CMR INSTITUTE OF TECHNOLOGY		USN						
Internal Assesment Test – II								
Sub:	Object Oriented Modeling And Design						Code:	18MCA43
Date:	20-05-2021	Duration:	90 mins	Max Marks:	50	Sem: IV	Branch:	MCA

Answer **ONE FULL QUESTION** from each part

		Marks	OBE CO	OBE RBT

Part – I

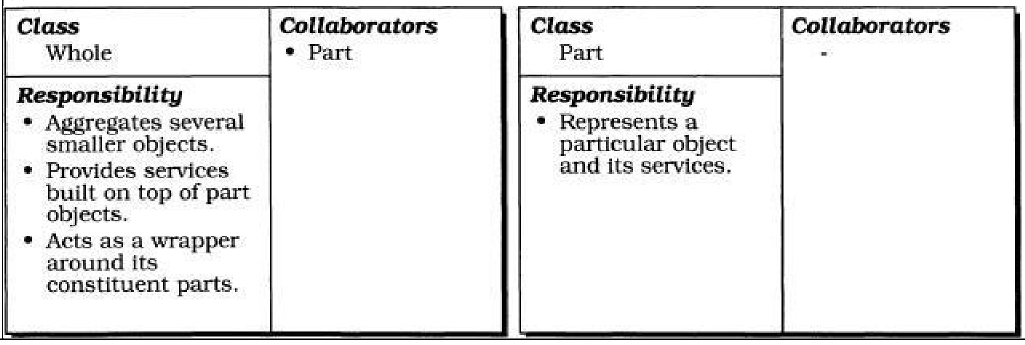
		Marks	CO	RBT
1	Explain the concept of whole-part design pattern with suitable example.	10	CO1	L2

The *Whole-Part* design pattern helps with the aggregation of components that together form a semantic unit. An aggregate component, the Whole1, encapsulates its constituent components, the Parts, organizes their collaboration, and provides a common interface to its functionality. Direct access to the Parts is not possible.
CONTEXT
Implementing aggregate objects.
PROBLEM
In almost every software system objects that are composed of other objects exist. For example, consider a molecule object in a chemical simulation system—it can be implemented as a graph of separate atom objects. Such aggregate objects do not represent loosely-coupled sets of components. Instead, they form units that are more than just a mere collection of their parts.
We need to balance the following *forces* when modeling such structures:
• A complex object should either be decomposed into smaller objects, or composed of existing objects, to support reusability, changeability and the recombination of the constituent objects in other types of aggregate.
• Clients should see the aggregate object as an atomic object that does not allow any direct access to its constituent parts.

SOLUTION
Introduce a component that encapsulates smaller objects, and prevents clients from accessing these constituent parts directly. Define an interface for the aggregate that is the only means of access to the functionality of the encapsulated objects, allowing the aggregate to appear as a semantic unit.
The general principle of the Whole-Part pattern is applicable to the organization of three types of relationship:
• An *assembly-parts* relationship, which differentiates between a product and its parts or subassemblies—such as the relationship of a molecule to its. atoms in our previous example. All parts are tightly integrated according to the internal structure of the assembly. The amount and type of subassemblies is predefined and does not vary.

• A *container-contents* relationship, in which the aggregated object represents a container. For example, a postal package can include different contents such as a book, a bottle of wine, and a birthday card. These contents are less tightly coupled than the parts in an assemblyparts relationship. The contents may even be dynamically added or removed.
• The *collection-members* relationship, which helps to group similar objects—such as an organization and its members. The collection provides functionality, such as iterating over its members and performing operations on each of them. There is no distinction between individual members of a collection—all of them are treated equally.

These relationships mimic relationships between objects in the real world. When modeling them with software entities, it is not always obvious which kind of relationship is appropriate. A molecule may be considered as an assembly composed of different atoms, but also as a container with atoms as its contents. Which relationship is most appropriate depends on the desired use of the aggregate.
It is important to note that these categorizations define relationships between objects, and not between data types.
STRUCTURE
The Whole-Part pattern introduces two types of participant:
A *Whole* object represents an aggregation of smaller objects, which we call *Parts*. It forms a semantic grouping of its Parts in that it coordinates and organizes their collaboration. For this purpose, the Whole uses the functionality of Part objects for implementing services.
Some methods of the Whole may be just placeholders for specific Part services. When such a method is invoked the Whole only calls the relevant Part service, and returns
.The Whole may additionally provide functionality that does not invoke any Part service at all.
Only the services of the Whole are visible to external clients. The Whole also acts as a wrapper around its constituent Parts and protects them from unauthorized access.
Each Part object is embedded in exactly one Whole. Two or more Wholes cannot share the same Part. Each Part is created and destroyed within the life-span of its Whole.



Class Whole	***Collaborators***
Responsibility • Aggregates several smaller objects. • Provides services built on top of part objects. • Acts as a wrapper around its constituent parts.	• Part

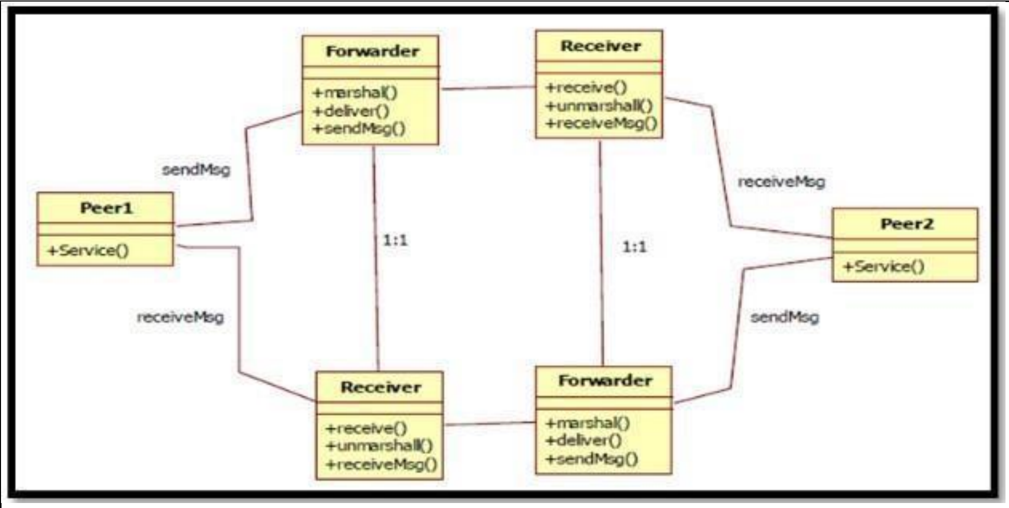
Class Part	***Collaborators***
Responsibility • Represents a particular object and its services.	-

