use case modeling and the scope can broadened.

4.1.7 Finding External Events

- All the scenarios are observed in order to find the external events.
- These external events are all inputs, decisions, interrupts, and interaction from the users.
- Transmitting the information to an object is also an event. For example submitting a form to the online course reservation system is an event.
- The flow of control might be affected by the series of events.
- Prepare a sequence diagram for each scenario. The sequence diagram clearly shows the sender and receiver of each event.
- In sequence diagram, each object is represented as a separate column.

For example : The sequence diagram for Online Course Reservation system is as shown below.

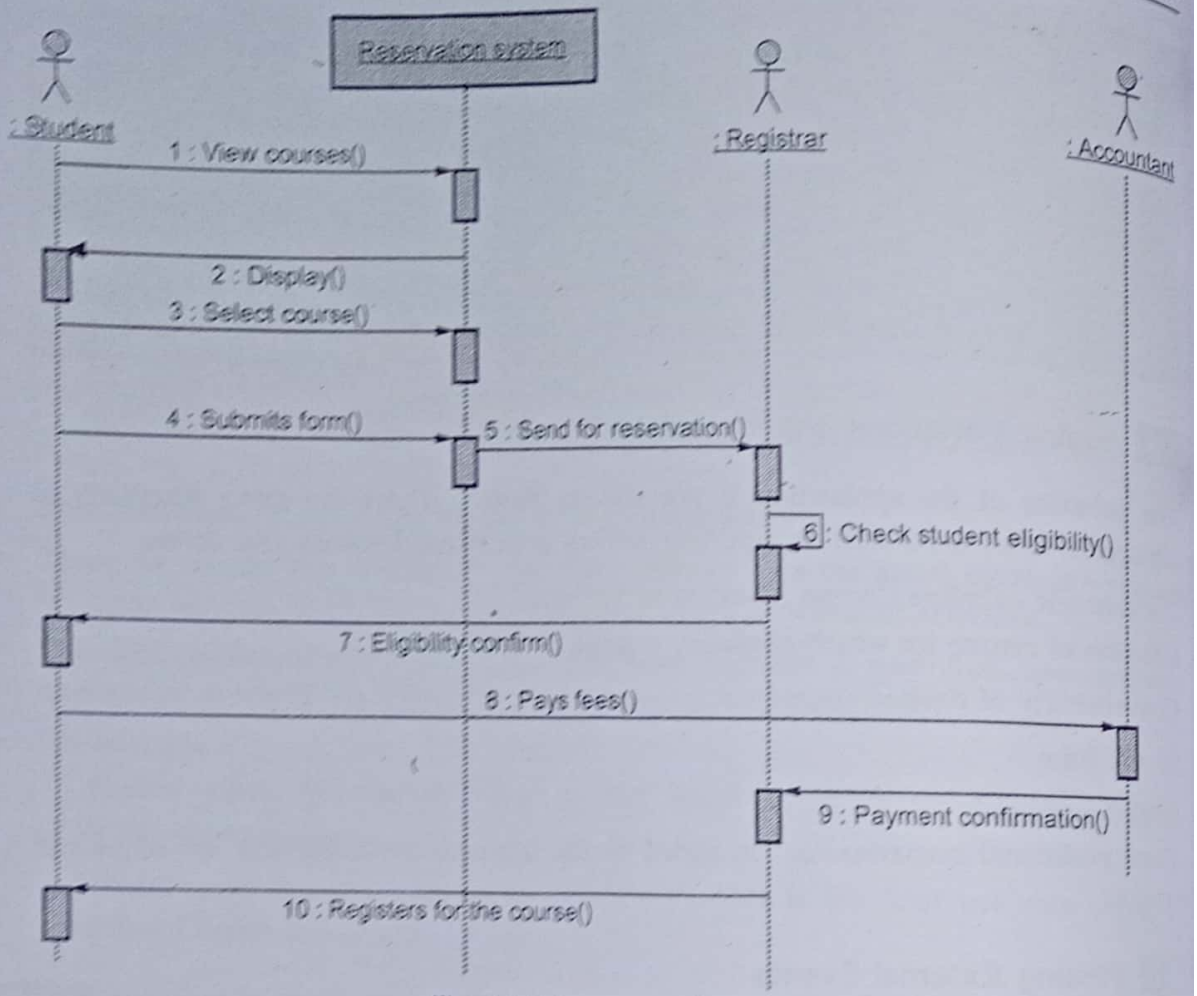


Fig. 4.1.2 Sequence diagram

The communication diagram represents the events in the scenario.

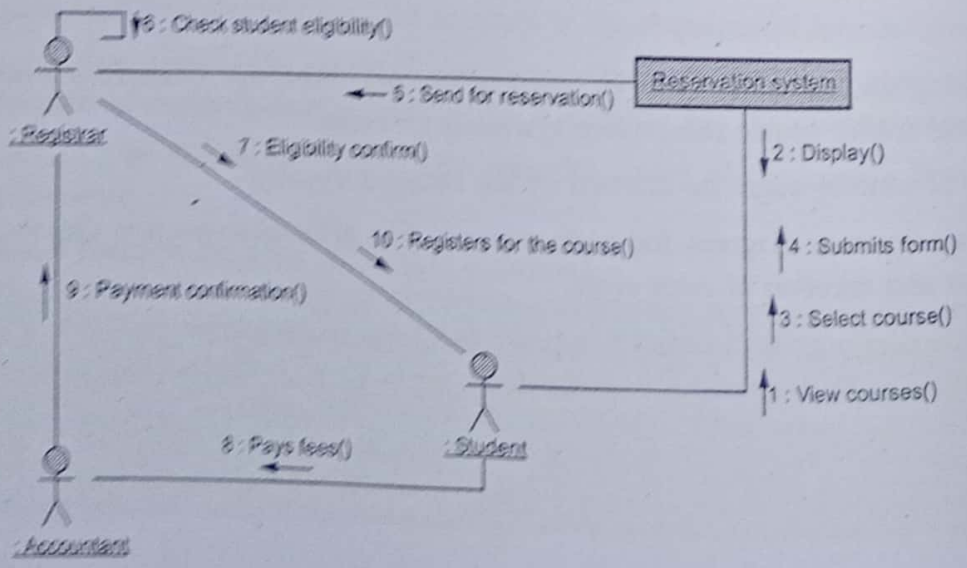


Fig. 4.1.3 Communication diagram

4.1.8 Creating Activity Diagrams

The sequence diagram represents the series of events that take place between the two objects. But it does not represent the alternative paths or decisions.

To show the decision points or alternative paths that can be possible within the communication are represented by the **activity diagrams**. Hence for more complex scenarios designing the activity diagram is a common practice. For example - The activity diagram for the Online Course Reservation System is as shown below.

(See Fig. 4.1.4 on next page)

4.1.9 Organizing Actors and Use Cases

The use case can be organized by using the relationship such as include, exclude and generalization within the use cases.

For example : The use case for Online Course Reservation system will be

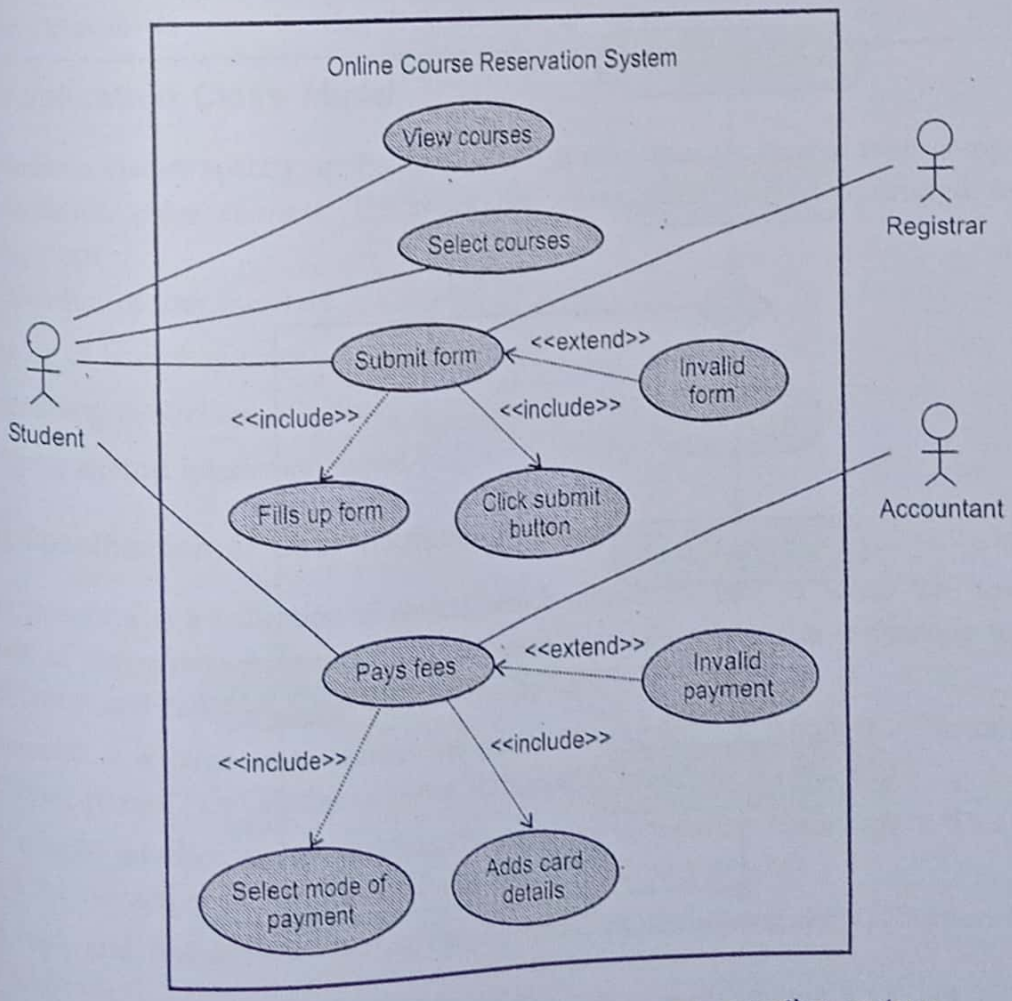


Fig. 4.1.5 Organizing use cases - online course reservation system

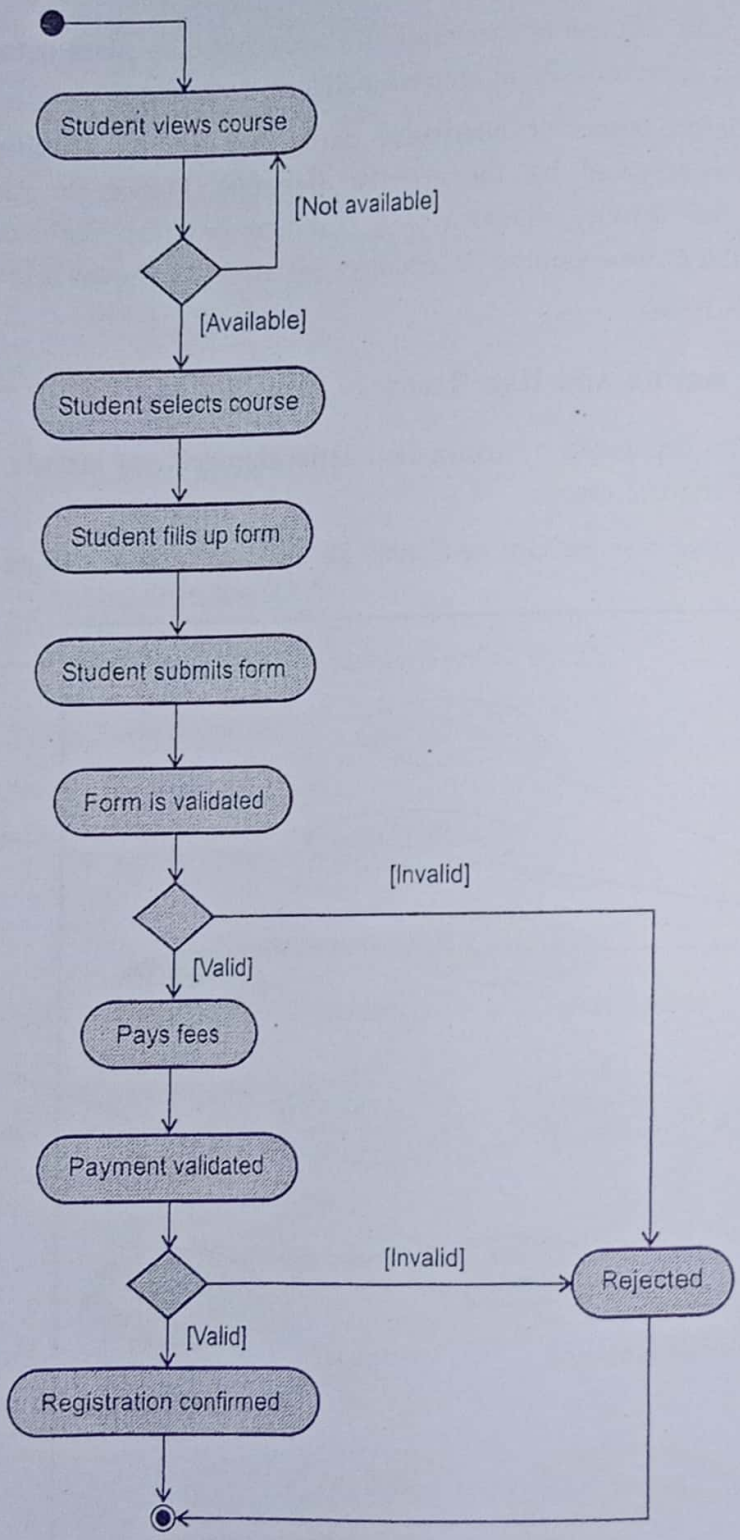


Fig. 4.1.4 Activity diagram for online course reservation system

4.1.10 Verifying against Domain Class Model

The actors, use cases and scenarios are all based on the classes and concepts from the domain model.

The application model must be tested against the domain class model. If both the models are consistent then it indicates that all the data and event parameters are considered consistently.

Review Questions

1. *Why interaction model is more important for application analysis ? Briefly explain the steps for preparing 'application interaction model'.*
2. *Explain the following steps while constructing an application interaction model.*
   a. Find use cases b. Find initial and final events.
3. *Explain the following steps in constructing an application interaction model with suitable example.*
   a) Determine the system boundary b) Find actors c) Find use cases

4.2 Application Class Model

Application classes specify applications. These applications are viewed from **computer implementation** point of view. The application class model can be constructed using following steps

- Specification of user interface
- Defining of boundary class
- Determining controllers
- Verifying against interaction model

4.2.1 Specification of User Interface

- **User interface** is a collection of objects which allows the user to access the domain objects of the system. Using the user interface user can submit the commands to the application and can access the applications options.
- **Command** is a large scale request for particular service. For example - "Search for particular phrase". Or "Update entry in database"
- The sample interface can be sketched out in order to visualize the interface. This also helps the developer in finding out if something is missing or what.
- The **look and feel** of user interface can be tested by decoupling the application from the interface.

- For example : For the **Online Course Reservation System** we can sketch one of the user interface as follows -

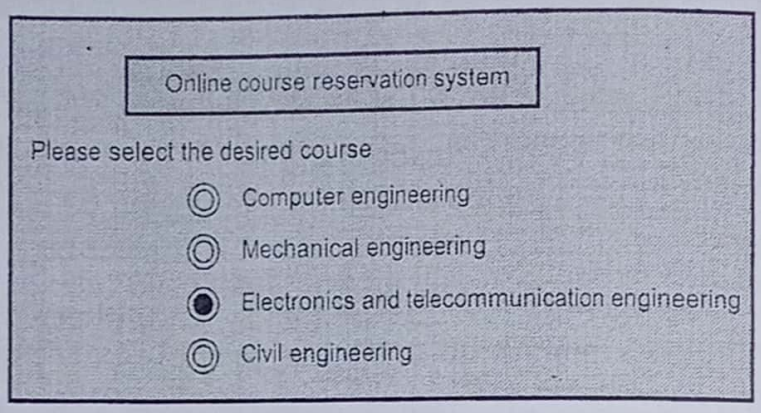


Fig. 4.2.1 GUI

4.2.2 Defining of Boundary Class

- The external sources can operate the system or the system can accept the information from external sources
- Boundary class is a class that helps in communication between a system and external source.
- The boundary class understands the format of one or more external sources and converts the information for transmission to the internal system. Similarly the information can also be sent from internal system to external sources.
- For example - The **CashCard** Boundary class can be defined in Online Course Reservation System.

4.2.3 Determining Controllers

- The **controller** is an active object that controls the internal operations of an application.
- It receives signal from outside world, then it invokes the required operations and sends the signals to the outside world.
- The controller represents a concrete behavior of an object.
- There can be one or more controllers within an application which are responsible to control the overall behavior of the application.
- For example : In **Online Course Reservation System** the checking eligibility of Student, Form validation and payment validation are the controllers.

4.2.4 Verifying against Interaction Model

When an application class model is built, think over the use case model. That means if some value is requested by the actor from the system , then that value must appear as an attribute of the user interface class.

For example : In Online Course Reservation System, User *selects the course(this is one use case)* - for that purpose **CourseName** must be the attribute of class **Course**.

Similarly if one object requests some value from another object within an application then it must be from the controller object.

For example : When the Form is submitted(Note that *Submit Form* is an use case) then the **Registrar** is another object that validates the form. Here the Registrar object acts as a controller which handles the form processing request.

4.3 Application State Model

- The application state model focuses on the application classes. It also helps in improving the domain state model.
- Using the application class model the classes can be used to identify the **states** and using the application interaction model the **events** can be identified.
- The event sequence for each class is organized using a state diagram.
- Design various state diagrams to match the common events.
- Finally check the state diagram against the class and interaction model to bring the consistency in the model.
- Following steps can be followed for designing the application state model -
  - Identify the application classes with states.
  - Find events
  - Design the state diagram
  - Check against other state diagram
  - Check against the class model
  - Check against the interaction model.

Let us discuss these steps in detail.

4.3.1 Identify the Application Classes with States

- In application class model the classes that are computer oriented and contain multiple operations from application point of view are considered for being the states.
- The **user interface classes** and **controller classes** are the good candidates for the state model.

- The boundary classes have the static nature and hence it is not desired to model the states using the boundary classes.
- For example - **Student** is a user interface class. Hence the various states that can be identified from the class Student are - *student viewing courses, student submitting form, student paying fees.*

4.3.2 Find Events

- From the interaction model, study the scenarios and find out the events. Do not overlook the common interactions and highlight common major events.
- **Contrast between domain and application processes for state models** - In **domain model**, the states are identified first and then the events because the domain model focuses on data. This data is grouped together to form the states and then the corresponding events are identified.
  In application model, the first events are identified and then the states are determined, because in application model the emphasis is on **behavior.**
- **For example :** In **Online Course Reservation System**, the events are clicking the course name for selection, clicking submit button to submit the form.

