USN | | | | | | | | | |

Internal Assessment Test 1 – Sept 2018

Sub:	Object-Oriented Modeling And Design Patterns					Sub Code:	16MCA51	Branch:	MCA
Date:	11/09/2018	Duration:	90 min's	Max Marks:	50	Sem / Sec:	V /A & B		OBE

Answer any FIVE FULL Questions

		MARKS	CO	RBT
1 (a)	What is Pattern? Discuss pattern categories in detail.	[10]	CO5	L2,L1
2 (a)	Explain about Client-Dispatcher-Server design pattern.	[10]	CO5	L4
3 (a)	Describe the pattern description template in detail.	[10]	CO5	L2
4 (a)	What is a pattern? Describe the design pattern's three part scheme that underlies every pattern.	[5]	CO5	L2
(b)	Explain the structure and implementation of command processor .	[5]	CO5	L3
5 (a)	Explain problem, solution, structure and dynamics of Forwarder Receiver's design pattern in detail.	[10]	CO5	L4
6(a)	Write in detail about the structure and dynamics of MVC pattern.	[10]	CO5	L2
7(a)	Define communication design pattern ? .	[2]	CO5	L1
(b)	Discuss the problem ,solution and variants of Publisher and Subscriber	[8]	CO5	L2
8(a)	Write the structure and dynamics of the Proxy design pattern.	[5]	CO5	L2
(b)	Explain the structure and implementation of Whole -Part design pattern	[5]	CO5	L4

ANSWER KEY

1 (a) What is Pattern? Discuss pattern categories in detail. [10]

Abstracting from specific problem-solution pairs and distilling out common factors leads to patterns.
Each pattern is a three part rule:
• a relation between a certain context
• a problem
• and a solution

Pattern Categories
Some patterns help in structuring a software system into subsystems. Other patterns support the refinement of subsystems and components, or of the relationships between them. Further patterns help in implementing particular design aspects in a specific programming language.
Patterns also range from domain-independent ones, such as those for decoupling interacting components, to patterns addressing domain-specific.

We group patterns into three categories:
Architectural patterns
Design patterns
Idioms
Each category consists of patterns having a similar range of scale or abstraction.

1. Architectural Pattern:

Architectural Patterns express a fundamental structural organization for software systems
It provides a set of predefined sub-systems, specifies their responsibilities, and includes rules for establishing relationships between them
Example: 3-tier database systems (Database, Intermediate DB Layer, User-Application), Client/Server, Component (Module) -Based Software Systems, Feedback Systems, Event-Driven Systems
The Model-View-Controller pattern is one of the best-known examples of an architectural pattern. It provides a structure for interactive software systems.

2. Design pattern :

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 .
Design patterns 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. The application of a design pattern has no effect on the fundamental structure of a software system, but may have a strong influence on the architecture of a subsystem.
Many design patterns provide structures for decomposing more complex services or components.
We group design patterns into categories of related patterns, in the following :

- Structural Decomposition: This category includes patterns that support a suitable decomposition of subsystems and complex components into cooperating parts. The Whole-Part pattern
- Organization of Work.: This category comprises patterns that define how components collaborate together to solve a complex problem. The Master-Slave pattern
- Access Control. Such patterns guard and control access to services or components. The Proxy pattern (263
- Management. This category includes patterns for handling homogenous collections of objects, services and components in their entirety. We describe two patterns: the Command Processor pattern and the View Handler pattern
- Communication. Patterns in this category help to organize communication between components. Two patterns address issues of inter-process communication: the Forwarder-Receiver pattern deals with peer-to-peer communication, while the Client Dispatcher-Server pattern describes location-transparent communication in a Client-Server structure and the Publisher-Subscriber pattern.

3. Idioms:

Idioms deal with the implementation of particular design issues.
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.
Most idioms are language-specific-they capture existing programming experience. Often the same idiom looks different for different languages, and sometimes an idiom that is useful for one programming language does not make sense in another. For example, the C++ community uses reference-counting idioms to manage dynamically allocated resources;

2(a) Explain about Client-Dispatcher-Server design pattern. [10]

Intent :
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.
Example : Imagine we are developing a software system ACHILLES for the retrieval of new scientific information.
Context A software system integrating a set of distributed servers, with the servers running locally or distributed over a network.

Problem When a software system uses servers distributed over a network it must provide a means for communication between them. In many cases a connection between components may have to be established before the communication can take place, depending on the available communication facilities. However, the core functionality of the components should be separate from the details of communication mechanisms. Clients should not need to know where servers are located.
We have to balance the following forces:

- A component should be able to use a service independent of the location of the service provider.
- The code implementing the functional core of a service consumer should be separate from the code used to establish a connection with service providers.

Solution : Provide a dispatcher component to act as an intermediate layer between clients and servers. The dispatcher implements a name service that allows clients to refer to servers by names instead of physical locations, thus providing location transparency. In addition, the dispatcher is responsible for establishing the communication channel between a client and a server.
Add servers to the application that provides services to other components. Each server is uniquely identified by its name, and is connected to clients by the dispatcher.
Clients rely on the dispatcher to locate a particular server and to establish a communication link with the server. In contrast to traditional Client-Server computing, the roles of clients and servers can change dynamically.