(OR)

2	What do you understand by communication patterns?	10	CO1	L2

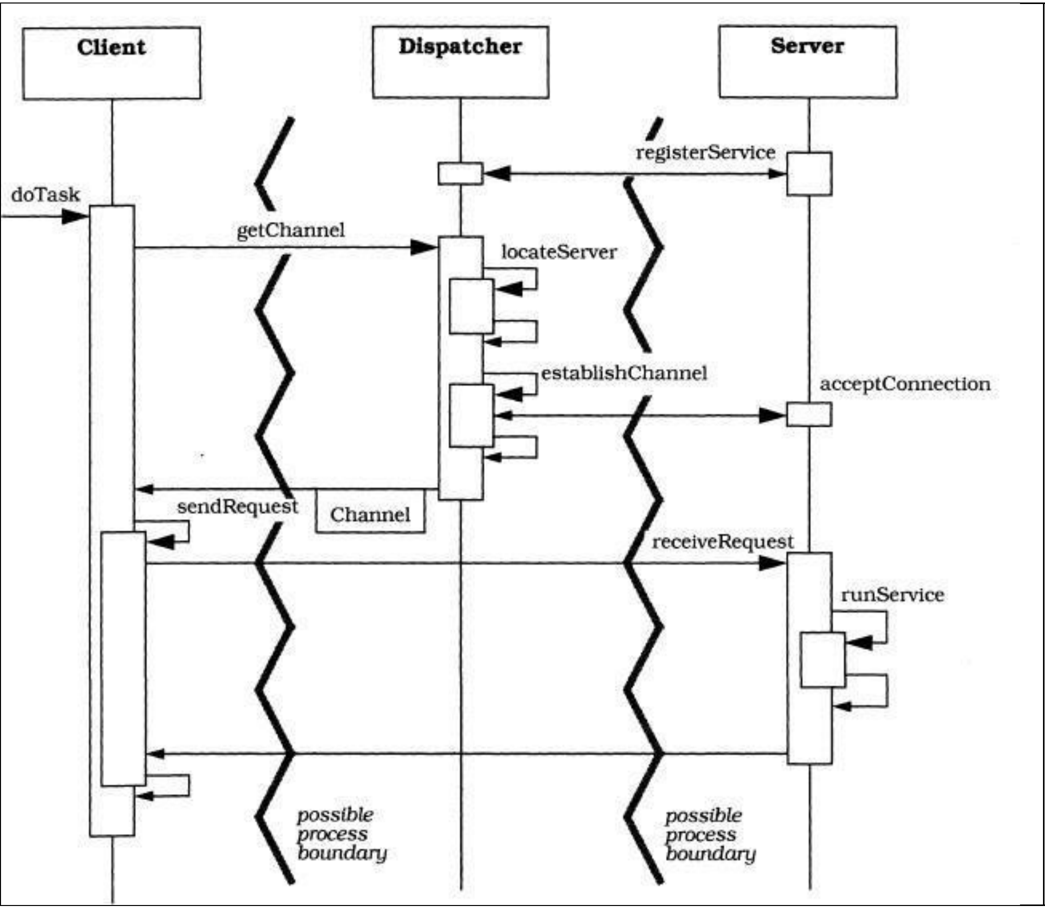
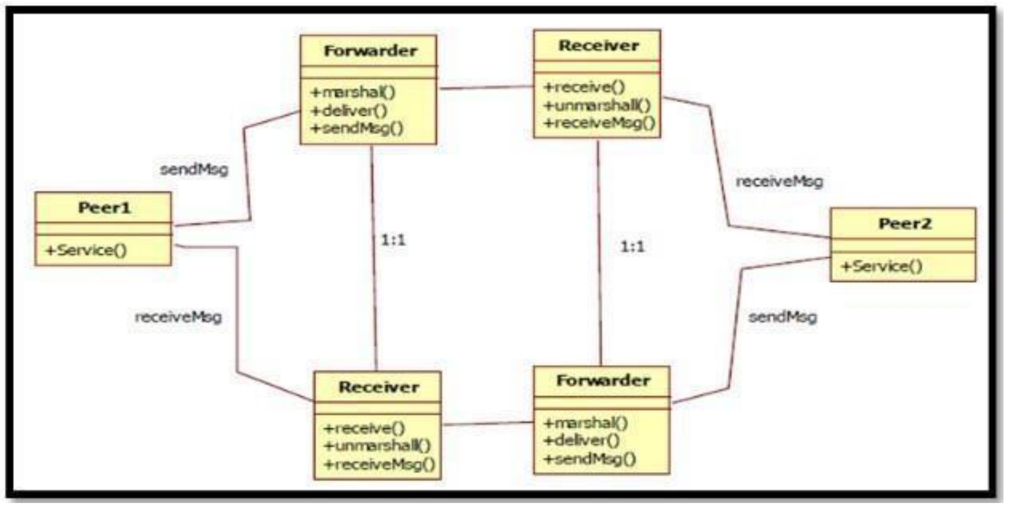
Communication. Patterns help to organize communication between components. Two patterns address issues of inter-process communication: the Forwarder-Receiver pattern and Client Dispatcher-Server pattern .

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



Participant Classes:
Peer components are responsible for application tasks. To carry out their tasks peers need to communicate with other peers.
Forwarder components are responsible for forwarding all these messages to remote network agents without introducing any dependencies on the underlying IPC mechanisms.
Receiver components are responsible for receiving messages. A receiver offers a general interface that is an abstraction of a particular IPC mechanism. It includes functionality for receiving and unmarshaling messages.
Example: A simple peer-to-peer message exchange scenario; Underlying communication protocol is TCP/IP

2. The *Client-Dispatcher-Server* design pattern introduces an intermediate layer between clients and servers, the dispatcher component. It provides location transparency by means of a name service, and hides the details of the establishment of the communication connection between clients and servers.
A server registers itself with the dispatcher component.
• At a later time, a client asks the dispatcher for a communication channel to a specified server.
• The dispatcher looks up the server that is associated with the name specified by the client in its registry.
• The dispatcher establishes a communication link to the server. If it is able to initiate the connection successfully, it returns the communication channel to the client. If not, it sends the client an error message.
• The client uses the communication channel to send a request directly to the server.
• After recognizing the incoming request, the server executes the appropriate service.
• When the service execution is completed, the server sends the results back to the client.