4.3.3 Design the State Diagram

For creating the state diagram -

- Choose the classes from the application class model, consider sequence diagram.
- Arrange events involving the class. The interval between two states is an event.
- Give name to each state. The name to the state must be meaningful.
- Every sequence diagram can be merged into a state diagram.
- If the sequence of events are repeated indefinitely in a sequence diagram, then it is called a **loop**. Then in a state diagram, for representing the loop there must be at least one state in a loop that have multiple transactions that are leaving the state.
- Merge the multiple sequence diagrams into the state diagram. Find the point in each sequence diagram where it diverges from previous one. Represent the new events as an alternate paths in state diagram.
- Make use of parameters and conditional transitions in order to simplify the state diagram.
- In a state diagram after considering the normal events add the variations and exception cases.
- If there are some complex interactions with independent inputs in the application then nested state diagrams can be used.

3.4 Check against Other State Diagram

Each state diagram is checked for its completeness and consistency.

Every event must have sender and receiver. The states must have predecessors and successors.

The states must have starting and termination points.

If objects are concurrent then the synchronizing states must be represented accurately.

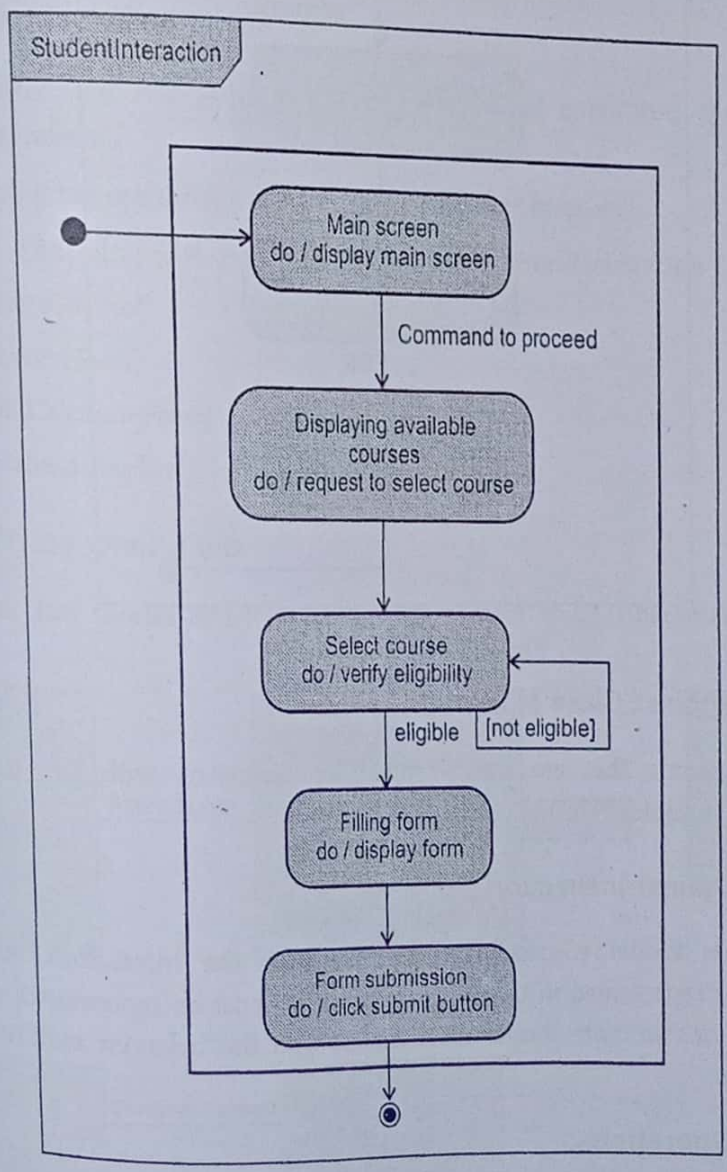


Fig. 4.3.1 (a) State model

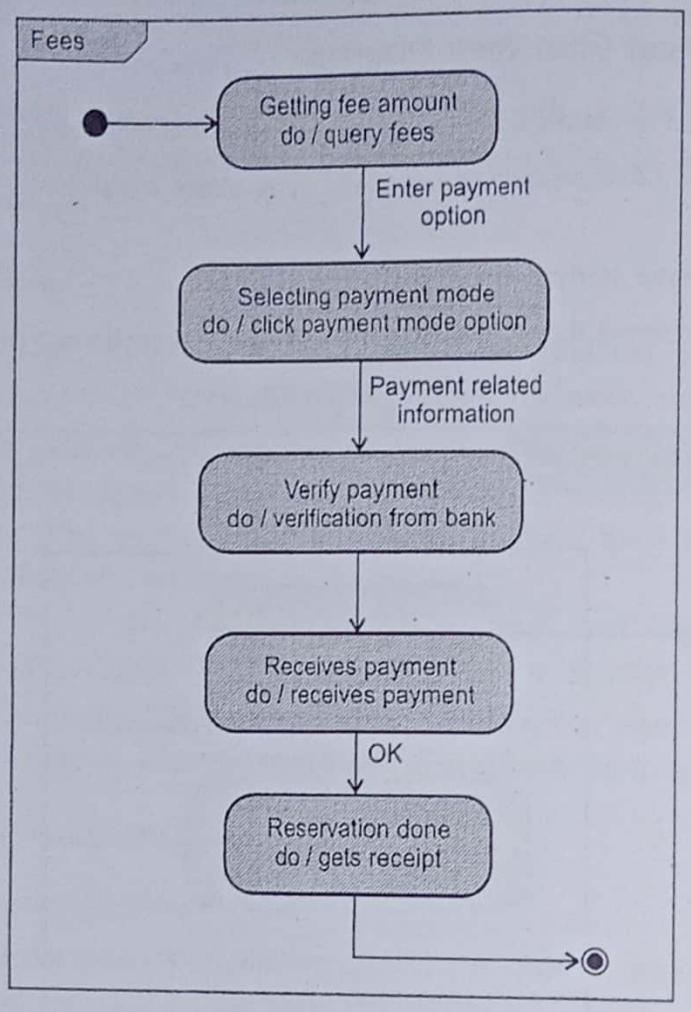


Fig. 4.3.1 (b) State model

4.3.5 Check against Class Model

The state diagrams that are created must be consistent with the **domain** and **application class models.**

4.3.6 Check against Interaction Model

When the **state model** is created, **check it against the interaction model.** Each behavior sequence represented in the sequence diagram must be represented in the state model. The behavior in state model must match with the behavior exhibited by the sequence diagram.

4.4 Adding Operations

Although adding operations to model is not of much importance, but still there are some useful operations and representing them in the model makes the model more

4.1 Operations from Class Model

The classes have attributes and associations and their values need to be read or written. Hence the significant set of operations can be obtained from class model.

4.2 Operations from Use Case

The use cases represent the scenarios. The complex functionalities of the system can be obtained from the use cases. These operations should correspond to the class model.

4.3 Shopping List Operations

Meyer suggested the real-world behavior by a list of operations that are called **shopping list operations.**

Due to shopping list operations the class definition gets broaden.

For example : Shopping List operations for **Online Course Reservation System are -**

- Student.SelectCourse()
- Student.ReserveSeat()
- Accountant.CalculateFees()
- CourseDatabase.display()

4.4 Simplifying Operations

Examine the class model and if variations are present in the operations then try to simplify them.

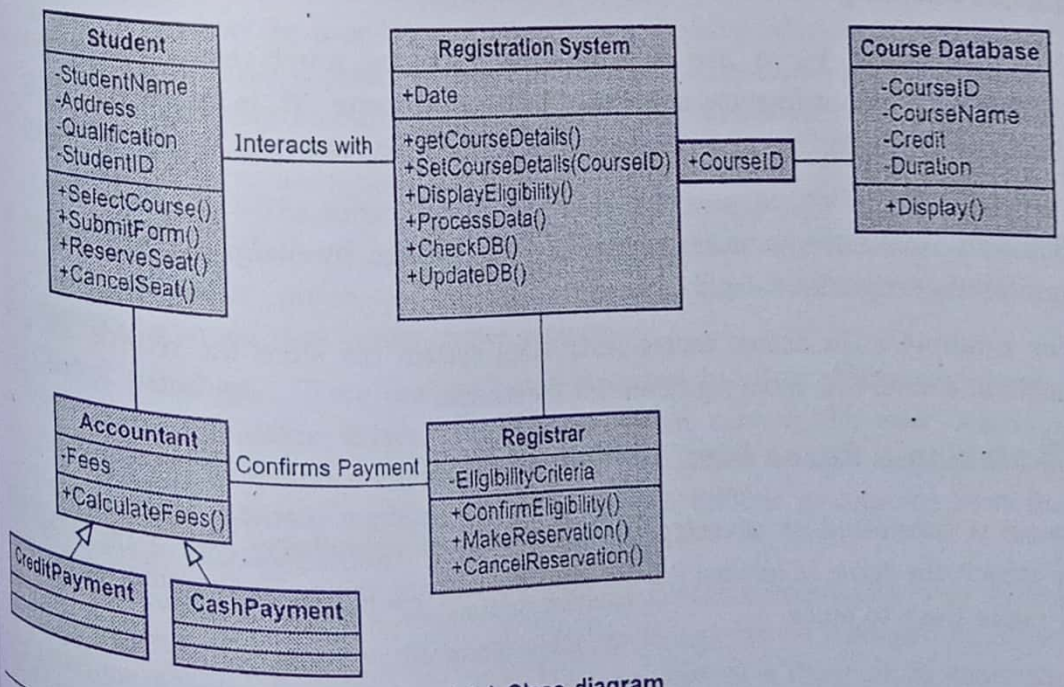


Fig. 4.4.1 Class diagram

Reduce number of distinct operations. Introduce the hierarchy of operations, if required. Make the operations more uniform and general.

Example : Fig. 4.4.1 indicates domain class model of **Online Course Reservation System** in which the operations are added.

(See Fig 4.4.1 on previous page)

Part II : System Design

4.5 Overview of System Design

System design is the first design stage in which the developers decide overall structure and style.

The **system architecture** decides the organization of system into subsystems. The system architecture also helps in taking some important decisions. These decisions are about -

1. Estimation of system performance
2. Making of reuse plan
3. Braking system into subsystems
4. Identifying concurrency
5. Allocation of subsystems
6. Management of data storage
7. Handling of global resources
8. Choosing software control strategy
9. Handling boundary condition
10. Common architectural style

Let us discuss these decisions in detail.

4.6 Estimating Performance

When planning for a new system, just make the rough estimation about its performance. This estimation need not be very accurate. It is made in order to understand the feasibility of the system. The performance estimate should be based on the common sense.

Similarly make the estimation for the data storage by judging the number of customers that might be using the system.

For example : The online course reservation system can store the records for 1000 students at a time who have registered for the course.

4.7 Making a Reuse Plan

- Reuse is considered as **advantage** to object oriented technology. There are two ways by which the reuse is applied - i) Reusing the existing things ii) Creating new things to make them to reuse.

Reusing the existing components is very easy than to create new things that will be used as reusable component in future.

Reusable things include models, libraries, framework and patterns.

4.7.1 Libraries

Library is a collection of classes these classes can be reused in developing various applications.

These classes must be carefully organized. They must be described in detail.

Following are some **good properties** of class libraries -

- **Coherent :** Using a well focused themes the class library must be organized.
- **Consistent :** The polymorphic operations must have the consistent names and signatures across classes.
- **Complete :** The class library must provide complete behavior for the desired theme.
- **Efficient :** A library must provide efficient implementation of algorithms.
- **Extensible :** It should be possible to define the subclasses for the classes in the library.
- **Generic :** A library must use parameterized class wherever is required.

Following are some **problems** that might arise when the classes from multiple libraries are integrated.

- **Argument validation :** The arguments can be validated in a group or individually. It is always better to check the arguments collectively when the command interface is used. But when the user interface is used then checking individual argument of the operation is always preferred. In a class library if some classes are checking the arguments in group and if some are checking them individually then it created complications.
- **Error handling :** The error handling is normally done by the methods. Some methods return the error code while handling the errors whereas some directly handle the error within the method. Thus use of different error handling mechanisms in the class library creates problem.
- **Control paradigm :** There are two types of control handling techniques - event driven and procedure driven. Using event driven controls the user interface invokes the application methods. With procedure driven control the application calls the user interface methods. Hence it becomes difficult to combine both the technique in one application.
- **Garbage collection :** There are various memory allocation techniques used in the class libraries. For instance the memory for the strings can be managed by either returning a pointer to a string or by returning a copy of a string or by returning

actual string. Similarly there can be various garbage collection techniques such as reference counting, mark and sweep or automatic garbage collector routines. Hence it becomes difficult for a application to choose proper class from the library of classes for handling the garbage collection.

- **Group operations** : The group operations are insufficient and incomplete. And usually the class libraries does not contain these group operations.
- **Name collisions** : There are chances that the class names, public attributes and operations can collide. It is a good practice to add distinguishing prefix names for the classes in the class library.

4.7.2 Frameworks

- **Framework** is a skeleton structure of a program. This framework is elaborated to build the complete application. In this elaboration the abstract classes are specialized as per the requirement of individual application.
- Along with the frameworks the class library can be used. Due to this the developer can pick up the required subclasses from the library instead of programming for them from scratch.
- Framework consists of classes, flow of control and shared invariants.
- The framework class libraries are application specific and not suitable for general use.

4.7.3 Patterns

- **Pattern** is a sample solution to a general problem.
- For various stages of software development life cycle there are patterns available. That means there are patterns for analysis, architecture, design and implementation.
- A pattern comes with guidelines.
- Following are some **benefits** of patterns -
  - Pattern is carefully considered and can be applied using past problems.
  - Pattern is more suitable to apply because it is correct and robust.
  - The language of pattern is familiar to many developers and the documentation about the pattern is easily available.
- **Difference between pattern and framework** - Pattern contains small number of classes and relationships whereas framework contains entire subsystem or application. It has much broader scope than pattern.

Questions

1. *Why reuse is considered as an advantage of object oriented technology ? One of the reusable thing, a library. Describe qualities of "good" class libraries.*
2. *What are reusable things ? Explain various difficulties that may arise in using the libraries while making a reuse plan?*
3. *How frameworks are used in making a reuse plan ?*
4. *Explain the notion of pattern. What are the benefits of using pattern in making a reuse plan.*
5. *Differentiate between pattern and framework*
6. *Explain following concept with reference to system design-Reusable components and their use.*

Breaking a System into Subsystems

A **subsystem** is a group of classes, associations, operations, events and constraints that are interrelated, well defined. Many subsystems have the interface with other subsystems.

The subsystem is known by its **services. A service** is nothing but the collection of functionalities that it provides. For example memory management system is a subsystem in operating system that provides the service of memory management.

Each subsystem has a well defined interface to other subsystem. This interface denotes all kinds of interactions and the information flow across the subsystem boundaries. But it hides the internal implementation of subsystem.

The design of each subsystem is independent of other subsystem.

There are two types of relationships between the two subsystems - 1) Client Server and 2) Peer to Peer.

In **client server relationship** the client invokes the server, the server then provide the service to this client. In this relationship, the client knows the server's interface but it is not necessary that the server should know the client's interface.

In **peer to peer relationship** each subsystem calls on another subsystem. In this communication it is not necessary that the calling subsystem should get the response immediately. In this communication each subsystem should know the interface of another subsystem. Hence this interaction is more complicated.

The **decomposition of the system** into subsystems can be into **layers and partitions.** Let us discuss each of this type of decompositions.

4.8.1 Layers

- A layered subsystem is a collection of tiers that are built one below the other. The lower layer provides the implementation basis for the layer just above it.
- The objects in each layer are independent of the other but these objects can communicate with each other.
- Each layer has a knowledge of the layer below it but it does not have the knowledge of the layer above it.
- The lower layer provides the service to the above layers. Hence client-server relationship exists between the lower and upper layers.
- Each layer has its own set of classes and operations. Each layer is implemented using the classes and operations of the lower layers.
- There are two types of layer architectures - **closed architecture** and **open architecture.**
- In **closed architecture,** each layer is built only using the immediate lower layer. The advantage of this architecture is that the dependencies between the layers are reduced and any changes in the layer can be made very easily.
- In **open architecture,** a layer can be built using any lower layer to any depth. The **advantage** of this architecture is that there is no need to redefine the operations at each level. But the **disadvantage** is that information does not get hidden. And changes in one subsystem can affect the higher subsystems.

4.8.2 Partitions

- Partitioning divide the system **vertically.**
- The subsystems created by partitioning are loosely coupled.
- Each subsystem provides some kind of service.
- Each subsystem provide a knowledge about other subsystem but this knowledge is not much deep. Hence avoid the design dependencies among the subsystems.

Difference between layers and partitions

- In layers the level of abstraction varies between the layers. Partition divides the system into the subsystems that have similar level of abstraction.
- The layers are dependant upon each other specially in client server relationship - either by open or closed architecture. However, partitions are peers and are independent or simply mutually dependant.

4.8.3 Combining Layers and Partitions

- A system can be broken into subsystems by combining the layers and partitions.
- A layer can be partitioned or a partition can be layered.
- Following Fig. 4.8.1 represents the block diagram of a typical application in which the layers and partitions are combined together.

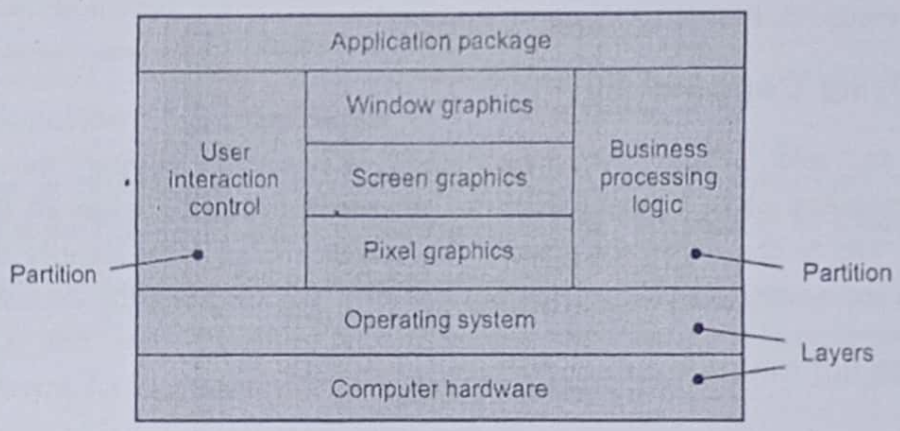


Fig. 4.8.1 Block diagram of typical application (combining layers and partition)

For example - The **online course reservation system** has two major subsystems as shown in Fig. 4.8.2

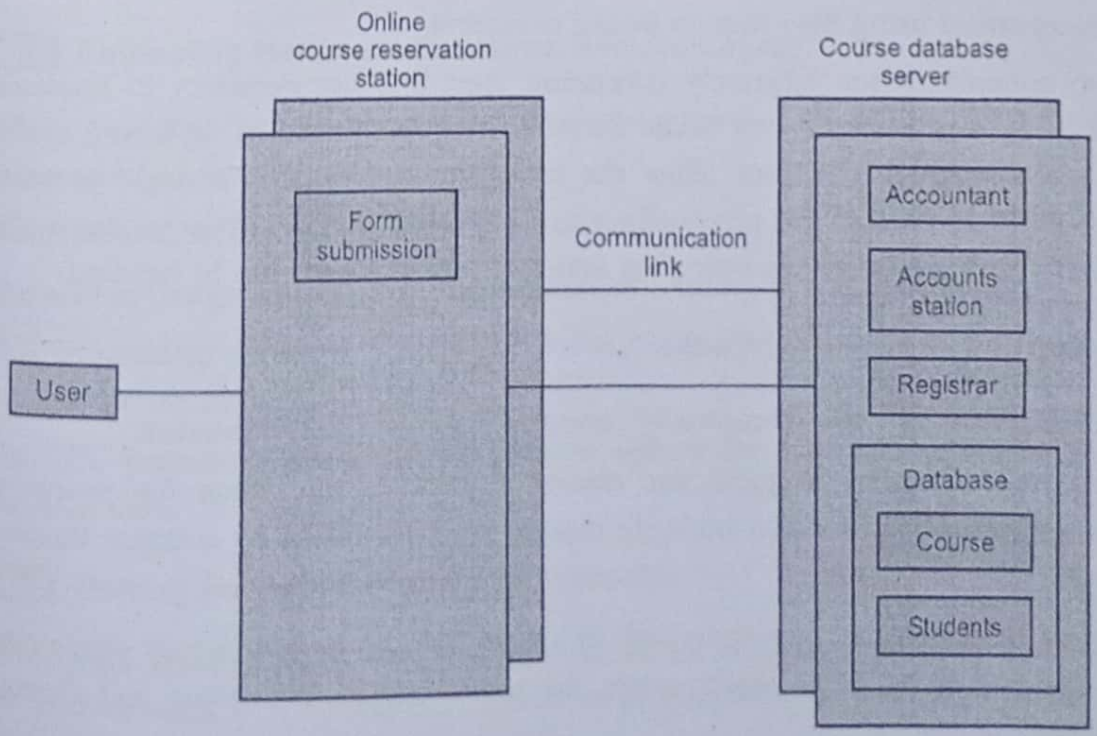


Fig. 4.8.2 Design of subsystems for online course reservation system

Review Questions

1. *What is layered system ? Layered architectures come in which two forms.*
2. *Explain following concepts with reference to system design :*
   i) Reusable components and their use
   ii) Methods of breaking system into subsystems.
3. *Explain following terms :*
   1. Frameworks 2. Patterns 3. Layers 4. Partitions.