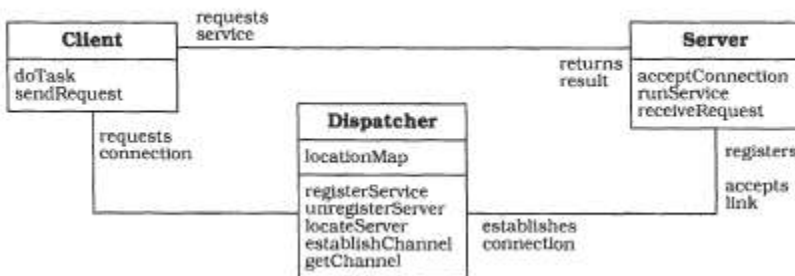
Structure

Class Client	***Collaborators***
Responsibility • Implements a system task. • Requests server connections from the dispatcher. • Invokes services of servers.	• Dispatcher • Server

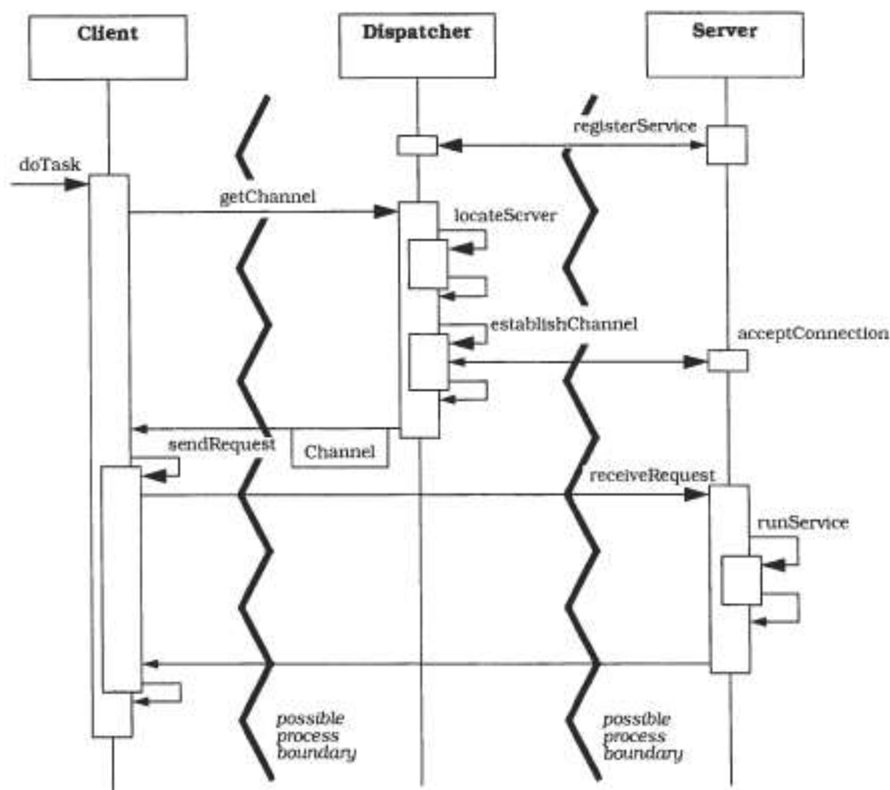
Class Server	***Collaborators***
Responsibility • Provides services to clients. • Registers itself with the dispatcher.	• Client • Dispatcher

Class Dispatcher	Collaborators • Client • Server
Responsibility • Establishes communication channels between clients and servers. • Locates servers. • (Un-)Registers servers. • Maintains a map of server locations.	

The static relationships between clients, servers and the dispatcher are as follows:



Dynamics A typical scenario for the Client-Dispatcher-Server design pattern includes the following phases:
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.

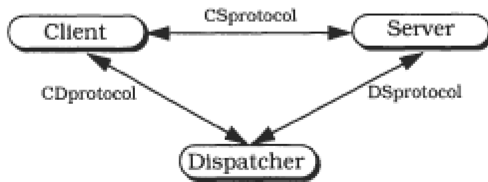


Implementation :

1 Separate the application into servers and clients. Define which components should be implemented as servers, and identify the clients that will access these servers.

2 ***Decide which communication facilities are required.*** Select communication facilities for the interaction between clients and the dispatcher, between servers and the dispatcher and between clients and servers. You can use a different communication mechanism for each connection, or you can use the same mechanism for all three.

3 ***Specify the interaction protocols between components.***



A protocol specifies an ordered sequence of activities for initializing and maintaining a communication channel between two components, as well as the structure of messages or data being transmitted. The Client-Dispatcher-Server pattern implies three different kinds of protocol. client about the failure. The dispatcher may try to establish a communication link several times before it reports an error.
This interaction could comprise the following steps:
- The client sends a message to the server using the communication channel previously established between them. To make this work, clients and servers need to share common knowledge about the syntax and semantics of messages they send and receive.
- The server receives the message, interprets it and invokes one of its services. After the service is completed, the server sends a return message to the client.
- The client extracts the service result from the message and continues with its task.

4 Decide how to name servers. The four-byte Internet IP address scheme is not applicable, because it does not provide location transparency. If IP addresses were used, a client would depend on the concrete location of the server. You need to introduce names that uniquely identify servers but do not carry any location information.

5 Design and implement the dispatcher, Determine how the protocols you introduced in step 3 should be implemented using available communication facilities.

6. Implement the client and server components according to your desired solution and the decisions you make about the dispatcher interface. Configure the system and either register the servers with the dispatcher or let the servers dynamically register and unregister themselves.

Example Resolved: In our scientific information example ACHILLES, a TCP port number and the Internet address of the host machine are combined to uniquely identify servers.

Variants

- Distributed Dispatchers. Instead of using a single dispatcher component in a network environment, distributed dispatchers may be introduced. In this variant, when a dispatcher receives a client request for a server on a remote machine, it establishes a connection with the dispatcher on the target node.
- Client-Dispatcher-Server with communication managed by clients. Instead of establishing a communication channel to servers, a dispatcher may only return the physical server location to the client. It is then the responsibility of the client to manage all communication activities with the server.
- Client-Dispatcher-Server with heterogeneous communication. It is not always possible to implement the communication between clients and servers using only one communication mechanism. Some servers may use sockets, while others use named pipes.
- Client-Dispatcher-Server. In this variant, clients address services and not servers. When the dispatcher receives a request, it looks up which servers provide the specified server in its repository, and establishes a connection to one of these service providers. If it falls to establish the connection, it may try to access another server providing the same service instead, if one is available.