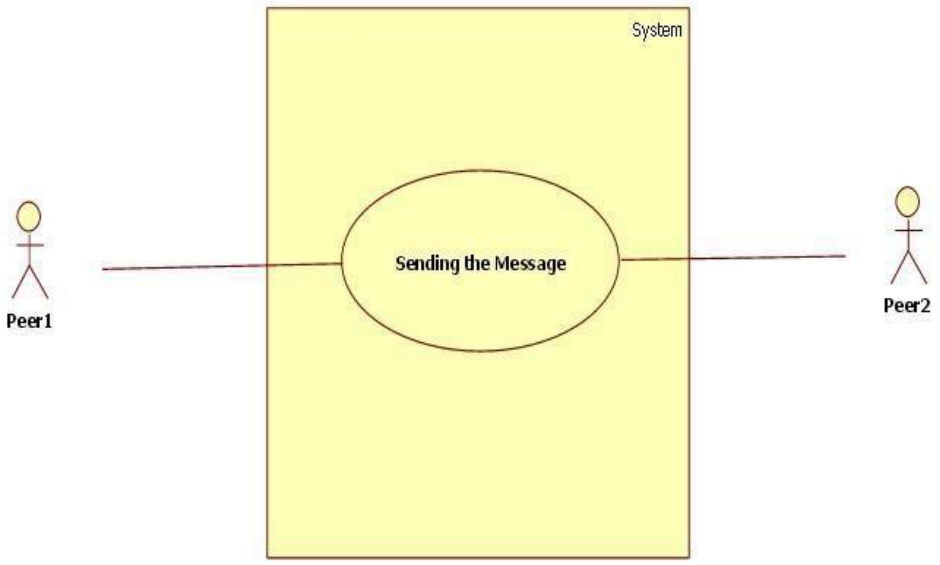
	 Client Dispatcher Server registerService doTask getChannel locateServer establishChannel acceptConnection sendRequest Channel receiveRequest runService possible process boundary possible process boundary			
Part – II				
3	Describe forwarder-receiver design pattern. Forwarder – Receiver Design Pattern Intent: The Forwarder-Receiver design pattern provides transparent inter-process communication for software systems with a peer -to-peer interaction model. It introduces forwarders and receivers to decouple peers from the underlying communication mechanisms. Structure:  Forwarder +marshal() +deliver() +sendMsg() Receiver +receive() +unmarshall() +receiveMsg() sendMsg Peer1 +Service() 1:1 1:1 receiveMsg Peer2 +Service() receiveMsg sendMsg Receiver +receive() +unmarshall() +receiveMsg() Forwarder +marshal() +deliver() +sendMsg() Participant Classes: Peer components are responsible for application tasks. To carry out their tasks peers need to communicate with other peers.	10	CO1	L1

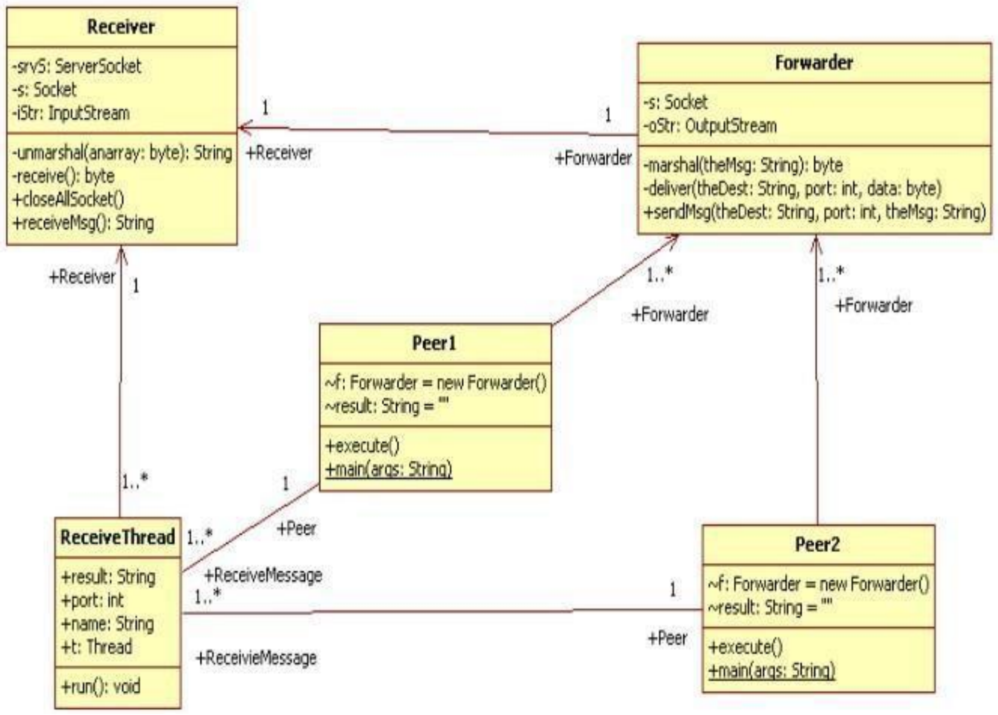
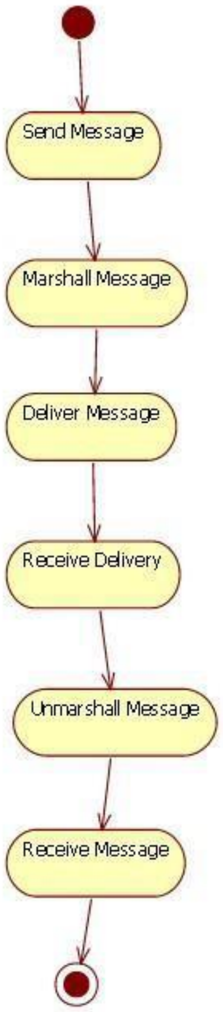
Forwarder components are responsible for forwarding all these messages to remote network agents without introducing any dependencies on the underlying IPC mechanisms.

Receiver components are responsible for receiving messages. A receiver offers a general interface that is an abstraction of a particular IPC mechanism. It includes functionality for receiving and unmarshaling messages.

