Unit III

CHAPTER 3

Advanced State Modeling and Interaction Modeling

University Prescribed 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.

Specific Instructional Objectives :

At the end of this lesson the student will be able to :

- Identify nested states.
- To draw state diagram and nested state diagram.
- Explain signals and signal generalization.
- Describe what a concurrency is?
- Differentiate amongst class model and state model.
- Describe use case model.
- Identify actors and their responsibilities, use cases.
- To draw use case diagram.
- Design and explain use case relationships.
- Design and explain sequence model.
- Explain procedural sequence model.
- Explain activity model.
- Describe special constructs for activity model.
- Design and explain a sequence diagram.
- Design and explain a activity diagram.

3.1 Nested States and Nested State Diagram

GQ. What are composite states? Describe categories of composite states.

- In order to model the behavior of the most complex software system, UML provides a strategy known as nested states.
- Nested states are nothing but the states which are nested inside other states in the scenario. For instance, when a washing machine is in the active state, there can be a nested state termed activate state. So in this scenario, we can say that machine is in active state as well as in activate state.
- ***Nested states*** are also known as ***composite states*** or ***substates***.
- The basic difference between a simple state and a composite state is that, there is no substructure in case of a simple state but, a composite state might hold sequential concurrent substates.
- The nested state can be graphically shown by the notation same as that of the normal state but there is a special compartment inside its body which is optional and is used for representing the software system substructure i.e., nested state machine as shown in the Fig. 3.1.1.

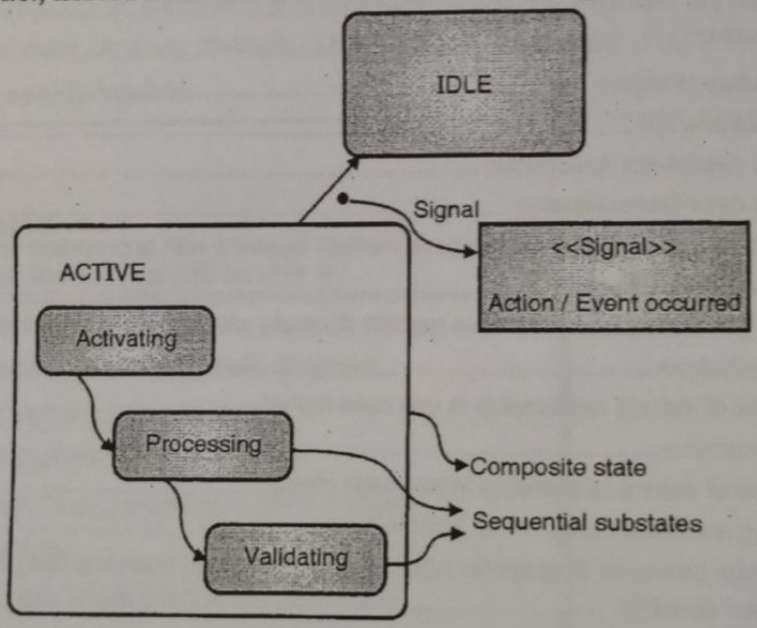


Fig. 3.1.1 : Nested States

- Nested states are basically categorized into two types :

1. Sequential substates.
2. Concurrent substates.

3.1.1 Sequential Substates

GQ. Explain sequential substate with the help of suitable example.

- Sequential substates are the type of nested states that collectively defines certain scenario and depicts a proper flow of activities, actions or operations involved in the scenario. For better understanding of sequential substates behavior, let us consider an example of ATM machine.

- As we have already discussed, any software system normally executes in four basic states : Idle state, Activating state, Active state and End state.
- In the scenario of an ATM machine, an idle state of an ATM machine system is that, the system is waiting for communication or interaction from end-users / customers. When customer presses the start transaction button on the ATM machine; user interface of a machine is in activating state and hence it is now ready for further transaction related activities.
- Active state comprises of all of the activities involved within a particular transaction that is handled by the customer and this active state is then followed by end state showing end of that particular transaction.
- In this scenario, ATM machine first of all validates the customer and then access will be given to the authorized customer in order to perform transaction. After successful login, customer might perform transaction which includes sub-activities like selecting the transaction, processing the transaction and then printing the receipt of the transaction.
- After printing the receipt of the current transaction, an ATM machine again returns back to the idle state. This scenario is shown in the Fig. 3.1.2 for better understanding of nested state diagram with sequential substates.

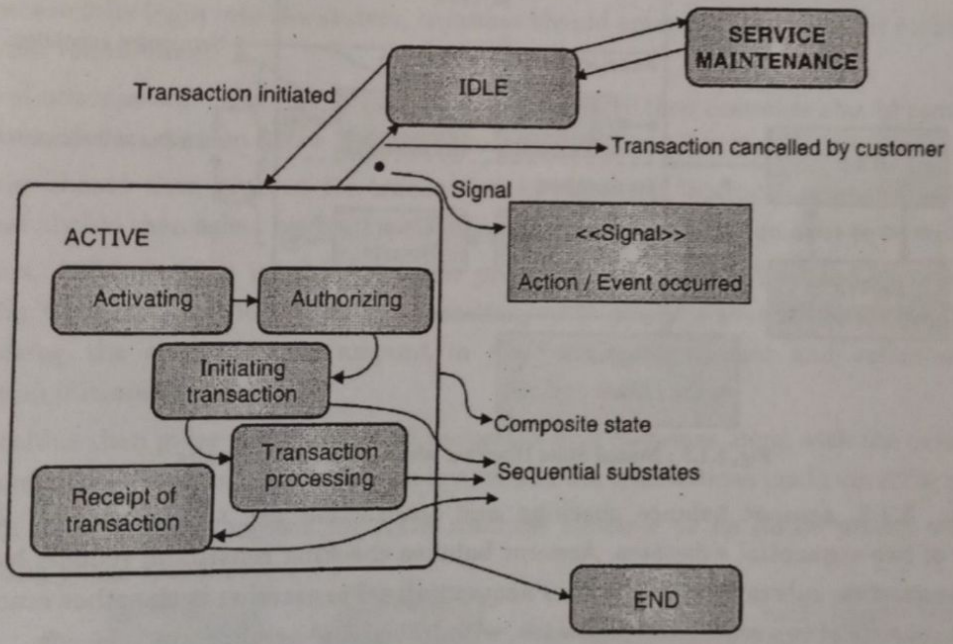


Fig. 3.1.2 : Nested State Diagram with sequential substates

- In the Fig. 3.1.2, substates like ***Initiating Transaction***, ***Transaction Processing*** and ***Receipt of Transaction*** are known as ***sequential substates***. All of these substates collectively represents the Active state of the ATM machine system and hence tells the detailed flow of activities involved in the execution of the system when the system is in Active state.
- A sequential substate might have at the most one initial state and one final state.

3.1.2 Concurrent Substates

- If we have to show two or more state machines that executes concurrently, then we can make use concurrent substates in order to represent the concurrency amongst two or more state machines.
- Use of sequential substates is too common in state modeling, but in extraordinary cases, we need specify concurrent substates to show parallel flow of execution in two nested state machines.
- A concurrent substate does not have an initial and final state. But, the sequential substates tha comprise concurrent substates may have an initial and final state.
- Execution of two or more concurrent substates continues in parallel. Each nested substate attains its final state ultimately. If one of the substate achieves its final state before the other substate then control of that particular concurrent substate pauses at its final state only and do not continues further As soon as second concurrent substate reaches at its final state, both of the substates then combine into one flow.
- Fig. 3.1.3 shows concurrent substates.

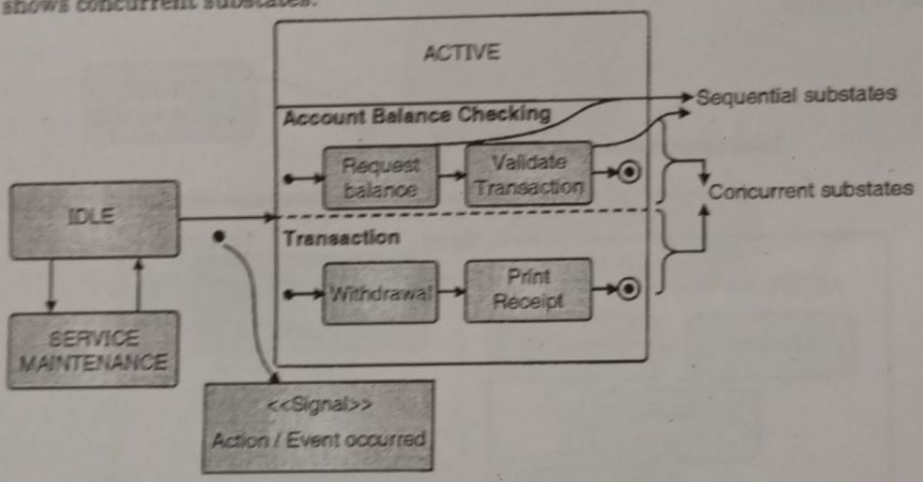


Fig. 3.1.3 : Nested State Diagram with concurrent substates

- In the Fig. 3.1.3, account balance checking and transaction are two concurrent substates which comprises of two sequential substates. Account balance checking consists of request balance state and validate transaction substates that executes sequentially. Transaction is the other concurrent substate that again consists of two sequential substates, withdrawal and print receipt.

3.1.3 Signal Generalization

- In advanced state model, we can arrange signals in generalization.
- The signal generalization mechanism offers use of different levels of abstraction in the state model.
- Each and every signal in the state model can be viewed as a leaf node or child node in the generalization hierarchy.

3.1.4 Concurrency

- The advanced state model indirectly supports and permits concurrency between several objects involved within the software system scenario.
- For achieving concurrency in state model, each and every object should share their features, operations and constraints with other objects in the system.
- Interdependency between several objects refers to concurrency in state model.

3.1.5 State Diagram Case Study : ATM Machine

GQ. Draw a state diagram for ATM system.

- As an example, we are going to study state diagram for an ATM Machine.
- At the outset, welcome message is displayed on the ATM Machine screen and ATM is in an idle state at this stage. When customer presses ENTER button on the screen, ATM requests for entering a card into the given slot.
- At this stage, ATM navigates from idle state to waiting state, since after insertion of a card, ATM asks customer for entering valid PIN on the screen.
- After successfully login into the system, customer should enter his/her choice for making transaction as per his/her requirement.
- In case of unsuccessful login (i.e., if entered PIN is invalid); then customer should remove card from the ATM slot and should again follow the same procedure for login into the system.
- Customer should then proceed for transaction by following instructions given on the ATM screen. Customer should then select his/her type of account and should enter amount to be withdrawn.
- After this, ATM machine is responsible for processing transaction initiated by a customer within a particular time instance. Within this transaction, ATM machine contacts to the bank's central server for checking the exact balance amount in the customer's account and validates that particular transaction initiated by customer.
- ATM machine then gives out the amount requested by a customer along with the printed receipt of the transaction. These are the basic activities involved in the transactions made via ATM machine.
- Fig. 3.1.4 depicts state diagram for ATM machine example at its initial phase, working phase and terminating phase respectively.

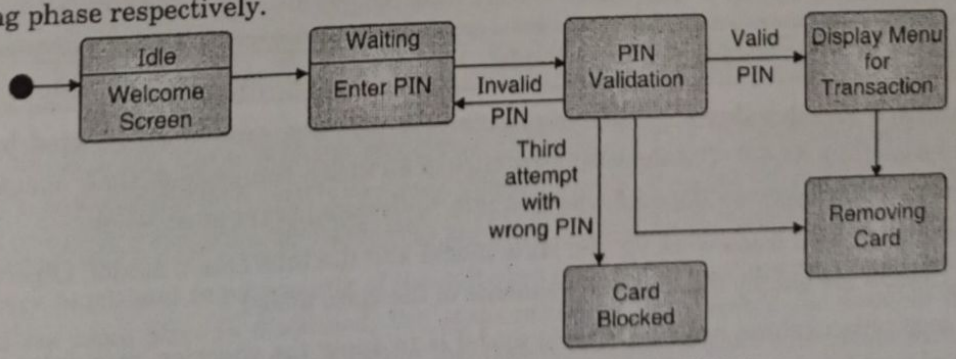


Fig. 3.1.4 : State transitions in ATM Machine at initial phase

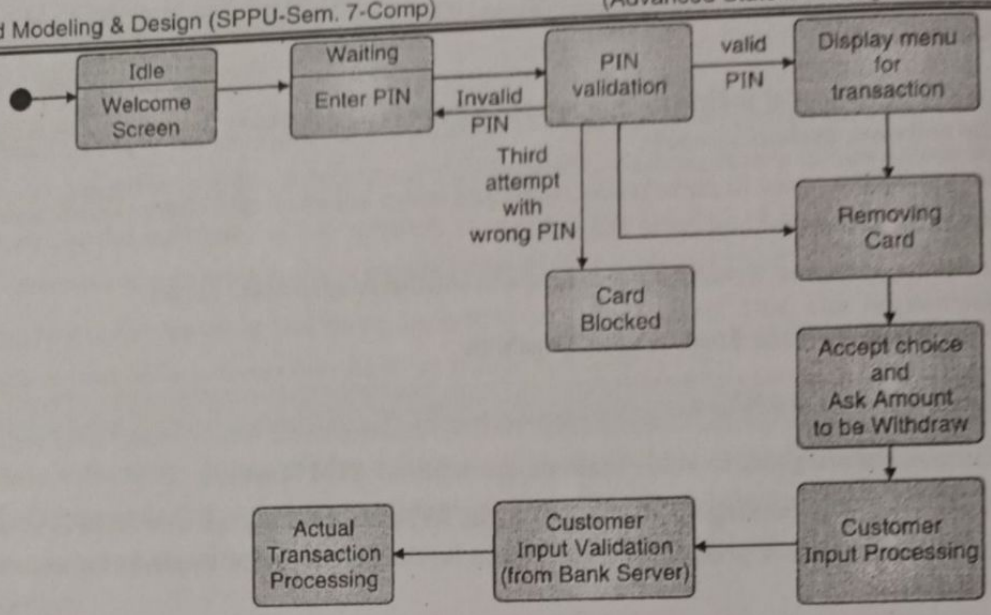


Fig. 3.1.5 : State transitions in ATM Machine at working phase

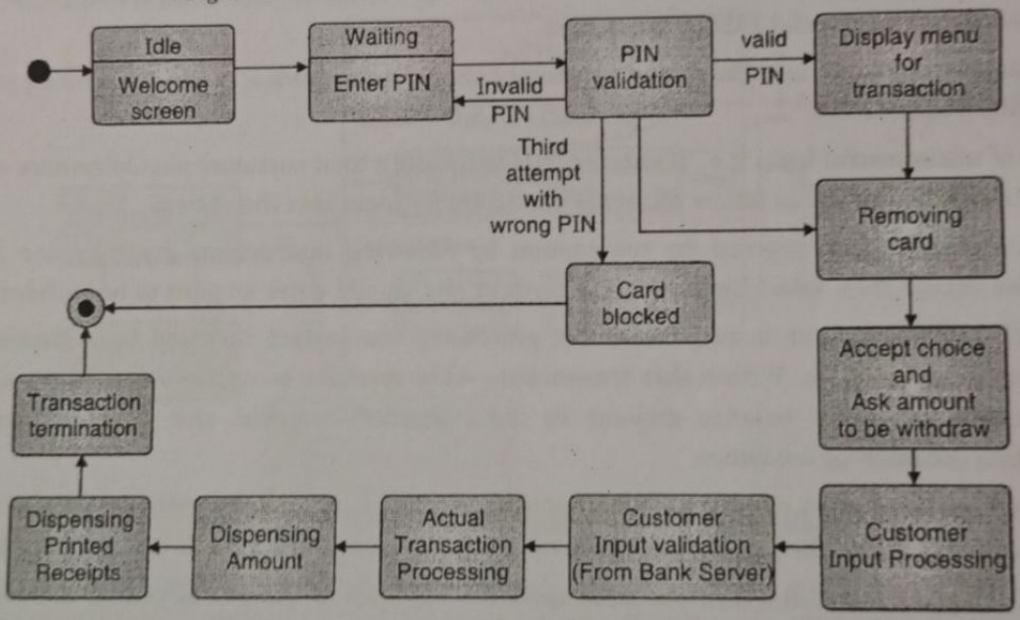


Fig. 3.1.6 : State transitions in ATM Machine at final phase (Overall State Diagram for ATM Machine)

3.1.6 Relation of Class Model and State Model

- The organization and detailed structure of objects involved in a system is depicted by means of the model known as class model. The detailed structure of an object comprise of their characteristics, their interactions to other objects in the scenario, and their individual tasks or services.
- The class model offers a framework for the state model and the interaction model. Objects are instances of classes. Objects are simply the building elements of the class model.
- The main motivation behind building a class model is to apply the theories, principles and conceptions from the real world that are significant and vital to the proposed software system.

- The class model must comprise of the things familiar to end users of the proposed system. Basically, the class model is articulated by means of class diagram.
- Objects from the class model are described with respect to sequencing and timing of operations in case of the state model. The state model describes those aspects of objects concerned with time and the sequencing of operations.
- The state model simply organizes and systemizes the states and the event involved in the proposed software system. The state model is dedicated for detailed description of sequence of tasks and services involved within the system operations and do not deals with the contents like details of operations and services, how different services and operations are executed during implementation and execution of the proposed system, etc. The state model is basically articulated by means of a state diagram.
- The state diagram represents the sequences of states and events that are tolerable in a proposed system for individual class of objects. The state diagram talks about the other models too.
- Tasks or jobs of a particular object from the class model are equivalent to the events and activities from the state diagram. References between state diagrams become interactions in the interaction model.

Guidelines for designing a State Diagram

- A state diagram should be simple enough and all the states, events and transitions involved in the software system scenario should be named from the vocabulary of the system for better understanding of a software system archetype.
- It should not contain any unnecessary transactions, events or states.
- Active classes and concurrent substates should be used in order to represent sequential flow and concurrency amongst the active objects in the software subsystem archetype.
- When something is happening into the system, a state or an event should be named with the help of the situation, condition or a particular point of time. Each state should be named with unique names for unique identification of the state in the system archetype.
- Actions and events involved in the scenario should be categorized in proper manner. Actions are the effects of the state transitions and events are the causes of the state transitions.
- Every state should have exit state, i.e., it should be conceivable to exit from each state.
- Actions and conditions should be used in proper situations and it is not at all compulsory to declare actions and conditions in the state archetype.

3.2 Interaction Modeling : Use Case Model

GQ. What do you mean by use case model? Describe in brief.

GQ. Explain the following terms with respect to use case model :
(a) Use case (b) Actor (c) System boundary

- The elementary functional requirements of the software system are explained with the help of use case modeling. As we have already discussed, the state modeling is dedicated for detailed description of sequence of tasks and services involved within the system operations and do not deals with the contents

like details of operations and services, how different services and operations are executed during implementation and execution of the proposed system, etc.

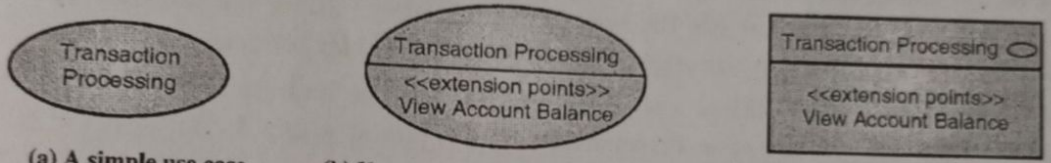
- But, initially software system designers should have a basic idea about the input given to the software system and output generated from the software system. So, use case models are used in primary phase of the software system design in order to define input to and output from the proposed software system.
- The use case model depicts functional requirement of the proposed system by means of the use cases and the actors.
- Use case model supports in the design of the software system by presenting intended behavior of the proposed system without any description about the implementation of the system.
- Let us discuss the three basic components of the use case model : use cases, actors and relationships in subsequent sections.

3.2.1 Use Cases

GQ. Write a short note on : Use case.
GQ. Draw and explain use case scenario in brief. Give suitable example.
GQ. List and explain the elements of use case scenario.

- In UML, the system requirements and functionality of the system are depicted with the help of use cases.
- Use cases are meant for specification of the interaction between the system itself and end users of the system which are termed as actors in UML.
- Basically, use case offers a detailed description of how the system is used.
- The set of activities and events in some proper sequence specifying the interaction amongst a system and its end users is known as a scenario.
- We might have to handle several scenarios in the execution of the system depending on the situations or scenarios involved within the proper execution of the system.
- For instance, in case of an ATM machine, access will be given to authorize customer only by authenticating the particular customer through his/her card and PIN. If login is successful, then customer will be in position to perform the transaction. But, if the login is failed, no access will be given to the customer and customer will not be able to perform the transaction through an ATM machine. So, both of these scenarios are different which are the things that might happen.
- All of the scenarios mentioned in the above example are different but are equivalent since the customer has the same goal in all of the scenarios i.e., to perform the transaction. This goal is only the main component behind definition of use cases.
- A use case is defined as a set of scenarios that collectively work to achieve a common user goal. It outlines a sequence of interactions amongst one or more actors and the system itself.
- In the above example, transaction processing can be one of the use case that strives for the successful transaction processing that is the goal of the customer.
- Each use case comes with its primary actor who is responsible for certain tasks involved in the scenario. More details of the actor are discussed in subsequent section of this chapter. Each use case should clearly mention the exact interaction between the actors and the system itself.
- Furthermore, use case guides us about what the system exactly does in response to the actor's activity and it is not devoted for detailing how system does it.

- At all times, a use case starts with input from a respective actor. Thus, an actor is responsible for giving input the system and the system is dedicated for giving response to the actor.
- A simple use case encompasses a single interaction between the actor and the system. But, a single use case can handle set of interactions between a particular actor and the system. More than one actor can be a part of the complex use cases.
- Graphically, a use case is shown with the help of an oval shape containing the use case name inside its body. Each use case should be named uniquely and use case name should describe the desired functionality of that particular use case.
- We can describe the use case in more detail by making partition inside the oval shape and details of the respective use case can be given in the second partition.
- The contents of the second partition are called as extension points. Use case can be depicted as a classifier also. Fig. 3.2.1 shows a simple use case notations in UML : (a) A simple use case (b) Use case with extension points and (c) Use case as a classifier.



(a) A simple use case (b) Use case with extension points (c) Use case as a classifier

Fig. 3.2.1 : Use Case notations in UML

- More fundamentally, any use case explanation consists of five entities:
  - The name of the use case.
  - The name of the actor.
  - A brief explanation of the use case.
  - A brief explanation of sequence of activities and events involved in a use case.
  - Preconditions and postconditions necessary for successful execution of the use case.
- More detailing of a particular use case can be done with the help of following use case scenario template :

Use case ID :	
Use case name :	
Use case description :	
Actors :	
Preconditions :	
Main sequence description :	
Alternative sequence description :	
Nonfunctional requirements :	
Postconditions :	
Frequency of use :	

Fig. 3.2.2 : Use Case Scenario Template

- As shown in the Fig. 3.2.2, software designers should describe each and every use case involved in the scenario of the system archetype during use case modeling.
  - ***Use case ID*** : Use case ID acts as an absolute unique identifier for unique identification of particular use case in the software system scenario. Even though use case name is used for unique identification of a particular use case, it may change over time. So, it is moderately necessary to have a unique identifier for the individual use case.
  - ***Use case name*** : A unique name should be given to each use case and it should specify the functionality of the use case. Use case name should be a verb since use cases specifies behavior of the system. Use case name should be descriptive but short.
  - ***Use case description*** : Use case should be described in brief in this section. This section should summarize the objectives of the use case.
  - ***Actors*** : actors can be classified as primary actors and secondary actors from the use case perspective. Primary actors are the main actors and they initiate the use case while secondary actors cooperates with use case after initiation of that particular use case.
  - ***Preconditions*** : Preconditions are simply nothing but the limitations that should be attained before triggering the use case. Preconditions do not permit actors for initiation of the use case unless and until all preconditions are satisfied.
  - ***Main sequence description*** : Actual flow of events and activities is described in this section.
  - ***Alternative sequence description*** : The use case may have alternative flow of execution along with the main flow. The alternative flow do not return to the main flow repeatedly. Possible alternatives to the main flow gives us the alternative flow for the execution of the use case.
  - ***Nonfunctional requirements*** : Security requirements and Performance requirements can be declared in this section for better understanding of the use case scenario.
  - ***Postconditions*** : Postconditions are the constraints that should be achieved after successful execution of a particular use case.
  - ***Frequency of use*** : As its name suggests, this section describes the frequency of use of a particular use case during single successful execution of the software system.
- For better understanding of the use case scenario template, refer Fig. 3.2.3 showing the use case scenario template for login into the ATM machine system for transaction.

NOTES

Use case ID:	01
Use case name:	Login into the ATM Machine.
Use case description:	After successful account opening within the system, the user will be provided with the unique account PIN with the help of which, user can get entry into the system for making transactions through ATM machine.
Actors:	Customer
Preconditions:	Customer should open an account in the bank and should avail the facility of using ATM. From ATM machines' perspective, ATM machine should have sufficient free memory available in order to launch the task.
Main sequence description:	Click the ENTRY button on the ATM machine screen. Enter PIN number for getting logged in into the system. After authentication and validation from server side, an authorized access will be given to the customer.
Alternative sequence description:	NIL
Nonfunctional requirements:	Unique Account PIN should be provided to the customer which will be used by the customer while performing transaction through the ATM machine.
Postconditions:	Authorized access to the system will be provided to the customer after authentication and validation from the server side.
Frequency of use:	01

Fig. 3.2.3 : Table for use Case Scenario Template for "Login into the ATM Machine"

3.2.2 Actors

- An actor is the external user of the system who is responsible for communication and coordination with the software system.
- It is not always necessary to have a human being as an actor within the use case scenario. We may have any other element or an external system as an actor.
- An actor have different representations in UML as shown in Fig. 3.2.4.

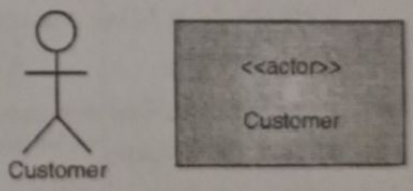


Fig. 3.2.4 : Actor notation in UML

- It is first of all necessary to recognize the role of an actor for better understanding of an actor.
- Actors are always ***external*** to the system.
- For identification of actors of a particular system, following questions should be taken into consideration :
  - Who are the end users of the system?
  - Who are the installers of the system?
  - Who provides information to the system?

- Is there any other cooperating or interacting system available in the scenario?
- Who maintains the system?

- From the business point of view, each actor should be named uniquely.
- Always, there is direct communication and coordination amongst actors and the system.
- We may have time as an actor in case we have to model the things that happen at a particular point of time in the system scenario.

3.2.3 System Boundary (Subject)

- It is essential to define boundaries of the system before moving towards the implementation and execution of the system. That is, software designers should define the things which are external to the system and the things which are the part of the system.
- So, the things which are the part of the system come inside the system boundary which is also known as a subject while the things which are external to the system are located outside the boundary of the system.
- The system boundary is graphically represented by means of a rectangular box, labeled with the name of the system. Since, actors are external to the system, they are drawn outside the system boundary. Use cases are drawn inside the system boundary.

3.2.4 Use case Relationships

- The possible relationships amongst actors and use cases in the use case diagram are as follows :
  - ***Use case/Actor Association*** **:** This relationship specifies that, the actor initiates the use case.
  - ***Use case Generalization*** **:** It indicates the generalization relationship between a specific use case and a general use case.
  - ***Actor Generalization*** **:** It indicates the generalization relationship between a specific actor and a general actor.
  - ***<<include>>*** **:** One use case can include its behavior from the other use case with the help of this relationship.
  - ***<<extend>>*** **:** One use case can extend its behavior with one or more other use cases with the help of this relationship.
- Let us have a brief discussion on use case relationships.

3.2.4.1 Use case/Actor Association

- Use case/Actor Association indicates that, the actor initiates a particular use case.
- Normally, one actor can be associated with one or more use cases.
- It actually depicts that, a specific output or result is supplied to the actor.
- An association between a use case and an actor is graphically shown as a solid line between them.

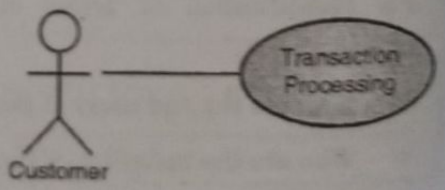


Fig. 3.2.5 : Use case/Actor Association

3.2.4.2 Use case Generalization

GQ. What is use case generalization in use case model? Illustrate with appropriate example.

- Use case generalization indicates the generalization relationship between a specific use case and a general use case.
- It should be used only in the simplification of the use case model.
- A general use case can inherit properties from their individual parent use case as well as change (override) properties.
- A general use case is simply a child use case and it automatically inherits all of the characteristics of its parent use case.
- Fig. 3.2.6 shows the UML notation for use case generalization.

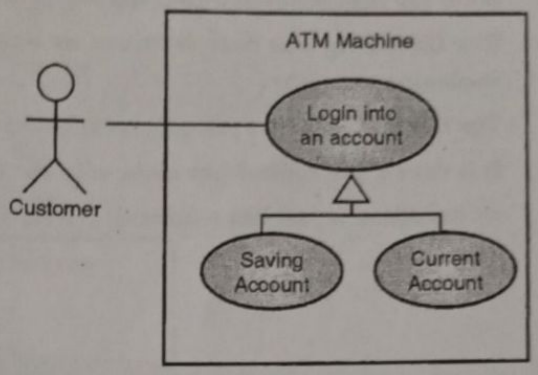


Fig. 3.2.6 : Use case Generalization

3.2.4.3 Actor Generalization

GQ. What is actor generalization in use case model? Illustrate with appropriate example.

- Actor generalization indicates the generalization relationship between a specific actor and a general actor.
- Just like use case generalization, this relationship should be used in the simplification of the use case model.
- Refer Fig. 3.2.7 for UML notation of actor generalization.

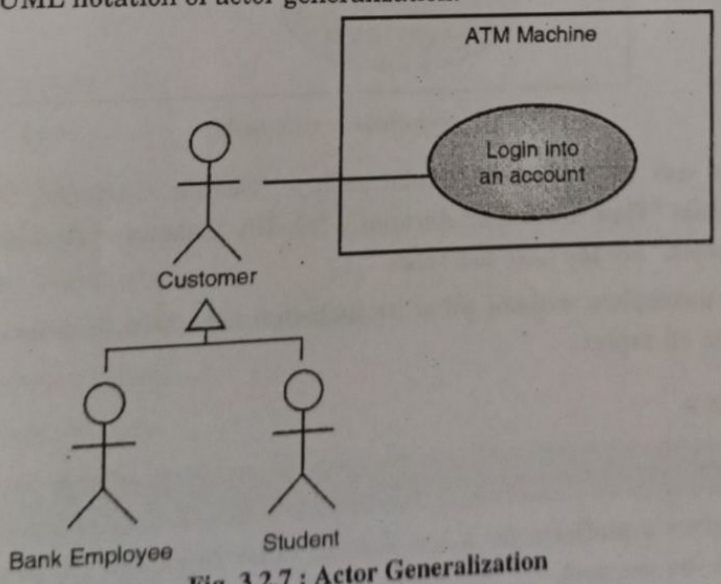


Fig. 3.2.7 : Actor Generalization

3.2.4.4 <<Include>>

GQ. Explain the use of include relationship in use case model.