Known Uses : Sun's implementation of **Remote Procedure Calls** is based upon the principles of the Client-Dispatcher-Server design pattern
The OMQ Corba (Common Object Request Broker Architecture) specification uses the principles of the Client-Dispatcher-Server design pattern for relining and instantiating the Broker architectural pattern.
Consequences The Client-Dispatcher-Server design pattern has several **benefits:**
Exchangeability of servers.
Location and migration transparency.
Re-configuration
Fault tolerance.
The Client-Dispatcher-Server design pattern imposes some **liabilities:**
Lower efficiency through indirection and explicit connection establishment.
Sensitivity to change in the interfaces of the dispatcher component.
Because the dispatcher plays the central role, the software system is sensitive to changes in the interface of the dispatcher.
See also
The ***Forwarder-Receiver*** design pattern can be combined with the Client-Dispatcher-Server pattern to hide the details of inter-process communication. While the Client-Dispatcher-Server pattern allows you to decouple clients and servers by supporting location transparency, it does not encapsulate the details of the underlying communication facilities.
The ***Acceptor and Connector*** patterns demonstrate a different way to decouple connection set-up from connection processing. Schmidt's patterns are more decentralized than our approach, which uses a centralized dispatcher.

3 (a) Describe the pattern description template in detail. [10]

Name The name and a short summary of the pattern.
Also Known As Other names for the pattern, if any are known.
Example A real-world example demonstrating the existence of the problem and the need for the pattern.
Context The situations in which the pattern may apply
Problem The problem the pattern addresses, including a discussion of its associated forces.
Solution The fundamental solution principle underlying the pattern.
Structure A detailed specification of the structural aspects of the pattern.
Dynamics Typical scenarios describing the run-time behavior of the pattern.
Implementation : Guidelines for implementing the pattern. These are only a suggestion, not an immutable rule. You should adapt the implementation to meet your needs, by adding different, extra, or more detailed steps, or by re-ordering the steps.
Example Resolved : Discussion of any important aspects for resolving the example that are not yet covered in the Solution, Structure, Dynamics and implementation sections.
Variants : A brief description of variants or specializations of a pattern.
Known Uses : Examples of the use of the pattern, taken from existing systems.
Consequences : The benefits the pattern provides, and any potential liabilities.
See Also : References to patterns that solve similar problems, and to patterns
that help us refine the pattern we are describing.

4 (a) What is a pattern? Describe the design pattern's three part scheme that underlies every pattern.

Abstracting from specific problem-solution pairs and distilling out common factors leads to patterns.
Each pattern is a three part rule:
• a relation between a certain context
• a problem
• and a solution

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.
Context:
The context extends the plain problem-solution by describing situations in which the problem occurs. The context of a pattern may be fairly general, for example 'developing software with a human-computer interface.' Specifying the correct context for a pattern is difficult. We find it practically impossible to determine all situations, both general and

specific, in which a pattern may be applied. **A** more pragmatic approach is to list all known situations where a problem that is addressed by a particular pattern can occur. It at least gives valuable guidance.

Problem :

This part of a 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.

Problem that should be considered when solving it, such as:

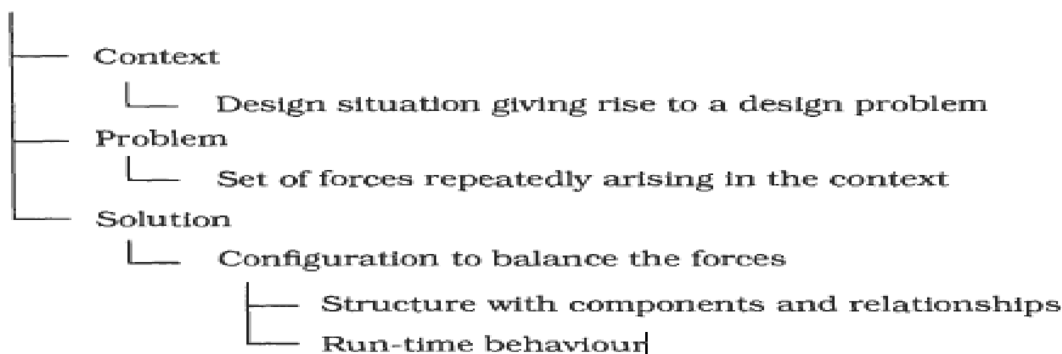
1.Requirements the solution must fulfill-for example, that peer-to-peer inter-process communication must be efficient.
2.Constraints you must consider-for example, that inter-process communication must follow a particular protocol.
3.Desirable properties the solution should have-for example, that changing software should be easy.

Solution:

The solution part of a pattern shows how to solve the recurring problem, or better, how to balance the forces associated with it. In software architecture such a solution includes two aspects.

Firstly, every pattern specifies a certain structure, a spatial configuration of elements. This structure addresses the *static* aspects of the solution. Since such a structure can be seen as a micro-architecture ,it consists, like any software architecture, of both components and their relationships.

Secondly, every pattern specifies run-time behavior. This run-time behavior addresses the *dynamic* aspects of the solution.



4b) Explain the structure and implementation of command processor.

Structure

The abstract command component defines the interface of all command objects. As a minimum this interface consists of a procedure to execute a command. The additional services implemented by the command processor require further interface procedures for all command objects. The abstract command class of TEDDI, for example, defines an additional undo method.