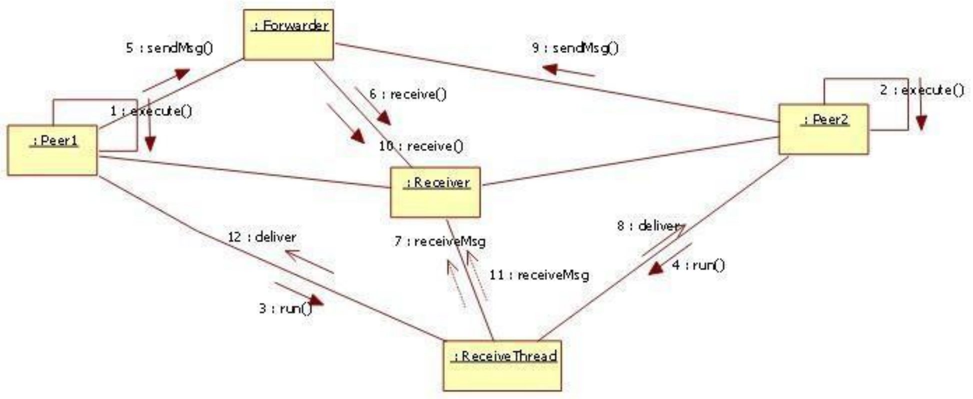
Example: A simple peer-to-peer message exchange scenario; Underlying communication protocol is TCP/IP

Use Case Diagram:



	Class Diagram:  Receiver -srvS: ServerSocket -s: Socket -iStr: InputStream -unmarshal(anarray: byte): String -receive(): byte +closeAllSocket() +receiveMsg(): String Forwarder -s: Socket -oStr: OutputStream -marshal(theMsg: String): byte -deliver(theDest: String, port: int, data: byte) +sendMsg(theDest: String, port: int, theMsg: String) Peer1 ~f: Forwarder = new Forwarder() ~result: String = "" +execute() +main(args: String) Peer2 ~f: Forwarder = new Forwarder() ~result: String = "" +execute() +main(args: String) ReceiveThread +result: String +port: int +name: String +t: Thread +run(): void +Receiver +Forwarder +Peer +ReceiveMessage +ReceivieMessage 1 1..* **Activity Diagram:**  Send Message Marshall Message Deliver Message Receive Delivery Unmarshall Message Receive Message			

Collaboration Diagram:



(OR)

4	Write short note on Proxy design pattern.	10	CO1	L2

The *Proxy* design pattern makes the clients of a component communicate with a representative rather than to the component itself. Introducing such a placeholder can serve many purposes, including enhanced efficiency, easier access and protection from unauthorized access.

CONTEXT

A client needs access to the services of another component2. Direct access is technically possible, but may not be the best approach.

PROBLEM

We do not want to hard-code its physical location into clients, and direct and unrestricted access to the component may be inefficient or even insecure. Additional control mechanisms are needed. A solution to such a design problem has to balance some or all of the following *forces*:

• Accessing the component should be run-time-efficient, cost-effective, and safe for both the client and the component.

• Access to the component should be transparent and simple for the client. The client should particularly not have to change its calling behavior and syntax from that used to call any other direct-access component.

• The client should be well aware of possible performance or financial penalties for accessing remote clients. Full transparency can obscure cost differences between services.

SOLUTION

Let the client communicate with a representative rather than the component itself. This representative—called a *proxy*—offers the interface of the component but performs additional pre- and post-processing such as access-control checking.

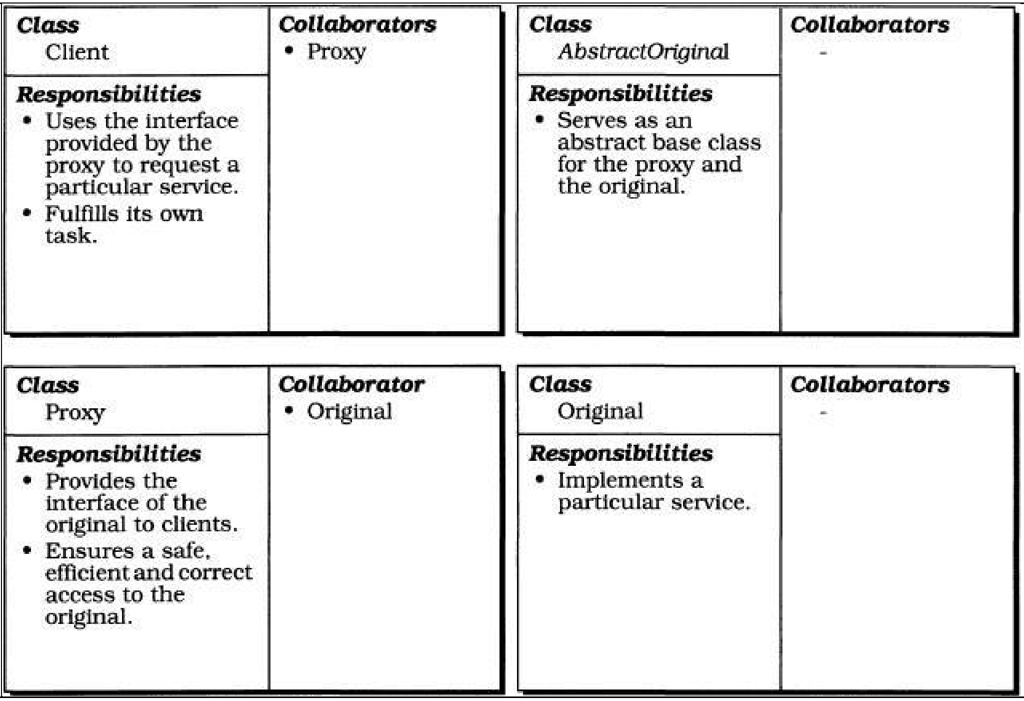
STRUCTURE

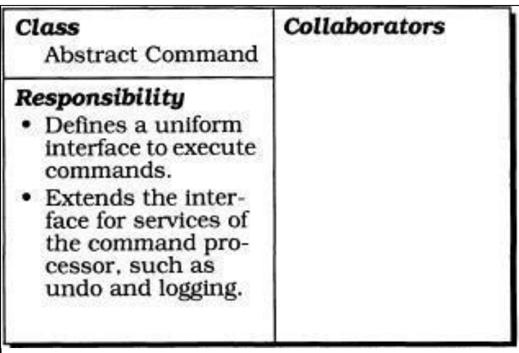
The *original* implements a particular service.

The *client* is responsible for a specific task. To do its job, it invokes the functionality of the original in an indirect way by accessing the proxy.

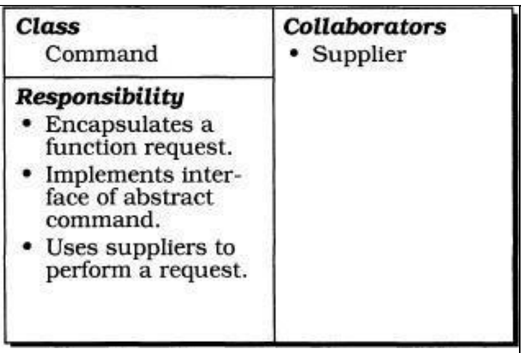
The *proxy* offers the same interface as the original, and ensures correct access to the original. To achieve this the proxy maintains a reference to the original it represents

The *abstract original* provides the interface implemented by the proxy and the original