- Include relationship gives us the facility to include the behavior of one use case into the flow of the other use case involved within the scenario.
- The ***including use case*** is known as ***base use case*** while ***included use case*** is considered as the ***inclusion use case***.
- The inclusion use case provides its behavior to the base use case.
- It is drawn as a dashed line along with the word <<include>> on the top of the line.
- << include>> is just like a function call having similar semantics and syntax as that of the function call

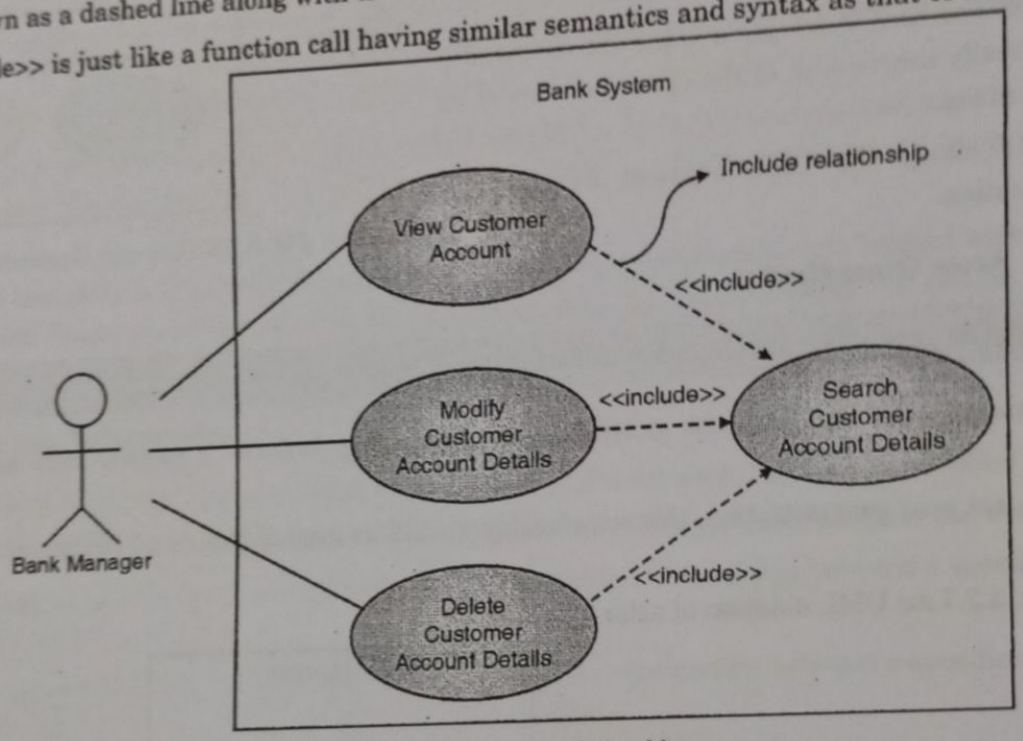


Fig. 3.2.8 : <<include>> relationship

- Fig. 3.2.8 represents use of <<include>> relationship. "Search Customer Account Details" is an inclusion use case while "View Customer Account", "Modify Customer Account Details" and "Delete Customer Account Details" are the base use cases.
- The base use case is incomplete without all of its inclusion use cases. But inclusion use cases may or may not be complete in all aspect.

3.2.4.5 <<extend>>

GQ. Explain the use of extend relationship in use case model.

- Extend relationship gives a platform for a use case in order to extend its behavior with one or more other use cases within the scenario.
- New behavior can be inserted into the existing use case with the help of <<extend>> relationship.
- The <<extend>> relationship should specify one or more extension points in the base use case.

- If certain specified conditions are satisfied, we can make use of this relationship to add some additional functionality in the base use cases.
- Fig. 3.2.9 shows <<extend>> relationship between "Open Bank Account" and "Add Joint Account Holder" use cases.

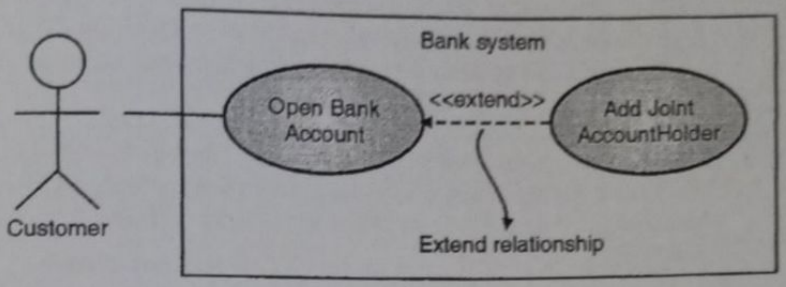


Fig. 3.2.9 : <<extend>> relationship

3.2.5 Use Case Diagram Case Study : ATM Machine

GQ. Draw and explain use case diagram for ATM System. Explain at least two use cases with the help of use case scenario.

- For ATM machine, following are the necessary actors and their respective use cases.
- Actors involved in the ATM scenario are :
  - Customer
  - Administrator (Third Party)
  - Bank
- Use cases involved in the ATM scenario are :
  - Login
  - Transaction
    - Balance Enquiry
    - Transfer Funds
    - Deposit Funds
    - Withdraw Cash
    - ATM Help (<<extend>>)
    - Customer Authentication (<<include>>)
    - Invalid PIN (<<include>>)
  - ATM Machine Maintenance
    - Upgrades (<<include>>)
    - Reporting (<<include>>)
    - Shut Down (<<include>>)
    - Diagnostics (<<include>>)
  - ATM Machine Repair
    - Diagnostics (<<include>>)

- Overall use case scenarios are described below for better understanding :

Use case ID:	01
Use case name:	Login.
Use case description:	After successful account opening within the system, the user will be provided with the unique account PIN with the help of which, user can get entry into the system for making transactions through ATM machine.
Actors:	Customer
Preconditions:	Customer should open an account in the bank and should avail the facility of using ATM. From ATM machines perspective, ATM machine should have sufficient free memory available in order to launch the task.
Main sequence description:	Click the ENTRY button on the ATM machine screen. Enter PIN number for getting logged in into the system. After authentication and validation from server side, an authorized access will be given to the customer.
Alternative sequence description:	NIL
Nonfunctional requirements:	Unique Account PIN should be provided to the customer which will be used by the customer while performing transaction through the ATM machine.
Postconditions:	Authorized access to the system will be provided to the customer after authentication and validation from the server side.
Frequency of use:	01

Fig. 3.2.10 : Use case Scenario – Login

Use case ID:	02
Use case name:	Transaction.
Use case description:	After successful login into the system, the user will be provided with the menu displayed on the ATM Machine screen for making transactions through ATM machine.
Actors:	Customer
Preconditions:	Customer should enter valid PIN for getting authorized access to the system.
Main sequence description:	Customer should select correct choice from the available options in the transaction menu. Furthermore, customer should fill correct necessary details related to transaction.
Alternative sequence description:	NIL
Postconditions:	Authorized access to the system will be provided to the customer after authentication and validation from the server side. Transactions can be of four types: Balance enquiry, Transfer funds, Deposit funds and Withdraw money. On the basis of the choice selected by a customer, required transaction will be successfully executed and customer will be provided with expected result. (For example; in case of money withdrawal, customer will be provided with amount and printed receipt of transaction.)
Frequency of use:	01

Fig. 3.2.11 : Use case Scenario - Transaction

Fig. 3.2.12 depicts detailed use case diagram for an ATM machine.

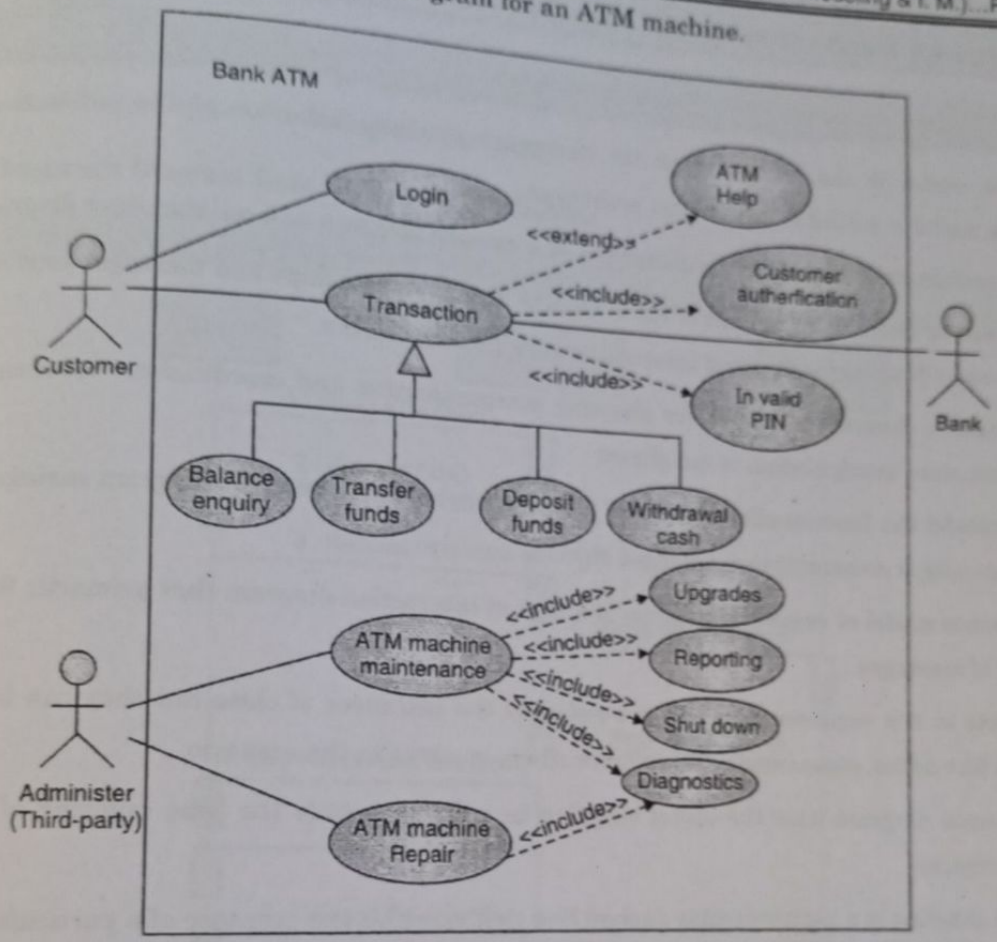


Fig. 3.2.12 : Use case diagram for ATM Machine

Guidelines for designing a Use Case Diagram

- All use cases involved in the scenario should be named uniquely and they should unitedly depict the overall behavior of the system.
- For better understanding, only important use cases should be shown.
- Only related actors should be shown in the diagram.
- System name should be given in such a way that, it should specify its purpose.
- Arrange the actors and use cases in the diagram in some proper sequence so that, roles of the actors and their behavior can be represented in right manner and it will also help to avoid lines that cross.
- Try to avoid showing too many relationships between use cases and respective actors.

3.3 Sequence Model and Procedural Sequence Model

GQ. What is sequence model? Explain with suitable example.

- Sequence model is the next variant for designing dynamic behavior of the software system. The sequence model is a kind of interaction archetype.
- The interaction diagrams typically comprises of sequence diagram and collaboration diagram.
- An interaction diagram comprises of set of objects, their relationships and messages sent amongst them and is mainly devoted for showing interaction between the objects.
- An interaction diagram is meant for showing communication and coordination between two or more objects but, data manipulation is not shown.
- To understand the functionality of each of the component, an interaction diagram mainly emphases on specific messages communicated amongst objects.
- The sequence model or sequence diagram is a type of interaction diagram that primarily focuses on time ordering of messages.
- The objects in the sequence diagram are not only the instances of class but they can be instances of elements like nodes, components and collaborations involved in the scenario.
- The sequence diagram uses the object timeline in order to specify the time ordering of the messages between objects.
- An object timeline is a perpendicular dashed line that specifies the presence of a particular object over a particular period of time for which an object is active in the system execution.
- All of the objects are arranges at the top of the diagram horizontally.
- The lifeline of each object involved is drawn from top to bottom of the diagram.
- Focus of control is another important element of a sequence diagram which is graphically represented as a rectangle that shows the time period during which an object is active or performing inside the system.
- As far as procedural sequence model is concerned, we should categorize objects into active objects and passive objects. Active objects have their own focus of control and are always activated.
- Also, we can depict conditions in a procedural sequence model.
- Implementation details of the proposed software system should be given in procedural sequence model for better understanding purpose.
- Also, in case of a procedural sequence model; active objects and passive objects should be distinguished; since, active objects are having their solitary focus of control and they are active in nature; while, passive objects are passive and they don't have their own focus of control.

- Other they additional features should be added as per the typical requirements of the proposed software system scenario.
- In conclusion, sequence diagram is meant for modeling the dynamic aspect of the software system and it gives flow of control by time ordering of messages.

3.3.1 Sequence Diagram Case Study : ATM Machine

GQ. Draw a detailed sequence diagram for ATM system scenario.

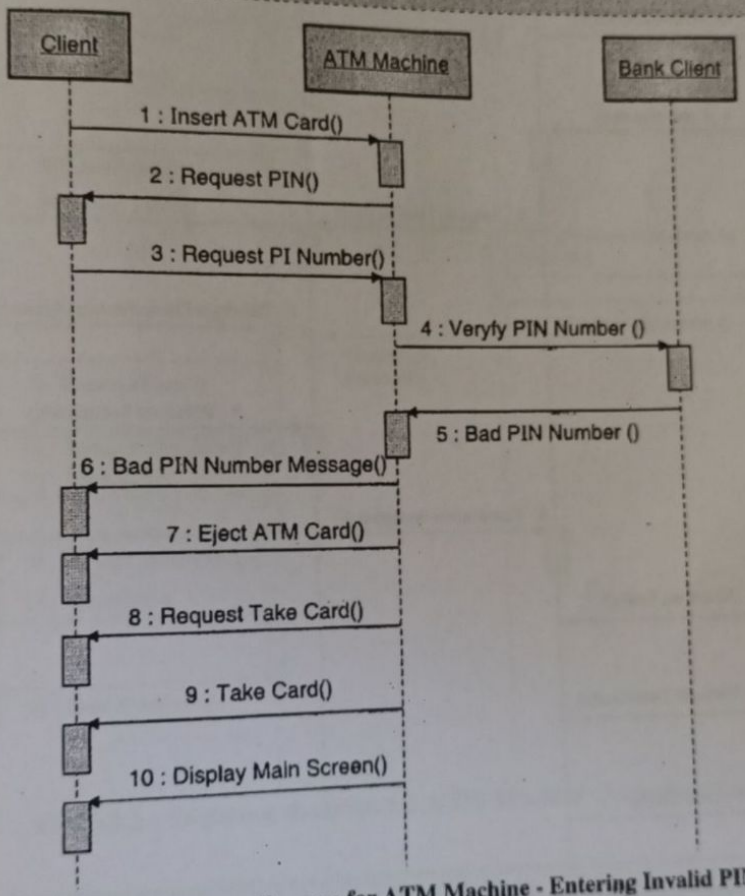


Fig. 3.3.1 : Sequence diagram for ATM Machine - Entering Invalid PIN

NOTES

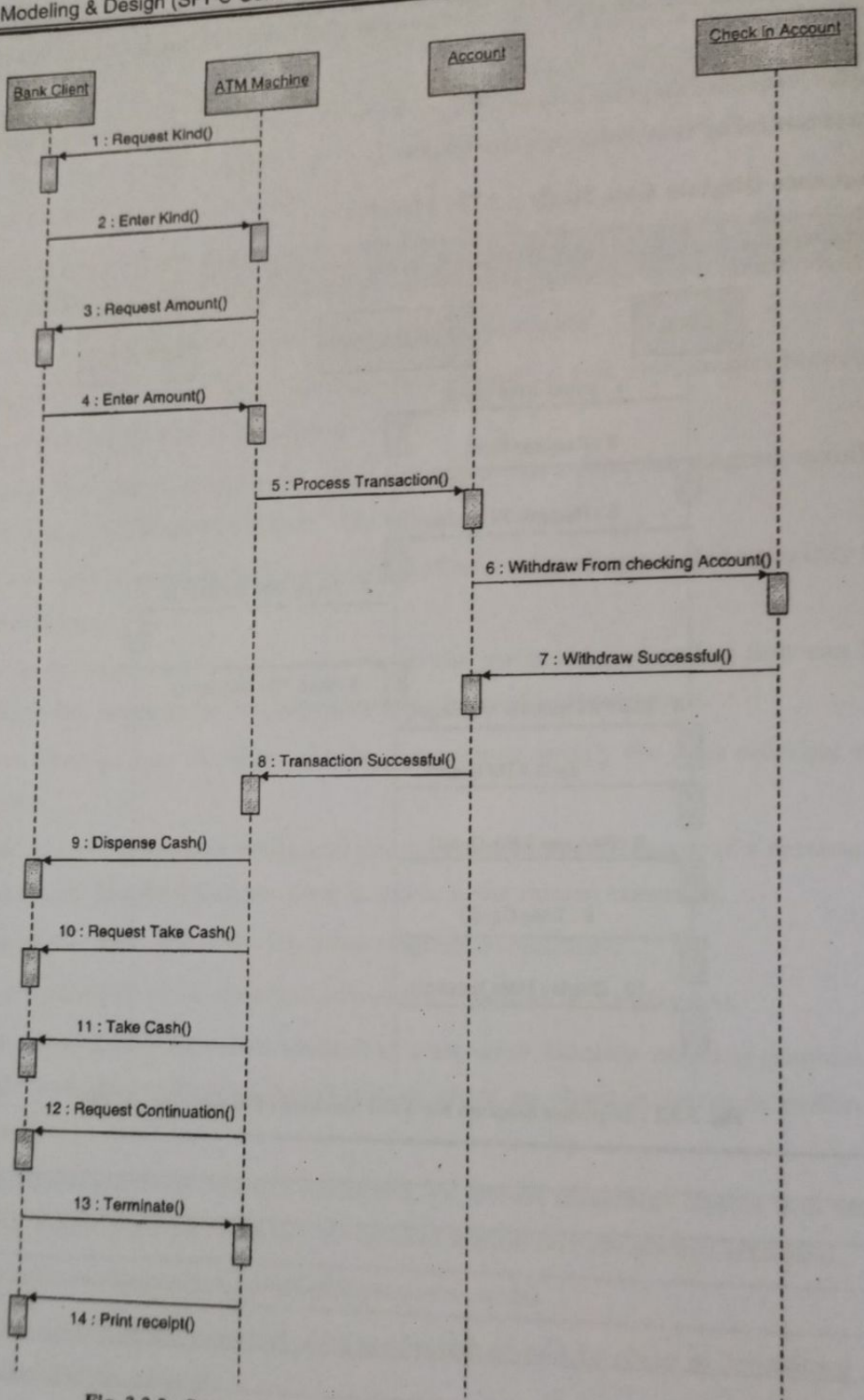


Fig. 3.3.2 : Sequence diagram for ATM Machine - Cash Withdrawal

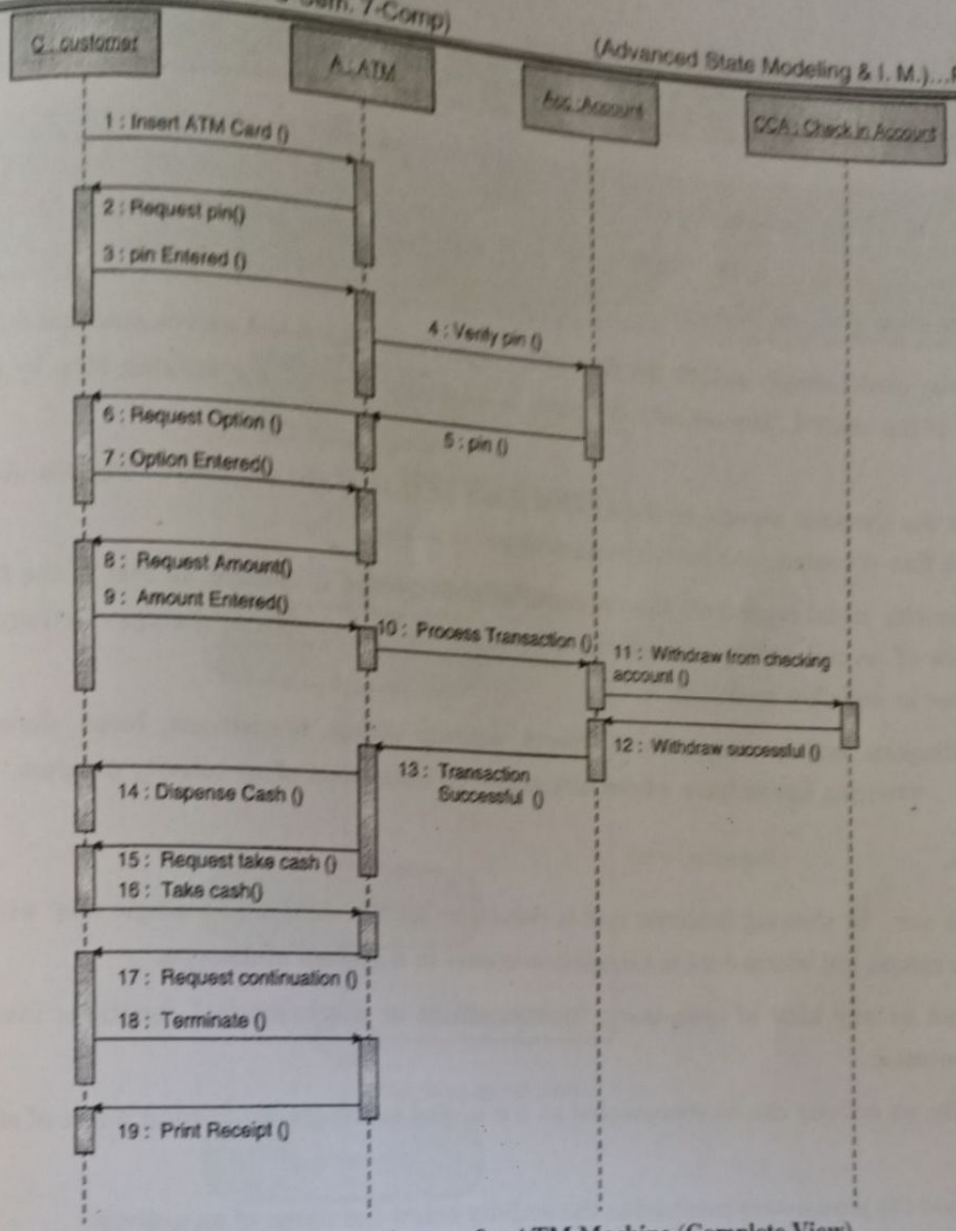


Fig. 3.3.3 : Sequence diagram for ATM Machine (Complete View)

Guidelines for designing a Sequence Diagram

- A sequence diagram should be used when we have to depict the flow of control within a system by time ordering of messages amongst objects.
- A sequence diagram should contain only important objects, nodes, components, collaborations or actors for better understanding.
- The lifeline for each of the object that is actively involved in the system execution should be set.
- Packages should be use in case of a large collection of sequence diagrams and each sequence diagram in this case should be named uniquely in order to differentiate it from other sequence diagrams.

3.4 Activity Models and its Special Constructs

GQ. Write a short note on Activity model.

GQ. What do you mean by activity? Consider the ATM system scenario. Identify and explain at least five activities involved in the ATM system scenario.

- The activity model simply models the flow of an object since an object moves amongst different states in the flow of the control. The activity diagram is basically devoted for making step by step executable systems.
- It models the dynamic aspects of the system same as that of the sequence diagram. Activity diagram focuses on flow of control from activity to activity.
- Mainly activity model represents the sequence of activity and it is same as that of the flowchart which shows flow of control within the activities involved in the system archetype. Activity diagrams are widely used in workflow modeling.
- Activity diagram mainly comprises of objects, activity states, transitions, loops, decision nodes and concu- -ctivities. Let us have a brief discussion on constructs of an activity diagram.

3.4.1 Ac

- Activity is use for showing behavior and it results in action. Action is a single step within an activity and is the compo..ent where data manipulation occurs in a system archetype.
- Actions can be any kind of operations, functionalities or mathematical functions involved within a system execution.
- Graphically, an activity can be represented as a rounded rectangle containing name of an activity in its body.
- We can show the parameters involved in the activity below the name of an activity.
- For instance refer Fig. 3.4.1.

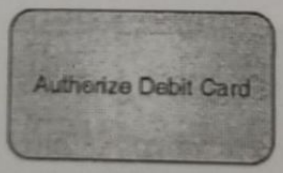


Fig. 3.4.1 : Simple Activity

NOTES

3.4.2 Activity Diagram Case Study : ATM Machine

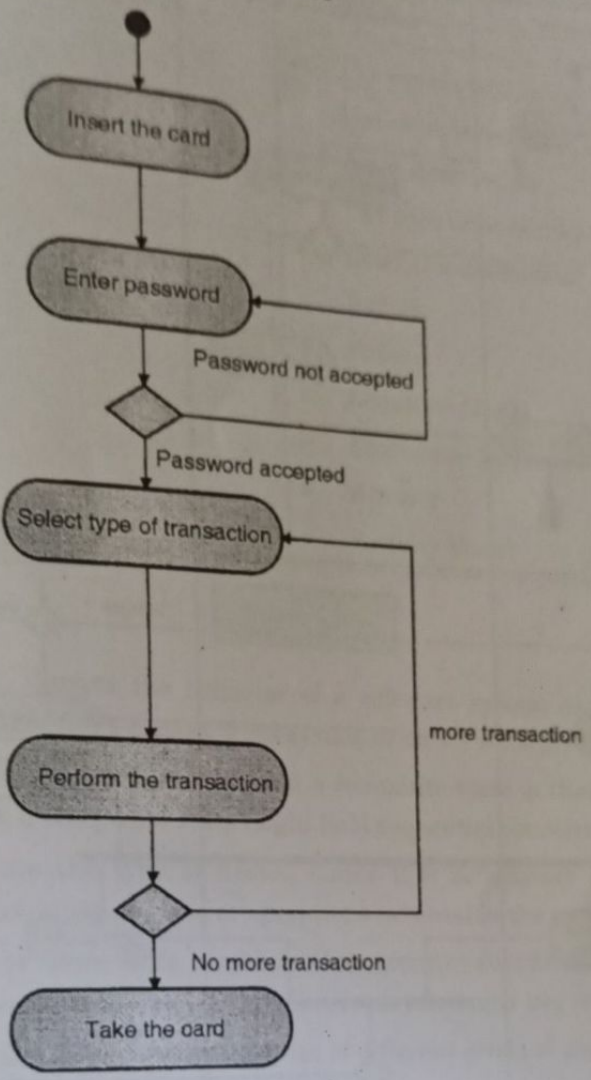


Fig. 3.4.2 : Simple Activity diagram for ATM Machine

Guidelines for designing Activity Diagram

- Activity diagram is intended for representing dynamic behavior of a system.
- For better understanding, only essential components should be taken into account and should be shown in the diagram.
- Name of activity diagram should mention its intention appropriately.
- All activities involved in the software system scenario should be arranged in some sequence.
- Activity modeling should comprise of primary flow, secondary considerations and object flows.

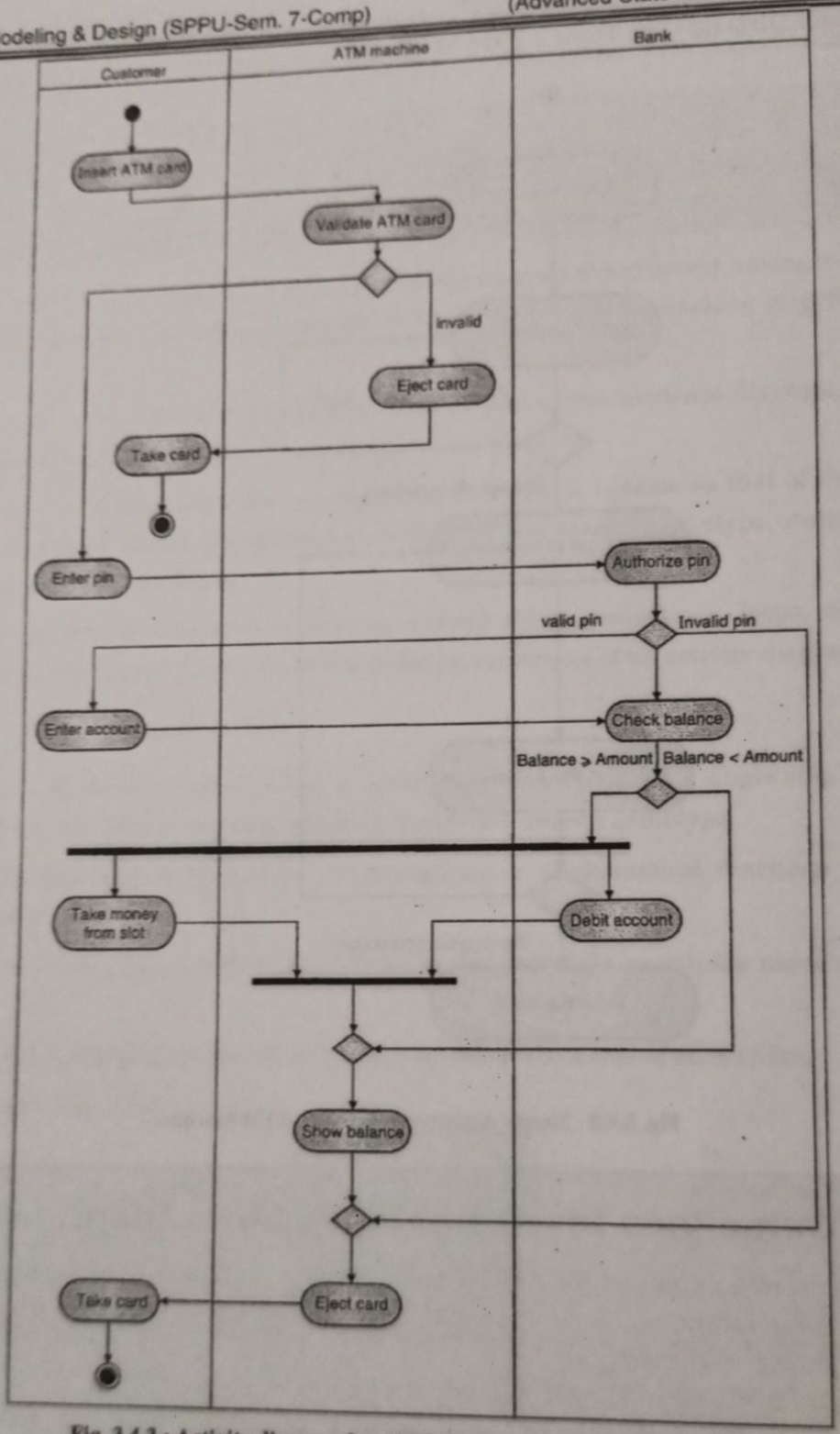


Fig. 3.4.3 : Activity diagram for ATM Machine with special constructs

Key Concepts

- Nested State
- Nested State Diagram
- Sequential substate
- Concurrent substate
- Class model
- State model
- Use case model
- Use case
- Actor
- System Boundary
- Use case relationships
- Use case Association
- Actor Association
- Use case Generalization
- Actor Generalization
- Include
- Extend
- Sequence Model
- Procedural Sequence Model
- Activity
- Activity Model

Summary

- ***State diagram*** in UML depicts the behavior of a software system and can be used to model the behavior of different elements like a class, a subsystem or entire software system.
- The basic difference between a simple state and a composite state is that, there is no substructure in case of a simple state but, a composite state might hold sequential concurrent substates.
- ***Sequential substates*** are the type of nested states that collectively defines certain scenario and depicts a proper flow of activities, actions or operations involved in the scenario.
- If we have to show two or more state machines that executes concurrently, then we can make use of concurrent substates in order to represent the concurrency amongst two or more state machines.
- The ***signal generalization*** mechanism offers use of different levels of abstraction in the state model.
- For achieving ***concurrency*** in state model, each and every object should share their features, operations and constraints with other objects in the system.
- The ***use case model*** depicts functional requirement of the proposed system by means of the use cases and the actors.
- ***Use case model*** supports in the design of the software system by presenting intended behavior of the proposed system without any description about the implementation of the system.
- ***Use cases*** are meant for specification of the interaction between the system itself and end users of the system which are termed as actors in UML.
- The set of activities and events in some proper sequence specifying the interaction amongst a system and its end users is known as a ***scenario***.
- An ***actor*** is the external user of the system who is responsible for communication and coordination with the software system.
- Actors are always ***external*** to the system in use case diagram.