For each user function we derive a *command component* from the abstract command. A command component implements the interface of the abstract command by using zero or more *supplier components.*

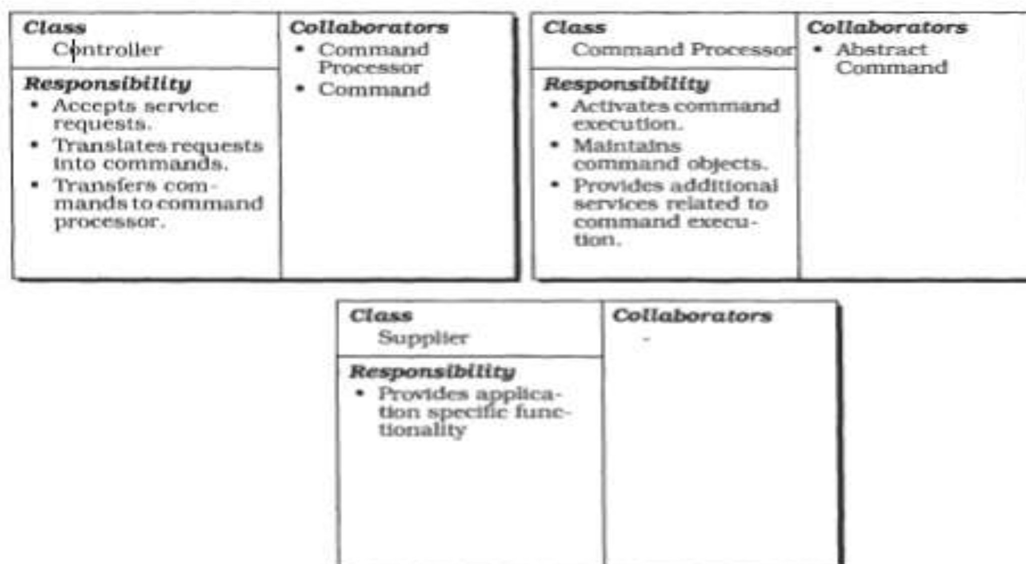
The commands of TEDDI save the state of associated supplier components prior to execution, and restore it in case of undo.

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.	

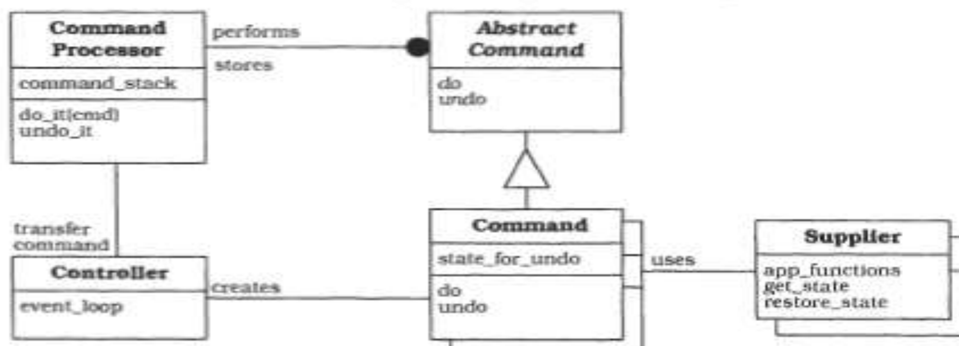
Class Command	***Collaborators*** • Supplier
Responsibility • Encapsulates a function request. • Implements interface of abstract command. • Uses suppliers to perform a request.	

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 *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 command processor remains independent of specific commands because it only uses the abstract command interface.
The *supplier* components provide most of the functionality required to execute concrete commands (that is. those related to the concrete command class, as opposed to the abstract command class). Related commands often share supplier components. When an undo mechanism is required, a supplier usually provides a means to save and restore its internal state. The component implementing the internal text representation is the main supplier in TEDDI.



The following diagram shows the principal relationships between the components of the pattern. It demonstrates undo as an example of an additional service provided by the command processor.



Implementation

1 ***Define the interface of the abstract command.*** The abstract command class hides the details of **all** specific commands. This class always specifies the abstract method required to execute a command. It also defines the methods necessary to implement the additional services offered by the command processor.
For the undo mechanism in TEDDI we distinguish three types of commands. They are modeled as an enumeration, because the command type may change dynamically, as shown in step **3:**
No change. A command that requires no undo. Cursor movement falls into this category.
Normal. A command that can be undone. Substitution of a word in text is an example of a normal command.
No undo. A command that cannot be undone, and which prevents the undo of previously performed normal commands.
2 Design the command components for each type of request that the application supports. There are several options for binding a command to its suppliers. The supplier component can be hardcoded within the command. or the controller can provide the supplier to the command constructor as a parameter.
3 Increase flexibility by providing macro commands that combine several successive commands.
4 Implement the controller component. Command objects are created by the controller. However, since the

controller is already decoupled from the supplier components, this additional decoupling of controller and commands is optional.

5 Implement access to the additional services of the command processor:

A user-accessible additional service is normally implemented by a specific command class. The command processor supplies the functionality for the 'do' method. Directly calling the interface of the command processor is also an option.

6 Implement the command processor component. The command processor receives command objects from the controller and takes responsibility for them. For each command object, the command processor starts the execution by calling the do method.

5a) Explain problem, solution, structure and dynamics of Forwarder Receiver's design pattern in detail.

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

Example: The company Dwarf Ware offers applications for the management of the computer networks. System consists of agent processes written in Java that run on each available network node.

Context – peer-to-peer communication

Problem forces –

1.The system should allow the exchangeability of the communication mechanisms.

2.The co-operation of components follows a peer-to-peer model, in which a sender only needs to know names of its receivers.