	 **Class** Client — **Collaborators** • Proxy **Responsibilities** • Uses the interface provided by the proxy to request a particular service. • Fulfills its own task. **Class** *AbstractOriginal* — **Collaborators** - **Responsibilities** • Serves as an abstract base class for the proxy and the original. **Class** Proxy — **Collaborator** • Original **Responsibilities** • Provides the interface of the original to clients. • Ensures a safe, efficient and correct access to the original. **Class** Original — **Collaborators** - **Responsibilities** • Implements a particular service.			
	PART - III			
5	Write short note on Command Processor. The *Command Processor* design pattern separates the request for a service from its execution. 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 provide services related to the execution of user functions, such as scheduling or undo. PROBLEM The following *forces* shape the solution: • Different users like to work with an application in different ways. • Enhancements of the application should not break existing code. • Additional services such as undo should be implemented consistently for all requests. SOLUTION The *Command Processor* pattern builds on the Command design pattern in [GHJV95]. Both patterns follow the idea of encapsulating requests into objects. Whenever a user calls a specific function of the application, the request is turned into a *command* object. A central component of our pattern description, the *command processor*, takes care of all command objects. The command processor schedules the execution of commands, may store them for later undo, and may provide other services such as logging the sequence of commands for testing purposes. Each command object delegates the execution of its task to *supplier* components within the functional core of the application. STRUCTURE The *abstract command* component defines the interface of all command objects.	10	CO1	L1

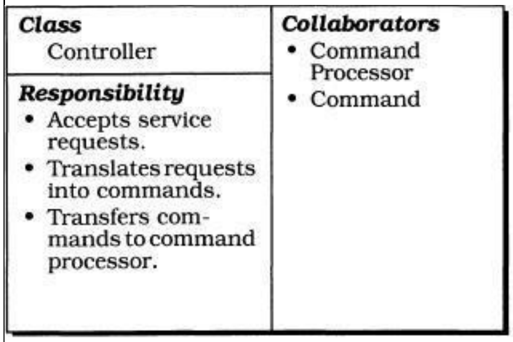


Class Abstract Command	Collaborators
Responsibility • Defines a uniform interface to execute commands. • Extends the interface for services of the command processor, such as undo and logging.	

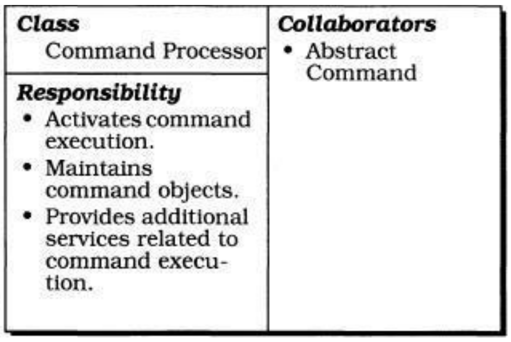


Class Command	Collaborators • Supplier
Responsibility • Encapsulates a function request. • Implements interface of abstract command. • Uses suppliers to perform a request.	

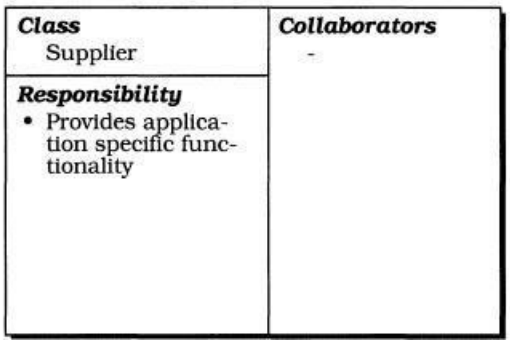
The *controller* represents the interface of the application. It accepts requests, such as ‘paste text,’ and creates the corresponding command objects. The command objects are then delivered to the command processor for execution. The controller of TEDDI maintains the event loop and maps incoming events to command objects.
The *command processor* manages command objects, schedules them and starts their execution. It is the key component that implements additional services related to the execution of commands.
The *supplier* components provide most of the functionality required to execute concrete commands.



Class Controller	Collaborators • Command Processor • Command
Responsibility • Accepts service requests. • Translates requests into commands. • Transfers commands to command processor.	



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



Class Supplier	Collaborators -
Responsibility • Provides application specific functionality	

(OR)

6	Discuss expanding states, nested states and reification with example. **Expanding States:**	10	CO1	L3

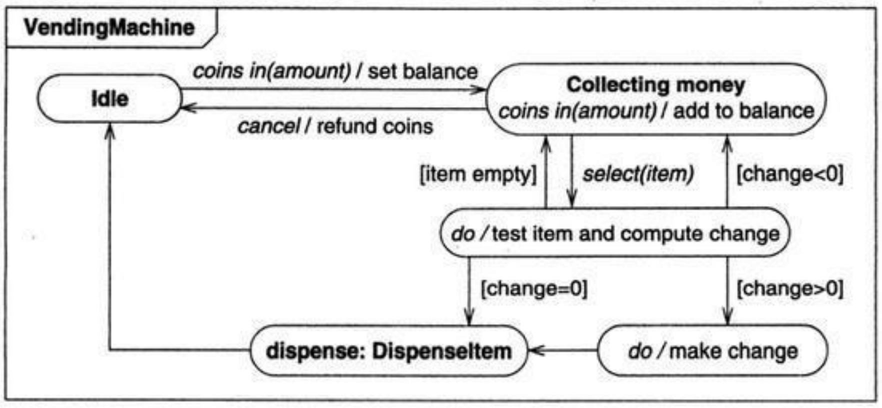


Fig. 4.54 Vending machine state diagram You can simplify state diagrams by using subdiagrams.

One way to organize a model is by having a high-level diagram with subdiagrams expanding certain states. This is like a macro substitution in a programming language. Fig. 4.54 shows such a state diagram for a vending machine. Initially, the vending machine is idle. When a person inserts coins, the machine adds the amount to the cumulative balance. After adding some coins, a person can select an item. If the item is empty or the balance is insufficient, the machine waits for another selection. Otherwise, the machine dispenses the item and returns the appropriate change.

Figure 4.55 elaborates the *dispense* state with a lower-level state diagram called a submachine. A ***submachine*** is a state diagram that may be invoked as part of another state diagram. The UML notation for invoking a submachine is to list a local state name followed by a colon and the submachine name. Conceptually, the submachine state diagram replaces the local state. Effectively, a submachine is a state diagram "subroutine."