4.9 Identifying Concurrency

Although in real world the objects are concurrent, during the software implementation many objects can be executed on single processor when all the objects are not active. Hence the important **goal of the system design** is to identify the objects that are active concurrently and identifying the objects that are mutually exclusive.

4.9.1 Identifying Inherent Concurrency

- Using the state model the concurrency between the objects can be identified.
- If the two objects receive the events at the same time without interacting then they are called inherently concurrent objects.
- If the events that the objects are receiving are not synchronized then those objects can not be operated using the common thread of control.
- If two subsystems are inherently concurrent then it is not necessary to implement them using separate hardware units. Because using the interrupts, operating systems and multitasking mechanisms allow the logical concurrency on a single processor. Using separate sensors the physically concurrent input can be given to the system. Similarly, using multitasking operating system, multiple inputs can be handled.

4.9.2 Defining Concurrent Tasks

- Although the objects are conceptually concurrent, they are interdependent.
- In the state diagram of particular objects, we can observe that the events are exchanged among them. The multiple objects can be handled by a single thread of control.
- **A thread of control** is a execution path in a state diagram on which only single object at a time is active. This object sends the events to another object and continue executing.

- On each thread of control only a single object at a time is active.
- The thread of control splits, if object sends an event and continue executing.

Review Questions

1. *Explain the concept of identifying concurrency in system design.*
2. *Explain the following terms :*
   1) Inherent concurrency 2) Thread of control.

4.10 Allocation of Subsystems

A system is broken down into the subsystems and each concurrent subsystem is allocated to some hardware unit or functional unit. This allocation is as follows -

1. Estimate the hardware resource requirements.
2. Choose the best possible hardware and software for implementing these subsystems (i.e. hardware-software tradeoffs).
3. Allocate software subsystems to processors.
4. Determine the physical connectivity between the units and implement the subsystem.

Let us discuss these steps in detail.

4.10.1 Estimating Hardware Resource Requirements

- When a higher performance is required then instead of single CPU, multiple processors or hardware units are used.
- Many parallel hardware units consume data and process them concurrently.
- The system designer must check the required processing power of CPU.
- This **processing power** = Number of transactions per second × Time required to process the transaction.
- The CPU processing power can be checked against the steady load as well as with the peak load.

4.10.2 Making Hardware - Software Tradeoffs

- During the system design, the decision about which subsystem must be implemented on hardware and which subsystem should be implemented on software - is taken.

- Following are the reasons why the subsystem should be implemented on hardware -
  1. **Cost :** The required functionality can be provided by a hardware unit to the subsystem. It is always cost effective to implement the subsystem on a hardware chip than to create a software unit. These hardware units can be accompanied with the sensors and actuators.
  2. **Performance :** The high performance systems are required by many subsystems. The most efficient hardware are available. The chips having Fast Fourier Transform are also available.
- There are **two common difficulties** in using the hardware system is that - i) Some external hardware gets imposed to the existing hardware system ii) Software imposes constraints on the subsystem.
- Using object oriented technology the external hardware being imposed is treated as objects.
- The compatibility, cost and performance issues can be considered.
- The system designer must think about the flexibility for future changes both for system design and for product enhancement.
- **Example -** As there is no special need for performance in online course registration system, the general purpose computers will also do for implementing the system.

4.10.3 Allocating Tasks to Processors

Allocating tasks to various processors is an important thing in system design. Following are the reasons for assigning tasks to the processors -

- **Communication limits :** Many times the data flow rate exceeds the available communication limits. For instance : In video gaming applications the high performance graphics devices require tightly coupled controllers because the large amount of gets generated in such applications.
- **Computation limits :** If a single processor is used then it has to handle large number of computations. Hence by attaching multiple processors the computational load can be divided. The communication cost gets reduced if highly interactive subsystems are attached to the main processor.
- **Logistics :** Certain tasks can be done by placing the processors at specific locations. Sometimes we need separate processors that are assigned to perform some independent activity. For instance - the database server must be placed at some location and is meant for performing particular activity.

4.10.4 Determining Physical Connectivity

In system design we must first determine the type of physical units required, then the number of such units required and finally how these units are connected together. Following are the **important issues in determining the physical connectivity** -

- **Connection topology :** Various physical units are connected by certain topologies. In class diagram we connect different classes by association links. These links normally correspond to the physical connections in system design. The client server relationship is popularly used in connecting the physical connections. The cost of important relations must be low.
- **Repeated units :** If there are several copies of repeated functional units then use the topology of repeated units. This will **enhance the overall performance**. The class model is not useful guide to decide the topology of repeated units because during the analysis process the design optimization is not focused and therefore we draw the class diagram in which the reuse of relationships is not much thought of. The topology of repeated units can be tree, star, matrix or some linear sequence.
- **Communications :** While determining the physical connectivity it is important to decide the communication protocols and type of connection channels. The interaction between various physical units can be synchronous, asynchronous or blocking. Depending on the type of interfacing the general interaction mechanism and protocol must be decided. The bandwidth and latency of the communication channels must also be estimated.

4.11 Management of Data Storage

- Data can be stored in data structures, files and databases. These different types of data storage offer trade-offs among cost, access time, capacity and reliability. For example - student admission record can be stored in databases.
- Files are simple and cheap to use as data storage. However, file operations are the low level operations.
- File implementation may vary from for different computer systems.
- Following are **the characteristics of data which is suitable for files** -
  - The data having high volume and low information density are suitable for file storage.
  - Data that needs a sequential access.
  - Data can be read fully into memory.
  - Data having simple structure and modest quantity are suitable for file storage.
- The Database Management System(DBMS) is useful for managing databases. Various type of databases are - relational databases and object oriented databases.

- The main **advantage** of using database is that due to databases the applications can be ported easily on different operating systems or on different hardware. But the **disadvantage** of databases is that integrating database with programming language is really complex.
- Following are **the characteristics of data which is suitable for database** -
  - Data that needs to be updated frequently and can be used by multiple users.
  - Data that needs to updated in some co-ordination via different transactions.
  - Data that is accessible by various application programs.
  - For handling large quantity of data with efficiency.
  - Data that needs security and proper authentication.
  - Data that is persistent and is highly valuable.
- Various applications that require Object Oriented Database Management System (OO-DBMS) are engineering applications, knowledge bases, electronic devices with embedded software, and multimedia applications. However, the relational database management is used more commonly.
- The relational database management provides good implementation of object oriented model.
- For example - In **Online Course Reservation System** the databases for registered-students, courses, accounts is created in relational databases. **ATM(Automatic Teller Machine)** makes use of relational database.

Review Questions

1. *What kind of data is suitable for files ? What kind of data is suitable for databases ? Does ATM (Automatic Teller Machine) use relational or object oriented database ?*
2. *Explain following concept with reference to system design management of data storage.*
3. *Discuss data storage management in system design.*

4.12 Handling Global Resources

- Various types of global resources are -
  - **Logical names :** The file names, class names, object identifiers represent the logical names.
  - **Shared data :** Databases are shared globally by many applications.
  - **Physical units :** The processors, disks drives, tapes or communication satellites.
  - **Space :** It includes disk space, workstation screen.

- The physical object is controlled by itself using some protocol.
- If the resource is a logical entity then there are chances are conflicts in access. For example a database can be updated from two different applications simultaneously.
- For avoiding these conflicts **guarding object** is used. The task of guardian object is to control access to several resources. Hence all access must pass through the guardian object. With the help of guardian object each object gets allocation to global shared resource.
- The shared global resource can be partitioned logically and each logical partition can be controlled by a separate guarding object. This provides an independent control over the access to shared global resource. For example - suppose there are multiple processors that allocate various object IDs. If each processors is assigned with some range of object ID, then each processor will generate the object ID within that range and there will not be any conflict.
- In real time applications, accessing global resources via guarding object is costly. In such applications clients need direct access to global resources. Hence the concept of lock is used.
- A lock is a logical object associated with some defined subset of resources. The lock holder will get the access to the resources directly. Here the job of guardian object is to allocate the locks. But after getting the lock user can interact to the resources directly. But this approach is not always preferred in sharing the global resources because there is a danger of having unauthorized access to the resources.
- For example - FormID, CourseID and StudentID are the global resources. While paying the fees the bank code and the account number must be unique.

4.13 Choosing a Software Control Strategy

- The events between the objects cause the interactions. The control can be implemented in software. This control is used to control the interactions between the objects.
- There are two types of control flows in software system -
  1. External control 2. Internal control.
- The **external control** manages the externally visible events among the objects. The external control is implemented in three ways -
  1. Procedure driven sequential 2. Event driven sequential 3. Concurrent.
- The **internal control** represents the flow of control within the procedure. Hence this control is neither sequential not concurrent.

Let us discuss above given various types of software controls in detail -

4.13.1 Procedure Driven Control

- In this type of control flow, the control lies within the program code (procedure) itself. The procedure requests for external input and waits. When the input is received by the procedure then the control gets activated within the procedure.
- The state of the system is defined with the help of stack of procedure calls, local variables and program counter.
- **Advantage :** The procedure driven control is easy to implement with conventional languages.
- **Disadvantage :**
  1) The concurrency is required when objects get mapped with sequential flow. The events need to be converted into the operations between the objects.
  2) Flexible user interfaces and control systems are difficult to build.
- **Applicable to :** This type of flow of control is suitable only for the **state models** that show **regular alteration** of input and output events.

4.13.2 Event Driven Control

- In event driven systems, the control lies within the dispatcher or monitor provided by a language, subsystem or operating system. The developers attach the applications to the events. The dispatcher then calls the procedures when particular event occurs. Finally the procedures return the control back to the dispatcher.
- In this method, the program counter and the stack can not preserve the state. For preserving the states the procedures must make use of the global variables.
- **Advantage :**
  1) The event driven control permits more flexible control than procedure driven systems.
  2) Event driven systems are modular and can handle error conditions better than the procedure driven control.
- **Disadvantage :** Event driven control is difficult to implement using the standard language. Hence it requires extra efforts to implement this type of control.

4.13.3 Concurrent Control

- When the controls reside within several independent objects each performing a separate task then the **concurrent control** gets executed.

- The events are implemented as **one way messages** between the objects.
- In concurrent systems, one task can wait for input but at the same time another task can continue its execution.
- If there are multiple CPUs then the tasks execute concurrently on several processors.

4.13.4 Internal Control

- Internal control gets executed when the many lower level operations get executed on same or another object.
- **Difference between internal and external control :** The internal and external control are almost same but there lies some differences between them.
  - In external interactions the objects wait for the occurrence of events. The objects are independent and the response of one object is not dependant upon the other. Whereas in internal interaction, the operations execute as a part of algorithmic implementation. Hence the response of these operations is predictable.
  - Internal interactions occur when procedures get executed. In this interaction the procedure call issues a request and waits for response. Whereas in external interactions, there are algorithms that invoke parallel processing with some computations. Each computation can be viewed as single thread of execution. Thus there can be multiple threads of execution for a single operation.

4.13.5 Other Paradigms

The procedural programming is the most commonly used paradigm. But there are several other forms of non procedural programming such as rule based systems, logic programming systems, knowledge base systems and so on. These non procedural programming systems have lot of future enhancement and now have the limited area of usage.

Review Question

1. *Explain how to choose software control strategy.*

4.14 Handling Boundary Conditions

For steady state behavior of the system the boundary conditions must be checked. Following are some issues that are used in handling the boundary conditions -

- **Initialization :** The system begins execution from the initial state to some steady state. In the initial state, the system initializes global variables, parameters, constant data and classes. During initialization only subset of functionality is available. The

system containing concurrent tasks is difficult to initialize because there are many independent objects that can be initialized.

- **Termination :** The termination is a state in which the task to be accomplished gets completed. In this state the external resources are released. In concurrent system it is necessary to note that particular task is terminated.
- **Failure :** It is an unexpected termination of the system. It arises due to internal breakdown, exhaustion of system resource or due to user errors. Failures occur due to the bugs in the system. The good design of the system suggests graceful exit on occurrence of fatal conditions. Due to which the remaining environment remains undisturbed.

4.15 Setting the Trade-off Priorities

- Trade-off means making the **compromise** in one goal in order to achieve some other more important goal.
- During the design of the system, it is the responsibility of the **system designer** to set the priorities of the system. For example - if we want the system to be faster, we make use of more memory. Here faster speed is achieved at the cost of more memory.
- The design trade-offs not only involve the software itself but the process of developing it.
- Sometimes the complete **software functionality need to be compromised** to make the system in use.
- While deciding the trade-offs the system designer has to establish the priorities of the system. For example - If the system is for implementing the mathematical model then accuracy is of at the most importance and not the speed.
- The **system design trade-off affects** the entire functionality of the system. The success of failure of the final product depends upon how well the decision for the trade-offs is taken.
- It is essential to decide the priorities of the system otherwise there will be wastage of system resources.
- Deciding the system priorities is a **vague** and one can not expect the accurate list of system priorities. However, with the help of proper judgment and by understanding the system requirements the trade-off can be made for the system.

4.16 Common Architectural Styles

System architecture is used for organization of the system components. Various commonly used architectural styles are -

1. Batch transformation
2. Continuous transformation
3. Interactive interface
4. Dynamic simulation
5. Real-Time system
6. Transaction manager

Let us discuss them in detail.

4.16.1 Batch Transformation

- **Batch transformation** performs the sequential computations. In this architectural style, the application gets some input and the answer is computed. There is no interaction with outside world. For example - compilers, payroll systems make use of batch transformation architecture.
- Creating state model for the applications that use batch transformations is of no use. But the class model for such applications is very important. The interaction model can also be created for such architectural style by documenting the computations and using the class model.
- Following Fig. 4.16.1 represents the stages in batch transformation architecture for the compiler -

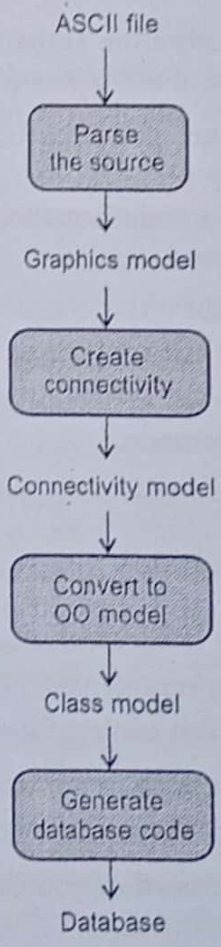


Fig. 4.16.1 Stages in batch transformation

- The **goal of** batch transformation is to define clean sequence of steps.
- Following are the steps used to perform the batch transformations -
  - Break the transformation into the stages. Each stage is performing some task of the transformation.
  - Create the class model for the input, output or between the pair of successive stages. Each stage must know the class model of its neighboring stage.
  - Detail out each stage until each operation gets simplified.
  - Restructure the stages to obtain the optimization.

4.16.2 Continuous Transformation

- In this architectural style the output varies when the input is getting changed.
- The output is continuously updated.
- Due to the limitation of time, computing entire set of output is not possible all the time, instead of that the output is computed incrementally.
- The applications that make use of continuous transformation architectural style are incremental compilers, windowing systems, system monitoring system or signal processing system.
- The class and interaction models are created for modeling the important classes and interactions.
- Using the **pipeline functions** the continuous transformation can be implemented. In this method the pipeline propagates the effect of input change.
- Following are the steps used to perform the continuous transformation by designing the pipeline -
  - Break the transformation into the stages. Each stage is performing some task of the transformation.
  - Create the class model for the input, output or between the pair of successive stages. Each stage must know the class model of its neighboring stage.
  - The incremental changes to each stage can be identified by differentiating each stage operation.
  - Add additional intermediate objects to obtain the optimization.

4.16.3 Interactive Interface

- In this type of architectural style the interactions between the system and the external devices are represented. The external devices are independent of the system and system can not control them.
- This type of architectural style includes only some part of the entire application. For example - query interface, control panel for simulation.
- The **communication protocol** used between the system and external devices is the most important factor of interactive interface style. Along with the communication protocol, other important issues are presentation of output, error handling, syntax of the interactions and flow of control within the system are important issues in interactive interface.
- The state models are very important modeling in interactive interface. The class model represents the interaction elements such as input, output or presentation formats. The interaction model represents how the state diagram interacts.

- Following are the steps used in designing the interactive interfaces -
  - Isolate interfaces classes from application classes.
  - Use predefined or library classes to interact with the external devices. For example - menu or forms or buttons are used to interact with external agents.
  - The program structure is created with the help of state model. The **interactive interfaces** are implemented using concurrent control or event driven control.
  - Separate out the physical and logical events. Many physical events correspond to one logical event. For example submit form is a logical event. To accomplish this event many physical events occur and those are - typing the command to invoke form, typing the input through keyboard to fill up form, and then clicking the submit button.
- Specify the application functions that are invoked by the interface. Gather the required information to implement the application functionality.