- ***Use case generalization*** indicates the generalization relationship between a specific use case and a general use case.
- ***Actor generalization*** indicates the generalization relationship between a specific actor and a general actor.
- ***Include*** relationship gives us the facility to include the behavior of one use case into the flow of the other use case involved within the scenario.
- ***Extend*** relationship gives a platform for a use case in order to extend its behavior with one or more other use cases within the scenario.
- The ***sequence model*** or ***sequence diagram*** is a type of interaction diagram that primarily focuses on time ordering of messages.
- The ***activity model*** simply models the flow of an object since an object moves amongst different states in the flow of the control.
- ***Activity*** is used for showing behavior and it results in action.
- ***Action*** is a single step within an activity and is the component where data manipulation occurs in a system archetype.

Chapter Ends...

❑❑❑

Unit IV

CHAPTER 4

User Application Analysis : System Design

University Prescribed 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 into sub-systems, Identifying concurrency; Allocation of sub-systems; Management of datastorage; 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.

Specific Instructional Objectives :

At the end of this lesson the student will be able to :

- Explain the software system performance estimation process.
- Define reuse and reuse plan.
- Make a reuse plan.
- Explain the organization of a system into subsystems.
- Explain what a concurrent inherency is.
- Identify concurrency inherent in the problem.
- Explain subsystems allocation to hardware platforms.
- Define data store management.
- Manage data stores.
- Handle global resources.
- Set trade-off priorities.
- Explain different architectural styles.
- Draw and explain a component diagram.
- Draw and explain a deployment diagram.

4.1 Estimating System Performance

GQ. Write a short note on: System performance estimation.

Tentative performance estimation should be done in system planning phase while developing a new software system.

This kind of performance estimation of a software system is known as a "back of the envelope" valuation.

- The performance estimation process should be fast and progressive in nature. The basic need for the performance estimation is that, the software developers should try to simplify the assumptions made during the requirements analysis phase of the software development.
- The main agenda behind the performance estimation of a software system is to determine the feasibility of a software system and it is not at all dedicated for attaining the greater accuracy in the final outcome.
- Basically, the software system performance estimation process comprises of two activities : prediction, approximation and estimation.
- That is, software developers should first of all predict and approximate the performance of a proposed system which is then followed by the thorough system performance estimation process.
- For the simplification of a performance estimation process, let us consider an ATM system example. Assume that, a particular network service provider provides an ATM network service to a bank having around 500 branches all over the country.
- Suppose there are an equivalent number of workstations or terminals in superstores and other markets. Suppose on a busy days; half of the workstations or terminals are abundantly working at once.
- Assume that, each purchaser takes at least one minute to accomplish a single session and maximum transactions initiated by customers encompass a single withdrawal or deposit.
- System designers can estimate an ultimate prerequisite of about 500 transactions per minute or per second. This may not be accurate, but it indicates that a system do not necessitate extraordinarily fast, progressive and advanced computer hardware.
- The computer hardware would become a giant dispute in case of online transactions. Software system developers can accomplish same estimate for the purpose of information storage. In this case, system designers should first of all count the number of customers involved in the particular scenario which is then followed by estimation of the information required by each customer.
- The necessities of information storage for the bank activities are supplementary and simple in comparison to the storage required for ATM computing power.
- In case of satellite based systems, the circumstances would be different where information storage and access bandwidth would be the crucial architectural disputes.

4.2 Making a Reuse Plan

GQ. Define Reuse plan. Explain in brief the following terms with respect to Reuse plan :
(a) Library (b) Framework (c) Pattern

- Reuse plan is considered as one of the key advantage of the object oriented technology but the reuse of system components or system itself does not happen unexpectedly. Reuse should be well planned.
- Two possible facets of reuse are :
  - With the help of existing things
  - By means of producing reusable new things.
- The reutilization of existing things is easier than the proposition of new things. Only the constraint is somebody must have designed the reusable things in the past so that, designers can make use of them for their present purpose.
- Only a limited number of software system developers produce new things for their use and rest of the system developers recycle the existing things. Reusable things contain libraries, models, frameworks and patterns.
- Reuse of model is the most practical form of reuse. The logic in a model can be applied to several problems.

4.2.1 Libraries

- A library is a collection of classes that are convenient and valuable in various circumstances. The collection of classes should be systematized and arranged judiciously in such a way that the system users can find them.
- Thorough and profound work is essential for good organization of contents or things involved in a particular scenario and sometimes it becomes challenging to reside all the things in place.
- Moreover, the classes must have precise and exhaustive explanations in order to assist end users for determination of their significance.
- Characteristics of good class libraries given by *Korson* are :

1. Efficiency	2. Genericity	3. Consistency
4. Completeness	5. Coherence	6. Extensibility

  - ***Efficiency*** : A library should offer alternative implementations of algorithms that trade space and time.
  - ***Genericity*** : A library should use parameterized class definitions where appropriate.
  - ***Consistency*** : Polymorphic operations should have consistent names and signatures across classes.
  - ***Completeness*** : A class library should offer complete behavior for the chosen themes.
  - ***Coherence*** : A class library should be organized about a limited, well-focused themes.
  - ***Extensibility*** : The user should be able to define subclasses for library classes.

- While integrating two or more class libraries from several sources, some problems mentioned below may arise.
  - Methods involved in one library can yield error codes to the calling routine.
  - Class libraries make use of several error handling mechanisms.
  - A class name collision is also one of the problem involved in class library.
  - Procedure-driven control and event-driven control can be assumed by different software system applications on the basis of their prerequisites. In this case, the key problem is we cannot combine both types of user interfaces in a particular software system application.

4.2.2 Frameworks

- Basically, a framework describes the broad approach to solve a particular problem. In short, a framework describes a complete architecture of a software system application.
- Hence, a software framework is used in order to design and develop a complete software system application and it can be expanded by software designers.
- As far as software modeling and design is concerned, frameworks are categorized into two broad categories : **Black Box frameworks** and **White Box frameworks**.
- In case of a Black Box framework, the internal structure of a software program is hidden from end user. End users can access only the general description of a software program.
- In White Box framework, software designers or end users can view all of the components and overall architecture of a framework.

4.2.3 Patterns

- A pattern is simply nothing but an established and confirmed solution to a general problem.
- Several patterns are meant for number of phases of the software development lifecycle. There are exclusive patterns for analysis, architecture, design and implementation phases of SDLC. Reuse can be attained by means of existing patterns.
- An individual pattern comes with rules and guidelines on when to use it, as well as trade-offs on its use. One of the major benefits of pattern is that a pattern has been judiciously considered by others and has already been applied to ancient problems.
- Subsequently, a pattern is more likely to be correct and robust than an unapproved and unverified custom solution. Correspondingly when a particular pattern is used, we tap into a language that is accustomed to many software system developers.
- A pattern is quite different from a framework. A pattern is usually a small number of classes and relationships. In contrast, a framework is much wider in scope and covers an entire subsystem or application.

4.3 Organizing a System into Subsystems

GQ. How can we organize a system into subsystems? Explain in detail.

- The very first step in software system design is to divide the system into number of pieces for better understanding of the software system scenario. Each key fragment of the system is known as a ***subsystem***.
- Each subsystem is based on some common theme, such as similar functionality, the same physical location, or execution on the equivalent kind of hardware.
- For instance, a spacecraft computer system might contain subsystems for life support, navigation, engine control, running scientific experiments and other activities.
- A subsystem is a group of classes, associations, operations, events and constraints that are unified, incorporated and have well-structured and defined small interface with other subsystems. A subsystem is typically acknowledged and approved by the numerous services provided by it.
- A service is a cluster of related functions that share some mutual purpose, such as I/O processing, drawing pictures or performing arithmetic.
- A subsystem articulates a rational technique of looking at a piece of the problem. For instance, the file system within an operating system is a subsystem that encompasses a set of associated abstractions that are generally independent of abstractions in other subsystems such as process management and memory management.
- Each subsystem has well-structured and defined interface to the rest of the system.
- Layers or Partitions can be used in order to divide the complete software system into subsystems in horizontal or vertical manner respectively. That is, software designers can organize the subsystems horizontally by means of layers and vertically by means of partitions.
- In both of these cases, all layers or partitions involved within a software system are interrelated with each other and interrelationships amongst these layers or partitions is of the type client server relationship.
- The key difference between partitions and layers is that, the subsystems defined with the help of partitions will have identical and same levels of abstraction whereas the subsystems defined by means of layers will have different levels of abstraction.
- One more difference is partitions are having peer to peer relationship amongst them and layers are having client server relationship in between them.
- Consequently, each partition can be considered as a self-governing peer and each layer is dependent on some other layer or group of layers.
- Furthermore, we can combine partitions and layers in order to get a software system as a whole. In conclusion, we can layer the partitions and partition the layers.

4.4 Identifying Inherent Concurrency in a Problem

- The state model is dedicated for detailed description of sequence of tasks and services involved within the system operations and do not deals with the contents like details of operations and services, how different services and operations are executed during implementation and execution of the proposed system, etc.
- The state model describes those aspects of objects concerned with time and the sequencing of operations. When two objects can get the events simultaneously deprived of interfacing, they can be said inherently concurrent.
- As far as implementation is concerned, all of the software objects are not concurrent, since one processor can support many objects. The state model is basically articulated by means of a state diagram.
- So, with the help of a state diagram, several objects involved in a scenario can be congregated into a solitary thread of control. A thread of a control is an end to end path through a set of state diagrams of discrete objects and only single object can be active at a particular instance of time.
- For instance, in case of an ATM system, the ATM machine is idle; when the bank is authenticating and confirming an account or processing bank transaction.
- If the central computer system proximately controls the ATM, the bank transaction object and an ATM object can be associated with each other as a solitary task.

4.5 Allocation of Subsystems to Hardware

- The necessity of advanced or superior performance basically forms the basis for the decision to use various processors or hardware functional units. The number of essential processors required totally depends on the speed of the machine and the volume of calculations.
- For instance, a parallel computing system produces too much data in a very short time span in order to handle in a solitary central processing unit. Many parallel machines should abstract the information or data handled within a system well before analysing a threat.
- The software system designer should assess and calculate the mandatory CPU processing power by means of computing the stable state load as the product of the number of transactions per second and the time required to process the particular transaction. The estimate will ordinarily be inaccurate or imprecise.
- The estimate should be increased in order to tolerate the transient effects due to random variations in load in addition to the synchronized bursts of activity.
- The sum of surplus capacity required depends on the adequate rate of failure due to inadequate resources.
- In case of an ATM system example, the ATM machine is mainly responsible for actual online transaction processing and providing a user interface to customer for communication and coordination amongst customer and ATM system.

- A solitary central processing unit is sufficient for an individual ATM machine. The main server of a bank can be considered as a routing machine fundamentally since it receives ATM requests and communicates them to proper bank computer situated at a particular branch of a bank.
- Sometimes, a large network may comprise of multiple processors in case of an ATM system scenario. The computer systems situated at the bank branch accomplish data processing and encompass database applications.
- Each device is an object that operates synchronously along with other objects these other objects may be hardware units or devices involved in a system.
- The software system developers should decide and agree upon the fact that, which subsystems will be implemented in software and which will be executed in hardware. Two reasons behind implementing subsystems in hardware are cost and performance.
- Much of the trouble of designing a software system comes from accomplishing externally enforced software and hardware constraints. Object oriented design affords no magic solution however the external packages can be demonstrated as objects.
- The software system designers should take into account the cost, compatibility and performance issues. Correspondingly, the system designers must think about flexibility for future modifications or alterations. The future modifications can be future software product versioning or improvements and software product design variations.
- In case of an ATM system example, no such performance issues are occurred.

4.6 Data Storage Management

GQ. Explain data storage management in software modeling.

- Number of substitutes are available for managing the data storage that we can use distinctly or in combination. For instance, databases, files, data structures, etc. Different types of data stores offer trade-offs between cost, access time, capacity, cost, reliability and access time.
- For instance, a personal computer system application can make use of files and data structures as per its requirement, an accounting can use a database to connect subsystems. Files are simple, modest, permanent and inexpensive.
- Implementations for sequential files are ordinarily standard however storage formats and commands for random-access files and indexed files fluctuate.
- Following kind of data is appropriate and can be suitable for files :
  - Sequentially accessed data.
  - Data that can be wholly read into memory.
  - Data with low data density.
  - Uncertain data with modest structure.
  - Data with high volume.
- Databases are the other type of data store which are typically managed, monitored and controlled by means of database management systems. Several types of database management systems like relational databases and object oriented databases are available.

- Following kind of data is appropriate and can be suitable for databases :
  - Larger amount of data that should be handled proficiently and skillfully.
  - Data that synchronized the updates through transactions.
  - Data that needs updates at satisfactory levels of detail by several users.
  - Data that must be accessed by numerous application programs.
  - Data that should be protected against malicious access.
  - Data that is long-lasting and highly valuable to an organization.
- The software system designers must consider object oriented database management systems merely for specialty domain applications that have a wide variety of data types or that must access low level data management primitives.
- These software system applications can be multimedia applications, embedded system applications, different engineering applications and many more.
- The software system designers should make use of relational database management systems for the software system applications that requires databases since the features of relational database management systems are adequate and satisfactory for utmost software system applications.
- Furthermore, if relational database management systems are used appropriately, they can offer a super execution of an object oriented archetype.

4.7 Global Resources Handling

GQ. Explain in brief : Global Resource Handling.

- The software system designer should recognize the global resources. The categories of global resources are mentioned in Table 4.7.1 below :

Table 4.7.1 : Types of global resources

Types of Global Resources	Examples
Physical Units	Processors, tape drives, communication satellites.
Space	A workstation screen, the buttons on a mouse.
Logical Names	Object IDs, filenames, class names.
Access to Shared Data	Databases.

- A physical unit like processor when considered as a resource, it can regulate and control overall activities involved in the scenario on its own.
- A global resource may also be segregated for self-governing control. The buttons on a mouse, a workstation screen can be the global resources of the type space.
- The entities dedicated for unique identification of a particular object in a given scenario can be deliberated as resources. For instance, class names in class model, object IDs or object names in object model, file names in file management system are the logical entities that are used as resources.
- In case of an ATM machine system, the account numbers of customers and the bank codes are global resources. Bank codes should be unique within the context of a bank.

4.8 Choosing a Software Control Strategy

GQ. Explain : (a) External control (b) Internal control.

- Basically, there are two types of control in a software system :

1. External control 2. Internal control

- The flow of the events between the objects involved in the software system scenario which are visible from outside are termed as external control flows.
- However, the control flow comprised by a process is known as an internal control flow.

4.8.1 External Control

GQ. What are the different categories of external control? Explain in brief.

- Moreover, there are three types of control for external events :

1. Procedure-driven Control 2. Event-driven Control 3. Concurrent Control

- The choice of control flow totally depends on the existing resources involved in the software system application.

4.8.1.1 Procedure-driven Control

- In a procedure-driven sequential system, control exists within the software program coding.
- The procedure driven control is quite easy to implement by means of conventional languages.
- It is the honest responsibility of a software system designer to translate events into operations among objects.
- The drawback of procedure driven control is, concurrency inherency is significant amongst objects involved in a scenario.

4.8.1.2 Event-driven Control

- An event-driven control is associated with the circumstances where the measurement method is inherently event-based in nature.
- Event-driven control offers adaptable and compliant control.
- From implementation point of view, it is more challenging to implement as compared to the procedure-driven control.
- As far as modularity of a software system is concerned, event-driven control flow supports for additional modularity for breaking of a software system into subsystems.

4.8.1.3 Concurrent Control

- Concurrent control guarantees that, the respective transactions involved within a scenario are accomplished and executed concurrently devoid of violating the integrity of data and hence, data remains whole, complete and uninterrupted within a scenario.

- It is also known as yes-no control or screening.
- Ultimately, there are three basic classes of concurrent control mechanisms: pessimistic, semi-optimistic and optimistic.
- Pessimistic concurrent control refers to the blocking of a transaction if that particular transaction violates the rules. Semi-optimistic concurrent control blocks transactions in some circumstances that may cause violation and optimistic concurrent control do not blocks the transaction but it postpones the respective transaction.

4.8.2 Internal Control

- Throughout the software system design, the software system developer expands operations on objects into lower-level operations on the same or other objects.
- Software system designers can make use of identical system execution mechanisms for developing internal objects, external objects and communication and coordination amongst internal and external objects.

4.9 Handling Boundary Conditions

- Three issues should be deliberated while handling boundary conditions which are initialization, termination and failure.
- At the outset, the software system should initialize parameters, constants and variables.
- Termination is generally modest as compared to the initialization since many internal objects can merely be unrestricted. A particular executing transaction or task should release the resources.
- Failure is the unintended closure of a software system. Usually, software development team members recognize that, a proposed developed software system is not working as per the requirements mentioned in the software requirements specification.
- The failures are candidly observed by software testers during thorough and systematic software testing of a software system. A misbehavior in the execution of a software system can be considered as an indication of the failure.

4.10 Setting Trade-off Priorities

- The trade-off priorities should be established for the good software system design. It is the responsibility of software system designer to set trade-off priorities for a software system.
- The task of designing trade-off priorities encompasses several types of software development techniques. If customer needs a delivery of a particular software module earlier than the outstanding software modules then customer should sacrifice the overall functionality of the software system as a whole.
- It is the duty of the software system designer to define the comparative significance of the several criteria as a guide for creation of design trade-offs.
- The software system designer does not form all the adjustments but launches the priorities for constructing respective software system arrangements.

4.11 Selecting an Architectural Style

GQ. Write a short note on : Architectural styles.
GQ. What do you mean by batch transformation and continuous transformation?
GQ. Explain :
(a) Interactive interface (b) Dynamic simulation
(c) Real time system (d) Transaction manager

- Numbers of architectural styles are commonly used in existing software systems.
- Following are some types of architectural styles :

1. Batch Transformation	2. Continuous Transformation
3. Interactive Interface	4. Dynamic Simulation
5. Real Time System	6. Transaction Manager

4.11.1 Batch Transformation

- In batch transformation, the information transformation is executed once on a complete input dataset. This architectural style accomplishes sequential computations.
- The main objective is to calculate an answer and is achieved by the application which is meant for receiving the inputs. Stress analysis of a bridge, payroll processing are the classic applications of batch transformation.
- In batch transformation, software developers should first of all breakdown the complete transformation into stages with each stage accomplishing one part of the particular transformation which is then followed by Formulation of class models (class diagram) for the input, output and in between each pair of successive stages.
- Next step involved is, expansion of each phase or level until the operations are straightforward to implement and finally reorganize the ultimate pipeline for optimization.

4.11.2 Continuous Transformation

- It is a system in which the output of the system is aggressively dependent on varying inputs. An uninterrupted transformation updates system outputs frequently.
- Windowing systems, signal processing are the classic applications of continuous transformation.
- In continuous transformation, the first step involved is to breakdown the complete transformation into stages with each stage accomplishing one part of the transformation. Then, we should describe and summarize input, output and intermediate models amongst all of the pairs of successive stages, as for the batch transformation.
- After successful breakdown of the complete transformation and defining inputs and outputs, we should discriminate each operation in order to update the incremental modifications or alterations to each level. Finally, we have to add transitional objects for optimization purpose.

4.11.3 Interactive Interface

- It is a system that is conquered by interactions amongst the external agents and the system itself.
- The external agents can be devices or humans.
- These external agents are self-governing and are independent of the system, so the system cannot control the agents.
- While using an interactive interface as an architectural style, we should first of all Segregate interface classes from the application classes.
- Predefined classes should be used for communication and coordination amongst the external agents.
- Next, we should separate out the logical events from physical events. Logical events correspond to multiple physical events. At last, we should identify and state the application functions that are invoked by the interface.

4.11.4 Dynamic Simulation

- This architectural style is dedicated for designing and modeling real world objects. Video games can be the classic examples of this type.
- The internal objects in the dynamic simulation correspond to real world objects and hence the class model (class diagram) is ordinarily significant.
- While selecting a dynamic simulation as an architectural style, we should diagnose active real-world objects along with the discrete events that relates to discrete interactions with the object from the class model (class diagram). Constant dependencies should also be predicted.

4.11.5 Real Time System

- The real time system is an interactive system with close-fitting or tight time constraints on actions.
- Real time system design is multifaceted and encompasses issues like interrupt handling, coordination of multiple central processing units, etc.

4.11.6 Transaction Manager

- It is a system dedicated for retrieval and storage of data.
- The transaction manager deal with several users who write and read data at the same time that is, concurrently.
- Data should be protected from unauthorized access and accidental loss.
- Inventory control, airline reservations, order fulfillment are the classic examples of the transaction manager.
- Steps involved in the transaction manager are :
  - Map the class model to the database structures.
  - Determine the units of concurrency.
  - Determine the unit of transaction.
  - Design concurrency control for transactions.

4.12 Component Diagram

GQ. Explain in detail the elements of a component diagram.

- A component diagram is basically meant for exhibiting the physical characteristics of the object oriented systems. It is also used for modeling the static implementation view of the software system.
- The physical belongings such as documents, tables, libraries, files, etc. that exist in a node are modeled with the help of a component diagram.
- A component diagram provides a framework for constructing the executable software systems and applications by means of forward and reverse engineering.
- A component diagram depicts a set of components and relationships among those components. Graphically, it is a collection of arcs and vertices.
- The main components of the component diagram includes :
  - Components
  - Interfaces
  - Relationships
    - Generalization
    - Realization
    - Dependency
    - Association
- Also, the component diagram may encompass constraints, packages, subsystems and notes likewise the other UML diagrams.
- The component diagram can be considered as a special type of a class diagram that mainly emphases on the components involved within a system.

Component

- A component is a physical and expendable part of a system that offers the realization of a set of interfaces.
- Graphically, it is represented as a rectangle with tabs.
- Each component involved within a system should be given a name (textual string) in order to identify a particular component uniquely from other components.
- The UML notation for component is depicted in Fig. 4.12.1.

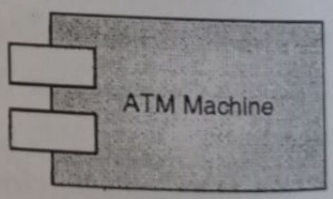


Fig. 4.12.1 : UML notation of Component

- Component shows the physical packaging of the logical elements such as collaborations and classes within a system.
- There are three types of components :
  1. **Work product components :** Examples are data files, source code files, etc.
  2. **Deployment components :** This type of components are essential for creation of execution components.
  3. **Execution Components :** Executable, Dynamic libraries, etc. are the typical examples.

Interface

- An interface is a collection of operations which are used for postulating a service of a particular component or a class.
- The relationship among interface and component is quite important.
- The interfaces are used as a glue for binding components altogether.
- Graphically, interface is depicted with the help of notation shown in the Fig. 4.12.2.

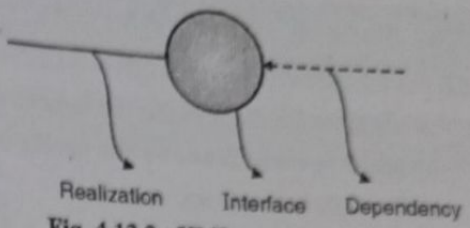


Fig. 4.12.2 : UML notation of Interface

Relationships

- Please refer Section 1.9.2 of this book.
- Case Study : ATM System
  - Fig. 4.12.3 depicts a component diagram for ATM system scenario.

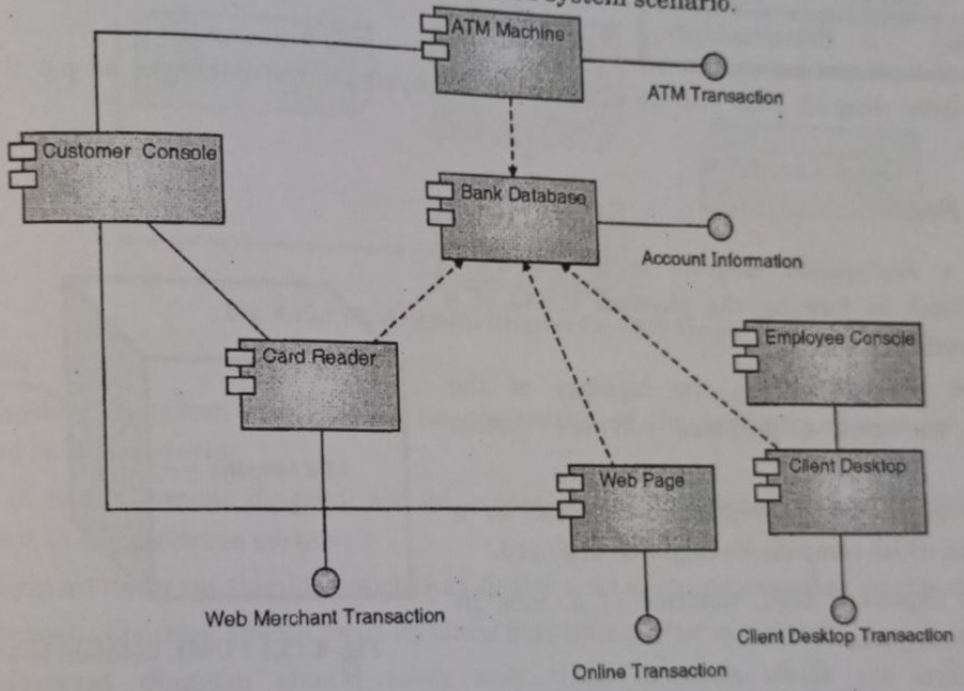


Fig. 4.12.3 : Component Diagram for ATM System

- In conclusion,
  - A component diagram should cover only those elements which are crucial for better understanding of the system.
  - It is focused on software systems static implementation view.

Guidelines for designing a Component Diagram

- A unique name should be given to each component involved in the scenario.
- In order to diminish the crossing lines, all the components and interfaces should be properly arranged.
- Notes and constraints should be used wherever necessary in order to draw attention to significant features of a software system.

4.13 Deployment Diagram

GQ. Explain the following with respect to Deployment diagram :
(a) Node (b) Association (c) Dependency

- A deployment diagram is the subsequent diagram meant for depicting the physical aspect of an object oriented system. It shows the configuration of run time processing nodes and the components involved in the software system.
- A deployment is basically used for modeling the static deployment view of a system. A deployment diagram provides a framework for constructing the executable software systems and applications by means of forward and reverse engineering.
- Following are the basic elements of a deployment diagram :

1. Nodes 2. Relationships

- A deployment diagram may contain components, subsystems and packages as per the necessity of a system.

4.13.1 Node

- Node in a deployment diagram is an important building block in forming the physical facets of a software system.
- Nodes are used to design the topology of the hardware on which a proposed software system executes.
- A solitary node basically represents a device or a processor on which components might be deployed.
- Fig. 4.13.1 depicts a UML notation of a node in deployment diagram.

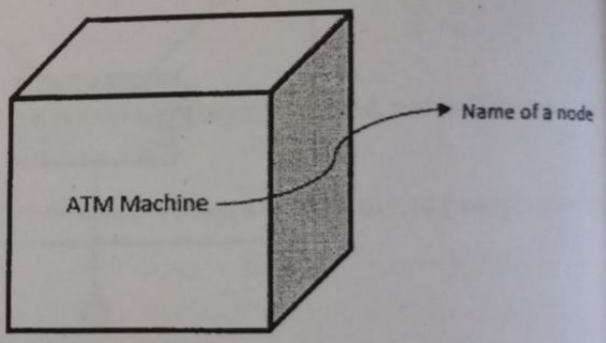


Fig. 4.13.1 : UML notation of a node

4.13.2 Relationship

1. Association 2. Dependency

4.13.2.1 Association

- Association is the most common type of relationship used amongst nodes involved in a deployment diagram.
- An association relationship shows a physical connection between nodes. For instance, shared bus, Ethernet line, etc.
- Association relationship can be furthermore used to define indirect connections.

4.13.2.2 Dependency

- Dependency is the other kind of relationship that can be used for showing the interdependency amongst nodes involved within a system scenario.
- **Case Study : ATM System**

GQ Draw Deployment diagram for ATM System.

- Fig. 4.13.2 shows a deployment diagram for ATM system.

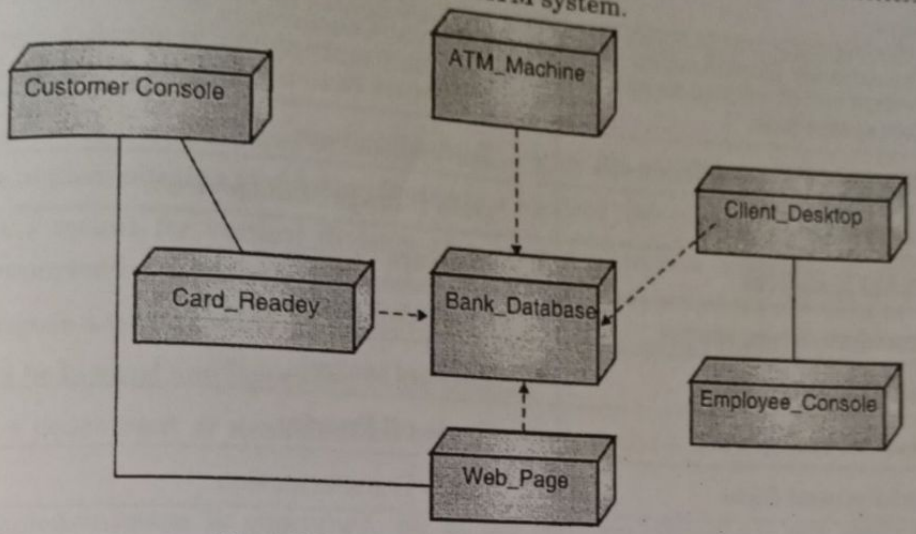


Fig. 4.13.2 : Deployment Diagram for ATM System

- In conclusion,
  - A deployment diagram should offer an abstraction of the hardware components and systems involved in the scenario.
  - Nodes in a deployment diagram should expose only those operations and attributes which are pertinent to the software system.
  - A deployment diagram should straightway deploy a set of components that exist in a solitary node.
  - A deployment diagram should focus on static deployment view of a system.
  - A deployment diagram should cover only those elements which are crucial for better understanding of the system.