3.The communication between peers should not have a major impact on performance.

Solution – peers may act as clients or servers. Therefore the details of the underlying IPC mechanisms for sending or receiving messages are hidden from peers by encapsulating all system-specific functionality into separate components. system specific functionalities are the mapping of names to physical locations, the establishment of communication channels and marshaling and un marshaling messages.

Structure

Class Forwarder	***Collaborators***
Responsibility • Provides a general interface for sending messages. • Marshals and delivers messages to remote receivers. • Maps names to physical addresses.	• Receiver

Class Receiver	***Collaborators***
Responsibility • Provides a general interface for receiving messages. • Receives and unmarshals messages from remote forwarders.	• Forwarder

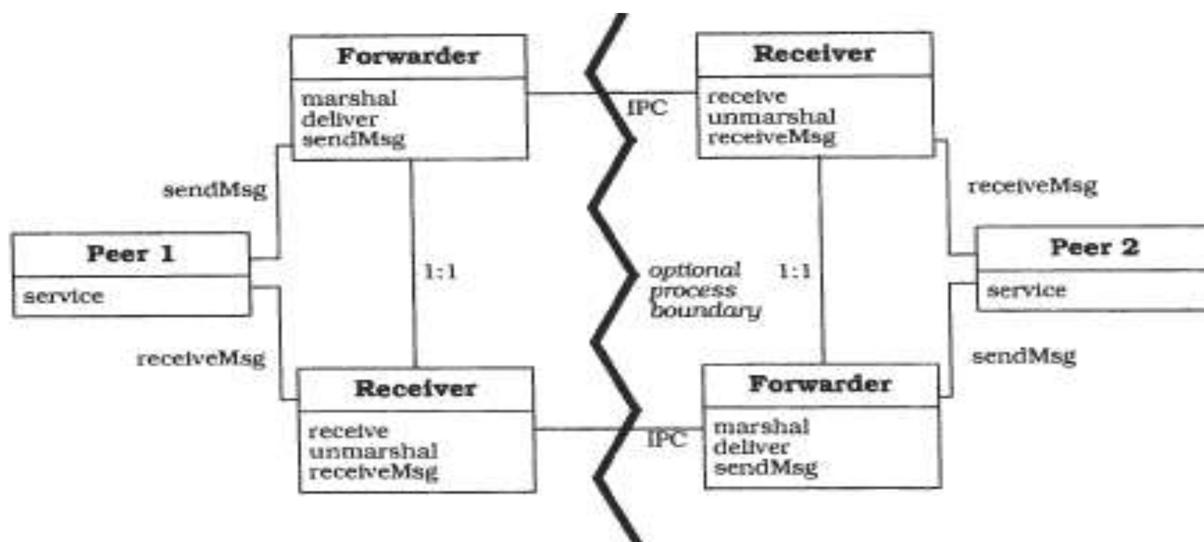
Class Peer	***Collaborators***
Responsibility • Provides application services. • Communicates with other peers.	• Forwarder • Receiver

The static relationships **in** the Forwarder-Receiver design pattern are shown in the diagram below.

To send a message to a remote peer, the peer invokes the method **sendMsg** of its forwarder, passing the message as an argument. The method **sendMsg** must convert messages to a format that the underlying IPC mechanism understands. For this purpose, it calls **marshal. sendMsg** uses **deliver** to transmit the IPC message data to a remote receiver.

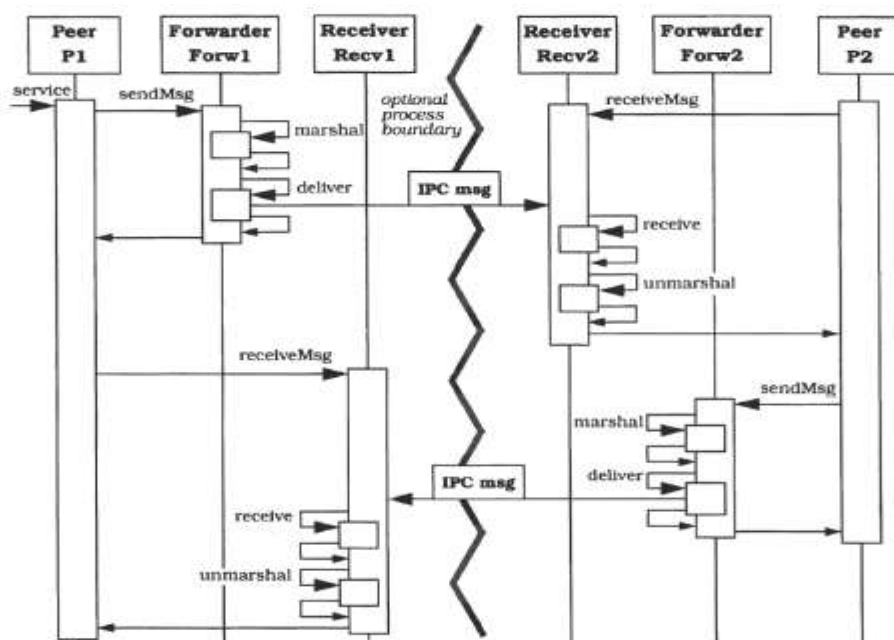
When the peer wants to receive a message from a remote peer, it invokes the **receiveMsg** method of its receiver, and the message is returned. **receiveMsg** invokes **receive,** which uses the functionality

of the underlying IPC mechanism to receive IPC messages. After message reception `receiveMsg` calls `unmarshal` to convert **IPC** messages to a format that the peer understands.