4.16.4 Dynamic Simulation

- The dynamic simulation models mimic the real-world objects. For example - Creating economic models, video games.
- The dynamic simulation can be modeled easily using the object oriented approach. Because the objects and their interactions can be identified directly from the application.
- There are **two ways to implement control** in dynamic simulation - 1) Explicit external controller controlling the objects of the application and 2) Objects that exchange information among themselves.
- The class diagram is very important and complex. The state model and the interaction diagrams are equally important for dynaic simulation.
- Following are the steps used in designing the dynamic simulation -
  - Identify the real-world **objects** from the class model. Identify the **attributes** of these objects.
  - Identify the discrete events. The discrete events correspond to the discrete interactions with objects. These discrete events are implemented as **operations.**
  - Identify the continuous dependencies between the real world objects. The attributes of these objects must be updated regularly in order to simulate the dependencies.
  - Simulation is driven by a **timing loop**. The discrete events between the objects are important part of these timing loops.
- 'Providing the required performance' is a crucial factor in designing of the dynamic simulation. Hence in practice, the system designer must estimate the computation cost and adequate resources.

4.16.5 Real-time System

- Real time system is an **interactive system with strict time constraint** on actions.
- There are two types of real time systems - Hard real time system and soft real time system.
- In **hard real time system**, the software must be highly reliable and the response must be within the specified time constraint. It includes the critical applications.
- In **soft real time system** also the software must be highly reliable but occasion violation of timing constraint is allowed.
- Examples of real time system are the applications based on data acquisition, communication, device control and process control.
- Real time system design is very complex and it includes various issues such as interrupt handling, prioritization of tasks and synchronizing the multiple processors.

4.16.6 Transaction Manager

- Transaction manager is a kind of system in which the main objective is to store and retrieve data.
- Transaction manager deals with the systems in which multiple users read or write data simultaneously.
- The transaction manager offers the security from unauthorized access to data or loss of data.
- The transaction manager often lie on the top of the database management systems. For example railway reservation systems, online banking systems, inventory control systems.
- The class model is the most important modeling for transaction manager. Sometimes the state model is used for determining the evolution of objects. Similarly the interaction modeling is also important for transaction manager.
- Following are the steps used in designing the transaction manager -
  - Map class model to database structure.
  - Determine the **concurrency** that means - identify the resources that are not shared by any other object.
  - If required, introduce new classes.
  - Determine the **transaction** that means - the way of accessing the resources.
  - Design the concurrency control for the transactions. In database management systems this type of control is often used, If any transaction is failed earlier then it must be retried until the successful transaction takes place.

Review Questions

1. *Which prototypical architectural styles are common in existing systems ?*
2. *Give the list of common architectural styles. Explain batch transformation in detail.*
3. *Why software architecture is so important in system design ? Enlist and briefly explain different architectural styles.*

4.17 Architecture of the ATM System

The ATM system architecture can be modeled as a fully distributed system architecture.

In distributed systems the components are globally distributed on multiple servers. The deployment diagram is used to model the current topology of the system and distribution of the components across the system.

Following figure shows the distributed system architecture of ATM systems.

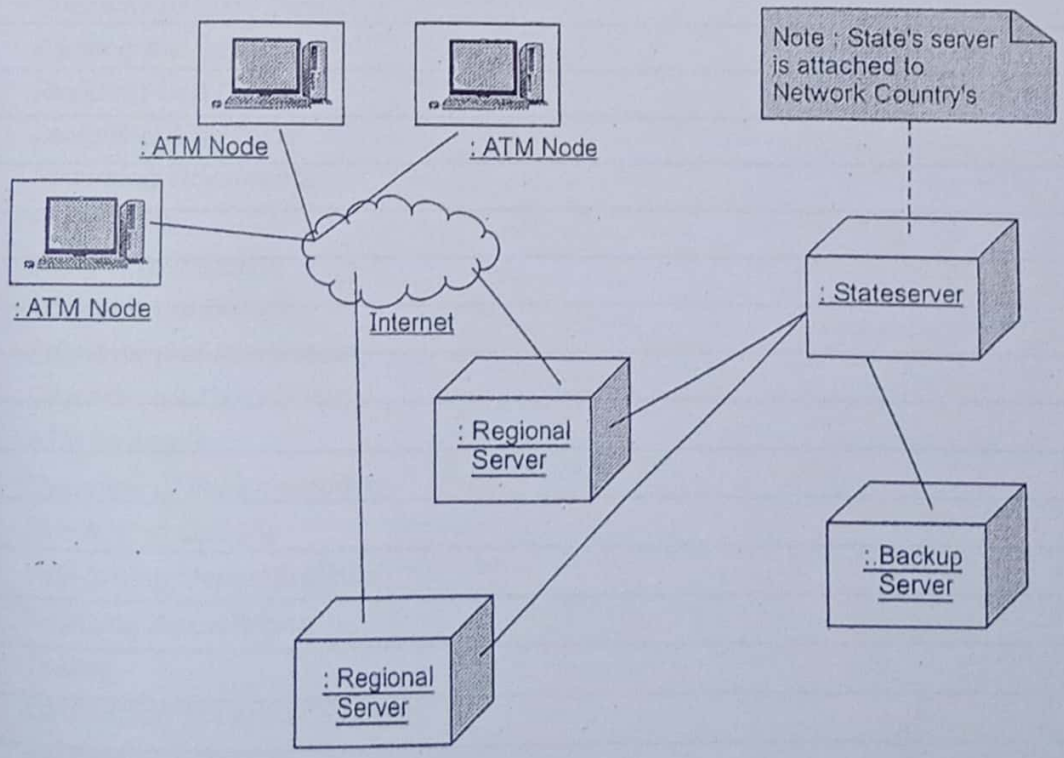


Fig. 4.17.1 Modeling fully distributed system

❑❑❑

Unit V

5 Class Design, Implementation Modeling and Legacy Systems

Syllabus

Class Design : Overview of class design; Bridging the gap; Realizing use cases; Designing algorithms; Recursing downwards, Refactoring; Design optimization; Reification of behavior; Adjustment of inheritance; Organizing a class design; ATM example. Implementation Modeling: Overview of implementation; Fine-tuning classes; Fine-tuning generalizations; Realizing associations; Testing. Legacy Systems : Reverse engineering; Building the class models; Building the interaction model; Building the state model; Reverse engineering tips; Wrapping; Maintenance.

Contents

Part I : Class Design

5.1 Overview of Class Design

- During the analysis phase the class design is created. The **purpose of class design** is to complete the definitions of classes and associations and choose the algorithms for operations.
- The analysis model describes the important **information about the system** and **high level operations** that the system must perform.
- The class design created in analysis model is directly mapped into the design. Later on the new additions are made in the class design.
- During the design the **main task for class design is choosing appropriate algorithms for the operations** of the classes and breaking the complex operations into simple ones.
- The **new classes** can be added to store the intermediate results.
- The Object oriented design is an iterative process. At each level the class design of one level of abstraction is created. At each level new operations, attributes and classes are added. Sometimes even new relationships are also added or existing relations are revised to simplify the class design.
- Following are the steps carried out for the class design -
  - Bridge the gap between high-level requirements to low level services.
  - Analyze use cases.
  - For each operation design an algorithm.
  - Recurse downwards to design operations.
  - Refactor the model into simpler class design.
  - Optimize access paths to data.
  - Reify behavior that must be manipulated.
  - Adjust class structure to increase inheritance.
  - Arrange classes and associations appropriately.

Review Question

1. *What is the purpose of class design ? Enlist the steps that are carried out during the class design.*

5.2 Bridging the Gap

- Every system has its own set of **features**. Similarly there are **several resources available** in the environment in which the system design is embedded. The system can offer various services when these features are properly utilized by the resources available. But there exists a gap between these **desired features** and **available resources**. Hence the designer has to build a bridge across the gap.
- Various features of the system can be obtained from use cases, system operations and services and application commands. Whereas, the resources are operating system infrastructure, class libraries, or previous application. This scenario can be illustrated by following Fig. 5.2.1.

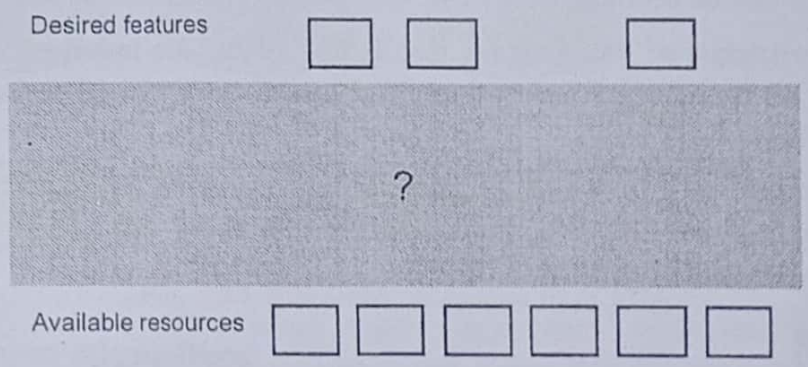


Fig. 5.2.1

- Each **feature** can be constructed from the **available resources**. For example - If the record of each student is required by the registrar then the database file containing the student record can be created. Thus making use of class libraries or databases or some applications so that the features can be made available as a service is called **bridging the gap**.
- But bridging the gap is not always so simple. Directly by making use of available resources one can not bridge the gap. In fact, the developer has to invent some **intermediate elements** so that the features can be used by the available resources to provide desired service. For instance- If you want to create a web site for online shopping. Then by using the programming language or spreadsheets you can not create such site. What you need is creating multiple elements that can be the part of your web sites. Furthermore, these elements need to be organized in multiple levels. The intermediate elements can be operations, classes or some other UML construct. Following Fig. 5.2.2 illustrates this idea. (See Fig. 5.2.2 on next page)
- Many times finding the intermediate element is not so simple. Many high level operations need to be decomposed. Multiple similar operations can be bound together in a single operation in order to bring the clarity in design. In short you have to

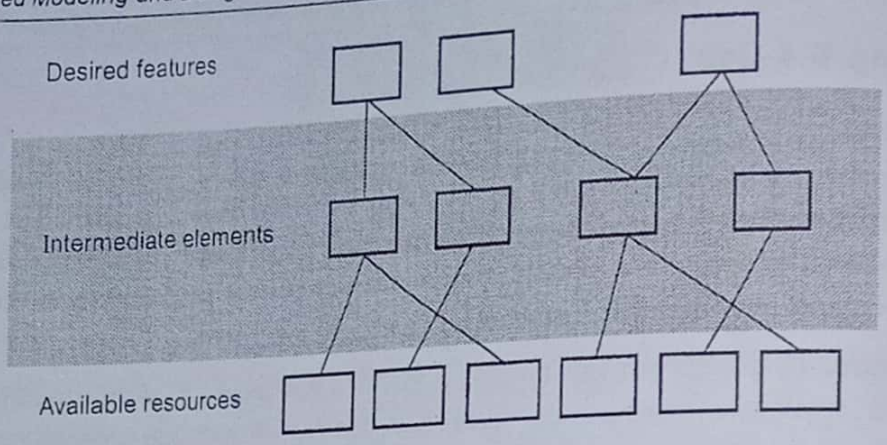


Fig. 5.2.2

rework on the existing intermediate elements so that the overall optimized and consistent design can be prepared.

- Thus the main **principle of bridging the gap** is that - Find the intermediate elements in a framework or class library, select them and then fit them together for providing particular service.
- Design is a creative work in which the intermediate elements are identified or new elements need to be created. This is called the **synthesis** of the design. Thus after analysis, the synthesis of the design is done.

Review Questions

1. *Why there is a need for bridging the gap in class design ?*
2. *Explain the use of intermediate elements in bridging the gap.*

5.3 Realizing Use Cases

Uses cases are meant for defining the behavior of the system.

The use cases are realized with the operations.

During the system design, the new operation and new object can be identified which provides the behavior.

Responsibility is something that object must do. For example - In washing machine - cleaning cloths is its responsibility of this machine.

Following steps are used to realize the use case -

Step 1 : List the responsibilities of use case or operation.

Step 2 : Each operation will have various responsibilities.

Step 3 : Define the operation for each responsibility cluster.

Step 4 : Assign the new lower level operations to classes.

For example - Consider the use case for ATM system. The ATM transaction can be performed using various operations such as cash deposit, withdrawal and checking balance amount. Following figure represents the use case diagram for the ATM system.

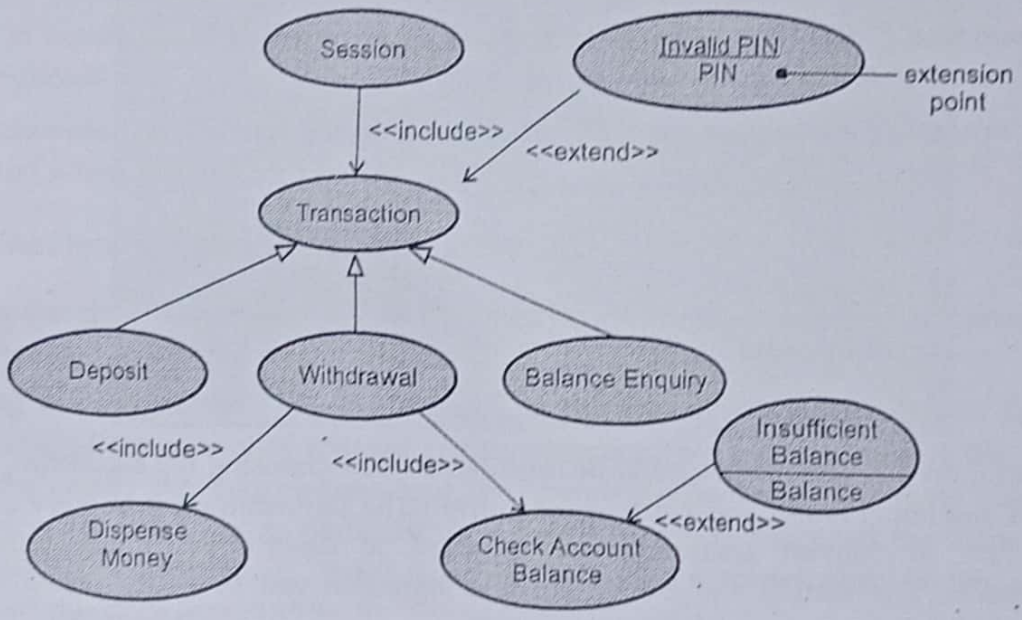


Fig. 5.3.1 Withdrawal of money from ATM

5.4 Designing Algorithms

- Every operation of class design is formulated to an algorithm.
- The **analysis** tells **what** the operations do but the **algorithms** tell **how** the operations do.
- For designing the algorithm **following steps are followed -**
  - Select the algorithm that reduces the overall cost of the operations.
  - Select the appropriate data structure for implementing the algorithm.
  - Define new internal classes and operations if required.
  - Assign the operations to appropriate class.

5.4.1 Selecting Algorithms

Normally, the operations traverse the class model in order to change the attribute values or associations. The Object Constraint Language (OCL) is commonly used to represent such traversal.

For more effective traversal of class design the **pseudo code** is used. Pseudo code is a kind of code using which the algorithmic steps can be expressed easily by avoiding programming details.

For example - For evaluation of the polynomial following pseudo code can be used -

```
Eval() returns RealNumber
{
  read the value for var x;
    /* Initialize var sum*/
  sum := 0;
  for each value in array p1 upto n terms
    sum := sum+p1[i].coef *pow( x, p1[i].expo);
 end for each
 return sum ;
}
```

Following are the factors that must be considered while choosing the algorithm.

- **Ease of Implementation and Understandability :** The algorithm must be very simple. Hence try to convert analysis model forward to design and make minimum adjustments. The simple and understandable algorithm can be easily coded in the programming language.
- **Computational Complexity :** The computational complexity denotes the efficiency of an algorithm. That means the computation complexity specified how much time the algorithm will take to execute or much memory will it occupy. The execution time can be constant, linear, quadratic or exponential. For example the linear search takes linear time i.e. n. Here n is the total number of elements in the list.
- **Flexibility :** The programs need to be extended quite often. In this process, some elements of the optimized algorithm needs to be changed. Making changes in optimized algorithm is very difficult. Hence any critical operation can be implemented in two ways - simple implementation but inefficient algorithm or complex implementation but efficient algorithm.
- **Example :** For **Online Course Reservation System** - Checking eligibility of students, validating the payment made by the student as fees are some complex operations for which algorithms can be designed.

5.4.2 Selecting Data Structures

Algorithms work with the help of data structures. During the analysis phase the focus is on the logical structure of information but during design the information can be represented with the help of data structure.

Various data structures are arrays, linked list, queues, stacks, trees, graphs,bags and so on. The data structures can also be used as a variation of the original data structure such as priority queue or binary search trees.

5.4.3 Selecting Internal Classes and Operations

The high level operations need to be decomposed into low level operations. Some of these low level operations can be shopping list operations.

Due to expansion of algorithms new classes can be added and hence new operations get introduced.

For example - In **Online Course Registration System**, the new class **Receipt** can be introduced when the Fees payment is accepted.

5.4.4 Assigning Operations to Classes

- When the class is meaningful then the operations of this class can be understood by anyone.
- During the design, new **internal classes** are getting introduced. In order to find out the name of the class that contains certain operation, we need to identify the objects and the collect the information about the operations performed by these objects. If only one object belongs to particular class then it can very easily tell the name of the operation it is performing. But if multiple objects are involved, then finding out the name of the class is very difficult.
- Following is a **set of questions** can be asked to understand the role of an object for possessing the important operations -
  - **Focal classes** : Many times, single important class is located centrally in the class model. This class serves the important set of operations.
  - **Analogy to real world** : If the object is a real-world object then the actions performed by these object help in finding out the important operations.
  - **Receiver of action** : Associate one operation with the target operation so that the important set of operations can be identified.
  - **Query Vs update** : Some objects are queried for just seeking the information where as there are objects that are accessed for updating the information. Thus the role of object leads to the type of operation.

Review Questions

1. *What steps should be performed while designing algorithms ?*
2. *What are the important factors that must be considered while selecting an algorithm ?*
3. *List and explain the steps to design algorithms with respect to class design.*

5.5 Recursing Downwards

Operations are arranged as layers. The operations at higher layers invoke the operations at lower layers. Thus the design process works from top to down. During the design process the higher level operations are focused and then the lower level operations are elaborated. Downward recursion can be done either by functionality and mechanism layers.

5.5.1 Functionality Layers

- Functional recursion means the high level functionality is broken into small functions having fewer operations. This decomposition must be logical. Combine similar operations and attach them to classes.
- **Dangers in functionality recursion :** If the higher level functionalities are broken arbitrarily then the lower level operations can not be related to the classes.
- Another danger is that this recursion depends heavily upon the statement of top level functionality. Any change in it can radically change the decomposition.
- To avoid these dangers, attach the appropriate operations to classes and make them more useful. An operation must be coherent and meaningful. It should not contain any arbitrary code. When operations are attached to the classes then there is less risk than the free floating functions.

5.5.2 Mechanism Layers

- Mechanism recursion means building the system to support various much needed mechanisms.
- Various mechanisms that a system needs are - storing information, coordinating objects, transmitting information, performing computations, and computing infrastructures. These mechanisms do not form the high level responsibilities of the system but they provide the support to perform high level responsibilities.
- These mechanisms are not directly the part of user needs but they create the support for the functionalities used for the user needs.
- Computing architecture includes various general purpose mechanisms such as data structures, algorithms and control patterns.
- There are some mechanisms that update the subjects and broadcast the changes to all the views.
- In large systems the functionality layers are mixed with mechanism layers. If the system is designed completely using the functionality layers then it becomes very sensitive i.e. a small change in the functionality can affect the overall system. If the system is designed completely using the mechanism layer then it becomes less useful. Hence the use of these two approaches in system design is always preferred.