Guidelines for designing a Deployment Diagram

- A unique name should be given to each node involved in the system scenario.
- In order to diminish the crossing lines, all the nodes and components should be properly arranged.
- Notes and constraints should be used wherever necessary in order to draw attention to significant features of a software system.
- Stereotyped elements should be used judiciously.
- A deployment diagram should offer a static deployment view of a system.

Key Concepts

• System Performance Estimation	• Reuse
• Reuse Plan	• Library
• Framework	• Pattern
• Layer	• Partition
• Concurrency Identification	• Inherent Concurrency
• Concurrent task	• Hardware Resource Requirements Estimation
• Hardware-Software Trade-offs	• Task Allocation
• Physical Connectivity	• Data Storage Management
• Global Resources	• Software Control
• Procedure-driven control	• Event-driven control
• Concurrent control	• Internal control
• Boundary condition	• Trade-off Priority
• Architectural Style	• Batch Transformation
• Continuous Transformation	• Interactive Interface
• Dynamic Simulation	• Real Time System
• Transaction Manager	• Component
• Interface	• Component Diagram
• Node	• Deployment Diagram

Summary

- The ***performance estimation*** process should be fast and progressive in nature. The basic need for the performance estimation is that, the software developers should try to simplify the assumptions made during the requirements analysis phase of the software development.
- The main agenda behind the performance estimation of a software system is to determine the feasibility of a software system and it is not at all dedicated for attaining the greater accuracy in the final outcome.
- Basically, the software system performance estimation process comprises of two activities: prediction, approximation and estimation.
- ***Reuse plan*** is considered as one of the key advantage of the object oriented technology; but, the reuse of system components or system itself does not happen unexpectedly. Reuse should be well planned.
- Two possible facets of reuse are :
  - With the help of existing things

- By means of producing reusable new things.

- Reusable things contain libraries, models, frameworks and patterns.
- Reuse of model is the most practical form of reuse. The logic in a model can be applied to several problems.
- A ***library*** is a collection of classes that are convenient and valuable in various circumstances.
- A ***framework*** is a structure of a program that must be expanded in order to design and develop a complete software application.
- A ***pattern*** is simply nothing but an established and confirmed solution to a general problem.
- A software system should be divided into a small number of subsystems for better understanding of the software system scenario.
- A ***layered system*** is a well-ordered set of tiers where each tier is built in terms of the tier below it and providing the implementation foundation for the tier situated above it.
- ***Partitions*** are meant for vertical division of a main system into a number of self-governing and inadequately coupled subsystems where each subsystem delivers some type of facility or service.
- We can decompose a system into subsystem by merging partitions and layers.
- Partitions can be layered and layers can be partitioned.
- All objects are concurrent or synchronized in the system analysis model, as in the real world and in hardware.
- As far as implementation is concerned, all of the software objects are not concurrent, since one processor can support many objects.
- Two objects are inherently concurrent if they can receive the event at the same time deprived of interacting.
- A ***thread of control*** is a path through a set of state diagrams on which only a single object is active at a time.
- A thread remains within a state diagram unless and until an object sends an event to another object and waits for another event.
- The thread passes to the receiver of the event until it ultimately returns to the original object.
- The thread splits if the object sends an event and continues executing.
- On each thread of control, only single object is active at a time.
- We can implement thread of control as a particular task in computer system.
- The number of essential processors required totally depends on the speed of the machine and the volume of calculations.
- Numbers of substitutes are available for managing the data storage that we can use distinctly or in combination. For instance, databases, files, data structures, etc.
- Different types of data stores offer trade-offs between cost, access time, capacity, cost, reliability and access time.
- ***Hardware control*** closely matches the analysis model however there are quite a lot of ways for implementing and executing control in a software system.

- ***Internal control*** concerns the flow of control contained by a process.
- In a ***procedure-driven sequential system***, control exists within the program code.
- Procedure request external input and then wait for it; when input arrives, control resumes within the procedure that made the call.
- The location of the program counter and the stack of procedure calls and local variables define the system state.
- The main benefit of procedure-driven control is that, it is quite easy to implement with conventional languages while the drawback is that it necessitates the concurrency inherent in objects to be mapped into a sequential flow control.
- In an ***event-driven sequential system***, control exists within a dispatcher or monitor that the language, subsystem, or operating system offers.
- In a ***concurrent system***, control exist simultaneously in several independent objects, each a separate task.
- Internal object interactions are similar to external object interactions since we can make use of the same implementation mechanisms.
- The software system designer should set priorities that will be used to guide trade-offs for the rest of the software system design.
- Number of architectural styles are commonly used in existing software systems:
  - Batch Transformation
  - Continuous Transformation
  - Interactive Interface
  - Dynamic Simulation
  - Real Time System
  - Transaction Manager
- In ***batch transformation***, the information transformation is executed once on a complete input set.
- A ***continuous transformation*** is a system in which the output of the system is aggressively dependent on varying inputs. A continuous transformation updates outputs frequently.
- An ***interactive interface*** is a system that is conquered by interactions amongst the external agents and the system itself. The external agents can be devices or humans.
- ***Dynamic simulation*** tracks or models real world objects.
- The ***real time system*** is an interactive system with close-fitting or tight time constraints on actions.
- A ***transaction manager*** is a system dedicated for retrieval and storage of data.
- A ***component diagram*** is basically meant for exhibiting the physical characteristics of the object oriented systems.
- A component diagram provides a framework for constructing the executable software systems and applications by means of forward and reverse engineering.
- A ***component*** is a physical and expendable part of a system that offers the realization of a set of interfaces.

- Three kinds of components are :
  - *Work product components* : Examples are data files, source code files, etc.
  - *Deployment components* : These types of components are essential for creation of execution components.
  - *Execution Components* : Executable, Dynamic libraries, etc. are the typical examples.
- An ***interface*** is a collection of operations which are used for postulating a service of a particular component or a class.
- A ***deployment diagram*** is the subsequent diagram meant for depicting the physical aspect of an object oriented system.
- A deployment diagram may contain components, subsystems and packages as per the necessity of a system.
- ***Node*** in a deployment diagram is an important building block in forming the physical facets of a software system.
- Nodes are used to design the topology of the hardware on which a proposed software system executes.
- ***Association*** is the most common type of relationship used amongst nodes involved in a deployment diagram.
- An association relationship shows a physical connection between nodes. For instance, shared bus, Ethernet line, etc.

Chapter Ends...

❑❑❑

Unit V

CHAPTER 5

Class Design, Implementation Modeling, Legacy Systems

University Prescribed 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 modes; Building the interaction model; Building the state model; Reverse engineering tips; Wrapping; Maintenance.

Specific Instructional Objectives :

At the end of this lesson the student will be able to :

- Identify the classes.
- Prepare data dictionary.
- Identify associations.
- Identify attributes of objects and links.
- Organize and simplify classes by means of inheritance.
- Verify access paths.
- Explain reconsideration of the level of abstraction.
- Grouping of classes into packages.
- Identify and define system boundary.
- Identify actors.
- Identify use cases.
- Define initial and final events.
- Define normal scenarios.
- Identify exceptions and variations in scenarios.
- Identify and define external events.
- Design activity diagram for use cases.
- Explain System Design.
- Describe Design Algorithms
- Describe Design Optimization
- Design Algorithms
- Design Optimization.

5.1 System Analysis

GQ. Write a short note on: Data Dictionary.

- These chapter emphases on fundamental concepts of Object Oriented System Analysis and System Design.
- Object oriented analysis methods makes use of the basic object oriented concepts in requirements analysis phase of the software development life cycle (SDLC) for analysis of the proposed software system.
- The static and dynamic archetypes of the proposed software system are designed in the system analysis phase. The main motive behind system analysis is to find out real-world objects and entities in the proposed system scenario and then plotting these objects as the software objects in the system scenario.
- The static object modelling in object oriented analysis comprises of object modelling and class modelling. The dynamic object modelling in object oriented analysis consists of state modelling, activity modelling, use case modelling and interaction modelling.
- Object modelling determines objects while class modelling determines classes to which the identified objects belong and present the relationships amongst classes and objects by means of class diagram.
- So in this chapter, we are going to study basics of system analysis phase in software development life cycle and fundamental issues of object oriented analysis and design.

5.1.1 Finding Classes

GQ. Give the detailed guidelines for finding and defining the classes involved in the software system scenario. Explain with suitable example.

- As we have already discussed, a class is a collection of objects those are having common characteristics. A class can be defined as a detailed explanation of a group of objects that share the equivalent attributes, relationships and operations.
- A class itself is not a specific object however; it is a complete set of objects.
- Typically, objects of a class are distinctive in nature and they are uniquely identified by means of variances in their attribute values and definite association to other objects of the same class. A class of a particular object is a fundamental feature of that object.
- For each object, it is possible to identify and distinguish its own class. Object oriented programming languages are capable of describing a class of an object at run time.
- Finding the appropriate classes in the proposed system scenario is quite essential in the system analysis phase of SDLC.
- For finding the classes, no stepwise procedure or algorithm is available in the literature. If we try to design such algorithm for finding out the classes, then it will be more dependable method of object oriented analysis and design. But, no such mechanism is available in the literature.
- So, analysis of classes is the whole sole responsibility of the personality known as ***software system analyst***. The experienced software system analysts may help out for defining the classes of the proposed software system in right manner.

- Sometimes, two or more classes may describe the same context then; we should choose one of them which is more relevant to the respective situation. So, better way chooses the class which is more descriptive. Also, the inappropriate classes should be removed.
- For instance, following is the possible list of classes in the case study of ATM Machine :
  - Customer/User
  - System User/Bank Employee
  - Credit/Debit Card
  - Receipt
  - ATM Machine
  - Banking Network
  - Central Server
  - Computer System at Bank
  - Bank Branch/Office
  - Customer Account Data
  - Transaction
- We can find out, define and design classes by using :
  - Noun/Verb Analysis
  - Class, Responsibilities and Collaborations (CRC) analysis and
  - Rational Unified Process (RUP) stereotypes
- Let us have a brief discussion on the said strategies for finding the classes.

5.1.1.1 Noun/Verb Analysis for Finding Classes

- For determining classes and their attributes/responsibilities, noun/verb analysis is the simplest strategy. We know that, the UML notation of a class consists of three sections: class name, class attributes and class operations.
- The class name depicts the name of an entity/actor/component/node/system involved in the system scenario. The class attributes are the characteristics or features of a particular class and are used for collection of objects having similar features and properties.
- While the class operations depicts the responsibilities of a particular class in the system archetype. In the procedure of noun/verb analysis for finding out the classes, noun phrases or nouns are dedicated for defining classes themselves or for defining attributes of respective classes and verb phrases or verbs depicts operations or responsibilities of the respective classes.
- System analysts have to be very careful in understanding the proposed system domain first and they should acquire better knowledge regarding system problems. The noun/verb analysis is the two-step procedure for finding the classes.
- First step is collection of the information and second is analysis of gathered information. In the beginning of the noun/verb analysis process, it is very essential to gather maximum system related information.
- System analysts can acquire system related information from different resources such as the proposed system vocabulary, the use case model, the requirements model, proposed system framework and many more.
- After collecting information from different data sources, analyze the information with the help of grammatical terms like noun phrases, nouns, verbs, verb phrases, etc.
- For instance, following is the categorization of the information in ATM Machine case study.
  - **Nouns :** Customer, Account, etc.
  - **Noun Phrases :** Customer ID, Account No., etc.

- **Verbs :** Open, Transfer, Manage, etc.
- **Verb Phrases :** Opening an account, Amount transfer, transaction, etc.

In conclusion, nouns and noun phrases may specify classes or class attributes whereas verbs and verb phrases specify operations or responsibilities of classes.

5.1.1.2 CRC Analysis

- CRC stands for Class, Responsibilities and Collaborations. CRC component comprises of three partitions: class name, responsibilities and collaborations.
- The responsibilities and collaborations are positioned at the same level where, responsibilities are placed in the left side and collaborations are placed in the right side.
- Responsibilities enlist the tasks, functions or operations for which a particular class is responsible and collaborations compartment enlists the other classes that collaborate with a particular class.
- Fig. 5.1.1 shows the structure of CRC.

Class Name : Bank Account	
Responsibilities	**Collaborations**
- Initiate Transaction	- Bank ATM
- Money Transfer	- Bank

Fig. 5.1.1 : CRC component

- CRC analysis strategy should be used along with noun/verb analysis process. CRC analysis is also a two-step procedure: Collecting information and Analyzing information.

5.1.1.3 RUP Stereotypes

- RUP stands for Rational Unified Process. Concept of analysis class is considered in this process. Analysis classes are the classes that signify an abstraction in the proposed system domain and are mapped to the real-world entities.
- Different types of analysis classes are used in finding the classes using RUP stereotypes. Entity class, control class and boundary class are the types of analysis classes.
- Entity class gives determined information about the particular entity involved within the system scenario.
- Control class summarizes use case modelling and helps in describing use case model in brief.
- Boundary class is used as a facilitator between proposed system and outside environment. It provides a platform for communication and coordination between the system and its environment. Fig. 5.1.2 shows different types of analysis class and their UML notations.

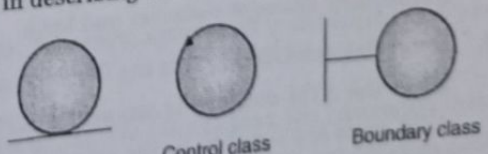


Fig. 5.1.2 : Types of Analysis class

- Entity class have a simple behaviour and are familiar with entities in the system.
- Control class acts as a controller and hence it controls the overall behaviour of the proposed system.

- Boundary class is used for defining the boundary of the proposed system. Boundary classes are furthermore categorized into three types: system interface class, user interface class and device interface class. All of these classes act as a mediator between systems, users and external devices respectively.

5.1.2 Data Dictionary Preparation

GQ. Prepare a data dictionary for ATM system scenario. Explain each element in brief.

- Data dictionary is the phrase related to 'data' and is the thing important for modelling all of the components involved in the system scenario.
- System analyst should briefly explain each class specifically in a passage of at least five statements that exactly describe the class. Also, scope of the class should be mentioned along with assumptions .
- Furthermore, the data dictionary specifies attributes, responsibilities, operations and associations. Let us have a brief discussion on data dictionary for ATM machine.
- We have already identified classes involved in the ATM machine example in Section 5.1.1. As far as the concept of data dictionary is concerned, data dictionary will define each of the class in brief.
- Once again, let us enlist the possible classes involved in the ATM machine.
  - Customer/User
  - ATM Machine
  - Bank Branch/Office
  - System User/Bank Employee
  - Banking Network
  - Customer Account Data
  - ATM Card
  - Central Server
  - Transaction
  - Receipt
  - Computer System at Bank

Now, let us define each class in brief in order to prepare data dictionary for ATM machine.

1. **Customer/User :** Customer in the ATM machine scenario is the human being and is the account holder in a particular bank. Customer/User might handle one or more accounts in the same bank. Customer/User can be an individual customer or a group of persons, organizations or corporations. The same person can have account in different banks but in that case, that particular customer is differently considered in different banks.
2. **ATM Machine :** It is the place from where customers can process their individual transactions by means of ATM card supplied by the bank for the purpose of unique identification of the customer. The ATM machine acts as a mediator between bank and customer and provides platform for handling transactions amongst customer and bank.
3. **Bank Branch/Office :** It is the financial organization that provides banking facilities to customers. As far as the ATM machine is concerned, ATM cards are important for transaction processing and these ATM cards are supplied by the respective bank only. Bank issues the ATM card to individual account holder and hence allows customer to handle the account in the bank.
4. **System User/Bank Employee :** These are the authorized and official personalities of a bank to handle the transactions initiated by a particular customer. Ultimately, these personalities are responsible for handling accounting and financial issues of an individual customer.

5. **Banking Network :** It consists of network components or nodes like ATM machines, banks central server connected to each other for communicating and coordinating the transactions initiated by individual customer. To handle the customers' transaction in real time is the whole and sole responsibility of the banking network and hence without banking network, the ATM machine services are unworkable and are of no use.
6. **Customer Account Data :** It is the detailed record of a particular customers' account and is used for handling transactions of that particular account.
7. **ATM Card :** It is the card provided by the bank for unique identification of a customer and transactions handled by a customer. ATM card comes with a card number, bank code on it and a PIN. The card number identifies a particular account while the bank code uniquely identifies the bank.
8. **Central Server :** It is the computer machine that dispatches transactions amongst the computer systems in the bank and ATM machines which are geographically distributed at different locations.
9. **Transaction :** It is a solitary request for performing operations on accounts of customers.
10. **Receipt:** It is the machine generated token and depicts record of transaction performed by a customer.
11. **Computer System at Bank :** It is the computer system used by bank employees like cashier, manager, etc. through which these employees can enter transactions for customers. These computer systems communicates and coordinates with the central bank server to authenticate and process the transactions.

5.1.3 Association Design : Finding Associations

- Association is a structural relationship amongst two classes and is static in nature. It shows fixed relationships between classes involved in a system.
- Association is shown by a solid line between two classes and it can be called as a binary association.
- Moreover, a reference from one class to the other class is an association.
- Associations can be mentioned in terms of verb phrases or verbs.
- Verb phrases used in communication, for specifying physical locations, for describing some conditions and many more can be used for showing association between two classes.
- In case of an association relation, the similar objects can form a group but all of the objects in a particular group are not totally dependent on each other. Therefore, association is considered as a weak form connection in object oriented programming.
- For example, consider a bus, number of passengers and a driver. We can say that, the passengers inside the bus and the driver are associated when they are in the bus because all of them occupy some space inside the bus and all of them move in same direction.
- A structural property of the problem should be described by an association.

5.1.4 Finding Attributes of Objects and Links

- An object itself is an instance of a class which outlines the set of characteristics or features that are unitedly used by all of the instances of the respective class.
- We can think of an object as a unified pack of data and function. The data is nothing but the information about an object and functions are known as ***operations***.
- An object have its own qualities and characteristics known as ***attributes***. For instance, a person has a name, a qualification, a color, an age and hobbies; a vehicle has a number, a manufacturer, a color, a shape, an owner and a price. The attribute values hold an object's information.
- Objects also can have behaviour. For instance, a vehicle can move from one location to other. Also, objects have their specific self-determining attributes. These attributes plays a vital role in unique identification of an object in a particular scenario.
- Attributes of an object depicts the state of an object. For example, the properties of a motorbike can be manufacturer, color, cost, number of gears, etc., and are the attributes of a motorbike object (Fig. 5.1.3).

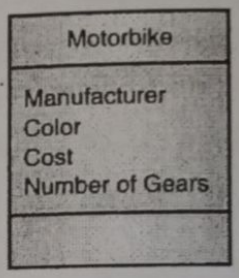


Fig. 5.1.3 : The attributes of a motorbike object

- Each attribute of an object can be represented in different ways in a programming language. For instance, manufacturer of a bicycle can be signified by a name of a manufacturer, unique commercial tax identification number or a reference to a manufacturer object.
- A link is defined as a semantic connectivity amongst two objects and it supports message passing mechanism for transfer of data from one object to other object since, objects requires communication between each other in an object oriented program.
- A link is a physical connectivity between two objects while an association is a physical connectivity between two classes. So, we can say that; link in object diagram is replaced by an association in class diagram and both are having same motive as far as their functionality is concerned. Moreover, a link is an instance of an association.
- Normally, most of the links are used for representing connectivity between two objects but it is quite possible to relate three or more objects with a single link. The links of an association relationship syndicates objects from the similar classes.

5.1.5 Organizing and Simplifying Classes using Inheritance

- Number of classes can be clustered into a single group and a class can inherit some of its characteristics from a parent class.
- Inheritance is the mechanism with the help of which specific elements can obtain their behaviour and structure from general elements involved within the system scenario.
- For organizing and simplifying classes with the help of inheritance, system analyst should first of all look after responsibilities, operations, and attributes that are common to two or more classes in a group of classes.

- Furthermore, system analysts should promote these common attributes, operations and responsibilities to a general class. System analyst can design new class for promoting these things but if possible, it is better to avoid inclusion of new class and system analyst should incorporate the necessary modifications in the existing classes only.
- It is possible for a class to have more than one super class in Unified Modelling Language. That means, a child can have more than one parent and this property is known as Multiple Inheritance.
- Multiple inheritances is not supported by all of the object oriented languages. C# and Java permits only single inheritance. A child class can inherit properties of its super classes in multiple inheritances.
- Implementation of multiple inheritances is possible in object oriented programming only if it is supported by the target implementation language. This can be considered as one of the design issue in object oriented programming and hence, implementation of multiple inheritances is totally dependent on the target object oriented language.
- The "***is kind of*** "relationship should be applied between the super classes and subclasses. Normally, super classes should not have a parent class in common in order to avoid cycles in the inheritance hierarchy.
- All the super classes involved in the scenario should be semantically disjoint. Sometimes, classes are defined to be "mixed in" with other classes with the help of the inheritance. Such types of classes are known as ***mixin classes***. Mixin classes supports multiple inheritances effectively.
- The multiple inheritances allows private inheritance and is powerful as compared to the single inheritance. Also, multiple inheritance permits mix-in inheritance. Repeated inheritance, clashes in names and complexity are some of the drawbacks of a multiple inheritance.
- In conclusion, inheritance is natural, sophisticated and graceful in nature and we can write generic code by making use of the concept of inheritance.
- But, it may not perform well during execution of the system and sometimes it is more complicated for better understanding.
- Furthermore, we can determine inheritance from the bottom up by searching for classes with similar operations, associations and attributes and this strategy is known as ***bottom-up generalization***. In this mechanism, system modellers should refine some classes or their respective attributes in order to fit them in a superclass and we should define a super class for each generalization in order to share the similar features.

5.1.6 Verification of Access Paths

- It is moderately essential to test and verify each access path in the system scenario. Software modellers should verify that, a particular access path produces fruitful outcome as per the requirement given by the end user.
- We can have independent classes in the system scenario for performing unique tasks and operations.
- Sometimes, these independent classes are not connected to any of the class involved in the class model and are disconnected in nature.

5.1.7 Reconsidering the Level of Abstraction

GQ. What is reconsideration of level of abstraction? List and explain several categories of abstraction.

- Abstraction is the fundamental way in which the definite facets of a particular problem are scrutinized selectively. The main objective of an abstraction is to segregate those facets of a particular problem which are essential for some purpose and destroy those features which are irrelevant.
- Abstraction should be purpose specific, for the reason that the purpose defines what is important, and what is not important. An abstraction signifies the crucial features of an object that differentiate it from all other types of objects.
- All abstractions are partial (incomplete) and erroneous (incorrect). An abstraction mainly concentrates on the exterior view of an object and assists for separation of an object's important behaviour from its implementation.
- The main motive behind an abstraction is to limit the universe so we can recognize and realize. Abstraction can be categorized into four types :
  - ***Entity abstraction :*** An object which denotes a useful model of a problem-domain entity or solution-domain entity.
  - ***Coincidental abstraction :*** An object that bundles a set of operations which are not interrelated with each other.
  - ***Virtual Machine Abstraction :*** An object that makes clusters of operations which are used by some superior level of control or operations that make use of secondary level set of operations.
  - ***Action abstraction :*** An object that offers a general set of operations, all of which implement the similar type of function.
- All abstraction has static as well as dynamic properties.

5.1.8 Grouping of Classes into Packages (Physical Packaging)

GQ. Give the guidelines for grouping of classes into packages.
GQ. List and explain types of dependencies in package.

- A class can be defined as a detailed explanation of a group of objects that share the equivalent attributes, relationships and operations.
- A class itself is not a specific object however, it is a complete set of objects.
- An object is characteristically an occurrence of a class. A class depicts a set of objects having similar attributes (properties), meanings, operations (functionality) and types of relationships.
- For example; employee, bicycle, fruit, college are the classes. Every employee working in a particular organization has employee-id, name, department, post, qualification, etc. as attributes that can be used for unique identification of a particular employee in an organization. As discussed in section 5.1.4 of this chapter, motorbike has characteristics like manufacturer, colour, cost, number of gears, etc. Each fruit has name, colour, shape and flavour. College has its own attributes as name, affiliation, type etc.
- When we have to extend the system to some more extent, it is quite necessary to organize all of these contents collectively into large containers. The package is defined as a container that organizes a model by grouping things. Hence, packages helps us to arrange all of the elements collectively for better understanding of the system archetype.

- The package is a widespread mechanism dedicated for organizing all of the building blocks of UML into groups. Normally, packages are used for presenting behavioural and structural views of the software system.
- Semantically associated elements are grouped by means of concept of package. We can have nested packages inside other packages.
- By rule, there should not be cycles in dependency relationships amongst packages involved in the scenario.
- The interfaces of the packages should be stable in case of the numerous dependencies in between packages.
- There are five categories of dependencies amongst packages.
  - ***<<import>>*** : The <<import>> dependency combine client and supplier namespaces and it implements a public merge.
  - ***<<access>>*** : The <<access>> dependency also combine client and supplier namespaces. Only the difference is that, it implements a private merge.
  - ***<<use>>*** : The <<use>> dependency represents the interrelationship between the elements within the packages and not the packages themselves.
  - ***<<merge>>*** : The <<merge>> dependency is used in met modeling only. We should not take into consider this kind of dependency in case of object oriented modeling and design.
  - ***<<trace>>*** : The <<trace>> dependency typically shows the interrelationship between models (archetypes) instead of depicting the relationship between elements of a system.

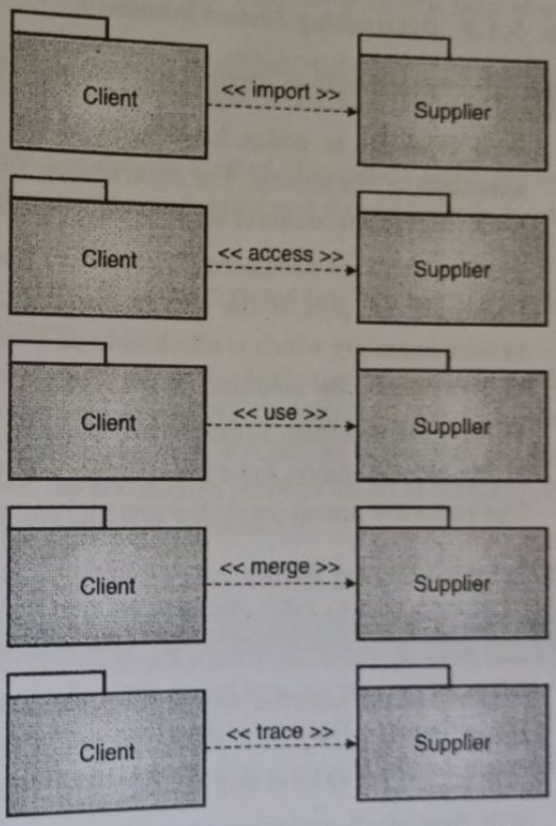


Fig. 5.1.4 : Categories of package dependencies

- We can merge classes from one package into another package by means of dependency. Package diagrams should be used in case of larger systems in order to get a detailed representation of the dependencies amongst key elements of a system.
- Dependencies within a proposed system can be handled effectively with the help of package diagram.
- For example, the detailed model of an ATM Machine can have following packages :
  - **Bank Employee :** Manager, Cashier, ATM Manager.
  - **Bank :** Branch, ATM.
  - **Account :** Type, Customer, Credit/Debit Card, Transaction Processing, etc.

Guidelines for Grouping of Classes into Packages

- A well-organized package is loosely coupled and normally less nested.
- Packages should not be too large for better understanding of the system archetype.
- Simple form of the package (i.e. folder) should be used for representation of a package. Detailed contents of the package (set of classes) should be given on the body of the package in extreme cases only.
- Only meaningful contents (objects, classes, relationships and interfaces) which are important to understand the system model should be shown explicitly.

5.1.9 Determining System Boundary

GQ. Write a short note on : Determining System Boundary.

- It is essential to define boundaries of the system before moving towards the implementation and execution of the system. For determining the system boundary; first of all, software designers should know the specific scope of the proposed software system.
- That is, software designers should define the things which are external to the system and the things which are the part of the system. So, the things which are the part of the system comes inside the system boundary which is also known as a subject while the things which are external to the system are located outside the boundary of the system.
- The system boundary is graphically represented by means of a rectangular box, labelled with the name of the system. Since, actors are external to the system; they are drawn outside the system boundary. Use cases are drawn inside the system boundary.

5.1.10 Finding Actors

GQ. Explain the term finding actors with respect to use case diagram.

- An actor is the external user of the system who is responsible for communication and coordination with the software system. It is not always necessary to have a human being as an actor within the use case scenario. We may have any other element or an external system as an actor.
- It is first of all necessary to recognize the role of an actor for better understanding of an actor. For finding actors of a particular system, following questions should be taken into consideration:
  - Who are the end users of the system?
  - Who are the installers of the system?
  - Who provides information to the system?
  - Is there any other cooperating or interacting system available in the scenario?
  - Who maintains the system?
- From the business point of view, each actor should be named uniquely. Always, there is direct communication and coordination amongst actors and the system.

We may have time as an actor in case we have to model the things that happen at a particular point of time in the system scenario. Actors are always external to the system.

There is direct interaction between actors and the system; i.e., actors can directly communicate and coordinate with the system.

Each actor should be described in short (in one or two sentences) for better understanding of the exact role of an actor within the system.

5.1.11 Finding Use Cases

Explain the term finding use cases with respect to use case diagram.

In UML, the system requirements and functionality of the system are depicted with the help of use cases.

Use cases are meant for specification of the interaction between the system itself and end users of the system which are termed as actors in UML.

Basically, use case offers a detailed description of how the system is used. The set of activities and events in some proper sequence specifying the interaction amongst a system and its end users is known as a scenario.

We might have to handle several scenarios in the execution of the system depending on the situations or scenarios involved within the proper execution of the system.

For instance, in case of an ATM machine, access will be given to authorize customer only by authenticating the particular customer through his/her card and PIN. If login is successful, then customer will be in position to perform the transaction. But, if the login is failed, no access will be given to the customer and customer will not be able to perform the transaction through an ATM machine. So, both of these scenarios are different which are the things that might happen.

All of the scenarios mentioned in the above example are different but are equivalent since the customer has the same goal in all of the scenarios i.e., to perform the transaction. This goal is only the main component behind definition of use cases.

A use case is defined as a set of scenarios that collectively work to achieve a common user goal. It outlines a sequence of interactions amongst one or more actors and the system itself.

In the above example, transaction processing can be one of the use case that strives for the successful transaction processing that is the goal of the customer.

Each use case comes with its primary actor who is responsible for certain tasks involved in the scenario. More details of the actor are discussed in subsequent section of this chapter.

Each use case should clearly mention the exact interaction between the actors and the system itself.

Furthermore, use case guides us about what the system exactly does in response to the actor's activity and it is not devoted for detailing how system does it.

At all times, a use case starts with input from a respective actor. Thus, an actor is responsible for giving input the system and the system is dedicated for giving response to the actor.