Dynamics

The following scenario illustrates a typical example of the use of a Forwarder-Receiver structure.
Two peers `pl` and `p2` communicate with each other. For this purpose, `PI` uses a forwarder `forw1` and a receiver `Recv2` handles all message transfers with a forwarder `Forw2` and a receiver `Recv2 :PI` requests a service from a remote peer`2.`Fo r this purpose, 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 the message.`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`2 .`Meanwhile. pl calls Its receiver Recvl to wait for a response.
P2 performs the requested service, and sends the result and the name of the recipient pl to the forwarder forw2.T he forwarder marshals the result and delivers it Recv l . Recvl receives the response from P2, unmarshals it and delivers it to P1.



6(a) Write in detail about the structure and dynamics of MVC pattern.

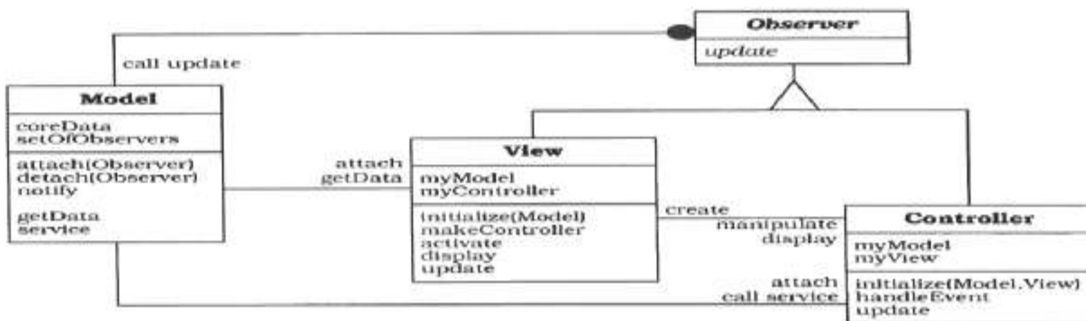
The Model-View-Controller architectural pattern (MVC) divides an interactive application into three components. The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model.

Structure:

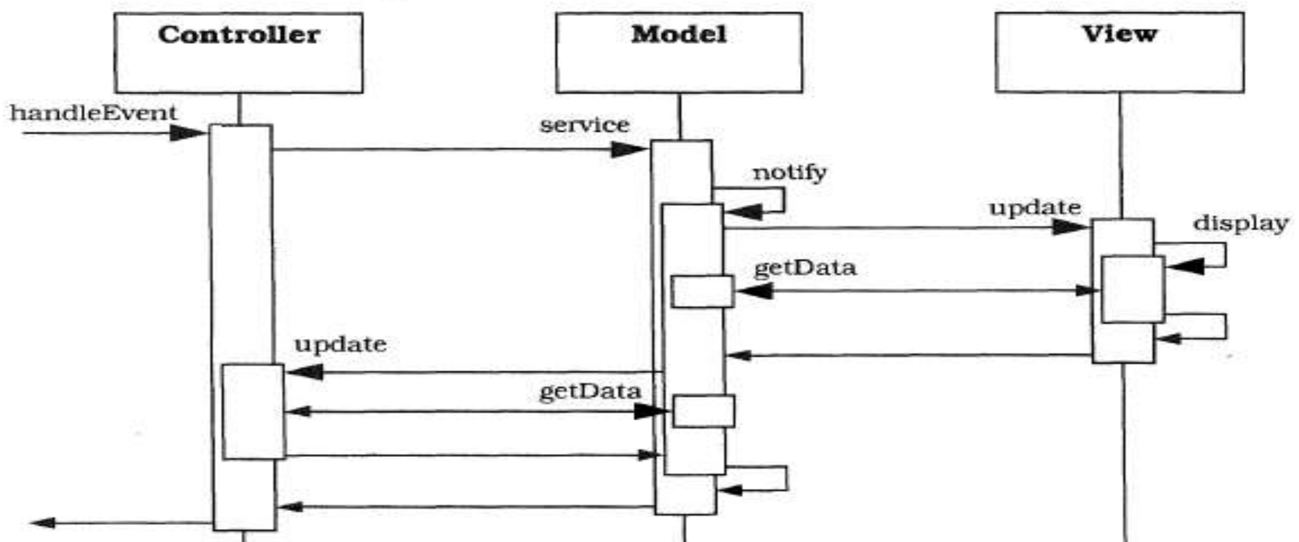
Class Model	*Collaborators*
Responsibility • Provides functional core of the application. • Registers dependent views and controllers. • Notifies dependent components about data changes.	• View • Controller

Class View	*Collaborators*
Responsibility • Creates and initializes its associated controller. • Displays information to the user. • Implements the update procedure. • Retrieves data from the model.	• Controller • Model

Class Controller	*Collaborators*
Responsibility • Accepts user input as events. • Translates events to service requests for the model or display requests for the view. • Implements the update procedure, if required.	• View • Model