- **Example** - The **Online Course Reservation System** includes several mechanisms such as updating the students database, monitoring the eligibility criterion, monitoring course information. Refunding the deducted amount on cancellation of reservation, computing the fees and so on.

Review Questions

1. *What are the two layers that are used in recursing downwards ?*
2. *Explain the danger in using functionality recursion.*
3. *Explain two ways in which downward recursion proceeds.*

5.6 Refactoring

- In the initial designs of the system we create a set of operations that are inconsistent, redundant and inefficient. Because it is just impossible to get the correct design in one pass.
- As the design evolves the operations and classes get identified and their use or purpose is getting understood by the developer. Hence it is essential to revisit the design, rework on the classes and operations and make them conceptually clear and coherent.
- Martin Flower defined **refactoring** as changes in the internals structure of software to improve the design without altering the external functionality.
- During the refactoring, the developers has to rework on the classes and operations in order to get improvement in design.
- Although refactoring is time consuming, it is essential part of any good engineering process.

Review Question

1. *Write short note on - Refactoring.*

5.7 Design Optimization

- It is always good practice to create the design and then to optimize it. Because creating the design and optimizing it simultaneously is very difficult. Run the application, measure the performance and then optimize it wherever needed. Although some amount of optimizations can also be done at the initial stage of the design.
- Normally the optimization for the critical area of the systems is done.

- The design model is built over the analysis model. The analysis model captures the logic of the system and the design model adds development details. The inefficient and semantically correct analysis model can be optimized to improve the performance. But optimized model becomes less likely reusable.
- Following **tasks** are carried out during design optimization -
  - Adding efficient access paths.
  - Rearranging execution order.
  - Saving the derived values.

  Let us discuss these tasks in detail -

5.7.1 Adding Redundant Associations

- Redundant associations do not add any useful information to the analysis model, hence such associations are not to be present in analysis model. However, in design the redundant associations are not considered to be useless. Infact they are utilized in such a way that the efficient access to the other classes can be possible. Hence in design, many associations from analysis model are rearranged, added or omitted for getting optimized performance.
- In **analysis model** the efficient network of associations is not created. But in **design model**, the **access patterns** and **relative frequencies** for accessing the class are the two factors that are considered for optimization.
- For example - Consider **Online Course Reservation System**. Following Fig. 5.7.1 represents some portion of analysis model. The operation Course.findSportman() returns the name of the students who play sports.

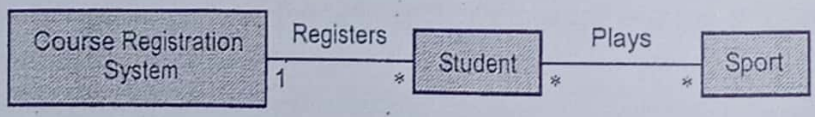


Fig. 5.7.1 Analysis model

- Now if there are 1000 students out of which only 10 students have skills for sports then for finding sport-persons a simple loop will execute for 1000 times. This leads to inefficient execution. To eliminate this problems some improvements can be made. For example we can create a hashed set for **play** instead of unordered linked list. Another solution is to create **index** for retrieving the objects that are accessed more frequently.
- Referring to above example, if we want to find out the students who speak German and play the sport then we need to iterate through the number of students for many times. Naturally we will get the less efficient model of execution. In order to improve the performance, we can use the **derived association**. The derived association does

not add useful information but is used for fast access. Following Fig. 5.7.2 illustrates this idea.

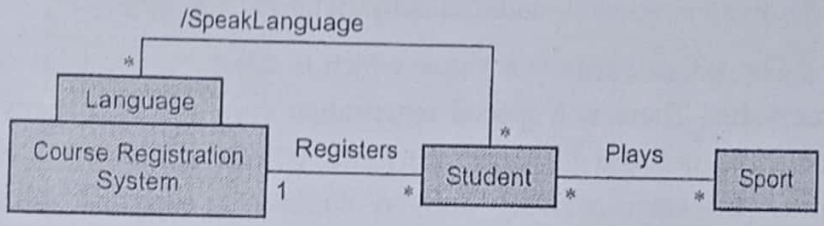


Fig. 5.7.2 Design model

- Following factors are considered while adding the redundant associations to the design model -
  - **Frequency of access** : For how many times the operation is accessed, is important to establish the association.
  - **Selectivity** : The selectivity is based on the **hits**. The **hits** is a number that denotes finding out of target object. If the selection criteria is met by an object then that target object is accessed. If the traversal rejects the number of objects then it just inefficient to find out the target object using simple loop.
  - **Fan out** : The fan-out denotes the average count on each **many** associations that are encountered in the traversal path. It basically represents the number of accesses to the last class along the path.

5.7.2 Rearranging Execution Order

- After determining the frequent traversal paths we need to optimize the algorithm itself. This optimization process mainly includes **elimination of dead paths.**
- **Narrow the search** as far as possible.
- Many times the **execution order of a loop** can be inverted from the original specification for optimization.
- Two **different derived associations** can be maintained for accessing the most frequent objects.

5.7.3 Saving Derived Values

- In order to store the derived attributes and avoid the recomputations new classes can be added to the design model. This storage or cache needs to be updated whenever the object dependant upon it gets changed.
- Following are the ways to handle the updates of cache,
  - **Explicit update** : The explicit code is added to the operations of source attributes so that the derived attributes that depend upon these source attributes get updated explicitly.

- **Periodic re-computation :** The derived attributes can be recomputed periodically because the application changes the values in bunches. Explicit update is not always possible and therefore periodic recomputation is preferred.
- **Active value :** The **active value** is a value which is automatically kept consistent with its source value. There is a special registration system which keeps track of all the derived attributes and its source attribute. Whenever, there is a change in source attributes the corresponding derived attribute is updated. Using some programming languages the active value can be obtained.

Review Questions

1. *Why class design is prepared ? Explain following concepts with respect to class design : i) Bridging gap ii) Designing algorithms iii) Design optimization.*
2. *Explain the tasks involved in design optimization.*
3. *What is the purpose of design optimization ? Briefly discuss the tasks of design optimization.*

5.8 Reification of Behavior

- Manipulation of behavior written in a code is not possible during the run-time(execution). But there are situations in which certain behavior needs to be modified at run time. Then we need to **reify the behavior.**
- **Reification** is a mechanism in which some entity which is not an object is promoted to an object.
- If the behavior is reified then we can store it, pass it to another operation or modify it.
- Due to reification the complexity of the model gets increased but it brings the lot of flexibility in the system design.
- The behavior can be reified by encoding it to an object and later on when it is run one can decode it. The runtime cost can be less if encoding invokes high level procedures. But if the entire behavior is encoded in some different programming language then the runtime cost can be significant.
- The example of reification can be illustrated by following class model. The **Algorithm** contains data structures and steps of execution. If one algorithm is converted into an object then this object can be passed to another algorithm and desired behavior of one algorithm can be used in another algorithm. Similarly, we can modify the behavior of one particular algorithm as per the requirement.
- There are number of behavioral patterns that are used to reify the behavior. These are **states, commands** and **strategies.**

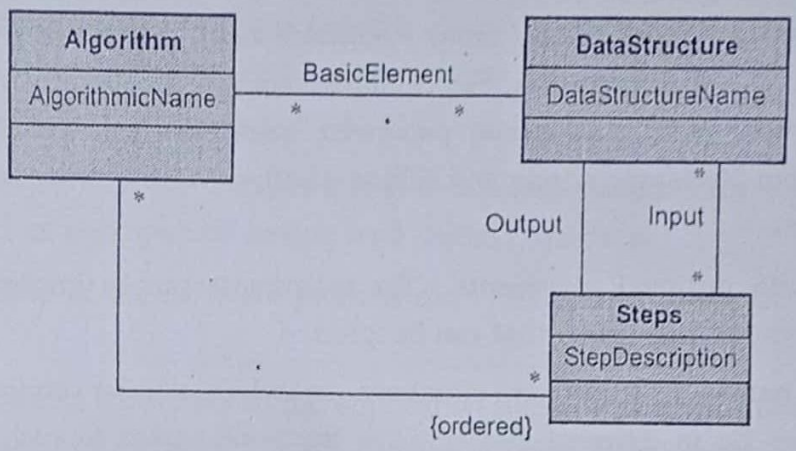


Fig. 5.8.1 Class model : Algorithm as an example of reification of behavior

- State represents the run-time interpreter. The **command** represents encoding sequence of requests and **strategy** represents the operations.

Review Questions

1. *Explain the concept of reification of behavior with the help of suitable example.*
2. *Explain the following terms in relation to class design.*
   a. Refactoring b. Reification.

5.9 Adjustment of Inheritance

As the class design progresses the inheritance can be adjusted using following steps -

- Rearranging classes and operations to increase inheritance.
- Abstracting out common behavior out of multiple classes.
- Using delegation when inheritance is invalid.

Let us discuss each in detail -

5.9.1 Rearranging Classes and Operations

- Many times the several classes define same operations, by adjusting the definition of operations the single inheritance can be adjusted.
- Before using the inheritance, all the operations must have same semantic and signature. The signature of an operation means the number of parameters, type of parameters and return type.

- Following are some adjustments that can be done to increase the inheritance -
  - **Operations having special cases :** Some operations have special cases with some special values with the arguments. Hence implement special operations using the general operations with appropriate parameter value. For example Shape is a general class but Rectangle, Circle and Ellipse are the shapes with specific values of its argument.
  - **Operations with optional arguments :** The inheritance can be implemented by adding the optional arguments that can be ignored.
  - **Inconsistent names :** There are situations in which similar attributes with different names lie in different classes. Give the same names to such attributes and move them under common ancestor. In short there should be consistency in the names of attributes and operations.
  - **Irrelevant operations :** There are several classes in a group that define some operations whereas there are some other classes for which these operations are least significant. Hence define only those operations that have some significance for that class. For example - Compute_area() is an important operation for the **Circle** class but it is unimportant for the **Line** class.
- **Example -** For **Online Course Reservation System** Student pays Fees. The class **Payment** can be supported by two specialized classes **CreditCard Payment** and **Cash Payment**.

5.9.2 Abstracting Out Common Behavior

- If two classes repeat several operations and attributes then it is possible that these two classes are specialized classes of some common thing.
- Whenever, there is a common behavior, then a common superclass can be created and specialized features can be added into the subclass. This transformation of class model is called **abstracting out** the common class.
- It is always good to have abstract super class and specialized subclasses. For example -

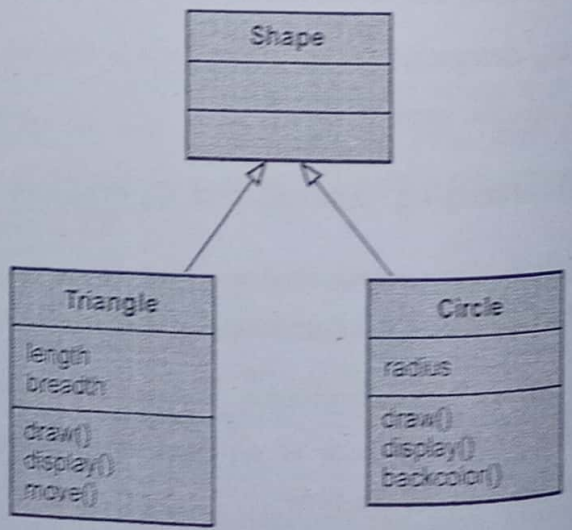


Fig. 5.9.1 Abstracting out

- Abstract super class have multiple benefits. Such super classes are useful for **sharing and reuse**. A class library of such abstract super classes can be created and can be reused by multiple applications.
- The abstract super class can be **split** into two different classes. Each separate class can be maintained as a component having well defined interface.
- The **extensibility of the software product** can be improved using the abstract super class.
- The use of abstract superclass in the application helps in **configuration management and maintenance of software product** because it helps in generating the **customized versions** of the software.
- For example - If we keep the **Payment** class as abstract then it can be specialized by adding the subclasses such as **ChequePayment, CashOnDeliveryPayment,** and **CreditCardPayment.** Thus the abstract superclass **Payment** can be used by multiple applications and customized sub classes can be created.

5.9.3 Using Delegation

- Inheritance is basically a generalization relationship. In this relationship the behavior of the superclass is shared by the subclass. In this case, the operations of subclass **override** the corresponding operations of superclass.
- Sometimes the programmers make use of inheritance in the implementation for ease of programming and there is no guarantee of the same behavior between the super and subclass. Sometimes the newly added class represents the same behavior that is present in the existing old class. In such situation, the programmer is tempted to use inheritance relationship by making the new class as a subclass. Due to this there are chances of providing unwanted operations from this inheritance hierarchy. This situation can be described as **inheritance of implementation**. It is not at all recommended in the design as it leads to some incorrect behavior.
- For example - A **TwoDArray** is a class that contains the two dimensional array element as a data value. Using the read_array() and print_array() operations the two dimensional array can be handled. If we add a new class say **Matrix** then it is obvious to associate this class with **TwoDArray** using the inheritance relationship. Although from implementation point of view, it seems very simple and convenient, but this could lead to some difficulties. For instance - if the matrix is sparse then

instead of representing it using Two dimensional array, another efficient representations can be used. But if we model these classes using inheritance relationship then it is not possible to use another efficient representation for the class **Matrix**. Hence instead of using inheritance some another association between the two classes are preferred. Refer Fig. 5.9.2

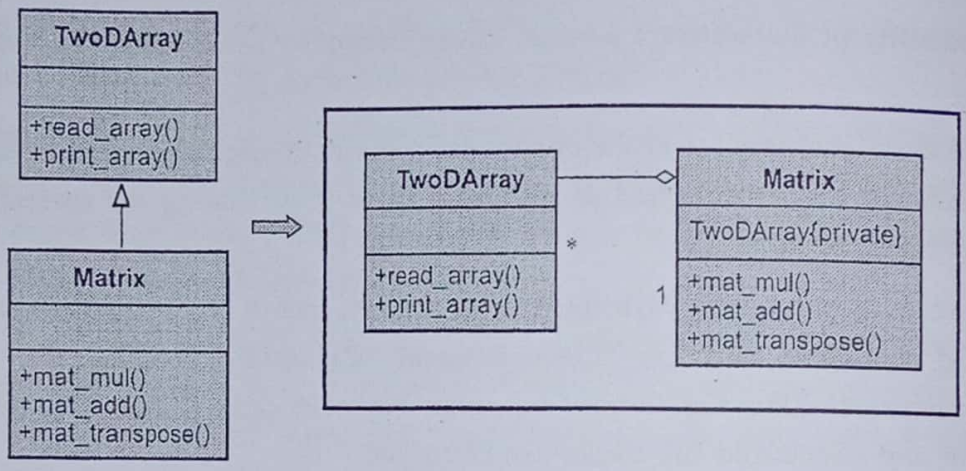


Fig. 5.9.2 Alternative design - Conversion from inheritance to aggregation

- **Delegation** is a mechanism in which only the useful operations from one object are cached and send them to another object. The **advantage** if using delegation is that only meaningful operations are delegated and there is no danger of inheriting meaningless operations.

Review Questions

1. *Explain the adjustments that are made to increase inheritance*
2. *Explain the concept of abstracting out the common behavior in adjustment of inheritance.*
3. *What do you understand by the term delegation used in adjusting the inheritance ?*
4. *What is the importance of adjustment of inheritance ? Discuss the steps of doing it.*

5.10 Organizing a Class Design

The organization of a class design can be improved using following steps -

Step 1 : Information hiding

Information hiding is a technique by which the internal specification is hidden from outside world.

There are following ways to hide information -

i) Do not directly access the foreign attribute.

ii) Limit the scope of traversal in class model.

iii) Define interfaces at high level of abstraction.

iv) Avoid cascading of method calls.

v) Hide external objects.

Step 2 : Coherence of entities

- Coherence is an important design principle. A class, operation or a package is coherent if it is organized on consistent plan and all the part of the design are following a common goal.
- An entity(such as class, operation or package) should have a single theme.
- The entity should not be a collection of unrelated parts.

Step 3 : Fine-tuning packages

- During analysis, the class model is partitioned into packages.
- The initial set of packages is not suitable for the final implementation. Because in later stages of development new classes may get added to the existing system model or some of the classes that are present during analysis may get refined or may get grouped with other classes.
- The packages should have well defined interfaces. These interfaces must be minimal in number.
- The interface between two packages must be an association that can co-relate the classes in one package with the classes in another package.
- While fine tuning the packages, the rule of thumb which is used for arrangement of classes within the package is - Arrange the closely related classes in one package and the classes that are unrelated must be present in different package.
- Packages should have some theme or some specific purpose.
- There must be strong coupling within the same package.

5.11 ATM Example

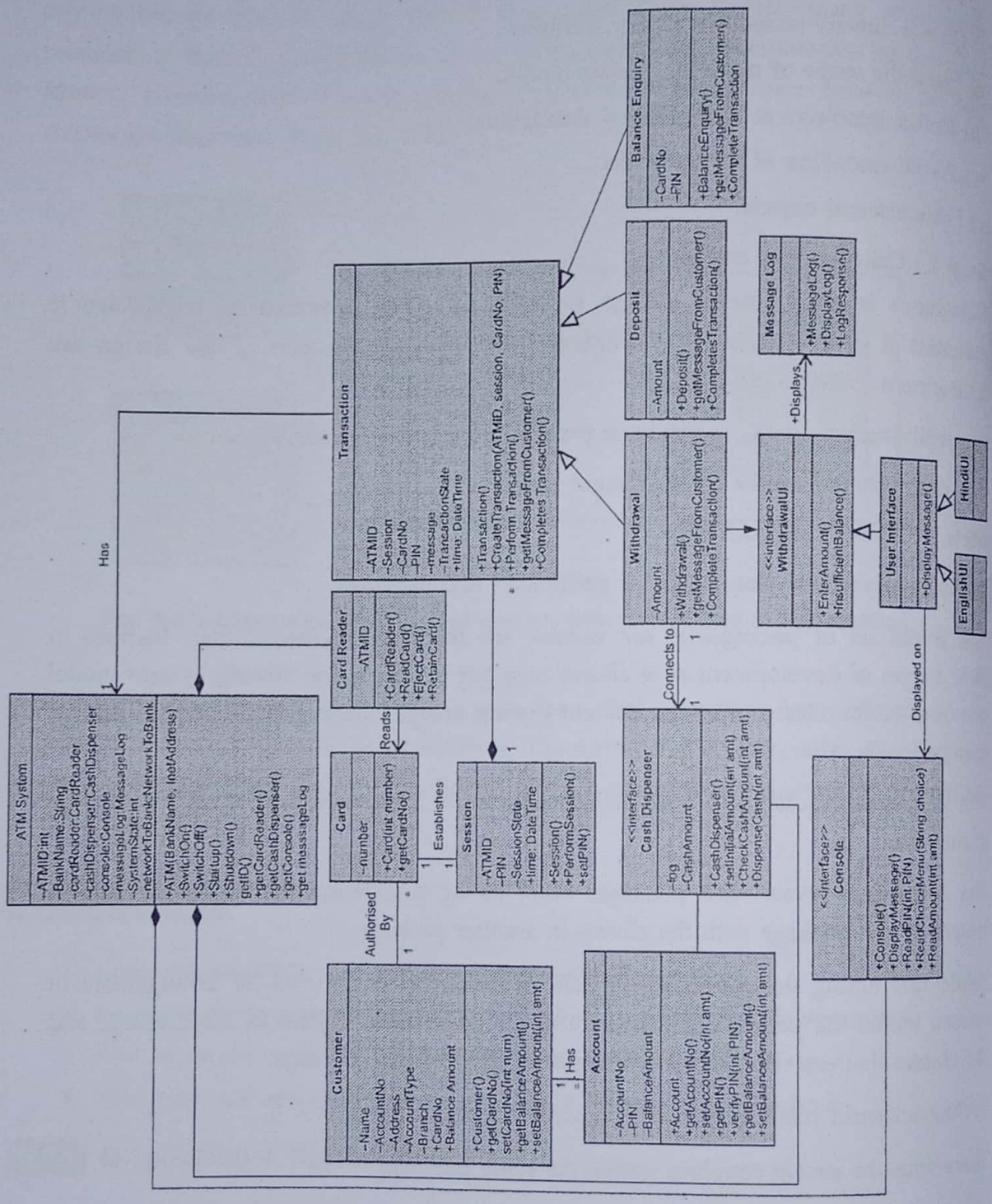


Fig. 5.11.1 Class diagram for ATM system

Part II : Implementation Modeling

5.12 Overview of Implementation

- Implementation is a **final stage** of development. In this stage a suitable programming language is used to create a **working model**.
- It must be straightforward and almost mechanical, because major decisions about building a model are taken during the design stage itself.
- Ideally, during implementation, the design decisions are simply translated into a source code using suitable programming language.
- Following are the steps that are followed in implementation modeling
  - Fine tuning the classes
  - Fine tuning generalizations
  - Realizing associations
  - Preparing for testing

5.13 Fine-tuning Classes

Before writing the code we need to fine tune the classes so that the code becomes simple to underastand. It also increases the performance of execution of the code.