A simple use case encompasses a single interaction between the actor and the system. But, a single use case can handle set of interactions between a particular actor and the system. More than one actor can be a part of the complex use cases.

- Graphically, a use case is shown with the help of an oval shape containing the use case name inside its body. Each use case should be named uniquely and use case name should describe the desired functionality of that particular use case.

5.1.12 Finding Initial and Final Events

GQ. Define event. Define and describe initial and final events for ATM system.

- Things that occur at a particular instance of time are known as events. Every single thing or action occurred at a particular time instance is termed as an event in UML.
- An event owns an activity or operation along with its time and location. As far as use case model is concerned, the proposed software system can be divided into small pieces on the basis of use cases.
- In use case model, use cases specifies the flow of execution of the system along with respective actors but it do not depict behaviour. So, events are used in order to represent the behaviour.
- The order of execution of events is better for understanding the behaviour of the system. Each use case involved in the scenario should be covered by means of sequence of events.
- From the use case model, we can define initial and final events by means of actors and use cases defined. Normally, each use case is dedicated for provision of some kind of service in the system scenario for proper execution of the software system.
- Defining initial events is simply nothing but the demand of service from respective use cases during execution.
- Sometimes, initial event causes execution of several activities in proper sequence that leads to fruitful outcome from the system. Same as that of the initial event, it is quite essential to define final event too.
- The final event simply determines the boundary line for a particular activity or set of activities in some sequence. i.e.; it tells us where to stop or where to terminate. Let us define initial and final events for ATM example.
- Main activities involved in the ATM are: Transaction initiation, Data transmission and Transaction processing.
- In case of transaction initiation, the initial event can be insertion of ATM card into the provided slot in the ATM machine. Final events can be either ATM machine holds ATM card in the slot till the end of the transaction or ATM returns the ATM card if PIN entered by the customer is invalid.
- During data transmission, initial event is customer requests account data and hence customer initiates the transaction while the final event is ATM machine communicates with the bank server, retrieves customers' account data and displays the same on ATM screen.
- During transaction processing, initial event is actual transaction initiation and final event can be transaction completion or transaction termination.

5.1.13 Preparing Normal Scenarios

GQ. What do you mean scenario. What is the purpose of defining the scenario? Explain with suitable example.

- A scenario is nothing but the ordered set of events or activities that occurs in between objects and it illustrates behaviour. Scenario can be defined as a definite path through a use case.

As discussed earlier, when internal objects communicates and coordinates with the external objects, an event occurs. So, certain message interaction amongst objects in the interaction model is considered as a scenario and it is not a use case. Each use case may comprise of several scenarios in case of a complex system.

Crucial or necessary sequences involved in a particular use case are considered as primary scenarios and alternative or substitute sequences are considered as secondary scenarios for each use case in the system archetype.

The data values involved in the activity are termed as event parameters. For example, in "enter username and password" activity; username and password are event parameters and these event parameters are exchanged amongst internal elements of the system and external agents.

Let us discuss scenarios for some of the use cases involved in the ATM machine example.

Scenario for transaction initiation comprises of following sub tasks :

1. The ATM machine shows Welcome message on the screen.
2. The ATM requests the customer to insert an ATM card into the slot.
3. The customer inserts an ATM card.
4. The ATM machine accepts the card and scanner inside the ATM machine scans and reads the serial number of an ATM card.
5. The ATM machine asks customer to enter the valid PIN.
6. The customer enters valid PIN.
7. The ATM machine accepts the PIN and communicates the same with the central server of the bank in order to validate the customer.
8. After validation, the ATM machine displays the main menu with the help of which the customer can initiate, process and terminate the transaction.

Data transmission scenario consists of :

1. The ATM machine accepts input from the customer.
2. The ATM machine requests customers' bank account data from the banks' central server.
3. The respective bank's server receives the request of account data from the customer.
4. The bank server retrieves the account data as per the customer's requirement/query.
5. The bank server then transfers the account data to the ATM machine.

Actual transaction processing scenario comprises of :

1. The ATM machine shows a main menu for transaction processing.
2. Let us say, the customer selects money/cash withdrawal option on the ATM machine screen.
3. The ATM machine asks for the amount of cash to be withdrawn.
4. The customer will then enter amount of cash; for e.g., 1000 Rs.
5. The ATM machine then confirms policy limits for cash withdrawal.
6. The ATM machine connects to the banks' central server and verifies that, the particular customers' bank account has adequate cash.
7. The ATM machine dispenses the cash and requests the customer to collect it.
8. The customer then collects the cash and gets the printed receipt of transaction.

- Subsequently, we have seen three use cases from the ATM machine which are : transaction initiation, data transmission, actual transaction processing and defined their scenarios.

5.1.14 Adding Variation and Exception Scenarios

- After defining normal scenarios, software system designers should give focus on special cases involved in the system archetype.
- The special cases contained in the system archetype can be lowest or extreme values, repetitive values, erroneous input or misplaced input and many more.
- In addition to that, error cases like illegal values, failures at particular stage, etc. should also be considered.
- For instance, the ATM machine may have following exceptions and variations for which we can define scenarios :
  - Card Read Error: The ATM card is not readable.
  - The ATM card is damaged and is not working.
  - The amount entered by the customer is invalid.
  - The ATM machine times out and waiting for response.
  - The ATM machine is out of cash.
  - The ATM machine is down/not working.
- Let us have a look at the scenario for one of the exception: Card read error.
  - The ATM machine shows Welcome message on the screen.
  - The ATM requests the customer to insert an ATM card into the slot.
  - The customer inserts an ATM card.
  - The ATM machine accepts the card and scanner inside the ATM machine scans and reads the serial number of an ATM card.
  - If the ATM card is damaged or is not working, the card read is displayed on the screen.
- In the same manner, we can write scenarios for remaining exceptions and variations mentioned above. For practice, students should try to write scenarios for the above exceptions and variations.

5.1.15 Finding External Events

- Previously, we have already discussed how to find initial and final events.
- Now, we have to define external events. Software modellers should start specification of external events with the help of scenarios defined.
- The scenarios will acts as a baseline for defining the external events.
- All types of inputs to the software system, possible interrupts, decisions and interactions between end users and external devices are useful for defining the external events.
- For better understanding, software modellers should design sequence diagram for each of the scenario.
- A sequence diagram should be used when we have to depict the flow of control within a system by time ordering of messages amongst objects.

A sequence diagram should contain only important objects, nodes, components, collaborations or actors.

In short, the sequence diagram evidently illustrates the sender and receivers of each event.

Fig. 5.1.5 shows the sequence diagram for actual transaction processing which is one of the scenarios that we have discussed in earlier sections of this chapter.

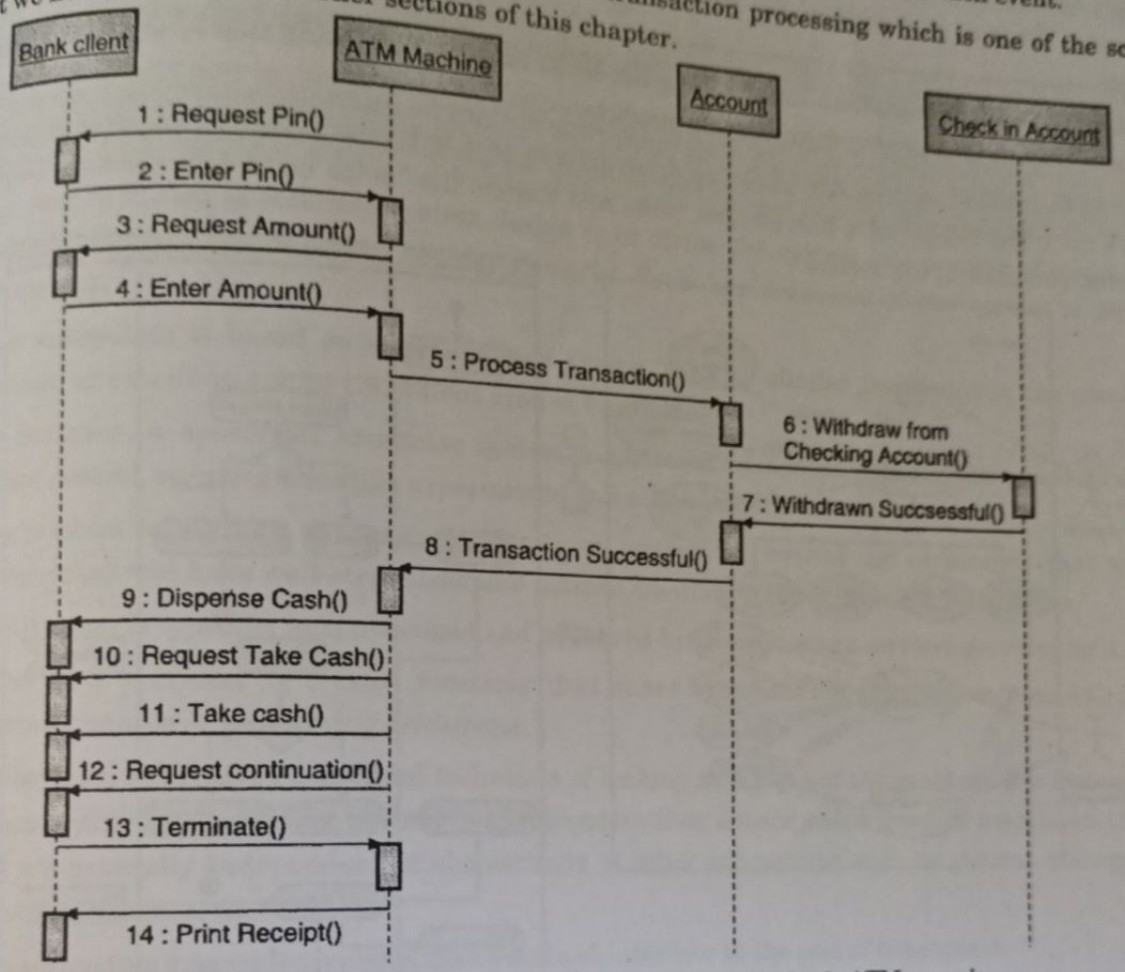


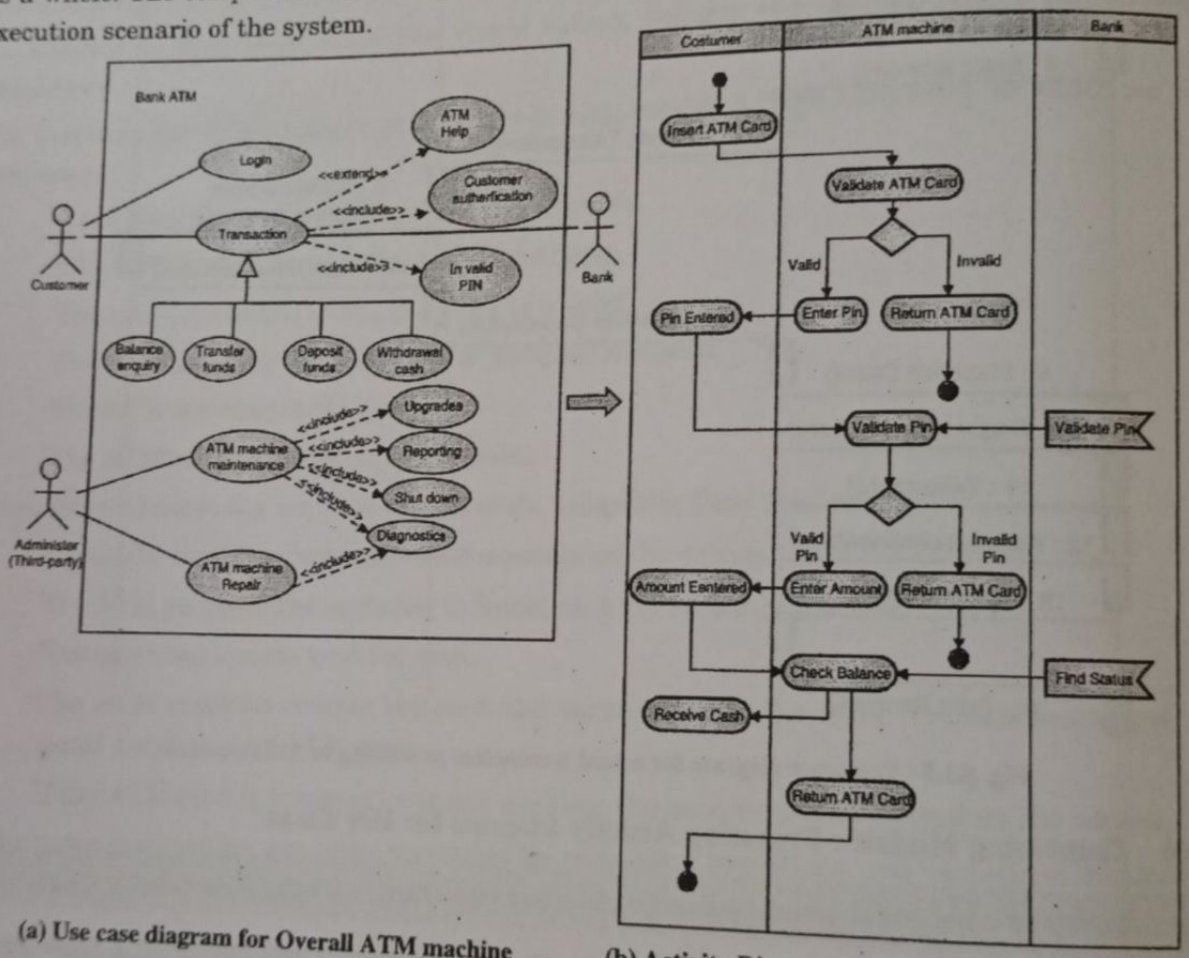
Fig. 5.1.5 : Sequence diagram for actual transaction processing in ATM scenario.

5.1.16 Combining Models : Preparing Activity Diagram for Use Cases

GQ. Explain in detail the process of preparing an activity diagram from use case diagram. Give suitable example.

- An interaction diagram comprises of set of objects, their relationships and messages sent amongst them and are mainly devoted for showing interaction between the objects.
- An interaction diagram is meant for showing communication and coordinate recursing on between two or more objects but, data manipulation is not shown. The interaction diagrams typically comprises of sequence diagram and collaboration diagram.
- The sequence model or sequence diagram is a type of interaction diagram that primarily focuses on time ordering of messages. The objects in the sequence diagram are not only the instances of class but they can be instances of elements like nodes, components and collaborations involved in the scenario.
- The sequence diagram uses the object timeline in order to specify the time ordering of the messages between objects. But, decision points and alternatives are not clearly shown in sequence diagram.

- That is, software modellers have to draw individual sequence diagram for each of the scenario involved in the system archetype. For instance, actual flow of interaction within the system will need separate sequence diagram and additionally we will need several sequence diagrams for decision points and errors.
- In order to tackle this problem, activity diagram helps us for representing step by step flow of execution of the system by considering the system as a whole.
- Activity diagram focuses on flow of control from activity to activity and hence it considers entire activity as a whole. The components like merges, forks and control flow helps us to depict overall step by step execution scenario of the system.



(a) Use case diagram for Overall ATM machine (b) Activity Diagram for overall ATM machine

Fig. 5.1.6 : Activity diagram from Use case diagram

- An activity diagram can be used to provide a detailed description of the use case. One or more sequential steps of a particular use case can be represented by means of an activity node.
- Normally, a high level activity node is used to show a single use case and then it can be separated into a separate activity diagram. Correspondingly, an activity diagram is a good option for showing proper ordering between use cases involved in the system scenario.
- So, use cases in use case diagram are replaced by activity nodes in the activity diagram, relations like association, generalization, include and extend in use case diagram are replaced by loops, arcs and decision points depending on the situation in the system scenario. An activity node depicting a use case can also be used for showing a link to an inclusion use case or exclusion use case.

After decomposition of a high level activity node, we can get low level activity diagram showing precise flow of a particular activity. An example of use case diagram and activity diagram for overall ATM machine is shown in Fig. 5.1.6.

5.2 System Design : Organizing a System into Subsystems

Q. What do you mean by System Design? Illustrate with suitable example.
Q. How can we organize a system into subsystems? Explain in detail.

- The very first step in software system design is to divide the system into number of pieces for better understanding of the software system scenario. Each key fragment of the system is known as a ***subsystem***.
- Each subsystem is based on some common theme, such as similar functionality, the same physical location, or execution on the equivalent kind of hardware.
- For instance, a spacecraft computer system might contain subsystems for life support, navigation, engine control, running scientific experiments and other activities.
- A subsystem is a group of classes, associations, operations, events and constraints that are unified, incorporated and have well-structured and defined small interface with other subsystems.
- A subsystem is typically acknowledged and approved by the numerous services provided by it.
- A service is a cluster of related functions that share some mutual purpose, such as I/O processing, drawing pictures or performing arithmetic.
- A subsystem articulates a rational technique of looking at a piece of the problem. For instance, the file system within an operating system is a subsystem that encompasses a set of associated abstractions that are generally independent of abstractions in other subsystems such as process management and memory management.
- Each subsystem has well-structured and defined interface to the rest of the system.
- Layers or Partitions can be used in order to divide the complete software system into subsystems in horizontal or vertical manner respectively.
- That is, software designers can organize the subsystems horizontally by means of layers and vertically by means of partitions.
- In both of these cases, all layers or partitions involved within a software system are interrelated with each other and interrelationships amongst these layers or partitions is of the type client server relationship.
- The key difference between partitions and layers is that, the subsystems defined with the help of partitions will have identical and same levels of abstraction whereas the subsystems defined by means of layers will have different levels of abstraction.
- One more difference is partitions are having peer to peer relationship amongst them and layers are having client server relationship in between them. Consequently, each partition can be considered as a self-governing peer and each layer is dependent on some other layer or group of layers.
- Furthermore, we can combine partitions and layers in order to get a software system as a whole. In conclusion, we can layer the partitions and partition the layers.

5.2.1 Identifying Inherent Concurrency in a Problem

- The state model is dedicated for detailed description of sequence of tasks and services involved within the system operations and do not deals with the contents like details of operations and services, how different services and operations are executed during implementation and execution of the proposed system, etc.
- The state model describes those aspects of objects concerned with time and the sequencing of operations. When two objects can get the events simultaneously deprived of interfacing, they can be said inherently concurrent.
- As far as implementation is concerned, all of the software objects are not concurrent, since one processor can support many objects. The state model is basically articulated by means of a state diagram.
- So, with the help of a state diagram, several objects involved in a scenario can be congregated into a solitary thread of control. A thread of a control is an end to end path through a set of state diagrams of discrete objects and only single object can be active at a particular instance of time.
- For instance, in case of an ATM system, the ATM machine is idle; when the bank is authenticating and confirming an account or processing bank transaction. If the central computer system proximately controls the ATM, the bank transaction object and an ATM object can be associated with each other as a solitary task.

5.2.2 Allocation of Subsystems to Hardware

- The necessity of advanced or superior performance basically forms the basis for the decision to use various processors or hardware functional units. The number of essential processors required totally depends on the speed of the machine and the volume of calculations.
- For instance, a parallel computing system produces too much data in a very short time span in order to handle in a solitary central processing unit. Many parallel machines should abstract the information or data handled within a system well before analysing a threat.
- The software system designer should assess and calculate the mandatory CPU processing power by means of computing the stable state load as the product of the number of transactions per second and the time required to process the particular transaction. The estimate will ordinarily be inaccurate or imprecise.
- The estimate should be increased in order to tolerate the transient effects due to random variations in load in addition to the synchronized bursts of activity.
- The sum of surplus capacity required depends on the adequate rate of failure due to inadequate resources. In case of an ATM system example, the ATM machine is mainly responsible for actual online transaction processing and providing a user interface to customer for communication and coordination amongst customer and ATM system.
- A solitary central processing unit is sufficient for an individual ATM machine. The main server of a bank can be considered as a routing machine fundamentally since it receives ATM requests and communicates them to proper bank computer situated at a particular branch of a bank.

- Sometimes, a large network may comprise of multiple processors in case of an ATM system scenario. The computer systems situated at the bank branch accomplish data processing and encompass database applications.
- Each device is an object that operates synchronously along with other objects these other objects may be hardware units or devices involved in a system.
- The software system developers should decide and agree upon the fact that, which subsystems will be implemented in software and which will be executed in hardware.
- Two reasons behind implementing subsystems in hardware are cost and performance. Much of the trouble of designing a software system comes from accomplishing externally enforced software and hardware constraints.
- Object oriented design affords no magic solution however the external packages can be demonstrated as objects. The software system designers should take into account the cost, compatibility and performance issues.
- Correspondingly, the system designers must think about flexibility for future modifications or alterations. The future modifications can be future software product versioning or improvements and software product design variations. In case of an ATM system example, no such performance issues are occurred.

5.2.3 Data Storage Management

GQ. Explain data storage management in software modeling.

- Number of substitutes is available for managing the data storage that we can use distinctly or in combination. For instance, databases, files, data structures, etc.
- Different types of data stores offer trade-offs between cost, access time, capacity, cost, reliability and access time.
- For instance, a personal computer system application can make use of files and data structures as per its requirement, an accounting can use a database to connect subsystems.
- Files are simple, modest, permanent and inexpensive.
- Implementations for sequential files are ordinarily standard however storage formats and commands for random-access files and indexed files fluctuate.
- Following kind of data is appropriate and can be suitable for files:
  - Sequentially accessed data.
  - Data that can be wholly read into memory.
  - Data with low data density.
  - Uncertain data with modest structure.
  - Data with high volume.
- Databases are the other type of data store which are typically managed, monitored and controlled by means of database management systems.
- Several types of database management systems like relational databases and object oriented databases are available.

- Following kind of data is appropriate and can be suitable for databases :
  - Larger amount of data that should be handled proficiently and skillfully.
  - Data that synchronized the updates through transactions.
  - Data that needs updates at satisfactory levels of detail by several users.
  - Data that must be accessed by numerous application programs.
  - Data that should be protected against malicious access.
  - Data that is long-lasting and highly valuable to an organization.
- The software system designers must consider object oriented database management systems merely for specialty domain applications that have a wide variety of data types or that must access low level data management primitives.
- These software system applications can be multimedia applications, embedded system applications, different engineering applications and many more.
- The software system designers should make use of relational database management systems for the software system applications that requires databases since the features of relational database management systems are adequate and satisfactory for utmost software system applications.
- Furthermore, if relational database management systems are used appropriately, they can offer a super execution, an object oriented archetype.

5.2.4 Global Resources Handling

GQ. Explain in brief : Global Resource Handling.

- The software system designer should recognize the global resources.
- The categories of global resources are mentioned in Table 5.2.1 :

Table 5.2.1: Types of global resources

Types of Global Resources	Examples
Physical Units	Processors, tape drives, communication satellites.
Space	A workstation screen, the buttons on a mouse.
Logical Names	Object IDs, filenames, class names.
Access to Shared Data	Databases.

- A physical unit like processor when considered as a resource, it can regulate and control overall activities involved in the scenario on its own. A global resource may also be segregated for self-governing control.
- The buttons on a mouse, a workstation screen can be the global resources of the type space. The entities dedicated for unique identification of a particular object in a given scenario can be deliberated as resources. For instance, class names in class model, object IDs or object names in object model, file names in file management system are the logical entities that are used as resources.

In case of an ATM machine system, the account numbers of customers and the bank codes are global resources. Bank codes should be unique within the context of a bank.

5.2.5 Control Implementation : Choosing a Software Control Strategy

GQ. Explain : (a) External control (b) Internal control

Basically, there are two types of control in a software system:

1. External control 2. Internal control

The flows of the events between the objects involved in the software system scenario which is visible from outside are termed as **external control flows.**

However, the control flow comprised by a process is known as an **internal control flow.**

5.2.5.1 External Control

GQ. What are the different categories of external control? Explain in brief.

Moreover, there are three types of control for external events:

1. Procedure-driven Control 2. Event-driven Control 3. Concurrent Control

The choice of control flow totally depends on the existing resources involved in the software system application.

1) Procedure-driven Control

- In a procedure-driven sequential system, control exists within the software program coding.
- The procedure driven control is quite easy to implement by means of conventional languages. It is the honest responsibility of a software system designer to translate events into operations among objects.
- The drawback of procedure driven control is, concurrency inherency is significant amongst objects involved in a scenario.

2) Event-driven Control

- An event-driven control is associated with the circumstances where the measurement method is inherently event-based in nature.
- Event-driven control offers adaptable and compliant control.
- From implementation point of view, it is more challenging to implement as compared to the procedure-driven control.
- As far as modularity of a software system is concerned, event-driven control flow supports for additional modularity for breaking of a software system into subsystems.

3) Concurrent Control

- Concurrent control guarantees that, the respective transactions involved within a scenario are accomplished and executed concurrently devoid of violating the integrity of data and hence, data remains whole, complete and uninterrupted within a scenario. It is also known as yes-no control or screening.

- Ultimately, there are three basic classes of concurrent control mechanisms : pessimistic, semi-optimistic and optimistic.
- Pessimistic concurrent control refers to the blocking of a transaction if that particular transaction violates the rules. Semi-optimistic concurrent control blocks transactions in some circumstances that may cause violation and optimistic concurrent control do not blocks the transaction but it postpones the respective transaction.

5.2.5.2 Internal Control

- Throughout the software system design, the software system developer expands operations on objects into lower-level operations on the same or other objects.
- Software system designers can make use of identical system execution mechanisms for developing internal objects, external objects and communication and coordination amongst internal and external objects.

5.2.6 Handling Boundary Conditions

- Three issues should be deliberated while handling boundary conditions which are initialization, terminati. and failure.
- At the outset, the software system should initialize parameters, constants and variables.
- Termination is generally modest as compared to the initialization since many internal objects can merely be unrestricted. A particular executing transaction or task should release the resources.
- Failure is the unintended closure of a software system. Usually, software development team members recognize that, a proposed developed software system is not working as per the requirements mentioned in the software requirements specification.
- The failures are candidly observed by software testers during thorough and systematic software testing of a software system. Misbehaviour in the execution of a software system can be considered as an indication of the failure.

5.2.7 Setting Trade-off Priorities

- The trade-off priorities should be established for the good software system design. It is the responsibility of software system designer to set trade-off priorities for a software system. The task of designing trade-off priorities encompasses several types of software development techniques.
- If customer needs a delivery of a particular software module earlier than the outstanding software modules then customer should sacrifice the overall functionality of the software system as a whole.
- It is the duty of the software system designer to define the comparative significance of the several criteria as a guide for creation of design trade-offs.
- The software system designer does not form all the adjustments but launches the priorities for constructing respective software system arrangements.

5.2.8 Selecting an Architectural Style

GQ. Write a short note on: Architectural styles.

- Numbers of architectural styles are commonly used in existing software systems.
- Following are some types of architectural styles :

1. Batch Transformation	2. Continuous Transformation	3. Interactive Interface
4. Dynamic Simulation	5. Real Time System	6. Transaction Manager

5.2.8.1 Batch Transformation

GQ. What do you mean by batch transformation?

- In batch transformation, the information transformation is executed once on a complete input dataset. This architectural style accomplishes sequential computations.
- The main objective is to calculate an answer and is achieved by the application which is meant for receiving the inputs. Stress analysis of a bridge, payroll processing are the classic applications of batch transformation.
- In batch transformation, software developers should first of all breakdown the complete transformation into stages with each stage accomplishing one part of the particular transformation which is then followed by Formulation of class models (class diagram) for the input, output and in between each pair of successive stages.
- Next step involved is, expansion of each phase or level until the operations are straightforward to implement and finally reorganize the ultimate pipeline for optimization.

5.2.8.2 Continuous Transformation

GQ. What do you mean by continuous transformation?

- It is a system in which the output of the system is aggressively dependent on varying inputs. An uninterrupted transformation updates system outputs frequently. Windowing systems, signal processing are the classic applications of continuous transformation.
- In continuous transformation, the first step involved is to breakdown the complete transformation into stages with each stage accomplishing one part of the transformation. Then, we should describe and summarize input, output and intermediate models amongst all of the pairs of successive stages, as for the batch transformation.
- After successful breakdown of the complete transformation and defining inputs and outputs, we should discriminate each operation in order to update the incremental modifications or alterations to each level. Finally, we have to add transitional objects for optimization purpose.

5.2.8.3 Interactive Interface

GQ. Explain : Interactive interface

- It is a system that is conquered by interactions amongst the external agents and the system itself. The external agents can be devices or humans. These external agents are self-governing and are independent of the system, so the system cannot control the agents.

- While using an interactive interface as an architectural style, we should first of all Segregate interface classes from the application classes.
- Predefined classes should be used for communication and coordination amongst the external agents. Next, we should separate out the logical events from physical events. Logical events correspond to multiple physical events. At last, we should identify and state the application functions that are invoked by the interface.

5.2.8.4 Dynamic Simulation

GQ. Explain : Dynamic simulation

- This architectural style is dedicated for designing and modeling real world objects. Video games can be the classic examples of this type.
- The internal objects in the dynamic simulation correspond to real world objects and hence the class model (class diagram) is ordinarily significant.
- While selecting a dynamic simulation as an architectural style, we should diagnose active real-world objects along with the discrete events that relates to discrete interactions with the object from the class model (class diagram). Constant dependencies should also be predicted.

5.2.8.5 Real Time System

GQ. Explain : Real time system

- The real time system is an interactive system with close-fitting or tight time constraints on actions.
- Real time system design is multifaceted and encompasses issues like interrupt handling, coordination of multiple central processing units, etc.

5.2.8.6 Transaction Manager

GQ. Explain : Transaction manager

- It is a system dedicated for retrieval and storage of data. The transaction manager deal with several users who write and read data at the same time that is, concurrently.
- Data should be protected from unauthorized access and accidental loss. Inventory control, airline reservations, order fulfilment are the classic examples of the transaction manager.
- Steps involved in the transaction manager are :
  1. Map the class model to the database structures.
  2. Determine the units of concurrency.
  3. Determine the unit of transaction.
  4. Design concurrency control for transactions.

5.2.9 Designing Algorithms

- As far as functional model is concerned, an algorithm should be defined and designed for each of the function or operation involved within a proposed software system.
- The software system analysis specification refers to the exact functionality or operation involved within a particular function while an algorithm tells us how that functionality or operation can be achieved.

- In order to define an algorithm, **following steps should be followed**:
  1. Select an algorithm in such a way that, it reduces the implementation costs of a particular function or operation.
  2. Suitable data structures should be selected and used for defining and implementing an algorithm.
  3. New internal classes and operations should be defined and designed if it is essential.
  4. Assign operations to suitable classes involved within a software system scenario.
- So, initially it is quite essential and is the best practice to define and design detailed algorithms for each and every functionality and operation of a proposed software system.

5.2.10 Design Optimization

- After designing an algorithm, next step is to implement an algorithm and optimize it. The system analysis archetype is used as the basic framework for implementation of a proposed software system.
- The system design model supports system analysis model in order to formulate the logical information about the proposed software system.
- Design optimization plays a vital role in the software system modelling by means of design of a correct logic which is then followed by implementation and execution of a particular software system.
- Every module or part of a software system involved within a scenario is quite important for implementation and proper execution of a particular functionality or operation involved within a software system. Basically, the system analysis model provides a base for the system design model.
- The system analysis model refers to the logic of a particular software system whereas the system design model is dedicated and meant for actual implementation and execution of a proposed software system.
- Following activities are involved in the process of design optimization and these activities should be attained in order to accomplish design optimization :
  1. Effective access paths should be made available.
  2. In order to achieve superior efficiency, the computation should be reorganized.
  3. In order to avoid recomputation, intermediary results obtained from the software system should be maintained.