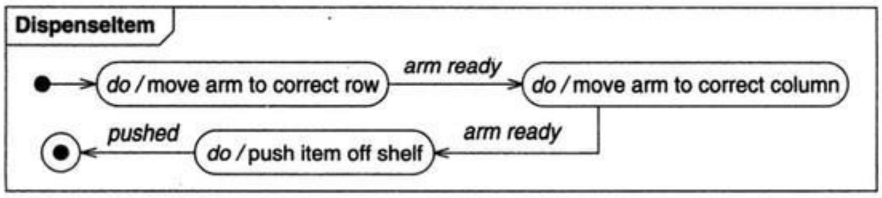
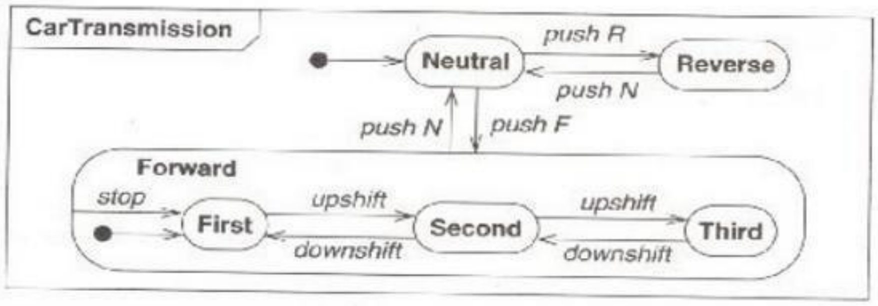
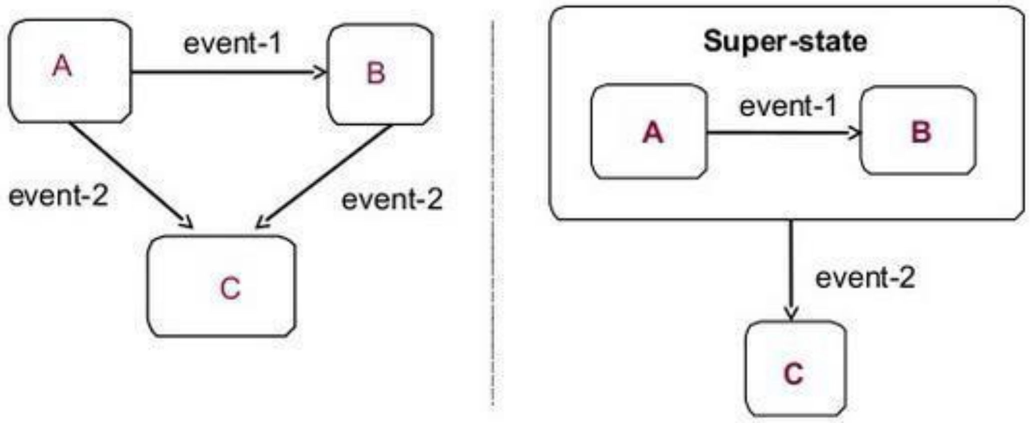


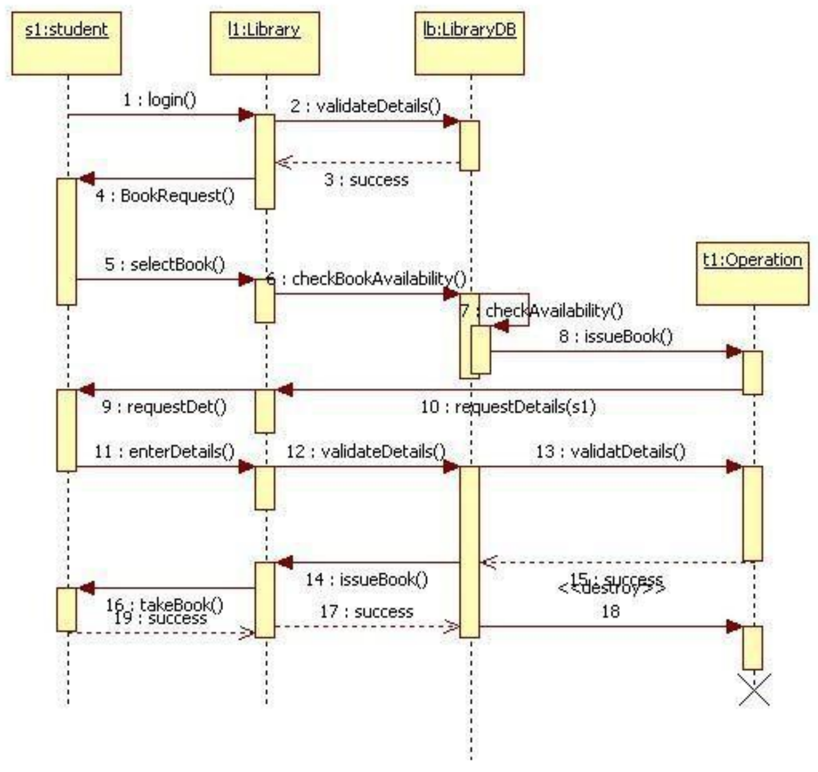
Fig. 4.55 *Dispense item* submachine of vending machine A lower-level state diagram can elaborate a state.

Nested States:

- Activities in states are composite items denoting other lower-level state diagrams
- A lower-level state diagram corresponds to a sequence of lower-level states and events that are invisible in the higher-level diagram.
- When one state is complex, you can include substates in it.
  - drawn as nested rounded rectangles within the larger state

	 CarTransmission Neutral push R Reverse push N push N push F Forward stop First upshift Second upshift Third downshift downshift Figure 6.5 Nested states. You can nest states to an arbitrary depth.  A event-1 B event-2 event-2 C Super-state A event-1 B event-2 C			
Part – IV				
7	Explain Sequence Diagram with an example. Sequence diagrams are simple subsets of interaction diagrams. They map out sequential events in an engineering or business process in order to streamline activities. Sequence diagrams can be useful reference diagrams for businesses and other organizations. Try drawing a sequence diagram to: Represent the details of a UML use case. Model the logic of a sophisticated procedure, function, or operation. See how tasks are moved between objects or components of a process. Plan and understand the detailed functionality of an existing or future scenario. Sequence diagrams are made up of the following elements: **Object** - this box shape represents a class, or object, in UML. They demonstrate how an object will behave in the context of the system. Class attributes should not be listed in this shape. **Activation boxes** - symbolized by a rectangle shape, an activation box represents the time needed for an object to complete a task. The longer the task will take, the longer the activation box becomes. **Actors** - represented by a stick figure, actors are entities that are both interactive with and external to the system. **Package** - also known as a frame, this is a rectangle shape that is used in UML 2.0 notation to contain interactive elements of the diagram. The shape has a small inner rectangle for labeling the diagram. **Lifeline** - a dashed vertical line that represents the passage of time as it extends downward. Along with time, they represent the sequential events that occur to an object	10	CO1	L2