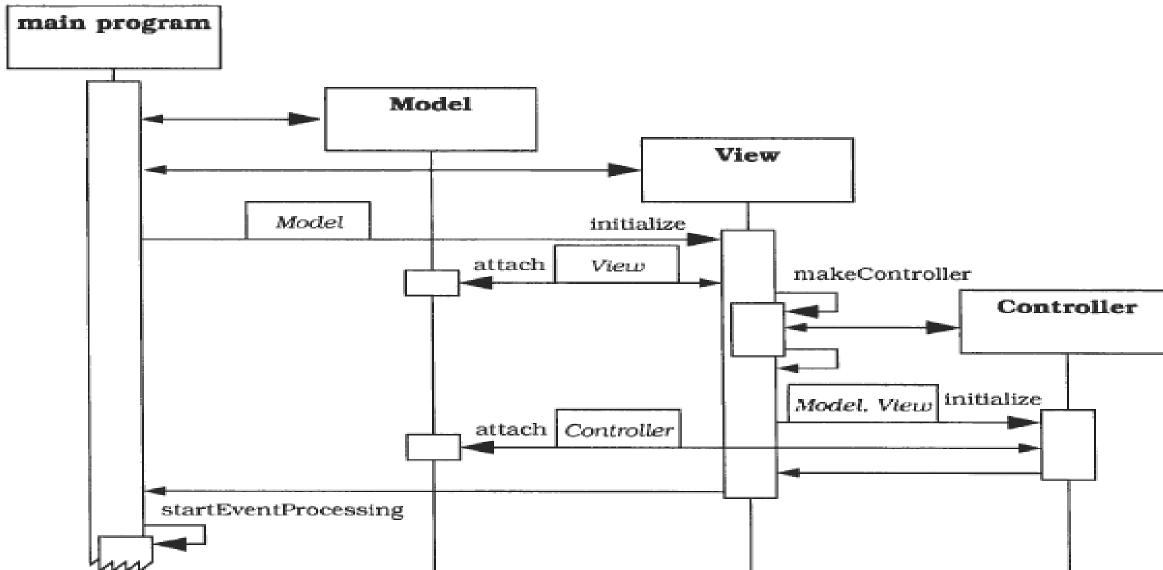


Dynamics: Scenario I : Only one view-controller pair of MVC



Scenario II

- The model instance is created, which then initializes its internal data structures.
- A view object is created. This takes a reference to the model as a parameter for its initialization.
- The view subscribes to the change-propagation mechanism of the model by calling the attach procedure.
- The view continues initialization by creating its controller. It passes references both to the model and to itself to the controller's initialization procedure.
- The controller also subscribes to the change-propagation mechanism by calling the attach procedure.
- After initialization, the application begins to process events.



7a) Define communication design pattern ? **[2]**

Communication. Patterns in this category help to organize communication between components. Two patterns address issues of inter-process communication: the Forwarder-Receiver pattern deals with peer-to-peer communication, while the Client Dispatcher-Server pattern describes location-transparent
communication in a Client-Server structure.

7b) Discuss the problem ,solution and variants of Publisher and Subscriber

The Publisher-Subscriber design pattern helps to keep the state of cooperating components synchronized. To achieve this it enables one-way propagation of changes: one publisher notifies any number of subscribers about changes to its state.

Problem :

A change-propagation mechanism is applicable in many contexts.
The solution should balance the following forces:
One or more components must be notified about state changes in a particular component.
The number and identities of dependent components is not known a priori, or may even change over time.
Explicit polling by dependents for new information is not feasible. The information publisher and its dependents should not be tightly coupled when introducing a change-propagation mechanism.

Solution:

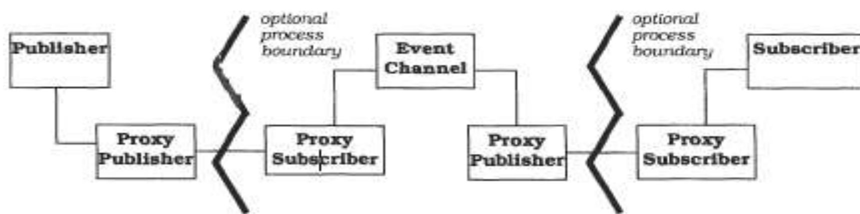
One component takes the role of the publisher (called subject]). All components dependent on changes in the publisher are its subscribers (called observers]). The publisher maintains a registry of currently-subscribed components. Whenever a component wants to become a subscriber it uses the subscribe interface offered by the publisher. Analogously, it can unsubscribe.
Whenever the publisher changes state, it sends a notification to all its subscribers. The subscribers in turn retrieve the changed data at their discretion.

Variants

Gatekeeper The Publisher-Subscriber pattern can be also applied to distributed systems. In this variant a publisher instance in one process notifies remote subscribers. The publisher may alternatively be spread over two processes. In one process a component sends out messages, while in the receiving process a singleton 'gatekeeper' de multiplexes them by surveying the entry points to the process. The gatekeeper notifies event-handling subscribers when events for which they registered occur.

The ***Event Channel*** variant is targeted at distributed systems. This pattern strongly decouples publishers and subscribers. In this variant, an event channel is created and placed between the publisher and the subscribers. To publishers the event channel appears as a subscriber, while to subscribers it appears as a publisher

The Producer- Consumer style of cooperation. In this a producer supplies information, while a consumer accepts this information for further processing. Producer and consumer are strongly decoupled, often by placing a buffer; between them. The producer writes to the buffer without any regard for the consumer. The consumer reads data from the buffer at its own discretion. The only synchronization carried out is checking for buffer overflow and underflow. The producer is suspended when the buffer is full, while the consumer waits if it cannot read data because the buffer is empty. Another difference between the Publisher-Subscriber pattern and the Producer-Consumer variant is that in the latter producers and consumers are usually in a 1 : 1 relationship.



8a) Write the structure and dynamics of the Proxy design pattern.