Following are the techniques used to fine tune the classes

(1) **Partition a class :** When the class is partitioned the information of a single class is split into two classes. For example - An **Employee** class can be partitioned into two classes namely **EmployeeHomeAddress** and **EmployeeCompanyAddress.** This partitioning can be represented by following figure -

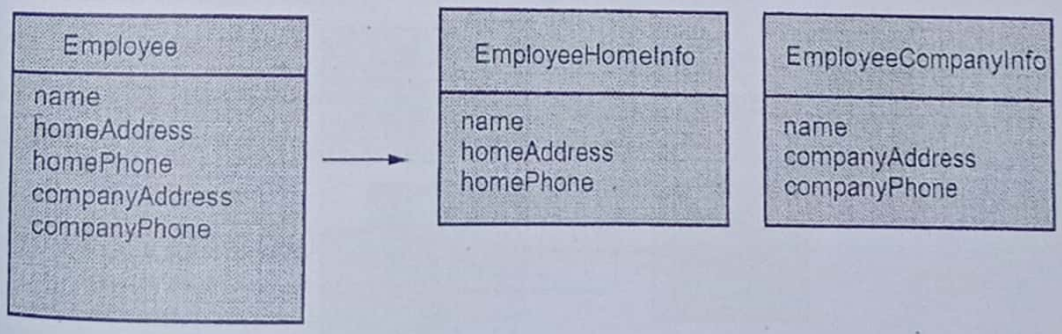


Fig. 5.13.1 Partitioning a class

(2) **Merge class :** Converse of partitioning the class is merging a class. If we have with us two classes namely **EmployeeHomeInfo** and **EmployeeCompanyInfo** then we can merge them into a single class named **Employee.** For example

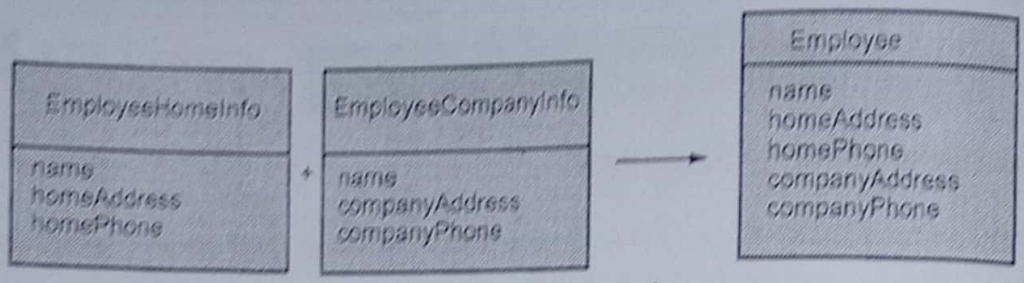


Fig. 5.13.2 Merging of two classes

(3) **Partition or merge attributes :** When a class gets fine tuned, sometimes we can adjust the attributes by partitioning or merging. Following representation illustrates it.

Fig. 5.13.3 Partitioning or merging of attributes

(4) **Promote an attribute or demote a class :** An entity can be represented as an attribute or a single class or as a several related classes. For instance, in the following figure, the Address can be represented as a single attribute, as a single class or as several related classes

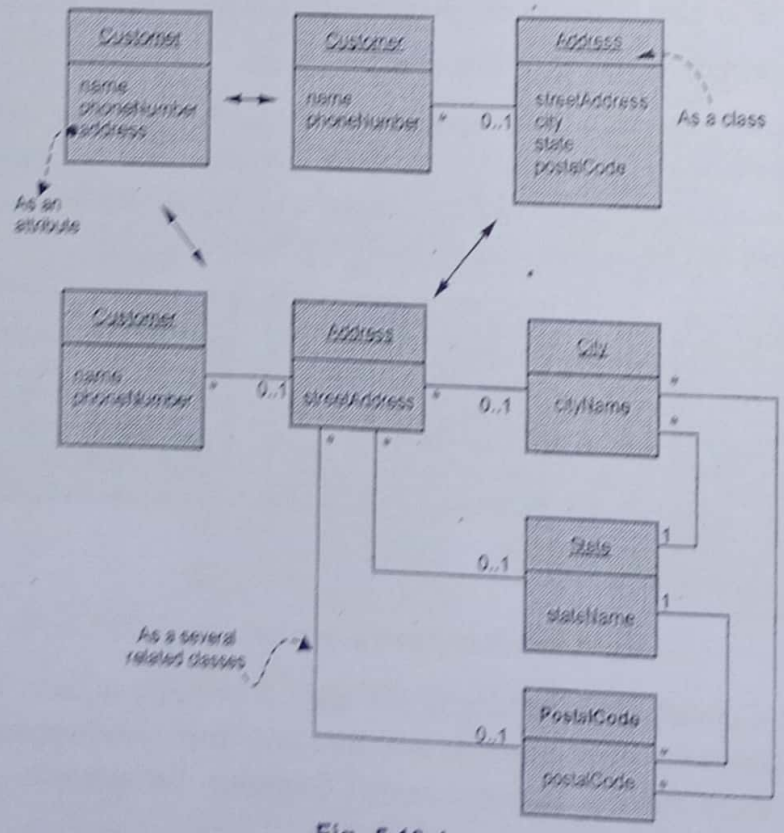


Fig. 5.13.4

Review Question

1. *Explain the techniques used to fine tune the classes.*

5.14 Fine-tuning Generalizations

Fine tuning generalization means either adding a generalization or removing it before coding. This makes the implementation simplified. For example - **Online course registration system,** while enroling for the course, the user can enrol as a **registred user** or as **Guest.** As per the type of user, the contents of the course will be displayed to him. Hence we need to add genealization to the user.

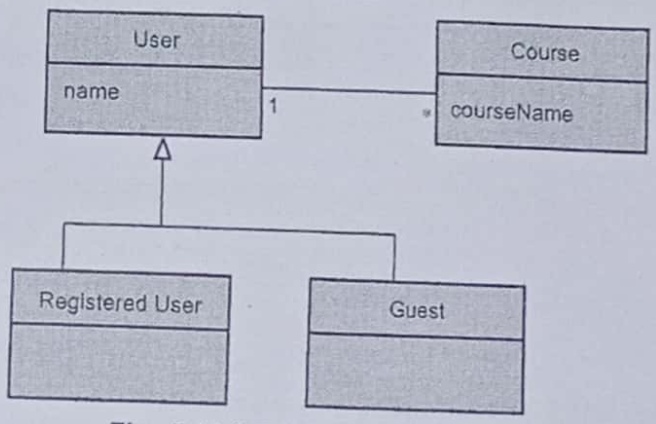


Fig. 5.14.1 Adding generalization

5.15 Realizing Associations

Associations provide the access path between two objects.

Inherently associations are bidirectional. But sometimes association need to be one directional. From implementation point of view it is necessary to know if the association one directional or bidirectional.

(1) **One way association :** If the association is one way association then it can be implemented using pointer or a reference. Following Fig. 5.15.1 shows the use of pointer in implementing one way association.

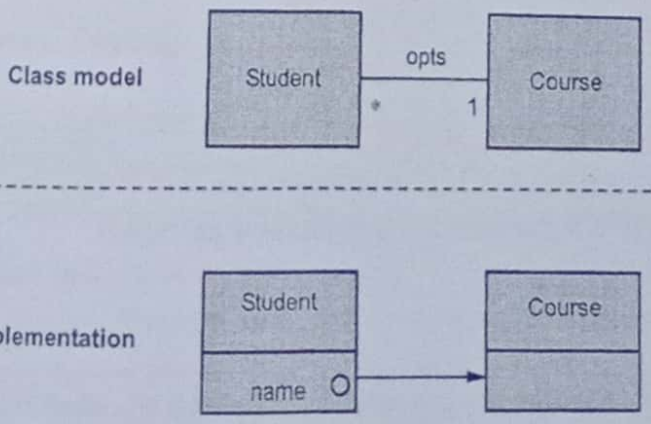


Fig. 5.15.1 Implementing one-way association with pointers

(2) **Two way association :** There are three approaches to their implementation using two way association -

- **Implement one way :** The implementation of one way traversal is using pointer. Then perform search when backward traversal is required.
- **Implement two way :** Implement with the pointers in both the directions. This allows fast access. For example - Consider following figure in which the implementation of two way association using pointer is represented.

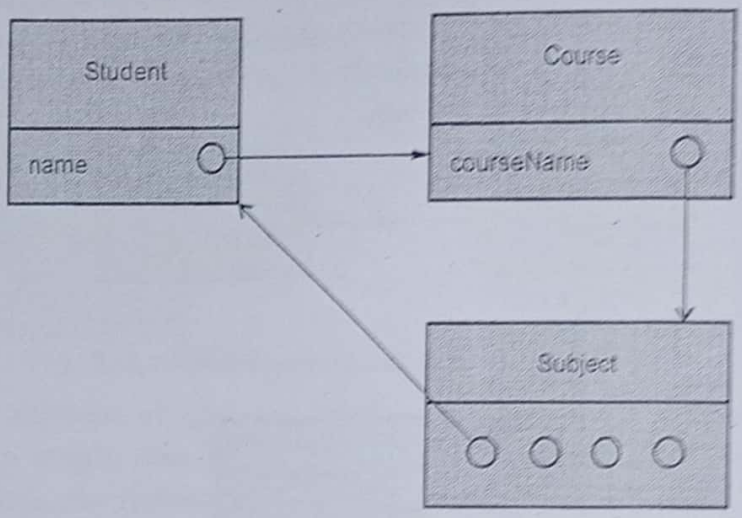


Fig. 5.15.2 Implementing two way association

- **Implementing association object :** This is the most general approach to implement association but it requires high programming skills.

Following figure represents how to implement an association object. The optsFor is an association between Student and Course class. It is implemented as an object.

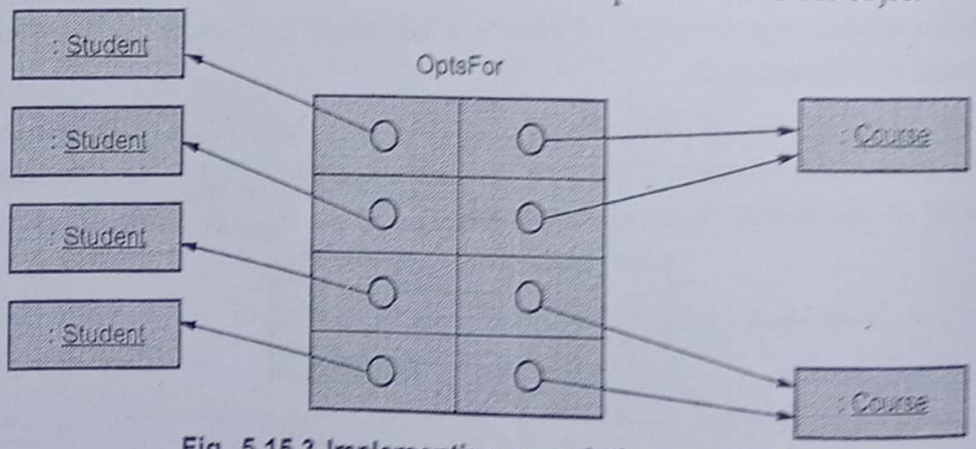


Fig. 5.15.3 Implementing association as object

Review Question

1. *Explain the techniques used to realize the generalization relationship of a class ?*

5.16 Testing

- Testing is a quality assurance mechanism used for catching the errors.
- Testing provides a measure of the quality for the software application. The number of bugs identified gives the measure of quality of software.
- Testing should be carried out at every stage of software development and not just during implementation.
- During **analysis**, test the model against the **user expectations**. During the **design**, test the **architecture** and simulate the performance. During **implementation** test the actual **code**
- Testing should progress from small pieces to ultimately the entire application.
- Developers start testing the classes, methods or functional modules. This is called **unit testing.**
- The next step is **integration testing.** In this testing how the classes and methods fit together is getting tested.
- The final step is **system testing** in which the entire application is tested.

5.16.1 Unit Testing

- The **unit testing** techniques are applied to detect the errors from each software component individually.
- The focus of unit testing is to uncover the errors in design and implementation.
- In **integration testing,** A group of dependent components are tested together to ensure their quality of their integration unit.
- The objective is to take unit tested components and build a program structure that has been dictated by software design.
- The unit testing and integration testing are normally carried out by developers because they understand detailed logic and possible sources of the errors.

5.16.2 System Testing

- System testing is defined as the testing conducted on the complete integrated system to evaluate the working of the system as a whole. The system testing is carried out in context of a **functional requirements and System requirements**
- The **functional testing** is a kind of testing that focuses on actual usage of the product. That means during functional testing the basic required functionality of the system is tested.

- Functionality testing is performed to verify that a software application performs and functions correctly according to design specifications. During functionality testing we check the core application functions, text input, menu functions and so on.
- The **non-functional testing** is always concentrating on customer expectations. The non-functional test cases target performance, resource utilization, usability, compatibility etc.
- System testing is performed on the basis of written test cases according to the information collected from the detailed design document, SRS and module specification.

Review Question

1. *Explain in brief : (1) Unit testing (2) System testing.*

Part III : Legacy Systems

5.17 Introduction to Legacy Systems

The Legacy systems are the older, large complex computer based systems which may be using the obsolete technology. These systems may include the older hardware, software, process and procedures. The legacy systems are business critical systems and are used for a long period (even though having obsolete technology) because it is too risky to replace them.

For example : Air Traffic Control (ATC) system which may be used from a long period. But it will create a great difficulty in managing the flight schedules, if such system is replaced by newer technology.

Various components of Legacy system are -

System hardware : Most of the legacy systems are written for the mainframe computers. These computers are now not in use and are expensive to maintain.

Supporting software : The supporting legacy software are older operating systems, compilers and utility software. But these software are generally obsolete.

Application software : It includes number of separate programs that include the business logic. These application software are sometimes referred as the legacy system.

Application data : In legacy systems the application data is accumulated and then used. This data may be stored in the files and sometimes it is not at all organized.

Business process : In order to meet the business objective some processes are executed. Such processes are called business processes. For example in Air Traffic Control system 'flight scheduling' is the business process.

Business policies and rules : It includes the set of rules and constraints on the business. Generally such policies and rules are documented.

5.17.1 Legacy System-layered Technology

The legacy software is comprised of various layers. It is as shown in following Fig. 5.17.1.

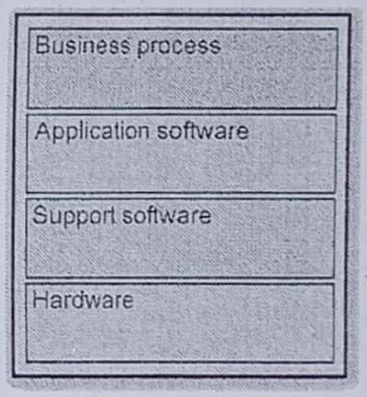


Fig. 5.17.1 Legacy system layers

Ideally **interfaces** are maintained for each layer of the legacy system so that if changes are made in some layer then the adjacent layers may not get affected. But in reality, changes in one layer affect the adjacent layers because -

1. If one layer is changed then new facilities get added. To take the advantages of these new facilities the adjacent layers may demand for the change.
2. Changes in software may cause the degradation in performance of the system. This performance then can be improved by changing the hardware. And if the hardware is changed for rapid performance then the corresponding software need to be changed.
3. Sometimes there is a drastic change in the system for example : the mainframe technology may be replaced by distributed system. Then in such a case changes in each layer of the legacy system are necessary.

Following are the techniques that are used to deal with legacy systems -

(1) Reverse engineering
(2) Wrapping
(3) Maintenance

Let us discuss them one by one.

5.18 Reverse Engineering

Reverse engineering is the process of **design recovery**. In reverse engineering, the data and architectural information is extracted from the source code.

5.18.1 Difference between Forward and Reverse Engineering

Sr. No.	Forward engineering	Reverse engineering
1.	The application is created using requirements.	From the application, the requirements are extracted.
2.	It more certain. That means when the developers are given the requirements, then the application is created for satisfy these requirements.	It is less certain. That means different requirements can be obtained from the implementation.
3.	It is prescriptive. Developers are told how to work.	It is adaptive. The reverse engineer must find out what the developer actually did.
4.	More time is required to accomplish.	Less time is required to accomplish it.
5.	The model must be correct and complete otherwise the application will fail.	The model can be imperfect.

5.18.2 Inputs to Reverse Engineering

Following are the inputs required by the reverse engineering process -

(1) **Programming code :** The source code is the rich source of information as an input. It helps to understand the flow of control and data structure. Comments, meaningful variable names, function and methods help to understand the application more deeply.

(2) **Data :** Using data the data structure is discovered.

(3) **Database structure :** The database structure specifies the data structure and many constraints precisely and explicitly.

(4) **Forms and reports :** Forms and report definitions are helpful if their binding to variables is available.

(5) **Documentation :** Documentation provide the context for reverse engineering. User manuals are very helpful. However while using the documentation one must be careful as it may be inconsistent with application code.

(6) **Application understanding :** By better understanding the application, the better interfaces can be created.

(7) **Test cases :** Test cases are used to exercise the normal flow of control of the application.

5.18.3 Outputs from Reverse Engineering

Following are the several useful outputs for reverse engineering

(1) **Models** : Model represents the software's scope and intent. It is used for understanding the original software and building the successor software.

(2) **Mappings** : Mapping of programming code to the state model or interaction model is possible.

(3) **Logs** : The log is a document that records the observations and pending questions. It is used for making decisions during the reverse engineering.