Key Concepts

• System Analysis	• Static object modeling
• Dynamic object modeling	• Software system analyst
• Noun/verb analysis	• Class, Responsibilities & Collaborations (CRC)
• CRC analysis	• Rational Unified Process (RUP)
• Entity class	• Control class
• Boundary class	• Data dictionary
• Association	• Inheritance

• Multiple inheritance	• Mix-in inheritance
• Abstraction	• Package
• Import dependency	• Access dependency
• Use dependency	• Merge dependency
• Trace dependency	• Actor
• Use case	• Initial event
• Final event	• Scenario
• External event	• Design Optimization

Summary

- The static and dynamic archetypes of the proposed software system are designed in the system analysis phase.
- The main motive behind ***system analysis*** is to find out real-world objects and entities in the proposed system scenario and then plotting these objects as the software objects in the system scenario.
- The static object modeling in object oriented analysis comprises of object modeling and class modeling.
- The dynamic object modeling in object oriented analysis consists of state modeling, activity modeling, use case modeling and interaction modeling.
- A class can be defined as a detailed explanation of a group of objects that share the equivalent attributes, relationships and operations.
- A class of a particular object is a fundamental feature of that object.
- For each object, it is possible to identify and distinguish its own class.
- For finding the classes, no stepwise procedure or algorithm is available in the literature.
- So, analysis of classes is the whole sole responsibility of the personality known as ***software system analyst***. The experienced software system analysts may help out for defining the classes of the proposed software system in right manner.
- In the procedure of ***noun/verb analysis*** for finding out the classes, noun phrases or nouns are dedicated for defining classes themselves or for defining attributes of respective classes and verb phrases or verbs depicts operations or responsibilities of the respective classes.
- CRC stands for Class, Responsibilities and Collaborations.
- CRC analysis strategy should be used along with noun/verb analysis process.
- CRC analysis is a two-step procedure: Collecting information and Analyzing information.
- RUP stands for Rational Unified Process.
- Concept of ***analysis class*** is considered in ***RUP***.
- ***Analysis classes*** are the classes that signify an abstraction in the proposed system domain and are mapped to the real-world entities.

- Entity class, control class and boundary class are the types of analysis classes.
- ***Entity class*** gives determined information about the particular entity involved within the system scenario.
- ***Control class*** summarizes use case modeling and helps in describing use case model in brief.
- ***Boundary class*** is used as a facilitator between proposed system and outside environment. It provides a platform for communication and coordination between the system and its environment.
- ***Data dictionary*** is the phrase related to 'data' and is the thing important for modeling all of the components involved in the system scenario.
- The data dictionary specifies attributes, responsibilities, operations and associations.
- ***Association*** is a structural relationship amongst two classes and is static in nature. It shows fixed relationships between classes involved in a system.
- ***Associations*** can be mentioned in terms of ***verb phrases*** or ***verbs***.
- ***Inheritance*** is the mechanism with the help of which specific elements can obtain their behavior and structure from general elements involved within the system scenario.
- For organizing and simplifying classes with the help of inheritance, system analyst should first of all look after responsibilities, operations, and attributes that are common to two or more classes in a group of classes.
- It is possible for a class to have more than one superclass in Unified Modeling Language. That means, a child can have more than one parent and this property is known as ***Multiple Inheritance***.
- ***Multiple inheritance*** is not supported by all of the object oriented languages. C# and Java permits only single inheritance.
- A child class can inherit properties of its superclasses in multiple inheritance.
- Normally, superclasses should not have a parent class in common in order to avoid cycles in the inheritance hierarchy.
- All the superclasses involved in the scenario should be semantically disjoint.
- Multiple inheritance permits mix-in inheritance.
- The main objective of an ***abstraction*** is to segregate those facets of a particular problem which are essential for some purpose and destroy those features which are irrelevant.
- ***Abstraction*** should be purpose specific, for the reason that the purpose defines what is important, and what is not important.
- An abstraction signifies the crucial features of an object that differentiate it from all other types of objects.
- The ***package*** is a widespread mechanism dedicated for organizing all of the building blocks of UML into groups.
- Normally, packages are used for presenting behavioral and structural views of the software system.
- The ***<<import>> dependency*** combine client and supplier namespaces and it implements a public merge.

- The ***<<access>> dependency*** also combine client and supplier namespaces. Only the difference is that, it implements a private merge.
- The ***<<use>> dependency*** represents the interrelationship between the elements within the packages and not the packages themselves.
- The ***<<merge>> dependency*** is used in metamodeling only. We should not take into consider this kind of dependency in case of object oriented modeling and design.
- The ***<<trace>> dependency*** typically shows the interrelationship between models (archetypes) instead of depicting the relationship between elements of a system.
- For finding actors of a particular system, following questions should be taken into consideration :
  - Who are the end users of the system?
  - Who are the installers of the system?
  - Who provides information to the system?
  - Is there any other cooperating or interacting system available in the scenario?
  - Who maintains the system?
- From the business point of view, each actor should be named uniquely.
- Actors are always external to the system.
- In UML, the system requirements and functionality of the system are depicted with the help of use cases.
- ***A use case*** is defined as a set of scenarios that collectively work to achieve a common user goal. It outlines a sequence of interactions amongst one or more actors and the system itself.
- Use cases are meant for specification of the interaction between the system itself and end users of the system which are termed as actors in UML.
- Use case offers a detailed description of how the system is used.
- More than one actors can be a part of the complex use cases.
- Things that occur at a particular instance of time are known as ***events***.
- An event owns an activity or operation along with its time and location.
- Defining ***initial events*** is simply nothing but the demand of service from respective use cases during execution.
- The ***final event*** simply determines the boundary line for a particular activity or set of activities in some sequence. i.e.; it tells us where to stop or where to terminate.
- A ***scenario*** is nothing but the ordered set of events or activities that occurs in between objects and it illustrates behavior.
- Scenario can be defined as a definite path through a use case.
- ***Activity diagram*** helps us for representing step by step flow of execution of the system by considering the system as a whole.
- An activity diagram can be used to provide a detailed description of the use case.
- Normally, a high level activity node is used to show a single use case and then it can be separated into a separate activity diagram.

- Use cases in use case diagram are replaced by activity nodes in the activity diagram, relations like association, generalization, include and extend in use case diagram are replaced by loops, arcs and decision points depending on the situation in the system scenario. An activity node depicting a use case can also be used for showing a link to an inclusion use case or exclusion use case.
- We can decompose a system into subsystem by merging partitions and layers.
- Partitions can be layered and layers can be partitioned.
- All objects are concurrent or synchronized in the system analysis model, as in the real world and in hardware.
- As far as implementation is concerned, all of the software objects are not concurrent, since one processor can support many objects.
- Two objects are inherently concurrent if they can receive the event at the same time deprived of interacting.
- A ***thread of control*** is a path through a set of state diagrams on which only a single object is active at a time.
- A thread remains within a state diagram unless and until an object sends an event to another object and waits for another event.
- The thread passes to the receiver of the event until it ultimately returns to the original object.
- The thread splits if the object sends an event and continues executing.
- On each thread of control, only single object is active at a time.
- We can implement thread of control as a particular task in computer system.
- The number of essential processors required totally depends on the speed of the machine and the volume of calculations.
- Numbers of substitutes are available for managing the data storage that we can use distinctly or in combination. For instance, databases, files, data structures, etc.
- Different types of data stores offer trade-offs between cost, access time, capacity, cost, reliability and access time.
- ***Hardware control*** closely matches the analysis model however there are quite a lot of ways for implementing and executing control in a software system.
- ***Internal control*** concerns the flow of control contained by a process.
- In a ***procedure-driven sequential system***, control exists within the program code.
- Procedure request external input and then wait for it; when input arrives, control resumes within the procedure that made the call.
- The location of the program counter and the stack of procedure calls and local variables define the system state.
- The main benefit of procedure-driven control is that, it is quite easy to implement with conventional languages while the drawback is that it necessitates the concurrency inherent in objects to be mapped into a sequential flow control.
- In an ***event-driven sequential system***, control exists within a dispatcher or monitor that the language, subsystem, or operating system offers.

- In a ***concurrent system***, control exist simultaneously in several independent objects, each a separate task.
- Internal object interactions are similar to external object interactions since we can make use of the same implementation mechanisms.
- The software system designer should set priorities that will be used to guide trade-offs for the rest of the software system design.
- Number of architectural styles are commonly used in existing software systems :
  - Batch Transformation
  - Continuous Transformation
  - Interactive Interface
  - Dynamic Simulation
  - Real Time System
  - Transaction Manager
- In ***batch transformation***, the information transformation is executed once on a complete input set.
- A ***continuous transformation*** is a system in which the output of the system is aggressively dependent on varying inputs. A continuous transformation updates outputs frequently.
- An ***interactive interface*** is a system that is conquered by interactions amongst the external agents and the system itself. The external agents can be devices or humans.
- ***Dynamic simulation*** tracks or models real world objects.
- The ***real time system*** is an interactive system with close-fitting or tight time constraints on actions.
- A ***transaction manager*** is a system dedicated for retrieval and storage of data.

Chapter Ends...

❑❑❑

Unit VI

CHAPTER 6

Design Pattern

University Prescribed 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.

Specific Instructional Objectives :

At the end of this lesson the student will be able to :

- Define Design Pattern.
- Document Design Pattern.
- Explain Relationships between patterns.
- Describe Patterns.
- Study and apply Communication Patterns: Forwarder-Receiver;
- Client-Dispatcher-Server; Publisher-Subscriber.
- Study and apply Management Patterns.
- Explain about Command processor.
- Describe View handler.
- Understand Idioms.
- Describe what can idioms provide?
- Explain Idioms and style.
- Describe where to find idioms.
- Explain Counted Pointer example.

6.1 INTRODUCTION TO DESIGN PATTERN

Q. Define : Design Pattern.

Basically, the object oriented software system design and development is quite rigid because of the activities involved for finding out the objects, classes, interfaces involved in the scenario and hence to discover proper relationships among these components in order to deliver the expected outcome as per the end user's requirements.

The software system design should be well specified and quantified and it should be more general to handle the requirements and problems in future.

The reuse of the software system is also a difficult task as compared to the basic software system design.

Design pattern is the concept that helps software designers for reusing the successful and fruitful architectures and designs of the software system.

With the help of design patterns, the software developers can evade the efforts required for reproducing some part of design or architecture while designing the proposed software system.

Design pattern provides a platform to apply the expertise and knowledge of other software developers to our precise problem.

Also, we as a software developer can communicate our expertise and knowledge with the software development community.

Each design pattern is simply nothing but the explanation of a specific method or technique that has already ascertained its effectiveness in the real world.

Nowadays, the design patterns are applicable to different areas like websites, distributed computing, concurrency, reuse, organizations, problem solving methodologies, etc.

Design pattern supports system designers in order to find out and select a specific design alternative from the available set of designs and hence software system designers can choose a particular design pattern for a specific problem.

At the outset, the design pattern was not concerned to the software engineering.

In 1960s, Christopher Alexander who was the architect of buildings by profession had written something about design patterns and architectures for urban planning.

Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice".

- Though, Alexander has stated the concept of design pattern with respect to the urban planning; it is applicable to the object oriented methodology too; for instance, doors and walls in building are replaced by interfaces and objects in object oriented methodology.
- A design pattern basically comprise of following elements :
  - Design pattern name
  - Problem
  - Solution
  - Consequences
- *Design pattern name* defines a design problem, solution and its consequences.
- *Problem* typically illustrates the conditions or situations in which the particular design pattern should be applied.
- *Solution* does not give a specific solution or implementation of a problem but, it explains the elements that forms the design, collaborations, relationships and responsibilities.
- *Consequences* are the trade-offs and results of application of a precise design pattern.
- Patterns help you build on the collective experience of skilled software engineers.
- They capture existing, well-proven experience in software development and help to promote good design practice.
- Every pattern deals with a specific, recurring problem in the design or implementation of a software system.
- Patterns can be used to construct software architectures with specific properties.

☞ **Properties of patterns for Software Architecture**

- A pattern addresses a recurring design problem that arises in specific design situations, and presents a solution to it.
- Patterns document existing, well-proven design experience.
- Patterns identify & and specify abstractions that are above the level of single classes and instances, or of components.
- Patterns provide a common vocabulary and understanding for design principles.
- Patterns are a means of documenting software architectures.
- Patterns support the construction of software with defined properties.
- Patterns help you build complex and heterogeneous software architectures
- Patterns help you to manage software complexity Putting all together we can define the pattern as:

In conclusion,

A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate.

6.2 TYPES OF DESIGN PATTERNS (FROM THE VIEWPOINT OF SOFTWARE MODELING AND DESIGN)

GQ. Explain : (a) Creational pattern (b) Structural pattern (c) Behavioral pattern

- In general, design patterns are categorized into three broad categories:

1. Creational patterns 2. Structural patterns 3. Behavioral patterns

6.2.1 Creational Patterns

- Creational pattern is the kind of design pattern that summaries the instantiation process.
- Creational design patterns helps software system designers in order to make a software system independent of how objects of a software system are generated and demonstrated.
- While making use of a creational pattern, the software system becomes more dependent on object composition as compared to the class inheritance. Hence, creational patterns plays a significant role in designing a software system application.
- Following is the detailed list of creational patterns :
  - Abstract Factory
  - Factory Method
  - Builder
  - Prototype
  - Singleton

6.2.2 Structural Patterns

- The structural design patterns are related to how the objects and classes are comprised to generate large structures.
- Structural patterns make use of the concept of inheritance in object oriented methodology.
- Following are the examples of structural design pattern :
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade
  - Flyweight
  - Proxy

6.2.3 Behavioral Patterns

- The Behavioral patterns are allied with algorithms and the assignment of responsibilities between objects.

- Behavioral design patterns defines patterns of objects and classes along with the patterns of communication and coordination between objects and classes involved in the software system scenario.
- The concept of inheritance is used for proper distribution of behavior amongst classes involved within a software system.
- Examples of Behavioral patterns are :
  - Template Method
  - Command
  - Memento
  - Strategy
  - Interpreter
  - Iterator
  - Observer
  - Visitor
  - Chain of Responsibility
  - Mediator
  - State

6.3 TYPES OF DESIGN PATTERNS (FROM THE VIEWPOINT OF SOFTWARE ARCHITECTURES)

GQ. List and explain different types of design pattern.

- Three-part schema that underlies every pattern:

  Context : a situation giving rise to a problem.

  Problem : the recurring problem arising in that context.

  Solution : a proven resolution of the problem.

Context

- The Contest extends the plain problem-solution dichotomy by describing the situations in which the problems occur.
- Context of the problem may be fairly general. For eg: —developing software with a human-computer interface‖. On the other had, the contest can tie specific patters together.
- Specifying the correct context for the problem is difficult. It is practically impossible to determine all situations in which a pattern may be applied.

Problem

- This part of the pattern description schema describes the problem that arises repeatedly in the given context.
- It begins with a general problem specification (capturing its very essence what is the concrete design issue we must solve?)
- This general problem statement is completed by a set of forces. The term force here denotes any aspect of the problem that should be considered while solving it, such as
  - Requirements the solution must fulfill
  - Constraints you must consider

- Desirable properties the solution should have.

Forces are the key to solving the problem. Better they are balanced, better the solution to the problem

Solution :

The solution part of the pattern shows how to solve the recurring problem(or how to balance the forces associated with it)

In software architectures, such a solution includes two aspects:

1. Every pattern specifies a certain structure, a spatial configuration of elements. This structure addresses the static aspects of the solution. It consists of both components and their relationships.
2. Every pattern specifies runtime behavior. This runtime behavior addresses the dynamic aspects of the solution like, how do the participants of the patter collaborate? How work is organized between then? Etc.

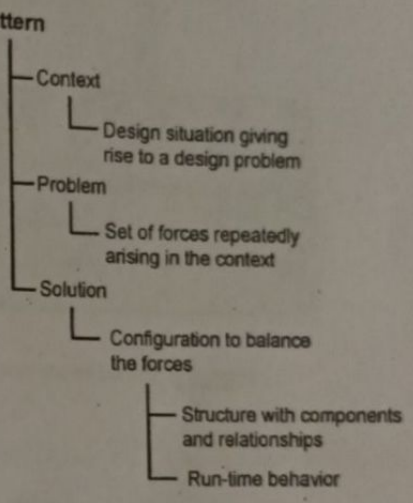


Fig. 6.3.1

- The solution does not necessarily resolve all forces associated with the Problem.
- A pattern provides a solution schema rather than a full specified artifact or blue print.
- No two implementations of a given pattern are likely to be the same.
- The Fig. 6.3.1 summarizes the whole schema.

Pattern Categories

- The patterns are grouped into three broad categories:
  1. Architectural patterns 2. Design patterns 3. Idioms
- Each category consists of patterns having a similar range of scale or abstraction.

Architectural patterns

- Architectural patterns are used to describe viable software architectures that are built according to some overall structuring principle.
- ***Definition :*** An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.
- Eg: Model-view-controller pattern.

Structure :

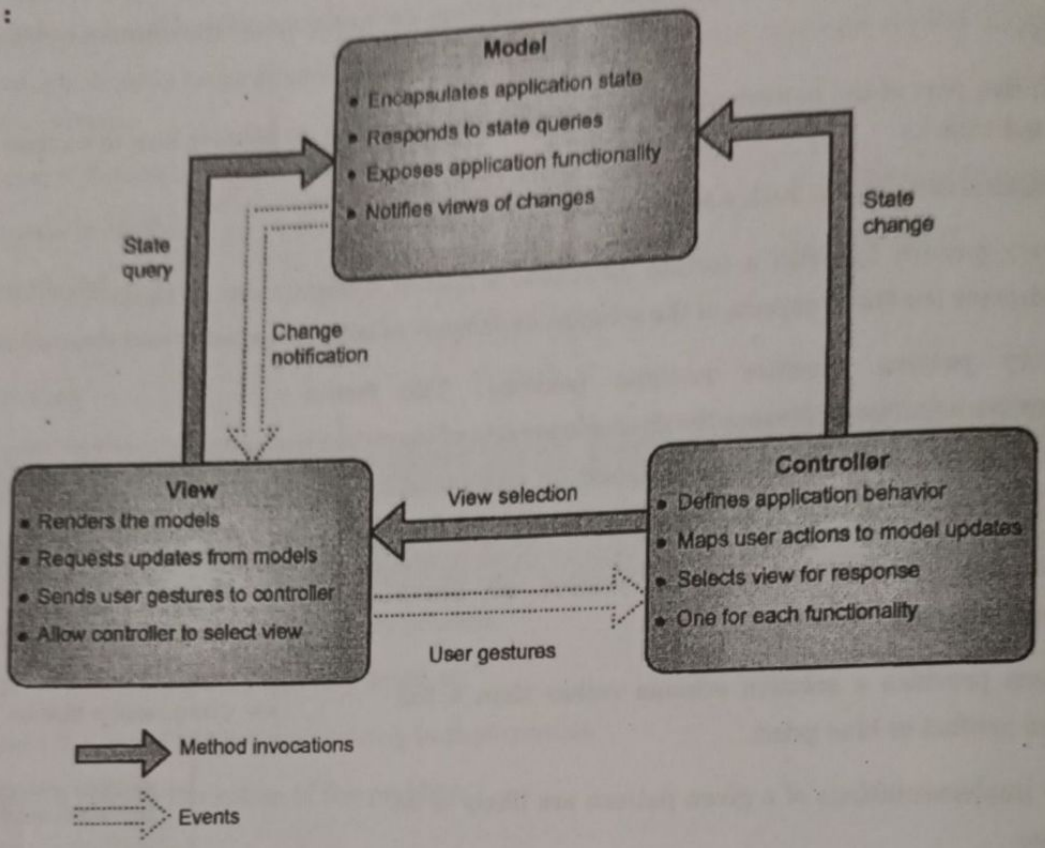


Fig. 6.3.2

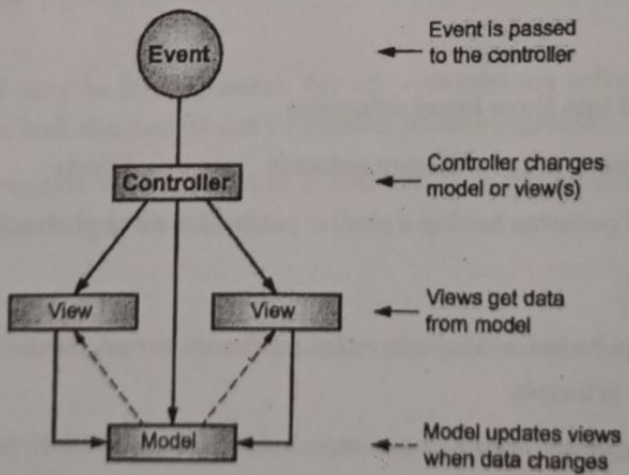


Fig. 6.3.3

Example :

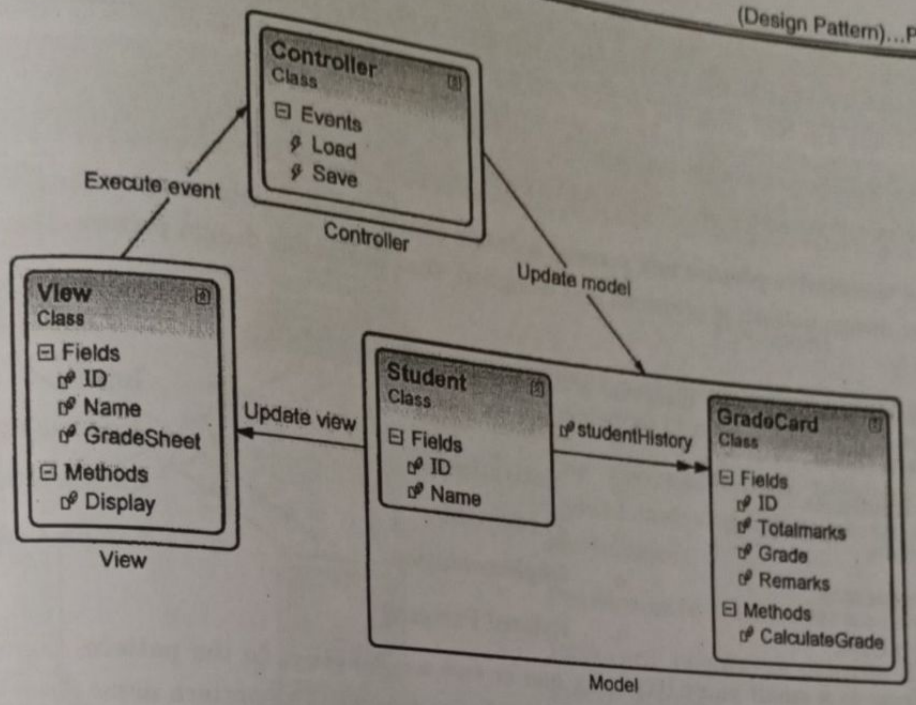


Fig. 6.3.4

Design patterns

- Design patterns are used to describe subsystems of a software architecture as well as the relationships between them (which usually consists of several smaller architectural units).
- *Definition:* A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them.It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular Context.
- They are medium-scale patterns. They are smaller in scale than architectural patterns, but tend to be independent of a particular programming language or programming paradigm.
- Eg: Publisher-Subscriber pattern.

Idioms

- Idioms deals with the implementation of particular design issues.
- ***Definition :*** An idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.
- Idioms represent the lowest- level patterns. They address aspects of both design and implementation.
- Eg: counted body pattern.

6.4 DESIGN PATTERN DESCRIPTION AND DOCUMENTATION

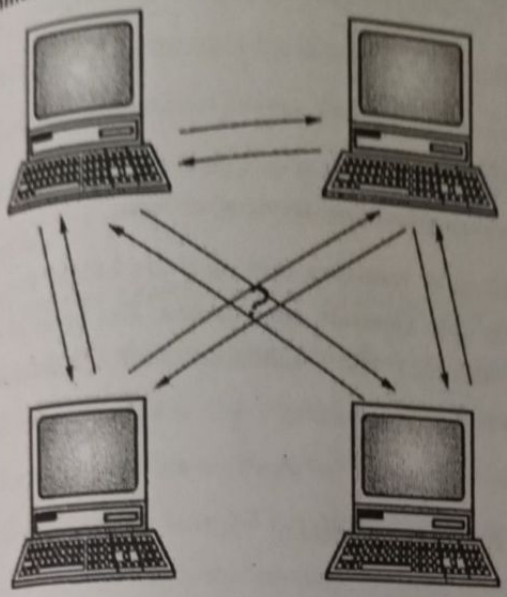
GQ. Explain the basic elements of design pattern.

GQ. State and explain the entities involved in a design pattern.

- It is quite essential to prepare and present a design pattern in a widely accepted format since, once the particular design pattern is proposed and designed; that particular design pattern is meant to be used extensively.
- So, the design pattern should comprise of following entities :
  - Pattern Name
  - Also Known As
  - Structure
  - Consequences
  - Known Uses
  - Classification
  - Motivation
  - Participants
  - Implementation
  - Related Patterns
  - Intent
  - Applicability
  - Collaborations
  - Sample Code
- *Pattern Name* is a small name (typically one or two words) given to the pattern. The pattern name should fundamentally specify the purpose of the design pattern. The pattern name should be unique in the particular application area.
- *Classification* means each deign pattern should be categorized under three broad categories of design patterns which are Creational patterns, Structural patterns and Behavioral patterns. Creational design patterns are concerned with how objects are created. Structural design patterns are concerned with placing objects together into a large structure. Behavioral design patterns are concerned with the relationship amongst objects in order to achieve a particular goal.
- *Intent* is a short explanation about the design pattern.
- *Also known as* field enlists aliases for the particular design pattern.
- *Motivation* field comprises of a detailed description of a design problem that is solved by the use of the design pattern.
- *Applicability* section guides about areas where the particular design pattern can be applied and how to identify and distinguish those areas.
- *Structure* mainly depicts how the design pattern actually works with the help of class diagrams and sequence diagrams.
- *Participants* segment comprises of short descriptions of the objects involved in the software system scenario and describes responsibilities of individual objects.
- *Collaborations* field comprise of explanation of collaborations among the participants.
- *Consequences* part consists of benefits and inadequacies of a design pattern.
- *Implementation* section comprises of useful tricks for implementation of a design pattern.
- *Sample Code* section consists of complete implementation of a particular design pattern.

- *Known Uses* field describes about, where the design pattern has been applied in the real world.
- *Related Patterns* encompasses all of the other available design patterns that are similar to a precise design pattern.

Communication pattern



Forwarder-Receiver (1)

Forwarder-Receiver

Problem

Many components in a distributed system communicate in a peer to peer fashion.

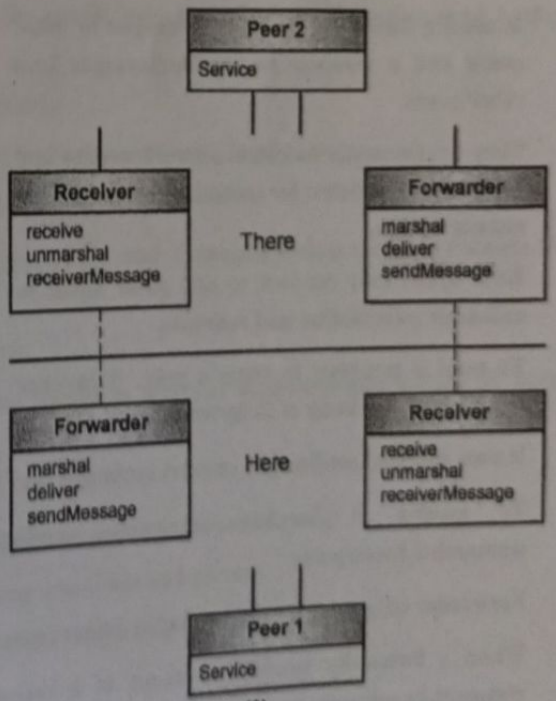
- The communication between the peers should not depend on a particular IPC mechanism;
- Performance is (always) an issue; and
- Different platforms provide different IPC mechanisms.

Fig. 6.4.1

☑ **Solution :**

Encapsulate the inter-process communication mechanism:

- *Peers* implement application services.
- *Forwarders* are responsible for sending requests or messages to remote peers using a specific IPC mechanism.
- *Receivers* are responsible for receivingIPC requests or messages sent by remote peers using a specific IPC mechanism and dispatching the appropriate method of their intended receiver.



Forwarder-Receiver (2)

Fig. 6.4.2

Intent

- "The Forwarder-Receiver design pattern provides transparent interprocesscommunication for software systems with a peer-to-peer interaction model.
- It introduces forwarders and receivers to decouple peers from the underlying communication mechanisms."

Motivation

- Distributed peers collaborate to solve a particular problem.
- A peer may act as a client - requesting services- as a server, providing services, or both.
- The details of the underlying IPC mechanism for sending or receiving messages are hidden from the peers by encapsulating all system-specific functionality into separate components. Examples of such functionality are the mapping of names to physical locations, the establishment of communication channels, or the marshaling and unmarshaling of messages.

Structure

- F-R consists of three kinds of components, Forwarders, receivers and peers.
- Peer components are responsible for application tasks.
- Peers may be located in different process, or even on a different machine.
- It uses a forwarder to send messages to other peers and a receiver to receive messages form other peers.
- They continuously monitor network events and resources, and listen for incomingmessages form remote agents.
- Each agent may connect to any other agent to exchange information and requests.
- To send a message to remote peer, it invokes the method sendmsg of itsforwarder.

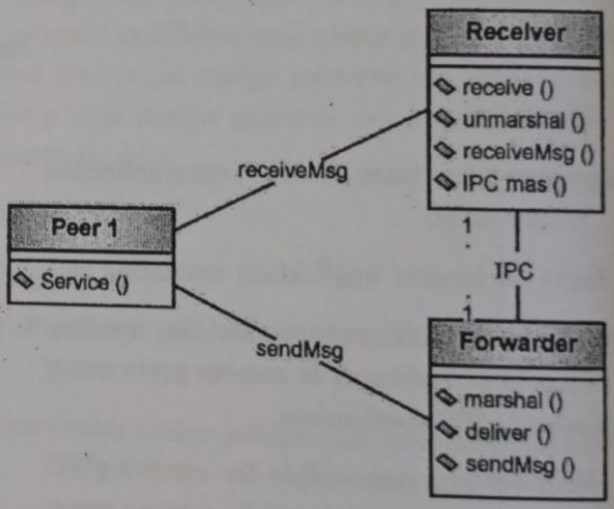


Fig. 6.4.3

- It uses marshal.sendmsg to convert messages that IPC understands.
- To receive it invokes receivemsg method of its receiver 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.

Kinds of messages are

1. **Command message :** instruct the recipient to perform some activities.
2. Information message- contain data.
3. Response message- allow agents to acknowledge the arrival of a message.

It includes functionality for sending and marshaling

Receiver components are responsible for receiving messages.