during the charted process. Lifelines may begin with a labeled rectangle shape or an actor symbol.
Option loops - a rectangle shape with a smaller label within it. This symbol is used to
model "if then" scenarios, i.e., a circumstance that will only occur under certain conditions.
Alternatives - used to symbolize a choice (that is usually mutually exclusive) between two or more message sequences. To represent alternatives, use the labeled rectangle
shape with a dashed line inside.
Messages - packets of information that are transmitted between objects. They may reflect
the start and execution of an operation, or the sending and reception of a signal.
o Synchronous messages - represented by a solid line with a solid arrowhead. This symbol is used when a sender must wait for a response to a message before it continues. The diagram should show both the call and the reply.
o Asynchronous messages - represented by a solid line with a lined arrowhead. Asynchronous messages are those that don't require a response before the sender continues. Only the call should be included in the diagram.
o Asynchronous return messages - represented by a dashed line with a lined arrowhead.
o Create messages - represented by a dashed line with a lined arrowhead. These messages are sent to lifelines in order to create themselves.
o Reply messages - represented by a dashed line with a lined arrowhead, these messages are replies to calls.
o Delete messages - represented by a solid line with a solid arrowhead, followed by an X symbol. This messages indicates the destruction of an object and is placed in its path on the lifeline.



(OR)

8	What do you mean by event and activity? Explain the signal event, change event and time event.	10	CO5	L2

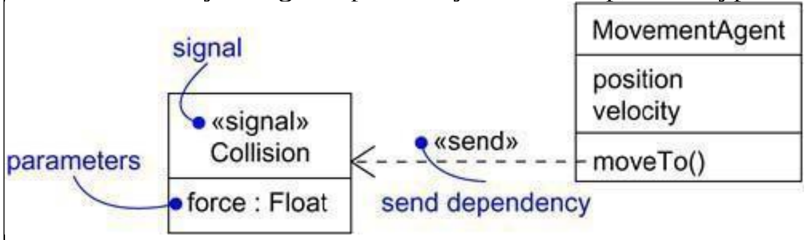
Q1 What do you mean by event and activity? Explain the signal event, change event and time event.

An **event** is an occurrence at a point in time, such as user depresses left button of mouse. An event happens instantaneously with regard to the time scale of an application. Events cause state changes which is shown in State Diagrams

An **activity** represents a business process. Fundamental elements of the activity are actions and
control elements (decision, division, merge, initiation, end, etc.). An action is an individual step
within an activity, for example, a calculation step that is not deconstructed any further, but that does not necessarily mean that the action cannot be subdivided in the real world.

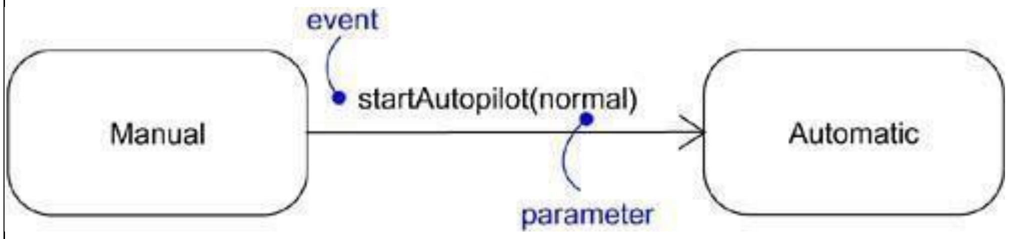
Signal Event

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



Call Event

a call event represents the dispatch of an operation
a call event is a synchronous event



Time and Change Events

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

Part – V

9	Define use case models? Explain use case diagram for vending machine. Hence describe guidelines.	10	CO5	L1

A ***use case*** is a coherent piece of functionality that a system can provide by interacting with actors. For example, a *customer* actor can *buy a beverage* from a vending machine. The customer inserts money into the machine, makes a selection, and ultimately receives a beverage. Similarly, a *repair technician* can *perform scheduled maintenance* on a vending machine.

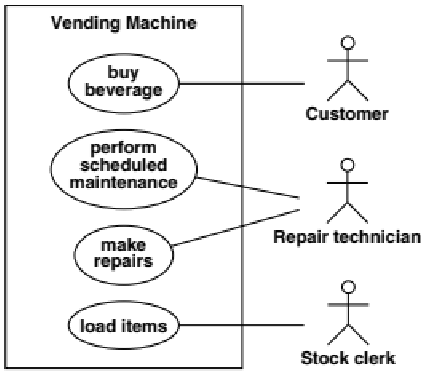
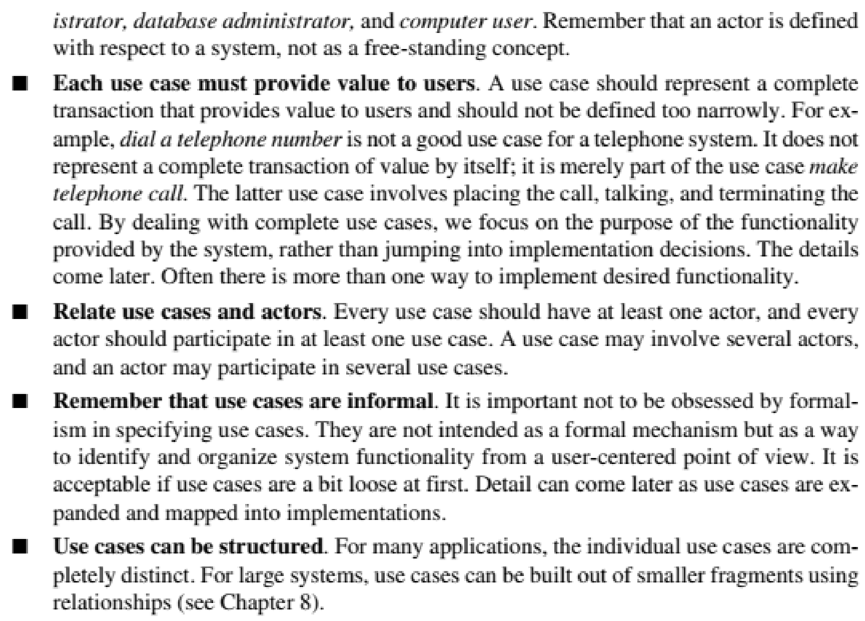


Figure 7.3 Use case diagram for a vending machine. A system involves a set of use cases and a set of actors.

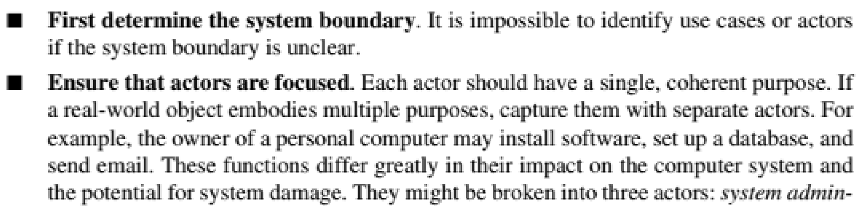
A rectangle contains the use cases for a system with the actors listed on the outside. The name of the system may be written near a side of the rectangle. A name within an ellipse denotes a use case. A "stick man" icon denotes an actor, with the name being placed below or adjacent to the icon. Solid lines connect use cases to participating actors. In the figure, the actor *Repair technician* participates in two use cases, the others in one each. Multiple actors can participate in a use case, even though the example has only one actor per use case.