Review Questions

1. *Enlist the inputs and outputs used for reverse engineering.*
2. *What is the difference between forward and reverse engineering ?*

5.19 Building the Class Models

The reverse engineering begins from class model. The classes and their relationships must be understood first in order to understand the class model. There are three distinct phases in building the class model

1. Implementation recovery 2. Design recovery 3. Analysis recovery

1. Implementation recovery :

- From the application create an **initial class model.**
- If the program is written in Object oriented language then the classes and generalizations can be directly used in the initial class model.
- If the program is not in an object oriented language, then the data structures and operations are studied in order to determine the classes.

2. Design recovery :

- The next step is to recover association.
- Typically the multiplicity in one direction is identified using the single pointer attribute in the implementation. The multiplicity in the reverse direction is typically not declared and it is determined by examination of the code.
- The multiplicity has lower limit of 0 or 1. The lower limit is 0 if target object is initialized somewhere in source code. The lower limit is 1 if the target object is initialized at the object creation time.

3. Analysis recovery :

- In analysis recovery phase, firstly the proper interpretation of the model is done, then it is refined and made more abstract.
- All the redundant information is eliminated or marked as redundant.
- If the source code is not an object oriented code then using similarities and differences in structure and behavior of the source code the generalization is infered.
- By careful study of code, the aggregation and composition is identified.
- Finally the packages are created to organize classes, associations and generalizations.

Review Question

1. *Explain the process of building class model during reverse engineering.*

5.20 Building the Interaction Model

- For building the interaction model it is necessary to understand the **behaviour** of the system.
- For understanding the behaviour, consider the **class model** and add the **methods** to it using **slicing** technique. A slice is a subset of a program that represents a specific behavior of the application. Then marks all the statements from the slice of the code. The sliced code is then converted from procedural representation to object oriented representation.
- The **activity digram** is build to represent the extracted method so that the sequence of processing and flow of data to various objects can be understood.
- From this activity diagram, further the **sequence diagrams** can be build for simplification purpose.

5.21 Building the State Model

- For studying the user inteface the state model is useful, otherwise there is no prominent use of a state model.
- For buillding the state model, the sequence diagrams are taken as input. Fold various sequence diagrams for a class together, by sequencing events and adding conditionals and loops.
- All possible **states** can be identified by studying the classes.
- Initation and termination of the state model corresponds to construction and destruction of objects.

5.22 Reverse Engineering Tips

During reverse engineering process the class, interaction and state models are build. Following are some useful tips used for this process.

(1) **Distinguish suppositions from facts :** Reverse engineering yields hypotheses. As reverse engineering proceeds you may need to revisit some of the earlier decisions and change them.

(2) **Use flexible process :** Adjust the reverse engineering process to fit the problem. Problem styles and the available input vary widely.

(3) **Expect Multiple interpretations :** Alternative interpretations can generate different models. If more information that is available, then less judgements can be made for reverse engineers.

(4) **Don't be discouraged by approximate results :** During reverse engineering one can extract around 80 percent of application's meaning. Hence during reverse engineering process one can not get the accurate results about the modeling. There can be lack of perfect information.

(5) **Expect odd constructs :** There is a possibility of building inaccurate or incomplete model during reverse engineering process.

(6) **Watch for a consistent style :** Software is designed using a consistent strategy. Observer the consistent style from the code during reverse engineering process.

5.23 Wrapping

- Wrapping is one of the technique used to deal with legacy systems.
- A wrapper is a collection of interfaces that control access to the system.
- It consists of set of **boundry classes** that provide the interfaces.
- The methods of boundry class call the existing system's operations.
- Normally the legacy code is complex and messy, but the boundry classes of wrappers hide the details from outside.
- For example - consider a web application. Suppose a legacy banking application which is written in COBOL and it is running on old hardware. Wrapping technique can treate the COBOL logic as OO method and it can be attached to modern Web application.

5.24 Maintenance

Software has following stages of maintenance -

(1) **Initial development :** Developers create the software.

(2) **Evolution** : The software undergoes major changes in functionality and architecture. Using **refactoring** technique the software quality can be maintained.

(3) **Servicing** : When requirement for software changes is limited to minor fixes, then at this stage software start becoming obsolete.

(4) **Phaseout** : The vendor starts thinking to demise the current software product.

(5) **Closedown** : At this stage, the product is removed from the market and customer is redirected to another software.

The software engineers goal is to slow decline of the software product.

Review Question

1. *Write short notes on - (1) Wrapping (2) Maintenance.*

❑❑❑

Unit VI

6 Design Pattern

Syllabus

What is a pattern and what makes a pattern? Pattern categories; Relationships between patterns; Pattern description Communication Patterns: Forwarder-Receiver; Client-Dispatcher-Server; Publisher-Subscriber.

Management Patterns: Command processor; View handler. Idioms: Introduction; what can idioms provide ? Idioms and style; Where to find idioms; Counted Pointer example

Contents

6.1 What is a Pattern?

- Pattern represents a reusable solution to a common design problem.
- The solution is specified by describing its constituent components, their responsibilities and relationships and the manner in which they collaborate.
- **Example of architectural pattern : Model-view-controller**

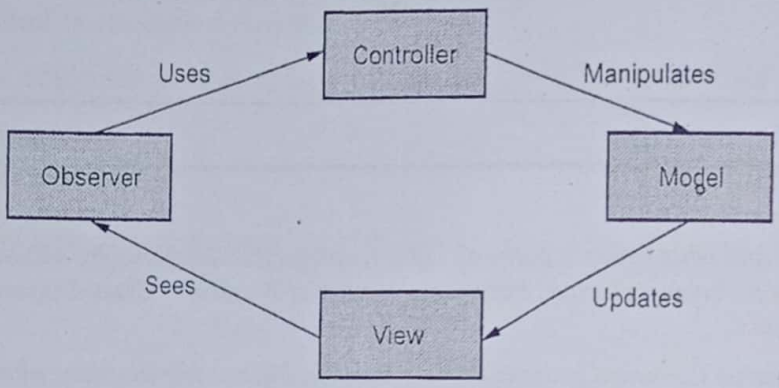


Fig. 6.1.1 Model view controller

In Model-view-architectural pattern there are three important classes - Model, View and Controller.

1. **Model** : This component is responsible for storing and managing the data. It encapsulates core data and functionality.
2. **View** : This component displays the information to the **user/observer**. The view is a visual representation of the data- like a chart, diagram, table, form. The view contains all functionality that directly interacts with the user - like clicking a button, or an enter event.
3. **Controller** : The controller connects the model and view. The controller converts inputs from the view to demands to retrieve/update data in the model. The controller receives input from view, uses logic to translate the input to a demand for the model, the controller passes data from the model back to the view for the **observer** to display.

Review Question

1. What is pattern ? Explain it with suitable example.

6.2 What Makes a Pattern ?

Pattern is described using three parts

1. Context 2. Problem 3. Solution

1. Context :

- It is basically a **situation** due to which the **problem is raised.**
- The context can be **general** or it can be **specific.**
- Specifying the correct context for a pattern is difficult because it is practically impossible to determine all the situations.
- A general approach for defining the context for a pattern is to make **a list of all known situations.** These are the situations for which particular pattern can occur.

2. Problem :

- This is a part of pattern description schema that occur repeatedly in a specific context.
- The term **force** is used to denote the aspect of the problem that should be considered while solving it. For instance -
  - The solution must satisfy all the requirements of the problem.
  - The solution must consider all the constraints
  - All the desirable properties must be present in the solution.
- For example - Model View Controller used in user interface applications must be easy to modify and its functionality must not be affected due to the modifications made in it.
- For a problem, the **set of forces** are repeatedly arising in the context. Forces are key to solve the problems. Better they are balanced, better is the solution to the problem.

3. Solution :

- This is a proven solution of the problem.
- The solution is considered as a configuration to balance the **forces.**
- There are two apsects of a solution -
  - **Static Aspect :** Every pattern specifies certain structure and configuration of elements. This addresses the static aspect of the solution. The components and their relationship is considered for the solution.
  - **Dynamic Aspect :** Every pattern specifies the runtime behavior. For example how many participants are taking part in solving the problem ? How the work is organized ?
- The solution does not necessarily resolve all the forces associated with the problem.
- No two implementations of the given pattern are likely to be the same.

Review Question

1. *Explain three important parts of the design pattern.*

6.3 Pattern Categories

The pattern is categorized in three groups -

1) Architectural pattern

2) Design pattern

3) Idioms

Let us discuss them.

1) Architectural patterns :

- This pattern describes software architecture that are built as per the structuring principles.
- It expresses a fundamental structural organizational schema for software systems.
- It provides the set of predefined subsystems, specifies their responsibilities.
- It includes rules and guidelines for organizing the relationships between them.

2) Design pattern :

- It provides scheme for redefining the subsystem or components of software system or the relationships between them.
- It describes a common structures of communicating components that solves general design problems within particular context.
- Design patterns are medium scale patterns.

3) Idioms :

- An idioms represent the lowest patterns.
- An **idiom** is a low-level pattern specific to a programming language.
- The idioms describe how to implement components or relationships among the components using the features of specific programming language.

Review Question

1. *Explain in brief the three different categories of the pattern.*

6.4 Relationships between Patterns

- Patterns do not usually exist as a single entity. The patterns are dependent on each other. A pattern may depend on a smaller pattern it contains or on larger patterns in which it is contained.
- If one pattern raises some problem then that problem can be solved by some other pattern.

For example -

- Consider the relationship between **Model View Controller** (an architectural pattern) and **Observer** (design pattern) pattern while developing a human computer interaction application.
- The user interfaces are created for this application.
- In a Model View controller pattern the **views** and sometimes even **controllers** depend on the state of **model**.
- The consistency among them must be maintained. Whenever the state of **model** changes we must update all its dependent **views** and **controllers**.
- Hence to change the user interface the **observer** (one of the design pattern) is used. There are two roles on **observer** pattern **subject** and **observers**. when a **subject** changes state, all registered **observers** are notified and updated automatically. In the human computer interaction application the **model** plays the role of **subject** and **views** and **controllers** play the role of **observers**.
- Thus both the **model-view-controller** and **observer** pattern work together to develop the interactive Human Computer Interface like applications.

6.5 Pattern Description

The pattern is described using the following format

Name - The name and a short summary of the pattern.

Also Known As (aka) - Other names for the pattern, if any are known.

Example - A real-world example demonstrating the existence of the problem and the need for the pattern.

Context - The situations in which the pattern may apply.

Problem - The problem the pattern addresses, including a discussion of its requirements.

Solution - The fundamental solution principle underlying the pattern.

Structure - A detail specification of the structural aspects of the pattern.

Dynamics - Describe scenarios of the pattern.

Implementation - Guidelines for implementing the pattern.

Variants - A brief description of variants or specializations of a pattern

Known Uses - Examples of the use of the pattern, taken from existing systems

Consequences - The advantages and disadvantages the pattern provides.

Review Question

1. *Explain the pattern description.*

6.6 Communication Patterns

The communcation pattern is used for communication between software modules.

There are two important aspects of communication patterns and those are -

1. Encapsulation 2. Location transperency

1. **Encapsulation :** Hiding the details of underlying communication mechanism from users is called encapsulation.
2. **Location transperency :** The technique in which the applications are allowed to access remote components without any knowledge of their physical location is known as location transperency.

There are three types of communication patterns -

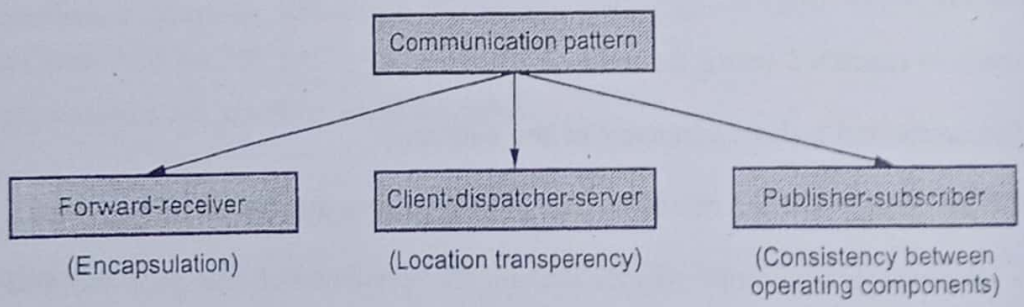


Fig. 6.6.1 Types of communication pattern

6.6.1 Forwarder-Receiver

Problem : Many components in distributed environment communicate in peer to peer fashion.

1) The communication between peers must not depend on a particular Interprocess communication mechanism.
2) Different platforms provide diferent Interprocess Communication (IPC) mechanism.
3) Performance of such communication is often low.

Solution : Encapsulate the interprocess commmunication mechanism.

Intent :

- The Forward-Receiver design pattern provides **transperent interprocess communication** for software systems with **peer to peer** interaction model.
- Using forwarders and receivers the peers are separated from underlying communication mechanisms.

Motivation :

- The distributed peers work together in collaboration to solve specific problem.
- A peer can be a client or a server. The client(peer1) requests for a service to a server(peer2). Similarly server(peer2) provides the services to the client(peer1).

Structure :

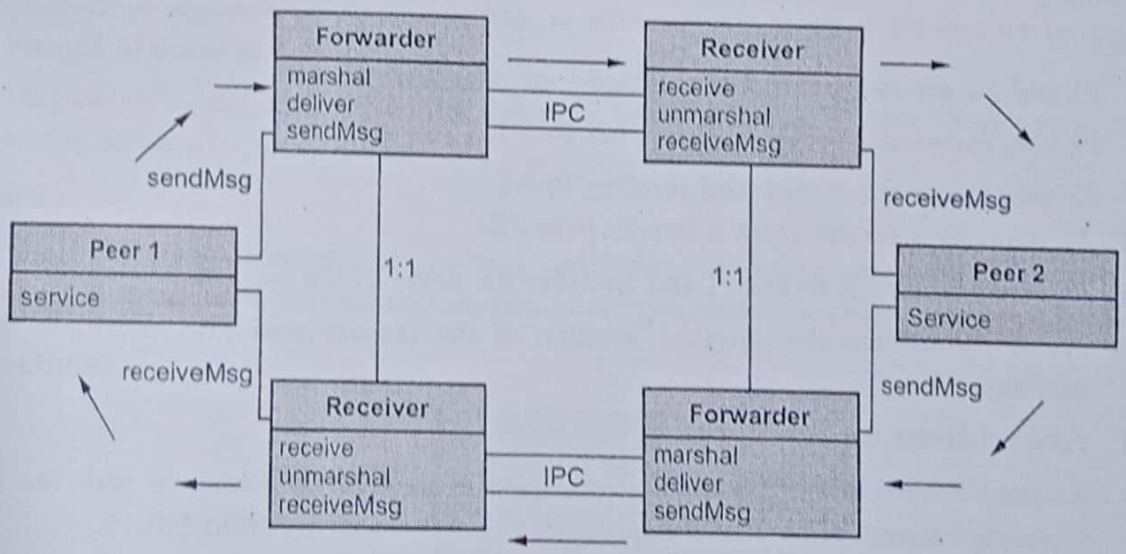


Fig. 6.6.2 Forwarder-receiver pattern

Participating classes :

- The forwarder, reciever and peers are three participating components.
- **Forwarder** components are responsible for forwarding all the messages to remote network agents without having dependencies on underlying IPC mechanisms.
- **Receiver** components are responsible for receiving the messages. At this stage receiving and unmarshalling of messages is done.
- **Peer** components are responsible for provinding application services. One peer component communicates with other peer component in collaboration with forwarder and receiver.
- The forwarder pattern and receiver pattern are the collaborators for peer class. i.e peer co-exist with the two communication patterns. The functional core of the

application will be represented be peer pattern. The communication mechanism overheads are handled by forwarder and receiver patterns.

- To send a message to remote peer2, the peer1 invokes the method **sendmsg** of its forwarder.
- It uses **marshal.sendmsg** to convert messages that IPC understands. To receive it invokes **receivemsg** method of its receiver and to unmarshal it uses **unmarshal.receivemsg**.
- Forwarder components send messages across peers. When a forwarder sends a message to a remote peer, it determines the physical location of the recipient by using its name-to-address mapping.

Dynamics :

Let,

- P1 and P2 are two peers that communicate with each other.
- P1 uses forwarder Forw1 and receiver Recv1.
- P2 uses forwarder Forw2 and receiver Recv2

1) P1 requests a service from a remote peer P2.
2) P1 sends a request to Forw1 and specifies the name of the receipant.
3) Forw1 determines the physical location of the remote peer and marshals the message.
4) Forw1 delivers the message to remote Recv2.
5) At some earlier time P2 has already requested its receiver Recv2 to wait for an incoming request. Now Recv2 receives the message arriving from Forw1.
6) Recv2 unmarshals the message and forwards it to its peer P2.
7) Meanwhile, P1 calls its Recv1 to wait for a response.
8) P2 performs the requested service and sends the result and the name of the receipent P1 to the forwarder Forw2.
9) The forwarder Forw2 marshals the result and delivers it to Recv1.
10) The receiver Recv1 receives the response from P2, unmarshals it and delivers it to peer P1.

Known Uses :

- TASC, a software development toolkit for factory automation systems, supports the implementation of Forwarder-Receiver structures within distributed applications.
- Part of the REBOOT project uses Forwarder-Receiver structures to facilitate an efficient IPC in the material flow control software for flexible manufacturing.

Consequences

1) Efficient interprocess communication
2) It supports the encapsulation

6.6.2 Client-Dispatcher-Server

Problem : A software system integrating a set of distributed servers, with servers running locally or distributed over a network.

1) The components should be to use a service **independent** of the location of the service provider.

Solution :

- Dispatcher implements a name service to allow clients to refer the servers by names instead of physical location.
- Dispatcher is responsible for establishing a communication channel between a client and a server.

Intent :

The client-dispatcher-server pattern provides inter-process communication for software systems in which the distribution of components is not known at compile time.

Structure :

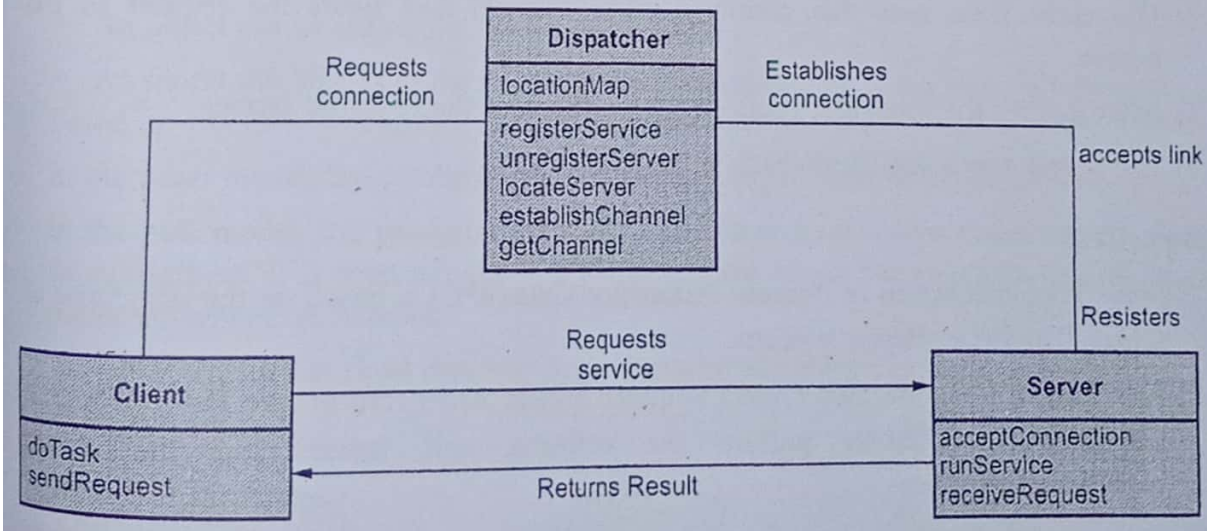


Fig. 6.6.3 Client-dispatcher-server

Participating Classes :

Dispatcher : The dispatcher class is responsible for establishing the communication channel between client and server. The dispatcher implements a **name service** that allows the clients to refer to the servers by name instead of physical locations. In this

way it helps to provide the location transperency. Clients rely on the dispatcher to locate a particular server.