It includes functionality for receiving and unmarshaling messages.

Dynamics

P1 requests a service from a remote peer P2.

It sends the request to its forwarder forw1 and specifies the name of the recipient.

Forw1 determines the physical location of the remote peer and marshals themessage.

Forw1 delivers the message to the remote receiver recv2.

At some earlier time p2 has requested its receiver recv2 to wait for an incoming request.

Now recv2 receives the message arriving from forw1.

Recv2 unmarshals the message and forwards it to its peer p2.

Meanwhile p1 calls its receiver recv1 to wait for a response.

P2 performs the requested service and sends the result and the name of therecipient p1 to the forwarder forw2.

The forwarder marshals the result and delivers it recv1.

Recv1 receives the response from p2, unmarshals it and delivers it to p1.Implmentation

Specify a name to address mapping.-/server/cvramanserver/.....

Specify the message protocols to be used between peers and forwarders.-class message consists of sender and data.

Choose a communication mechanism-TCP/IP sockets

Implement the forwarder.- repository for mapping names to physical addresses-desitination Id, port no.

sendmsg(dest, marshal(the mesg))

Implement the receiver – blocking and non-blockingrecvmsg() unmarshal(the msg)

Implement the peers of the application – partitioning into client and servers.

Implement a start up configuration- initialize F-R with valid name to address mapping

Benefits and liability

Efficient inter-process communication

- Encapsulation of IPC facilities
- No support for flexible re-configuration of components.

Known Uses

- This pattern has been used on the following systems: 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.
- ATM-P implements the IPC between statically-distributed components using the Forwarder-Receiver pattern..)

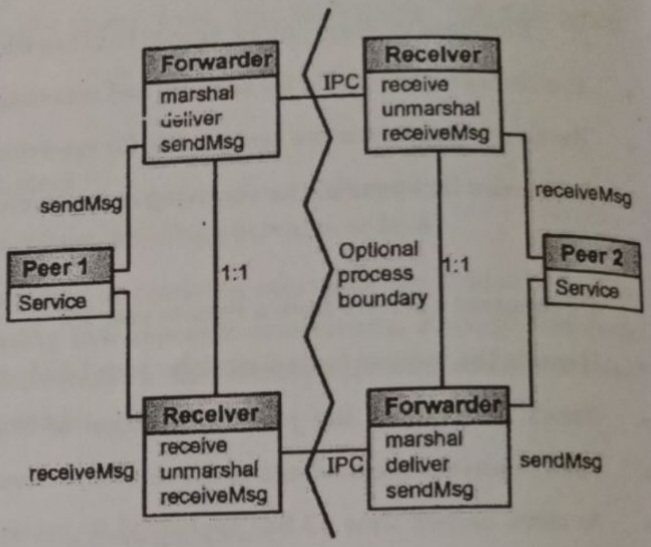


Fig. 6.4.4

- In the Smalltalk environment BrouHaHa, the Forwarder-Receiver pattern is used to implement interprocess communication.

Design Patterns Management

- Systems must often handle collections of objects of similar kinds, of service, or even complex components.
- **E.g.1** Incoming events from users or other systems, which must be interpreted and scheduled approximately.
- **E.g.2** When interactive systems must present application-specific data in a variety of differentway, such views must be handled approximately, both individually and collectively.
- In well-structured s/w systems, separate manager components are used to handle suchhomogeneous collections of objects.
- For this two design patterns are described

1. The Command processor pattern
2. The View Handler pattern

1. Command Processor

- The command processor design pattern separates the request for a service from itsexecution.
- A command processor component manages requests as separate objects, schedules their execution, and provides additional services such as the storing of request objects for later undo.

Context :

Applications that need flexible and extensible user interfaces or Applications that provides services related to the execution of user functions, such as scheduling or undo.

Problem

- Application needs a large set of features.
- Need a solution that is well-structured for mapping its interface to its internalfunctionality
- Need to implement pop-up menus, keyboard shortcuts, or external control of applicationvia a scripting language
- We need to balance the following forces:
  - Different users like to work with an application in different ways.
  - Enhancement of the application should not break existing code.
  - Additional services such as undo should be implemented consistently for all requests.

☑ **Solution**

- Use the command processor pattern
- Encapsulate requests into objects
- Whenever user calls a specific function of the application, the request is turned into a command object.
- The central component of our pattern description, the command processor componenttakes care of all command objects.
- It schedules the execution of commands, may store them for later undo and may provide other additional services such as logging the sequences of commands for testing purposes.
- **Example :** Multiple undo operations in Photosho

Structure

- Command processor pattern consists of following components:
  - The abstract command component
  - A command component
  - The controller component
  - The command processor component
  - The supplier component

Components

Abstract command Component:

- Defines a uniform interface of all commands objects
- At least has a procedure to execute a command

- May have other procedures for additional services as undo, logging,...

Class Abstract Command	Collaborators
Responsibility • Defines a uniform interface Interface to execute commands • Extends the interface for services of the command processor such as undo andlogging	

A Command component:

- For each user function we derive a command component from the abstract command.
- Implements interface of abstract command by using zero or more suppliercomponents.
- Encapsulates a function request
- Uses suppliers to perform requests
- E.g. undo in text editor : save text + cursor position

Class Command	Collaborators • Supplier
Responsibility • Encapsulates a functionrequest • Implements interface of abstract command • Uses suppliers to perform requests	

The Controller Component :

- Represents the interface to the application
- Accepts service requests (e.g. bold text, paste text) and creates the correspondingcommand objects
- The command objects are then delivered to the command processor for execution

Class Controller	*Collaborators* • Command Processor • Command
Responsibility • Accepts service requests • Translates requests intoCommands • Transfer commands to command processor	

Command processor Component:

- Manages command objects, schedule them and start their execution
- Key component that implements additional services (e.g. stores commands for laterundo)
- Remains independent of specific commands (uses abstract command interface)

Class	Collaborators
Command Processor	• Abstract Command
Responsibility	
• Activates command execution • Maintains command objects • Provides additional services related to command execution	

The Supplier Component:

- Provides functionality required to execute concrete commands
- Related commands often share suppliers
- E.g. undo : supplier has to provide a means to save and restore its internal state

Class	Collaborators
Supplier	
Responsibility	
Provides application specific functionality	

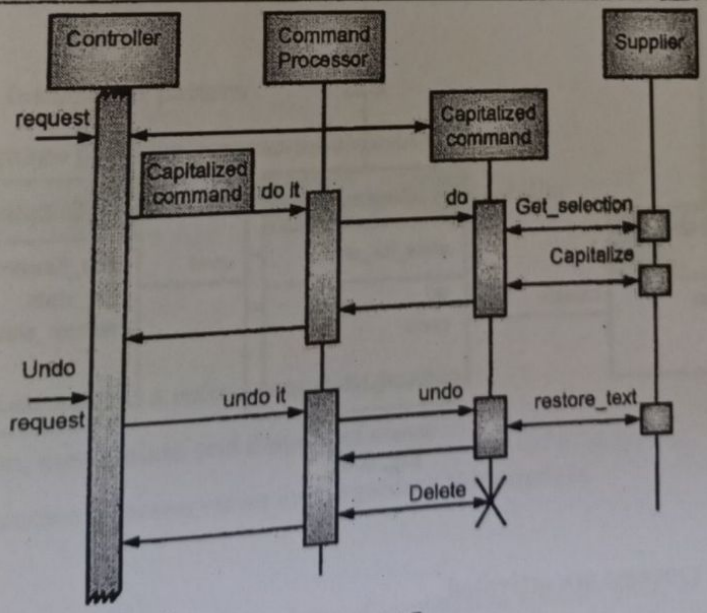


Fig. 6.4.5

The following steps occur:

- The controller accepts the request from the user within its event loop and creates a capitalize command object.
- The controller transfers the new command object to the command processor for execution and further handling.
- The command processor activates the execution of the command and stores it for later undo.
- The capitalize command retrieves the currently-selected text from its supplier, stores the text and its position in the document, and asks the supplier to actually capitalize the selection.
- After accepting an undo request, the controller transfers this request to the commandprocessor.
- The command processor invokes the undo procedure of the most recent command.
- The capitalize command resets the supplier to the previous state, by replacing the saved text in its original position
- If no further activity is required or possible of the command, the command processor deletes the command object.

Component structure and inter-relationships

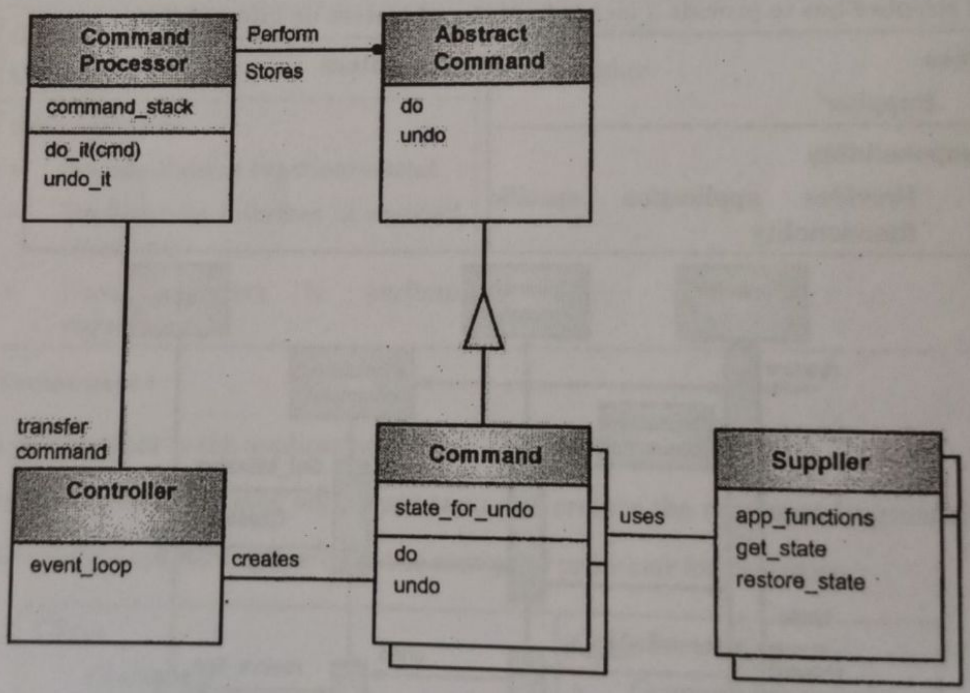


Fig. 6.4.6

Strengths

- Flexibility in the way requests are activated
  - Different requests can generate the same kind of command object (e.g. use GUI orkeyboard shortcuts)

- Flexibility in the number and functionality of requests
  - Controller and command processor implemented independently of functionality of individual commands
  - Easy to change implementation of commands or to introduce new ones
- Programming execution-related services
  - Command processor can easily add services like logging, scheduling,...
- Testability at application level
  - Regression tests written in scripting language
- Concurrency
  - Commands can be executed in separate threads
  - Responsiveness improved but need for synchronization

Weaknesses

- Efficiency loss
- Potential for an excessive number of command classes
  - Application with rich functionality may lead to many command classes
  - Can be handled by grouping, unifying simple commands
- Complexity in acquiring command parameters

Variants

- Spread controller functionality
  - Role of controller distributed over several components (e.g. each menu button creates acommand object)
- Combination with Interpreter pattern
  - Scripting language provides programmable interface
  - Parser component of script interpreter takes role of controller

2. View Handler

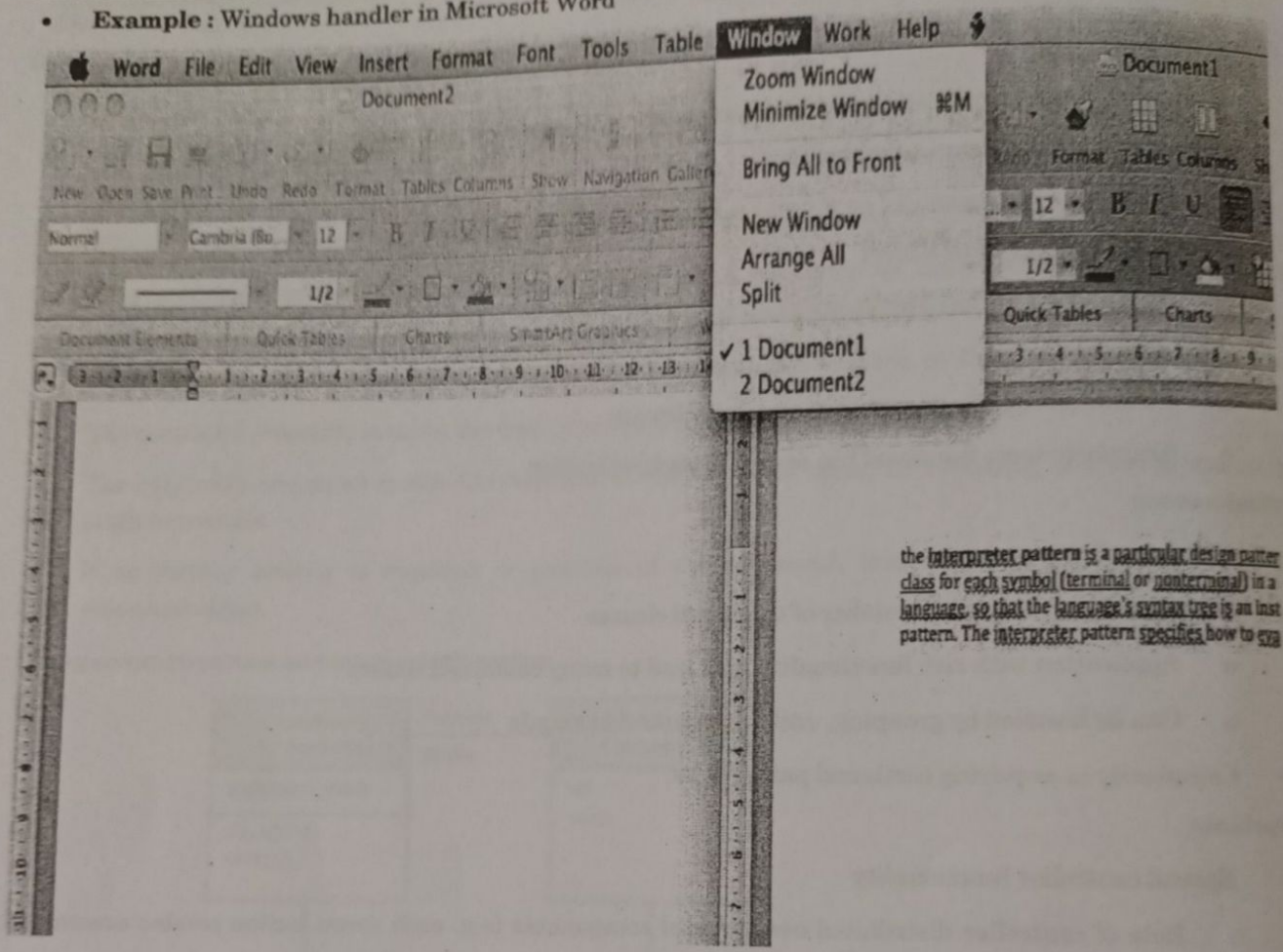
Goals

- Help to manage all views that a software system provides
- Allow clients to open, manipulate and dispose of views
- Coordinate dependencies between views and organizes their update

Applicability

- Software system that provides multiple views of application specific data, or that supportsworking with multiple documents

- **Example :** Windows handler in Microsoft Word



View Handler and other patterns

- MVC
- View Handler pattern is a refinement of the relationship between the model and its associatedviews.
- PAC
- Implements the coordination of multiple views according to the principles of the View Handler pattern.

Components

- View Handler
- Is responsible for opening new views, view initialization
- Offers functions for closing views, both individual ones and all currently-open views
- View Handlers patterns adapt the idea of separating presentation from functional core.

- The main responsibility is to Offers view management services (e.g. bring to foreground, tile all view, clone views)
- Coordinates views according to dependencies

Class: View Handler	Collaborators
Responsibility • Opens, manipulates, and disposes of views of a software system.	• Specific View

Components

Abstract view

- Defines common interface for all views
- Used by the view handler : create, initialize, coordinate, close, etc.

Class: Abstract View	Collaborators
Responsibility • Defines an interface to create, initialize, coordinate, and close a specific view.	

Components

- Specific view
- Implements Abstract view interface
- Knows how to display itself
- Retrieves data from supplier(s) and change data
- Prepares data for display
- Presents them to the user
- Display function called when opening or updating a view

Class: Specific View	Collaborators • Supplier
Responsibility • Implements the abstract interface.	

Components

Supplier

- Provides the data that is displayed by the view components
- Offers interface to retrieve or change data
- Notifies dependent component about changes in data

Class	Collaborators
Supplier	• Specific View • View Handler
Responsibility	
• Implements the interface of the abstract view-one class for each view onto the system.	

- The OMT diagram that shows the structure of view handler pattern Component structure and inter-relationships

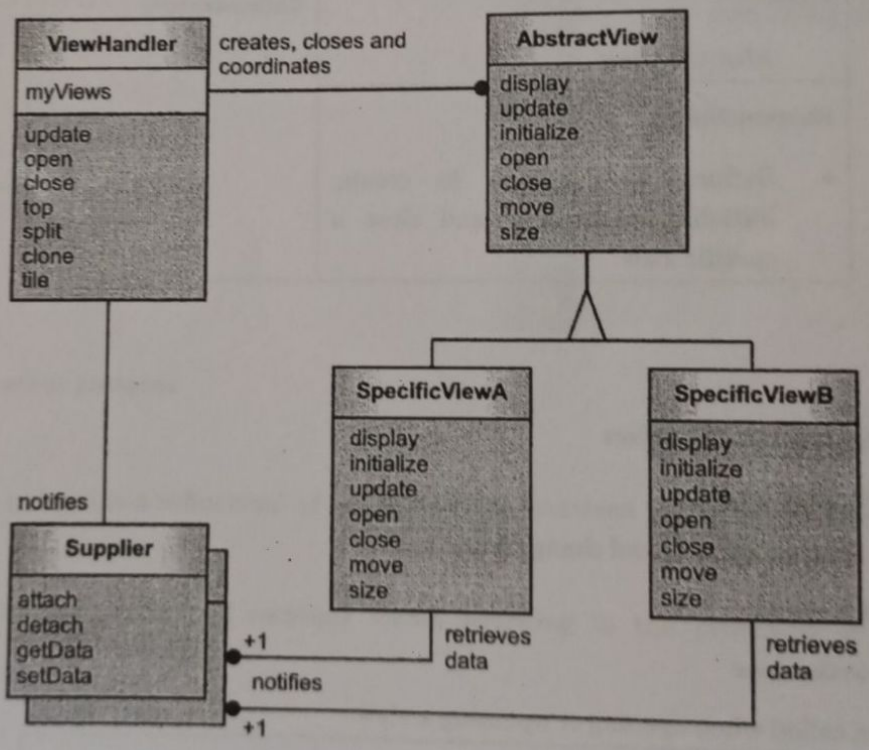


Fig. 6.4.7

Two scenarios to illustrate the behavior of the View Handler

- View creation
- View tiling

Both scenarios assume that each view is displayed in its own window.

Scenario I : View creation

Shows how the view handler creates a new view. The scenario comprises four phases:

- A client-which may be the user or another component of the system-calls the viewhandler to open a particular view.
- The view handler instantiates and initializes the desired view. The view registers with the change-propagation mechanism of its supplier, as specified by the Publisher-Subscriber pattern.
- The view handler adds the new view to its internal list of open views.
- The view handler calls the view to display itself. The view opens a new window,retrieves data from its supplier, prepares this data for display, and presents it to the user. **Interaction protocol**

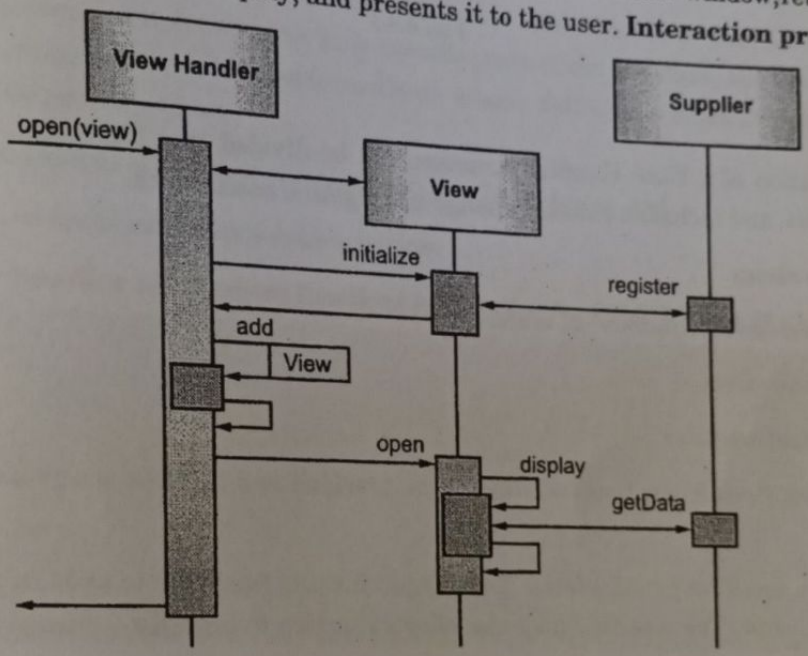


Fig. 6.4.8

Scenario II : View Tiling

Illustrates how the view handler organizes the tiling of views. For simplicity, we assumethat only two views are open. The scenario is divided into three phases:

- The user invokes the command to tile all open windows. The request is sent to the viewhandler.
- For every open view, the view handler calculates a new size and position, and calls its resize and move procedures.
- Each view changes its position and size, sets the corresponding clipping area, and refreshes the image it displays to the user. We assume that views cache the image they display. If this is not the case, views must retrieve data from their associated suppliers

Interaction protocol

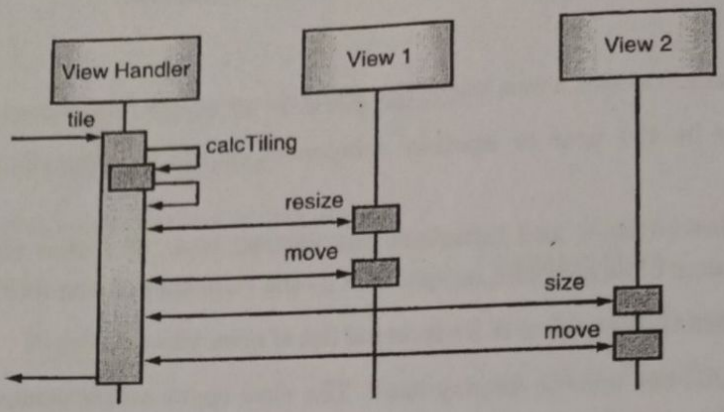


Fig. 6.4.9

Implementation

The implementation of a View Handler structure can be divided into four steps. We assume that the suppliers already exist, and include a suitable change-propagation mechanism.

1. Identify the ***views.***
2. Specify *a common interface for all* ***views.***
3. Implement the views.
4. *Define the* ***view handler***

- Identify the ***views.*** Specify the types of views to be provided and how the user controls each individual view.
- Specify *a common interface for all* ***views.*** This should include functions to open, close, display, update, and manipulate a view. The interface may also offer a function to initialize a view.
- The public interface includes methods to open, close, move, size, drag, and update a view, as well as an initialization method.

Implementation

```
class AbstractView {
protected :
   //Draw the view
   virtual void displayData( ) = 0;
   virtual void displayWindow (Rectangle boundary) = 0;
   virtual void eraseWindow ( ) = 0;
public :
   //Constructor and Destructor
   AbstractView ( ) {};
   ~AbstractView ( ) {};
```

```
//Initialize the view
void initialize ( ) = 0;
// View handling with default implementation
virtual void open (Rectangle boundary) { /* ... */ };
virtual void close ( ) { /* ... */ };
virtual void move (Point point) { /* ... */ };
virtual void size (Rectangle boundary) { /* ... */ };
virtual void drag (Rectangle boundary) { /* ... */ };
virtual void update ( ) { /* ... */};
};
```

- ***Implement the views.*** *Derive a separate class from the* ***AbtrsactView*** class for each specific type of view identified in step 1. Implement the view-specific parts of the interface, such as the **displayData** *()* method in our example. Override those methods whose default implementation does not meet the requirements of the specific view.

In our example we implement three view classes: *Editview, Layoutview,* and

- *Thumbnailview*, as specified in the solution section.
- Define the **view handler :** Implement functions for creating views as Factory Methods.

```
class ViewHandler {
  //Data structures
  Struct ViewInfo {
    AbstractView* view;
    Rectangle    boundary;
    bool         iconized;
};
```

- The view handler in our example document editor provides functions to open and close views, as well as to tile them, bring them to the foreground, and clone them. Internally the view handler maintains references to all open views, including information about their position and size, and whether they are iconize.

```
  Container<ViewInfo*> myViews;
  //The singleton instance
  static ViewHandler* theViewHandler;
  //Constructor and Destructor
  ViewHandler ( );
  ~ViewHandler ( );
Public :
  //Singleton constructor
  static ViewHandler* makeViewHandler( );
```

```
    //Open and close views
    void open (AbstractView* view);
    void close(AbstractView* view);

    //Top, clone, and tile views
    void top (AbstractView* view);
    void clone( ); // Clones the top-most view
    void tile ( );
};
```

```
void ViewHandler :: openView (AbstractView* view) {
    ViewInfo*    viewInfo = new ViewInfo ( );

    //Add the view to the list of open views
    viewInfo ->view        = view;
    viewInfo ->boundary = defaultBoundary;
    viewInfo-> iconized    = false;
    myViews.add(viewInfo);

    // Initialize the view and open it
    view->initialize( );
    view ->open(defaultBoundary);
};
```

Strengths

Uniform handling of views

- All views share a common interface
  - Extensibility and changeability of views
- New views or changes in the implementation of one view don't affect other component
  - Application-specific view coordination
- Views are managed by a central instance, so it is easy to implement specific view coordination strategies (e.g. order in updating views)

Weaknesses

- Efficiency loss (indirection)
- Negligible

- Restricted applicability : useful only with
- Many different views
- Views with logical dependencies
- Need of specific view coordination strategies

Variant

- View Handler with Command objects
- Uses command objects to keep the view handler independent of specific view interface
- Instead of calling view functionality directly, the view handler creates an appropriatecommand and executes it

Known uses

- Macintosh Window Manager
- Window allocation, display, movement and sizing
- Low-level view handler : handles individual window
- Microsoft Word
- Window cloning, splitting, tiling...

Idioms

- idioms are low-level patterns specific to a programming language
- An idiom describes how to implement particular aspects of components or the relationships between them with the features of the given language.
- Here idioms show how they can define a programming style, and show where you can find idioms.
- A programming style is characterized by the way language constructs are used to implement a solution, such as the kind of loop statements used, the naming of program elements, and even the formatting of the source code

What Can Idioms Provide?

- A single idiom might help you to solve a recurring problem with the programming language you normally use.
- They provide a vehicle for communication among software developers.(because each idiom has a unique name)
- idioms are less 'portable_ between programming languages

Idioms and Style

If programmers who use different styles form a team, they should agree on a single coding style for their programs. For example, consider the following sections of C/C++ code, which both implement a string

```
copy function for 'C-style' string
void strcopyRR(char *d, const char *s)
{ while (*d++=*s++) ; }

void strcopyPascal (char d [ I , const char s [I )
{ int i ;
for (i = 0: s[il != '\O 1 : i = i + 1)
{ d[i] = s [i]; }
d[i] = _\0' ; /* always asign 0 character */
}/* END of strcopyPascal */Idioms and Style
```

- A program that uses a mixture of both styles might be much harder to understand and maintain than a program that uses one style consistently.
- Corporate style guides are one approach to achieving a consistent style throughout programs 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.
- Idioms from conflicting styles do not mix well if applied carelessly to a program. 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 single style guide can therefore be unsuitable for large companies that employ many teams to develop applications in different domains.
- A coherent set of idioms leads to a consistent style in your programs.
- Here is an example of a style guide idiom from Kent Beck's *Smalltalk Best Practice Patterns :*

  Name : Indented Control Flow

  Problem : How do you indent messages?

  Solution : Put zero or one argument messages on the same lines as their receiver.foo isNil

  2 + 3

  a < b ifTrue: [. . .]

- Put the keyword/argument pairs of messages with two or more keywords each on its own line, indented one tab.

  a < b

  ifTrue: [. . .]

  ifFalse: [. . . I

- Different sets of idioms may be appropriate for different domains.
- For example, you can write C++ programs in an object-oriented style with inheritance and dynamic binding.
- In some domains. such as real-time systems, a more 'efficient' style that does not use dynamic binding is required.
- A single style guide can therefore be unsuitable for large companies that employ many teams to develop applications in different domains.
- A style guide cannot and should not cover a variety of styles.

Where Can You Find Idioms?

- 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++ .
- You can find 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.

6.5 CASE STUDY: STUDY OF GOF DESIGN PATTERNS AND EXAMPLES

This section describes the GOF design patterns namely:

1. Strategy Design Pattern	2. Observer Design Pattern
3. State Design Pattern	4. Adapter Design Pattern

6.5.1 Strategy Design Pattern

- **Pattern Name :**
  - Strategy
- **Classification :**
  - Behavioral Pattern
- **Intent :**
  - Strategy pattern defines a family of algorithms, summarize each algorithm and make then substitutable.
- **Also Known As :**
  - Policy

- **Motivation :**
  - In majority of the situations, classes vary in their behavior.
  - For getting facility of selecting an algorithm in run time, the algorithms should be segregated and organized in separate classes in case of fluctuating classes.
- **Applicability :**
  - Example : Robotics Application.
  - In the robotics application, at the outset; a simple application is produced to simulate an arena where number of robots are interacting. Following can be the classes involved in this scenario. (Refer Fig. 6.4.1 for better understanding.)
    1. **IBehavior (Strategy) :** It is an interface which defines the behavior of a robot.
    2. **Concrete strategies :** Aggressive Behavior, Defensive Behavior and Normal Behavior defines a precise behavior. These classes requires information transferred from several inbuilt sensors like position, etc. in order to decide and define the action of a particular class.
    3. **Robot :** It is the context class. It gets necessary context information from inbuilt sensors and passes required information to the strategy class.
- **Structure :**

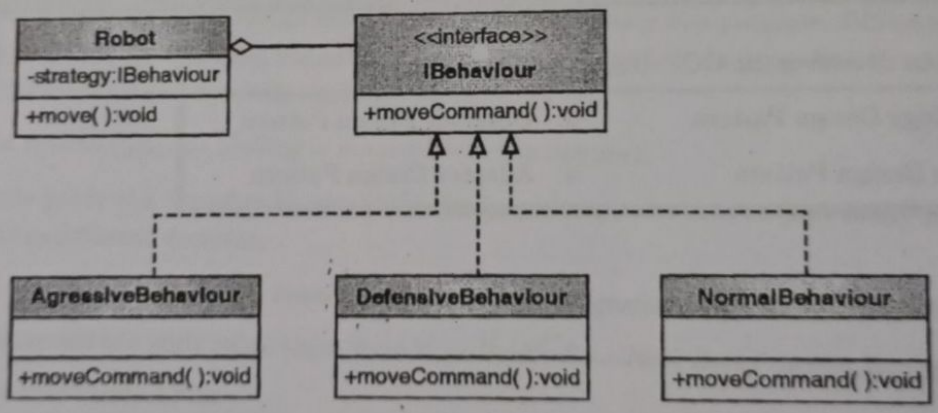


Fig. 6.5.1 : Strategy Pattern Example : Robotics Application

- **Participants**

  Following are the participants involved in Robotics application.