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

Structure:

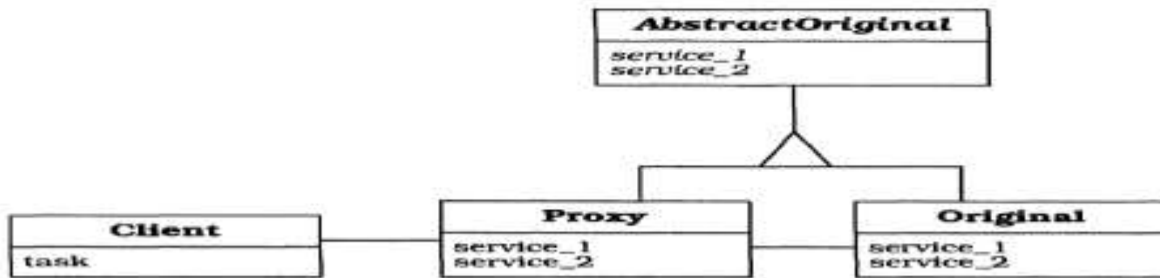
Class Client	Collaborators
Responsibilities • Uses the interface provided by the proxy to request a particular service. • Fulfills its own task.	• Proxy

Class *AbstractOriginal*	Collaborators
Responsibilities • Serves as an abstract base class for the proxy and the original.	-

Class Proxy	Collaborator
Responsibilities • Provides the interface of the original to clients. • Ensures a safe, efficient and correct access to the original.	• Original

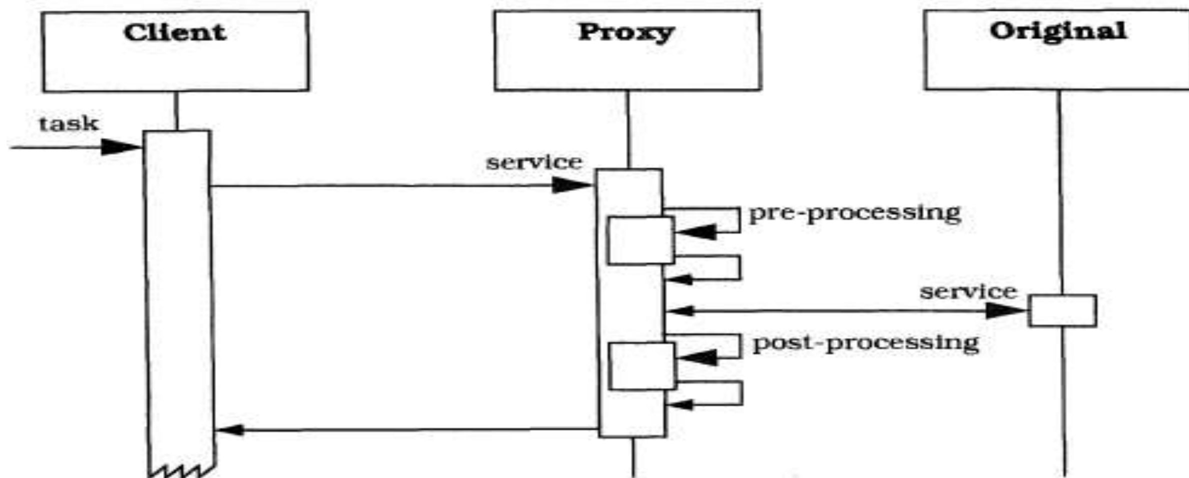
Class Original	Collaborators
Responsibilities • Implements a particular service.	-

The following OMT diagram shows the relationships between the classes graphically:



Dynamics : The actions performed within the proxy differ depending on its actual specialization.

- While working on its task the client asks the proxy to carry out a service.
- The proxy receives the incoming service request and pre-processes it. This pre-processing involves actions such as looking up the address of the original, or checking a local cache to see if the requested information is already available.
- If the proxy has to consult the original to fulfill the request, it forwards the request to the original using the proper communication protocols and security measures.
- The original accepts the request and fulfills it. It sends the response back to the proxy.
- The proxy receives the response. Before or after transferring it to the client it may carry out additional post-processing actions such as caching the result, calling the destructor of the original or releasing a lock on a resource.



8b) Explain the structure and implementation of Whole -Part design pattern

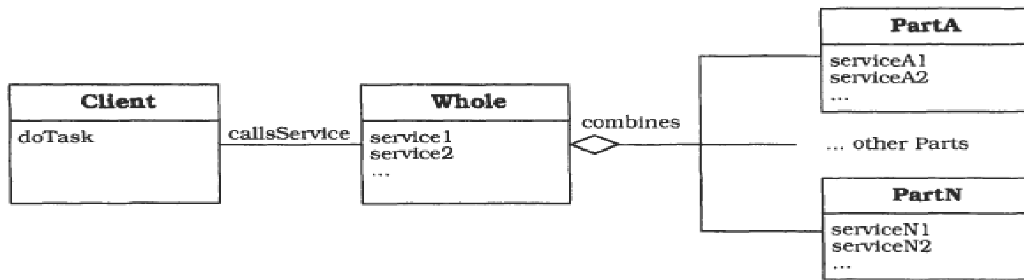
The Whole-Part design pattern helps with the aggregation of components that together form a semantic unit. An aggregate component, the whole, encapsulates its constituent components, the Parts, organizes their collaboration, and provides a common interface to its functionality. Direct access to the Parts Is not possible.

Structure :

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.	-

The static relationships between a Whole and its Parts are illustrated in the OMT diagram below:



Implementation:

1.Design the public interface of the Whole.

2.Separate the Whole into Parts, or synthesize it from existing ones. There are two approaches to assembling the Parts you need—either assemble a Whole 'bottom-up' from existing Parts, or decompose it 'top-down' into smaller Parts:

- The bottom-up approach allows you to compose Wholes from loosely-coupled Parts that you can later reuse when implementing other types of Whole.
- The top-down approach makes it is possible to cover all of the Whole 's functionality. Partitioning into Parts is driven by the services the Whole provides to its clients, freeing you from the requirement to implement glue code.

3. If you follow a bottom-up approach, use existing Parts from component libraries or class libraries and specify their collaboration.

4. If you follow a top-down approach, partition the Whole 's services into smaller collaborating services and map these collaborating services to separate Parts.

5.Specify the services of the Whole in terms of services of the Parts. In the structure you found in the previous two steps, the Whole is represented as a set of collaborating Parts with separate responsibilities.

6. Implement the Parts.

7. Implement the Whole.