Guidelines

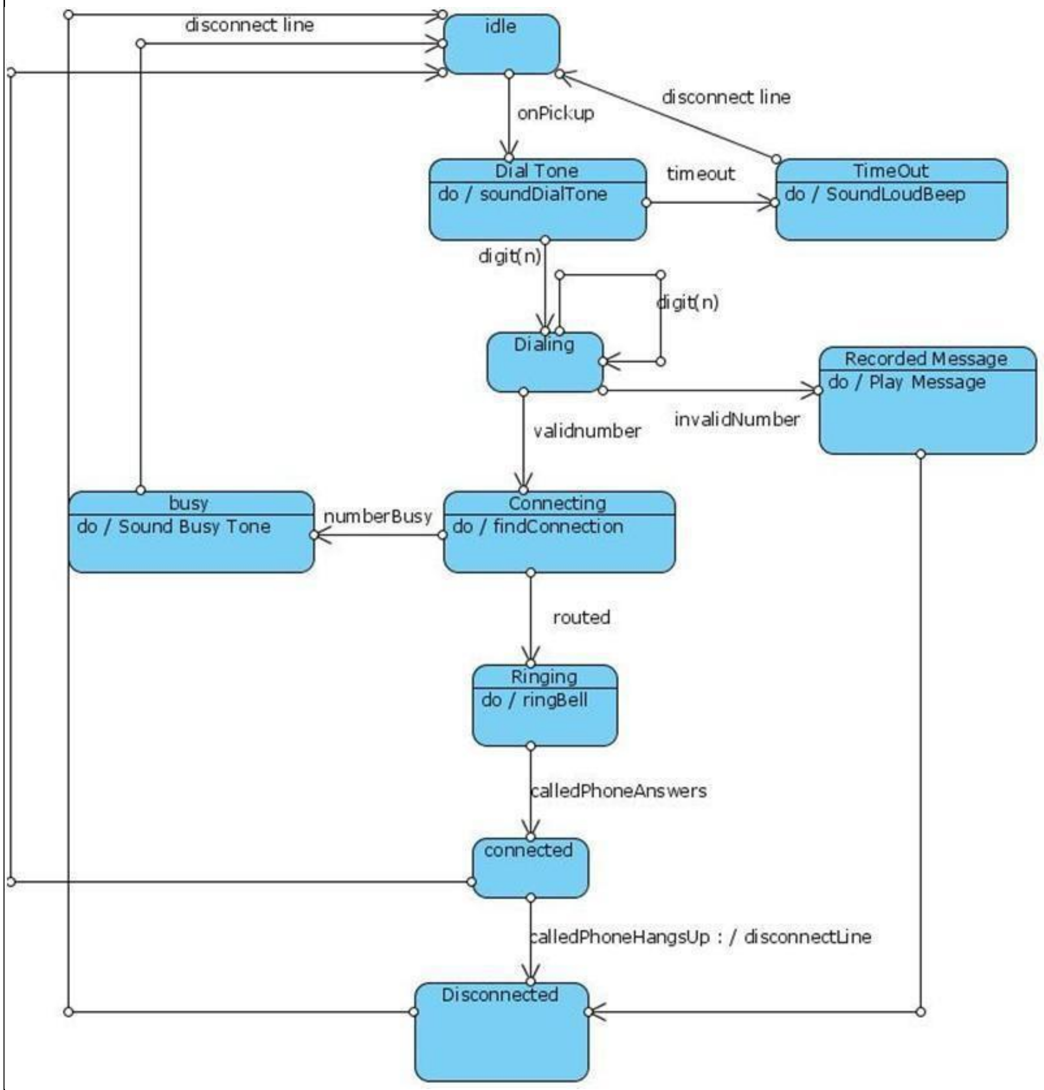


istrator, database administrator, and *computer user*. Remember that an actor is defined with respect to a system, not as a free-standing concept.

- **Each use case must provide value to users**. A use case should represent a complete transaction that provides value to users and should not be defined too narrowly. For example, *dial a telephone number* is not a good use case for a telephone system. It does not represent a complete transaction of value by itself; it is merely part of the use case *make telephone call*. The latter use case involves placing the call, talking, and terminating the call. By dealing with complete use cases, we focus on the purpose of the functionality provided by the system, rather than jumping into implementation decisions. The details come later. Often there is more than one way to implement desired functionality.
- **Relate use cases and actors**. Every use case should have at least one actor, and every actor should participate in at least one use case. A use case may involve several actors, and an actor may participate in several use cases.
- **Remember that use cases are informal**. It is important not to be obsessed by formalism in specifying use cases. They are not intended as a formal mechanism but as a way to identify and organize system functionality from a user-centered point of view. It is acceptable if use cases are a bit loose at first. Detail can come later as use cases are expanded and mapped into implementations.
- **Use cases can be structured**. For many applications, the individual use cases are completely distinct. For large systems, use cases can be built out of smaller fragments using relationships (see Chapter 8).



- **First determine the system boundary**. It is impossible to identify use cases or actors if the system boundary is unclear.
- **Ensure that actors are focused.** Each actor should have a single, coherent purpose. If a real-world object embodies multiple purposes, capture them with separate actors. For example, the owner of a personal computer may install software, set up a database, and send email. These functions differ greatly in their impact on the computer system and the potential for system damage. They might be broken into three actors: *system admin-*

(OR)				
10	What is the use of state diagram? Draw the state diagram of a telephone line A state diagram, also called a state machine diagram or statechart diagram, is an illustration of the states an object can attain as well as the transitions between those states in the Unified Modeling Language (UML). In this context, a state defines a stage in the evolution or behavior of an object, which is a specific entity in a program or the unit of code representing that entity. State diagrams are used to give an abstract description of the behavior of a system. This behavior is analyzed and represented as a series of events that can occur in one or more possible states.  disconnect line idle onPickup disconnect line Dial Tone do / soundDialTone timeout TimeOut do / SoundLoudBeep digit(n) digit(n) Dialing Recorded Message do / Play Message invalidNumber validnumber busy do / Sound Busy Tone numberBusy Connecting do / findConnection routed Ringing do / ringBell calledPhoneAnswers connected calledPhoneHangsUp : / disconnectLine Disconnected	10	CO5	L2