  - Robot
  - IBehavior
  - AgressiveBehavior
  - DefensiveBehavior
  - NormalBehavior
  -
- **Collaborations**
  - IBehavior (Strategy) and Robot (Context) interact to implement the selected algorithm. As far as Robotics application is concerned, a robot (Context) can forward all information to the

- This information may be requested by the algorithm, when a call is given to the algorithm.
- In contrast, a robot (Context) may forward itself as an argument to operations of IBehavior (Strategy).

- **Consequences**
  - A Strategy design pattern offers a facility to select implementations for the similar behavior.
  - The number of objects involved in a particular software system application can be augmented and improved with the help of strategy design pattern.
  - The best thing about a strategy design pattern is that, client programmers may implement and execute their individual strategies without making use of any kind of existing strategy.
  - Strategy eradicates the conditional statements involved in the scenario.
  - In order to get access to numerous behaviors and algorithms, a strategy design pattern offers the substitute for sub classing of a context class.
  - It is the responsibility of strategy pattern to check whether all of the algorithms involved within a scenario are using the similar strategy interface.
- **Implementation :**
  - Table 6.5.1 illustrates components comprised in a Strategy design pattern.

Table 6.5.1 : Components of a strategy design pattern

Sr. No.	Component	Description
1	Context	Encompasses a reference to a Strategy object.
2	Strategy	Interface for supported algorithms is defined with the help of Strategy.
3	Concrete Strategy	Compulsorily implements an algorithm.

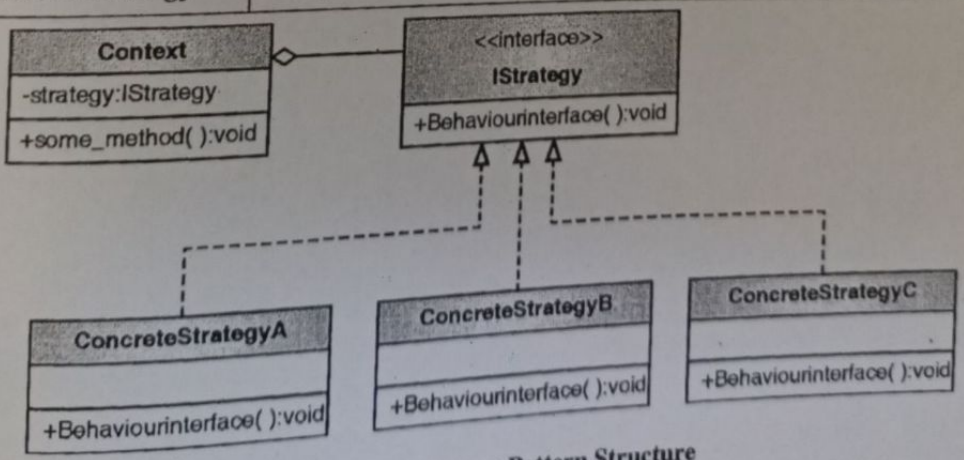


Fig. 6.5.2 : Strategy Pattern Structure

- **Sample Code :**
  - Java code for the Robotics Application example :

```
public interface IBehavior {
    public int moveCommand();
}
public class AgressiveBehavior implements IBehavior{
    public int moveCommand()
    {
        System.out.println("\tAgressiveBehavior: if find another robot attack it");
        return 1;
    }
}
public class DefensiveBehavior implements IBehavior{
    public int moveCommand()
    {
        System.out.println("\tDefensiveBehavior: if find another robot run from it");
        return -1;
    }
}
public class NormalBehavior implements IBehavior{
    public int moveCommand()
    {
        System.out.println("\tNormalBehavior: if find another robot ignore it");
        return 0;
    }
}
public class Robot {
    IBehaviorbehavior;
    String name;
    public Robot(String name)
    {
        this.name = name;
    }
    public void setBehavior(IBehaviorbehavior)
    {
        this.behavior = behavior;
    }
```

```
publicIBehaviorgetBehavior()
{
   returnbehavior;
}
public void move()
{
   System.out.println(this.name + ": Based on current position" +
         "the behavior object decide the next move:");
   int command = behavior.moveCommand();
   // ... send the command to mechanisms
   System.out.println("\tThe result returned by behaviorobject " +
         "is sent to the movement mechanisms " +
         " for the robot "" + this.name + """);
}
public String getName() {
   return name;
}
public void setName(String name) {
   this.name = name;
}
}
public class Main {
  public static void main(String[] args) {
    Robot r1 = new Robot("Big Robot");
    Robot r2 = new Robot("George v.2.1");
    Robot r3 = new Robot("R2");
    r1.setBehavior(new AgressiveBehavior());
    r2.setBehavior(new DefensiveBehavior());
    r3.setBehavior(new NormalBehavior());
    r1.move();
    r2.move();
    r3.move();
    System.out.println("\r\nNewbehaviors: " +
        "\r\n\t'Big Robot' gets really scared" +
        "\r\n\t, 'George v.2.1' becomes really mad because" +
        "it's always attacked by other robots" +
        "\r\n\t and R2 keeps its calm\r\n");
    r1.setBehavior(new DefensiveBehavior());
```

```
        r2.setBehavior(new AgressiveBehavior());
        r1.move();
        r2.move();
        r3.move();
    }
}
```

- **Known Uses :**
  - Borland's ObjectWindows makes use of strategy in dialog boxes in order to make sure that the end user enters valid information. For instance, numbers might have to be in a definite range and a numeric entry field should accept only digits. Confirming that the entered string is correct may require a table look-up.
- **Related Patterns :**
  - Flyweight Pattern

6.5.2 Observer Design Pattern

- **Pattern Name :**
  - Observer
- **Classification :**
  - Behavioral Pattern
- **Intent :**
  - Define a one-to-many dependency amongst objects so that, when one object changes state, all its dependents are warned and updated automatically.
- **Also Known As :**
  - Dependents
  - Publish-Subscribe
- **Motivation :**
  - Object oriented programming is all about objects and their interaction. Without considering the state of objects, we cannot talk about object oriented programming. The Observer design pattern can be used whenever a subject has to be observed by one or more observers.
- **Applicability :**
  - The observer pattern can be used when a change in one object needs modifications in other subsequent objects and system designers do not know how many objects require to be modified and improved.
  - It can be used to notify other objects without making any kind of expectations or assumptions about who these objects are. That is, we don't want these objects tightly coupled.

- **Structure :**

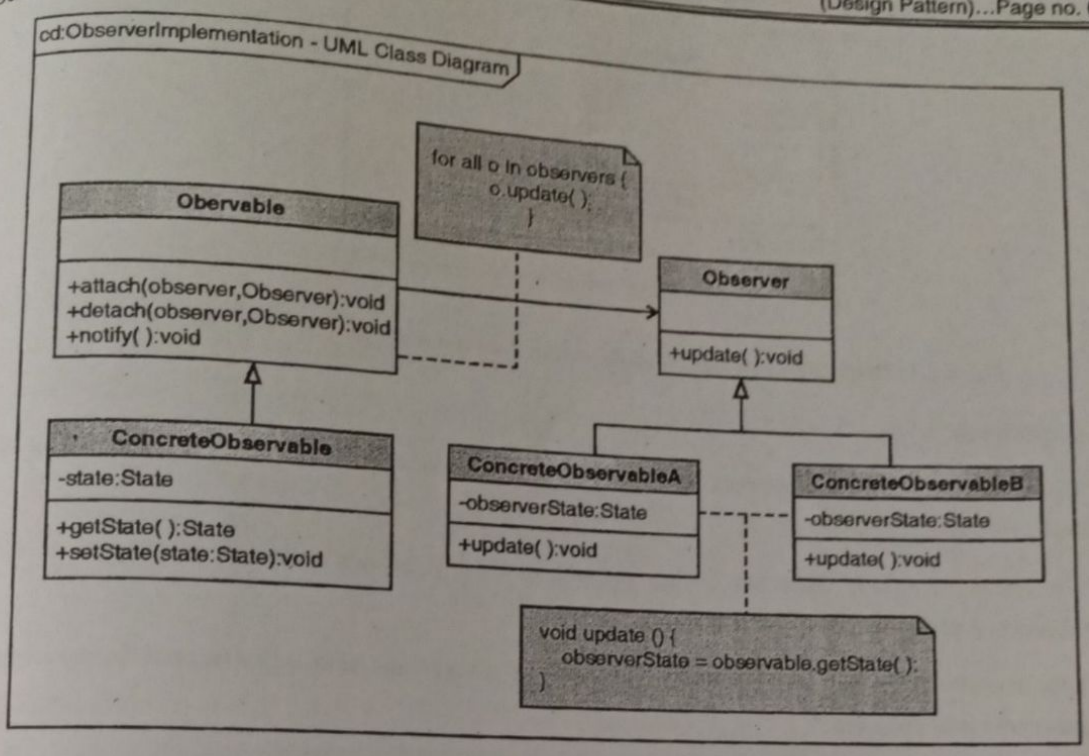


Fig. 6.5.3 : Observer pattern structure

- **Participants :**
  - **Observable :** It is also known as subject. It knows its observers. Any number of observers can observe an observable.
  - **Observer :** It defines an updating interface for objects that should be informed about modifications in an observable.
  - **ConcreteSubject :** It stores state of interest to ConcreteObservable objects. Also, it sends a notification to its respective observers when its state changes.
  - **ConcreteObservable :** It stores state that should stay consistent with observables. It maintains and manages a reference to a ConcreteSubject object. It implements the Observer updating interface to keep its state consistent with the Observables.
- **Collaborations :**
  - The following sequence diagram demonstrates the collaborations between a subject and two observers :

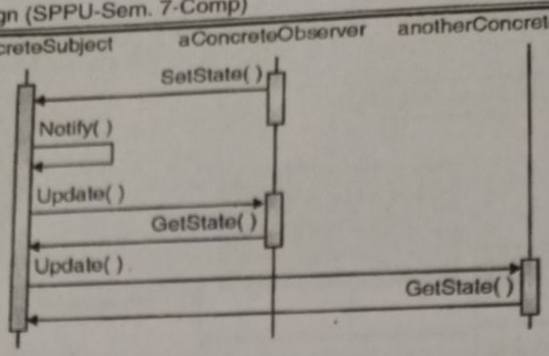


Fig. 6.5.4 : Sequence Diagram : Collaboration between a subject and two observers

- **Consequences**
  - The coupling between observers and subjects involved in a software system scenario is abstract and negligible.
  - The major benefit of Observer design pattern is that, it allows software developers to alter the Observers and subjects independently.
  - That is, new observers can be added deprived of alterations and adjustments in other involved observers and subjects.
- **Implementation**
  - The participant classes in the Observer pattern are :
    1. **Observable :** It is also known as subject. It knows its observers. Any number of observers can observe an observable.
    2. **Observer :** It defines an updating interface for objects that should be informed about modifications in an observable.
    3. **Concrete Subject :** It stores state of interest to Concrete Observable objects. Also, it sends a notification to its respective observers when its state changes.
    4. **Concrete Observable :** It stores state that should stay consistent with observables. It maintains and manages a reference to a Concrete Subject object. It implements the Observer updating interface to keep its state consistent with the Observables.
- **Sample Code :**

```
class Observable{
  ...
  int state = 0;
  intadditionalState = 0;
  publicupdateState(int increment)
  {
```

```
        state = state + increment;
        notifyObservers();
    }
    ...
}

class ConcreteObservable extends Observable{
    ...
    publicupdateState(int increment){
        super.updateState(increment); // the observers are notified
        additionalState = additionalState + increment; // the state is changed after the notifiers are updated
    }
    ...
}
class Observable{
    ...
    int state = 0;
    intadditionalState = 0;
    public void final updateState(int increment)
    {
        doUpdateState(increment);
        notifyObservers();
    }
    public void doUpdateState(int increment)
    {
        state = state + increment;
    }
    ...
}
class ConcreteObservable extends Observable{
    ...
    publicdoUpdateState(int increment){
        super.doUpdateState(increment); // the observers are notified
        additionalState = additionalState + increment; // the state is changed after the notifiers are updated
    }
    ...
}
```

- **Known Uses :**
  - Smalltalk Model/View/Controller (MVC) : The user interface framework in the Small talk environment.
  - User interface toolkits like InterViews, Unidraw and Andrew Toolkit.
- **Related Patterns :**
  - Mediator Pattern
  - Singleton Pattern

6.5.3 State Design Pattern

- **Pattern Name :**
  - State
- **Classification :**
  - Behavioral Pattern
- **Intent :**
  - Let an object to change its behavior when it's internal state changes.
  - The object will appear to change its class.
- **Also Known As :**
  - Objects for States
- **Motivation :**
  - Let us consider an example scenario using a mobile. With respect to alerts, a mobile can be in different states. For example, vibration and silent. Based on this alert state, behavior of the mobile changes when an alert is to be done. Which is a suitable scenario for state design pattern. Following class diagram is for this example scenario.
  - Fig. 6.5.5 depicts a class diagram for Mobile scenario using a state design pattern.

NOTES

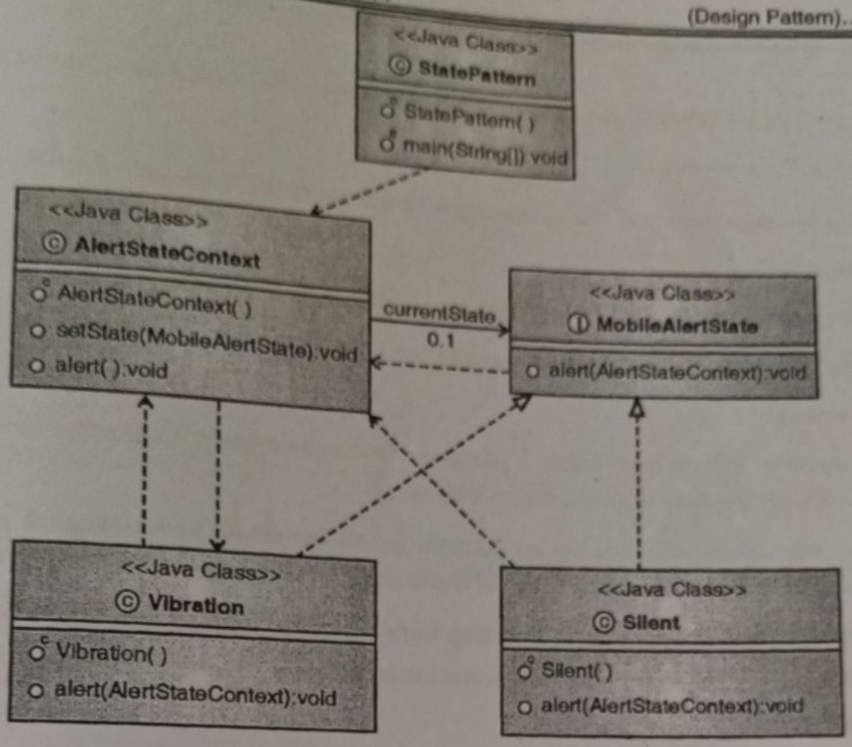


Fig. 6.5.5 : Class Diagram for Mobile scenario using State Design Pattern

- **Applicability**
  - When a behavior of a particular object depends on its state, a state design pattern should be used.

Structure :

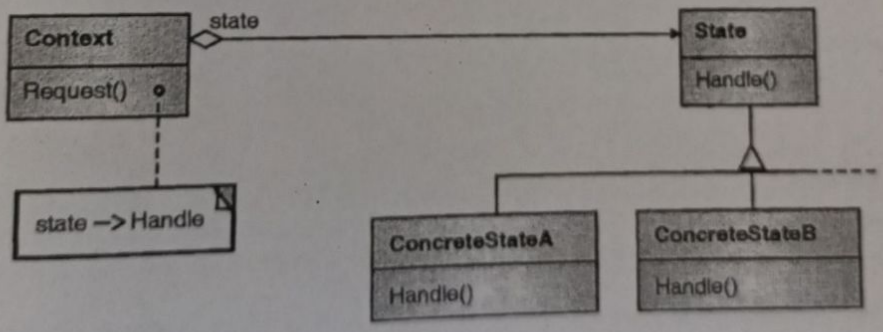


Fig. 6.5.6 : State Design Pattern Structure

- **Participants :**
  - Context :
    1. Outlines the interface of client's interest.
    2. Upholds an instance of a Concrete State subclass which articulates the current state.

- **State :**

  Describes an interface for encapsulating the behavior allied with a particular state of the Context.

- Concrete State subclasses :

  Implements a behavior accompanying with a state of the Context.

- **Collaborations :**
  - Context is the most important interface for clients.
  - A context can pass itself as an argument to the State object handling the request.
- **Consequences :**
  - The number of objects involved in a particular software system application can be augmented and improved with the help of State design pattern.
  - The key advantage of State design pattern is, all behaviors accompanying with a state can be placed into a solitary object by means of State design pattern.
  - Use of State pattern supports in avoiding unpredictable or unreliable states in a software system scenario.
- **Sample Code :**
  - Let us consider an example of TV Remote with a simple button for performing ON/OFF activity. If the state is ON, it will turn on the TV and if state is OFF, it will turn off the TV.
  - We can implement this scenario in Java as given below :

```
packagecom.journaldev.design.state;
public class TVRemoteBasic {
private String state="";
public void setState(String state){
this.state=state;
}
public void doAction(){
if(state.equalsIgnoreCase("ON")){
System.out.println("TV is turned ON");
}else if(state.equalsIgnoreCase("OFF")){
System.out.println("TV is turned OFF");
}
}
public static void main(String args[]){
TVRemoteBasic remote = new TVRemoteBasic();
remote.setState("ON");
remote.doAction();
```

```
remote.setState("OFF");
remote.doAction();
  }
}
  publicclassTVStartStateimplementsState {
  publicvoiddoAction() {
    System.out.println("TV is turned ON");
  }
}
publicclassTVStopStateimplementsState {
  publicvoiddoAction() {
    System.out.println("TV is turned OFF");
  }
}
```

- **Known Uses :**
  - This design pattern is used in drawing editor frameworks namely HotDraw and Unidraw.
- **Related Patterns :**
  - Flyweight Pattern
  - Singleton Pattern

6.5.4 Adapter Design Pattern

- **Pattern Name :**
  - Adapter
- **Classification :**
  - Structural Pattern
- **Intent :**
  - Adapter design pattern converts the interface of a class into another interface clients expect.
  - It allows classes to work together that could not otherwise because of incompatible interfaces.
- **Also Known As :**
  - Wrapper
- **Motivation :**
  - The Adapter pattern is adapting between objects and classes.
  - It is used to be an interface which is nothing but the bridge between two objects.

- **Applicability :**
  - The Adapter pattern should be used when software designers are in need of several existing subclasses.
  - Also, it should be used while producing a reusable class which is intended for alliance with distinct classes.
- **Structure :**
  - Fig. 6.5.7 illustrates a class adapter pattern.
  - Multiple inheritance is used in a class adapter.

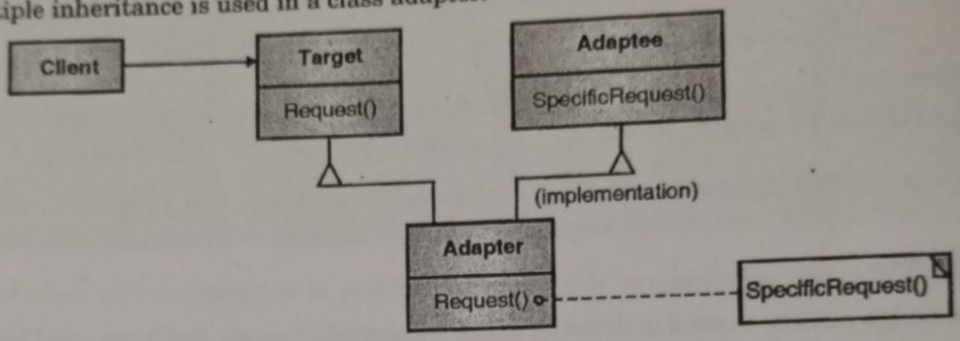


Fig. 6.5.7 : A class adapter pattern structure

  - Fig. 6.5.8 depicts an object adapter pattern.

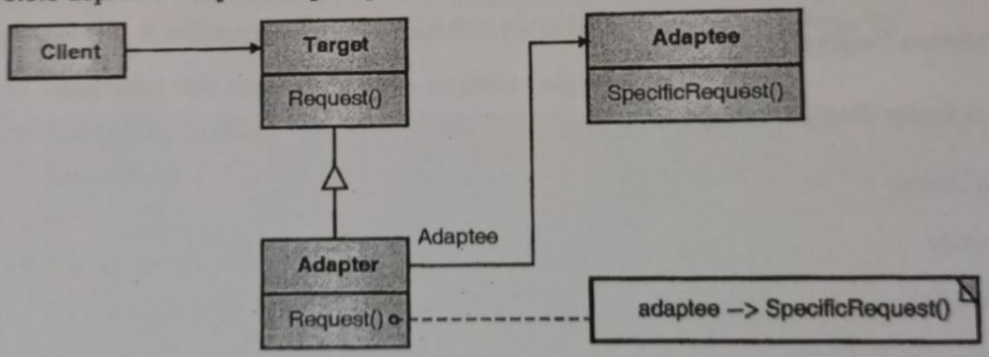


Fig. 6.5.8 : An object adapter pattern structure

- **Participants :**
  - Table 6.5.2 shows elements involved in the Adapter design pattern.

Table 6.5.2 : Elements of Adapter design pattern

Sr. No.	Element	Denotation
1	Client	Works along with objects that are compatible to the target interface.
2	Target	Outlines the domain specific interface used by client.
3	Adapter	Adapts the Adaptee's interface to the target interface.
4	Adaptee	Describes an existing interface that demands for adapting.

- **Collaborations :**
  - Clients call operations on an Adapter instance.
  - In turn, the adapter calls Adaptee operations that carry out the request.
- **Consequences :**
  - A class adapter adapts Adaptee to Target as a result of promising to a concrete Adapter class.
  - A class adapter presents a solitary object.
  - In association with an object adapter, a solitary Adapter can work with numerous Adaptees.
  - Also, the Adapter can enhance the functionality of all Adaptees gradually.
- **Implementation :**
  - Fig. 6.5.9 depicts the class diagram for the Adapter pattern.

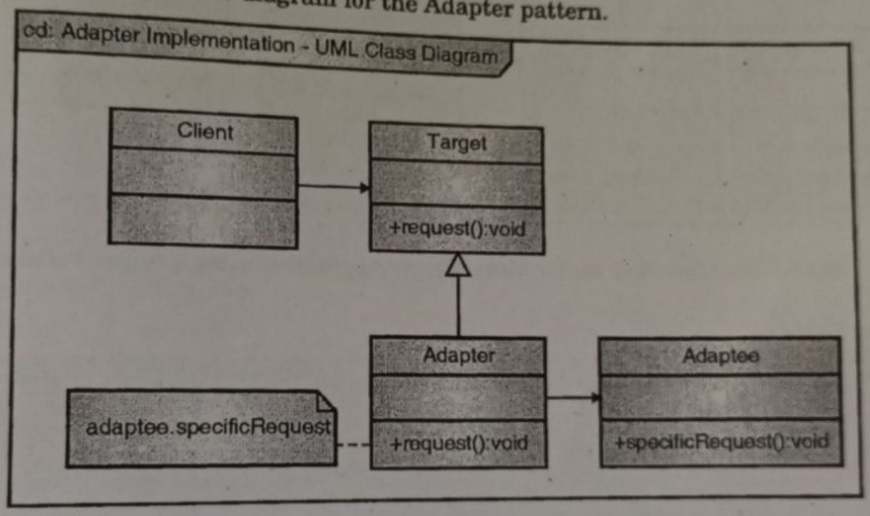


Fig. 6.5.9 : Class diagram for Adapter Pattern

  - Components involved are :

1. Target 2. Adapter 3. Adaptee 4. Client

  - Refer Table 6.4.2 for more details.

- **Sample Code :**
  - Below is the class diagram and source code in Java using Adapter pattern for implementing a media player.

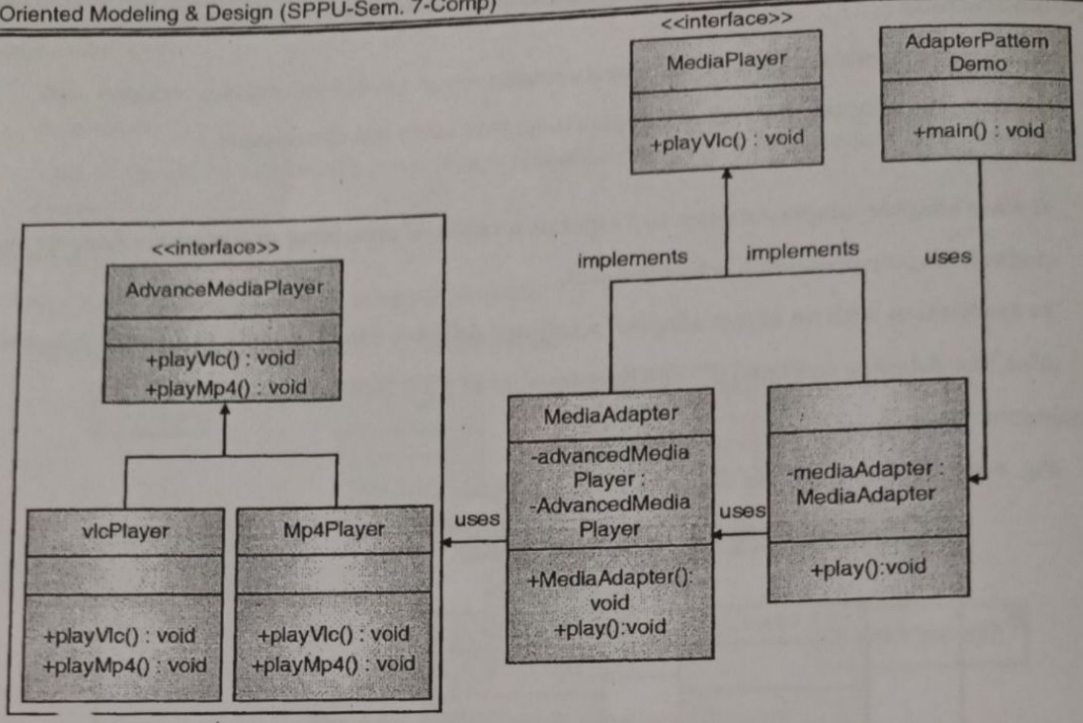


Fig. 6.5.10 : Class diagram for Media Player Application using Adapter Pattern.

- **Source Code :**

```
public interface AdvancedMediaPlayer {
public void playVlc(String fileName);
public void playMp4(String fileName);
}
public class VlcPlayer implements AdvancedMediaPlayer{
  @Override
public void playVlc(String fileName) {
System.out.println("Playing vlc file. Name: " + fileName);
  }
  @Override
public void playMp4(String fileName) {
    //do nothing
  }
}
public class Mp4Player implements AdvancedMediaPlayer{
  @Override
public void playVlc(String fileName) {
    //do nothing
```

```
    }
    @Override
public void playMp4(String fileName) {
System.out.println("Playing mp4 file. Name: "+ fileName);
    }
}
public class MediaAdapter implements MediaPlayer {
AdvancedMediaPlayeradvancedMusicPlayer;
publicMediaAdapter(String audioType){
if(audioType.equalsIgnoreCase("vlc") ){
advancedMusicPlayer = new VlcPlayer();
}else if (audioType.equalsIgnoreCase("mp4")){
advancedMusicPlayer = new Mp4Player();
    }
  }
   @Override
public void play(String audioType, String fileName) {
if(audioType.equalsIgnoreCase("vlc")){
advancedMusicPlayer.playVlc(fileName);
    }
else if(audioType.equalsIgnoreCase("mp4")){
advancedMusicPlayer.playMp4(fileName);
    }
  }
}
public class AudioPlayer implements MediaPlayer {
MediaAdaptermediaAdapter;
  @Override
public void play(String audioType, String fileName) {
    //inbuilt support to play mp3 music files
if(audioType.equalsIgnoreCase("mp3")){
System.out.println("Playing mp3 file. Name: " + fileName);
    }
    //mediaAdapter is providing support to play other file formats
else if(audioType.equalsIgnoreCase("vlc") || audioType.equalsIgnoreCase("mp4")){
mediaAdapter = new MediaAdapter(audioType);
mediaAdapter.play(audioType, fileName);
    }
```

```
else{
System.out.println("Invalid media. " + audioType + " format not supported");
    }
  }
}
public class AdapterPatternDemo {
public static void main(String[] args) {
AudioPlayeraudioPlayer = new AudioPlayer();
audioPlayer.play("mp3", "beyond the horizon.mp3");
audioPlayer.play("mp4", "alone.mp4");
audioPlayer.play("vlc", "far faraway.vlc");
audioPlayer.play("avi", "mind me.avi");
  }
}
```

- **Known Uses**
  - A drawing application based on ET++ makes use of the Adapter pattern.
  - ET++ Draw reuses the ET++ classes for text editing with the help of a TextShape adapter class.
  - Pluggable adapters are common in Smalltalk.
  - Meyer's "Marriage of Convenience" is a kind of class adapter.
- **Related Patterns**
  - Bridge Pattern
  - Proxy Pattern
  - Decorator Pattern

Key Concepts

• Design Pattern	• Creational Pattern
• Structural Pattern	• Behavioral Pattern
• Communication Pattern	• Forward-Receiver
• Client-Dispatcher-Server	• Publisher-Subscriber
• Management Pattern	• Command Processor
• View Handler	• Idioms
• Strategy Pattern	• Observer Pattern
• State Pattern	• Adapter Pattern

Summary

- Design pattern is the concept that helps software designers for reusing the successful and fruitful architectures and designs of the software system.
- Design pattern provides a platform to apply the expertise and knowledge of other software developers to our precise problem.
- Each design pattern is simply nothing but the explanation of a specific method or technique that has already ascertained its effectiveness in the real world.
- Design pattern supports system designers in order to find out and select a specific design alternative from the available set of designs and hence software system designers can choose a particular design pattern for a specific problem.
- A design pattern basically comprise of following elements :
  - Design pattern name
  - Problem
  - Solution
  - Consequences
- In general, design patterns are categorized into three broad categories :
  - Creational patterns
  - Structural patterns
  - Behavioral patterns
- **Creational pattern** is the kind of design pattern that summaries the instantiation process.
- **Creational design** patterns helps software system designers in order to make a software system independent of how objects of a software system are generated and demonstrated.
- The structural design patterns are related to how the objects and classes are comprised to generate large structures.
- Structural patterns make use of the concept of inheritance in object oriented methodology.
- The **Behavioral patterns** are allied with algorithms and the assignment ofresponsibilities between objects.
- Behavioral design patterns defines patterns of objects and classes along with the patterns of communication and coordination between objects and classes involved in the software system scenario.

Chapter Ends...

❑❑❑