Client : The task of client is to perform domain specific tasks. Before sending requests to the server the client asks the dispatcher for the communication channel. The client uses this channel to communicate with server.

Server : Server provides the set of services to the client. It is registered with the dispatcher by its name and address. A server may be located on the same machine on which the client is located or it may be located on another machine which is accessible via network.

Dynamics :

1) The server registers itself with the dispatcher component.
2) At a later time, the client asks the dispatcher for communication channel. The client wants this communication channel in order to talk to the server.
3) The dispatcher then looks up its registery for the name of the server with which client wants to communicate.
4) The dispatcher then tries to establish the connection with server. If the connection is established successfully with the server, then that communication channel is returned to the client. Otherwise, dispatcher sends the error message to the client.
5) The client then uses this communication channel and sends the request to the server.
6) After getting the incoming request, server executes the required service.
7) The server sends the result back to the client.

Known Uses :

1) Sun's implementation of Remote Procedure Calls(RPC) is based on the principles of Client-Dispatcher-Server pattern.
2) The OMG Corba(Common Object Request Broker Architecture) uses the principle of Client-Dispatcher-Server pattern for refining and instantiating the broker architectural pattern.

Consequences :

1) Exchangability of servers is possible.
2) Location and migration transperency exists.
3) It supports fault tolerence mechanism.

6.6.3 Publisher-Subscriber

Problem : A commonly arising situation is that data changes at one place and there exists many components depending on this data. For example in some GUI based application, if some internal data gets changed, then multiple views based on this data get affected.

Solution :

- To solve this problem there must be some dedicated component who can inform the changes in the data to all the dependent components. Hence, in the publisher subscriber pattern, one dedicated component takes the role of the **publisher**(also called as subject). All the components dependent on changes in the publisher are called its **subscribers**(also called as observer).
- The publisher maintains a registry of currently subscribed components. Whenever a component wants to become a subscriber, it makes use of **subscribe interface** offered by publisher.
- Whenever the publisher changes the state, it sends the notification to all its subscribers.
- The subscribers then retrieve the changed data.
- Following are the freedoms used in publisher-subscriber pattern -
  - The publisher can decide which internal state changes it will notify to its subscribers.
  - An object can be subscriber to many publishers.
  - An object can take both the roles - it can be publisher or it can be a subscriber.
- Based on publisher-subscriber pattern the **push and pull models** are derived.
- In the push model, the publisher sends all the changed data to its subscribers.
- In the pull model, the publisher only sends minimal information about the change to its subscribers. It is then subscribers' responsibility to get the required data from the publisher whenever needed.
- The push model has rigid dynamic behavior whereas the pull model is flexible.

Structure :

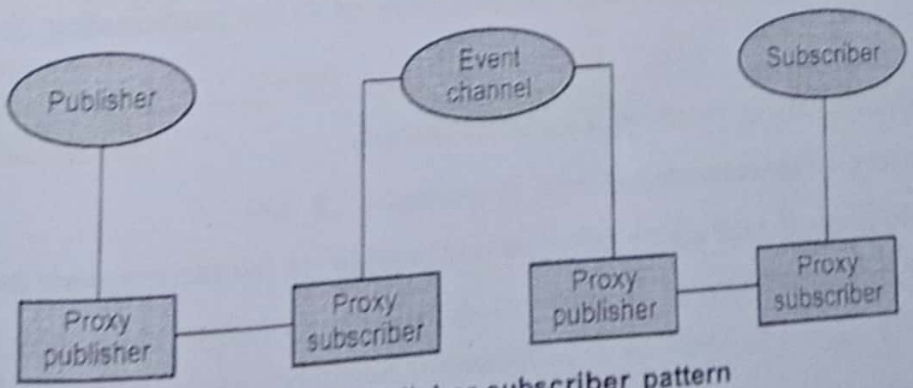


Fig. 6.6.4 Publisher-subscriber pattern

Known Uses :

Event messaging : Publisher-Subscriber pattern is widely used in delivery logistics. As we shop online more frequently for a wider variety of goods, package delivery has become commonplace. Logistics companies need to use delivery resources more efficiently. To optimize delivery, dispatching systems need up-to-date information on where their drivers are. Pub/sub-event messaging helps logistics companies do this.

Consequences :

Advantages :

1) There is a support for distributed system.
2) Extensibility

Disadvantages :

1) Less efficient
2) Complex pattern to implement

Review Question

1. *Explain any one communication pattern in detail with suitable example.*

6.7 Management Patterns

In software system there are collection of objects that provide different kind of services. In a well-structure software systems, separate manager component are often used to handle such homogeneous collection of objects.

There are two design patterns based on this approach -

1. Command processor 2. View handler

6.7.1 Command Processor

Command Processor separates the request for a service from its execution. It Manages requests as separate object. It schedules the execution of all the participating objects.

Example :

- In a text editor, there is multi-level undo mechanism.
- It allows future enhancements.
- The user-interface of text editor offers several means of interaction such as keyboard input or pop-up menus.

- The editor program has to define various **callback** procedures that are called automatically for every user interaction.

Context :

1) Applications that need flexible and extensible user interfaces.

2) Applications that provide services related to scheduling or undo operations.

Problem :

- An application needs a large set of features.
- It also needs a solution that is well-structured for mapping its interface to its internal functionality.
- Application implements pop-up menus, keyboard shortcuts, or external control of application via a scripting language.
- Forces or requirements for such application can be as follows -
  - Provide **different ways for different users** to work with an application.
  - Enhancements of the application should not change the existing code.
  - Services such as undo should be implemented consistently for all requests.

Solution :

- Use the Command Processor pattern.
- Encapsulate requests into objects.
- When user calls a function, the request is turned into a command object.
- The Command processor component takes care of all command objects. It schedule the execution, provides undo services and other additional services

Structure :

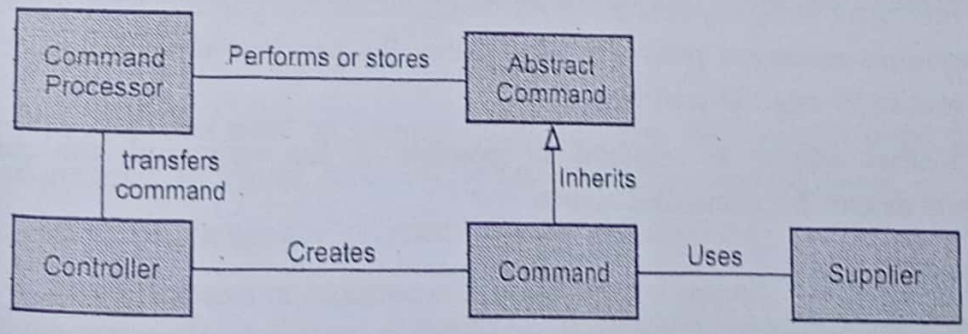


Fig. 6.7.1 Command processor

Participating Classes :

Abstract Command component defines the interface to all commad objects. It extends the interface for service of the command processor such as undo and logging.

Command encapsulates a function request. It uses suppliers to perform a request.

Controller represents the interface of the application. It accepts the command and respond accordingly. For example if it accepts the command "Paste text" then it creates corresponding command object. It transfers the command to the command processor.

Command processor performs the command execution. It also provides additional services related to command execution. The command processor maintains command objects.

Supplier components provide most of the functionality required to execute concrete commands.

When an undo mechanism is required, a supplier usually provides a means to save and restore its interal state

Dynamics : To understand the working of command processor pattern Consider an instance in which user makes an undo operation after capitalizing the text.

1) The **controller** accepts the request from the user within its event loop and creates a 'capitalize' command object.
2) The controller transfers the new command object to the **command processor** for execution.
3) The **command processor** activates the execution of the command and stores it for later undo operation.
4) The capitalize command retrieves the currently-selected text from its **supplier**, stores the text and its position in the document. The command then instructs the supplier to actually capitalize the selection.
5) After accepting an undo request, the **controller** transfers his request to the **command** processor.
6) The **command processor** invokes the undo procedure of the most recent command.
7) The capitalize command resets the **supplier** to the previous state, by replacing the saved text in its original position.
8) If no further activity is required or possible of the command, the command processor deletes the command object.

Known Uses :

MacApp use the pattern to do undo.

Consequences :

Advantages :

1) There is flexibility in the way requests are activated.
2) The testability in application level.
3) Concurrency can be achieved.

Disadvantages :

1) Complexity exists in acquiring the command parameters.
2) Less efficient.

6.7.2 View Handler

This pattern helps to manage all the views that a software system provides.

Intent :

For a GUI based applications or a multimedia systems one system has different views. The view handler help to manage all views that a software system provides. The view handler also allows the clients to open, manipulate or dispose the views.

Example :

A multi-document editors allows the user to work on several documents simultaneously.

Each document is displayed in a separate window. Changes in one window may affect the other window.

Context :

A software system that provides multiple views for application specific data.

Problem :

- The applications that need to support multiple views are required more often. Such applications need additional functionality for managing different views.
- All the views in such application need to be co-ordinated.
- Following are the forces or requirements of such applications
  - Individual view implementation should not be dependent on others.
  - Managing multiple views must be easy for client components.
  - View implementation can be added during the lifetime of the system.

Solution :

A view handler component manages all the views that a software system provides.

Structure :

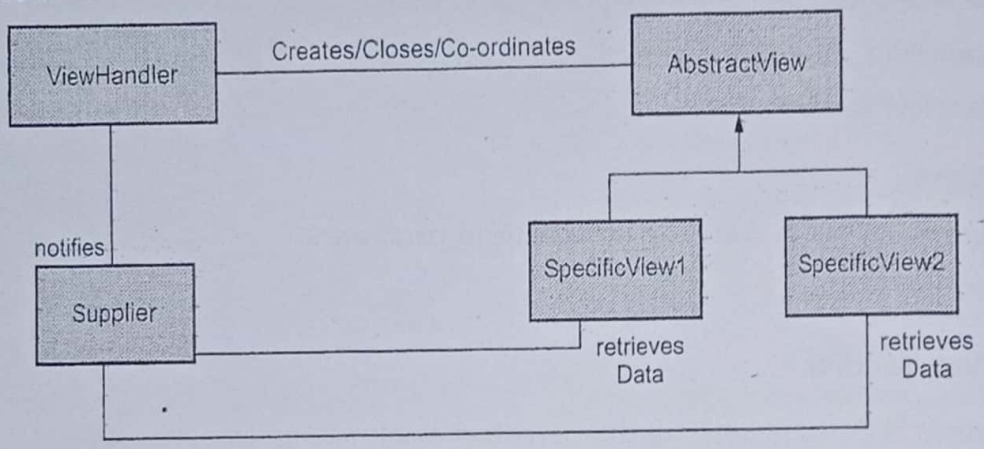


Fig. 6.7.2 View handler

Participating Classes :

View Handler plays a central role in this pattern. It handles opening and closing of the views. The client specifies the view he/she wants. The important task of view handler is to offer view management services.

AbstractView component defines an interface that is common to all views. The **viewHandler** uses the this interface for creating, coordinating and closing views.

SpecificView1 and **SpecificView2** are the components that are derived from **AbstractView** interface. These components have their own **display** functions. This display function is called when opening or updating the view. These components retrieve data from **Supplier** component.

Supplier components provide the data that is displayed by view components. They notify dependent components about change to their internal state.

Dynamics : There are two scenarioes -

Scenario 1 : ViewHandler creates a new view

1) A client invokes the view handler to open a particular view.
2) The view handler instantiates and initializes the desired view.
3) The view registers with the change-propagation mechanism of its supplier.
4) The view handler adds the new view to its internal list of open views.
5) The view handler then calls the view to display.
6) The view open a new window, retrieves data from its supplier, prepares this data for display, and presents it to the user.

Scenario 2 : ViewHandler organizes the tiling of views

1) The client uses a command to tile all open windows. This request is sent to view handler.
2) The view handler calculates the size and position of all the opened views and calls its **resize** and **move** procedures.
3) Each view changes its **position** and **size** and sets corresponding clipping area. It then refreshes the image and it is then displayed to the user.
4) View obtains the data from the supplier whenever required.

Known Uses :

1) Microsoft Word makes use of ViewHandler pattern for cloning, splitting and tiling windows.
2) Macintosh Window Manager is a part of Macintosh toolbox that can be compared with view handler component.

Consequences :

Advantages :

1) It helps in **uniform handling of views**
2) **Extensibility and changeability** of views is possible.
3) There is application Specific **view coordination.**

Disadvantages :

1) It has **restricted applicability** if different views are needed.
2) The view handler component introduces a level of indirection between the clients that want to create views. This results in **loss of performance.**

Review Questions

1. *Explain the command processor pattern in detail*
2. *What is view handler pattern ? Explain it in detail.*

6.8 Idioms

6.8.1 Introduction

- Idioms are low-level patterns which are specific to a programming language.
- Idioms describe how to solve implementation specific problem using some programming language.

- Idioms can directly address the concrete implementation of a specific design pattern.
- Idioms represent the competent use of programming language features.
- The idioms ease communication among developers and speed up software development and maintainance.

6.8.2 What can Idioms Provide ?

- Single idiom might help to solve a recurring problem with a programming language being used. For example - it might help to perform memory management, object creation, use of library component and so on.
- Each idiom has a unique name and therefore they provide a vehicle for communication among software developers.
- Idioms are less portable among the programming languages.

6.8.3 Idioms and Style

- Programmers working in a team, must agree on a single coding style for programs.
- For example -

```
void strcpy1(char *d,const char *s)
{
        while(*d++ = *s++)
}
void strcpy2(char d[], const char s[])
{
        int i;
        for(i=0;s[i]!='\0';i++)
                d[i] = s[i];
                d[i] = '\0';
}
```

Above code is for string copy operation. Both the implementations have different style. A programmer can select any desired style as per his/her requirement. But if a mixture of both the styles is used then it is much harder to understand the code. Managinging this mixed style of programming is very difficult.

- It is always recommended to follow a consistent style throughout the program. This style of programming might be developed by teams.
- Style guides that contain collected idioms work better. They not only give the rules, but also provide insight into the problems solved by a rule. They name the idioms and thus allow them to be communicated.

- Different sets of idioms may be appropriate for different domains. example, you can write C++ programs in an object-oriented style with inheritance and dynamic binding. In real time system dynamic binding is not used which is required.
- A coherent set of idioms leads to a consistent style in developers program.

6.8.4 Where to Find Idioms ?

- A good collection of Smalltalk programming wisdom in the idioms presented in Kent Beck's columns in the **Smalltalk Report**. Beck defines a programming style with his coding patterns that is consistent with the Smalltalk class library, so you can treat this pattern collection as a Smalltalk style guide.
- Idioms that form several different coding styles in C++ can be found for example in Coplien's Advanced C++ Barton and Neck man's Scientific and Engineering C++ and Meyers' Effective C++ .

6.8.5 Counted Pointer Example

The counted pointer idiom is used for memory management of dynamically allocated shared objects in C++.

Context : Memory management of dynamically allocated instances of a class.

Problem :

In C++, the objects can be passed as parameters to the functions. Normally the pointers or references to the objects are passed as parameters. However, this mechanism there are chances that the references that are passed not no longer valid or do not exists at all.

Following are **the forces or requirements** that arise during the passing of parameters which are actually the pointers to the dynamically allocated objects -

- Passing object by value is inappropriate for a class.
- There might be the situation in which several clients may want to access the same object of a class.
- There may be a situation of dangling references. The dangling reference is basically the object that is been deleted and still programmer try to access it.
- If shared object is not required further and there is a requirement to destroy it and release the memory acquired by this object.

Solution :

The counted pointer idiom is used to perform memory management of shared objects by introducing **the reference counting mechanism.**

- There are two classes used in this mechanism. One object, called the **Handle**, manages the interface, while another object called the **Body** provides the application logic.
- The **Body** is the object that will be referenced and shared. A **reference count** in the **Body** keeps track of number of pointersto it by objects.
- Handle is a only class in the system that is allowed to have a reference directly to the body. In other words, all the references to the body are made through the handle.
- Accesses to the body through the handle manipulate the reference counter. It increments the reference counter when a new class points to the handle and decrements the reference counter when a reference to the handle is eliminated.
- The handle objects are passed by value throughout the system, which causes them to be allocated and destroyed automatically.

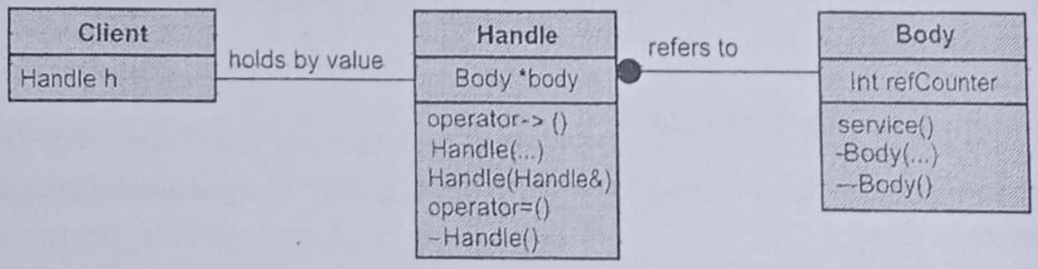


Fig. 6.8.1 Representation of handle body idiom

Implementation : Following are the steps that are pefromed for implementing the counted pointer example -

1) Make a constructor and destructor of the **Body** class. It is denoted as Body() and ~Body() respectively.
2) Make the **Handle** class a friend to the **Body** class so that the **Handle** class can access the private members of the **Body** class.
3) Extend the Body class with a **reference counter**. (Denoted by a variable refCounter).
4) Add a data member to the **handle** class as a pointer that points to the object of the Body class.
5) Implement the Handle class' copy constructor and its assignment operator by copying the Body object pointer and incrementing the reference counter of the shared Body object. The copy constructor is **Handle(Handle&)**
6) Also implement the destructor of Handle class. It is used to decrement the reference counter and delete the Body object when reference counter reached to zero.
7) By overloading **operator -> ( )** in the **Handle** class, its objects can be used syntactically as if they were pointers to Body objects.

8) Extend the Handle class with one or more constructors. These constructors are used to create Body instances to which the Handle refers. Each of these constructors initializes the reference counter to one.

Variants :

- Reference counting idiom
- Counted Body

Review Questions

1. *What is idiom ? What can an idiom provide ?*
2. *Explain the counted pointer idiom in detail.*

❑❑❑