Sub:	Object-Oriented Modeling And Design Patterns					Sub Code:	13MCA51	Branch:
Date:	18/09/2017	Duration:	90 min’s	Max Marks:	50	Sem / Sec:	V /A & B	

Answer any FIVE FULL Questions

		MARKS
1 (a)	What is Pattern? Discuss pattern categories in detail.	[10]
2(a)	Explain about Client-Dispatcher-Server design pattern.	[10]
3 (a)	Describe the pattern description template in detail.	[10]
4 (a)	Write a program in JAVA to implement the Publisher-Subscriber pattern .	[6]
(b)	Discuss structure and variants of the Publisher-Subscriber design pattern?	[4]
5 (a)	Explain the structure and implementation of command processor with a diagram.	[10]
6 (a)	Write in detail about the structure and dynamics of View Handler.	[10]
7(a)	Define communication design pattern ? .	[2]
(b)	Describe the various components of Forwarder Receiver design pattern in the structure along with the Dynamics.	[8]
8(a)	Write the problem and solution of the Master-Slave design pattern.	[5]
8(b)	Explain the structure of this design pattern	[5]

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]

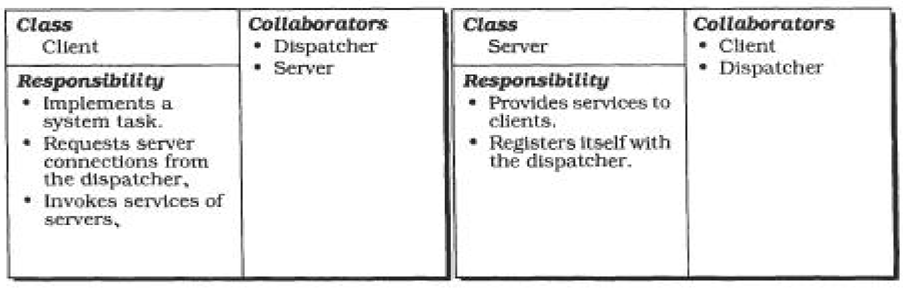
Intent :
The Client-Dispatcher-Server design pattern introduces an intermediate layer between clients and servers, the dispatcher component. It provides location transparency by means of a name service, and hides the details of the establishment of the communication connection between clients and servers.
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.

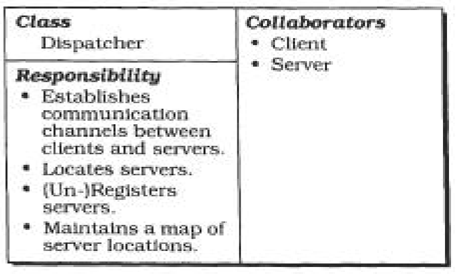
Solution : Provide a dispatcher component to act as an intermediate layer between clients and servers. The dispatcher implements a name service that allows clients to refer to servers by names instead of physical locations, thus providing location transparency. In addition, the dispatcher is responsible 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



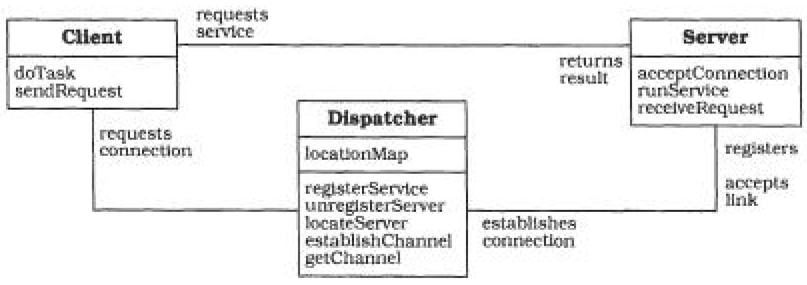
Class Client	*Collaborators*
Responsibility • Implements a system task. • Requests server connections from the dispatcher. • Invokes services of servers.	• Dispatcher • Server

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

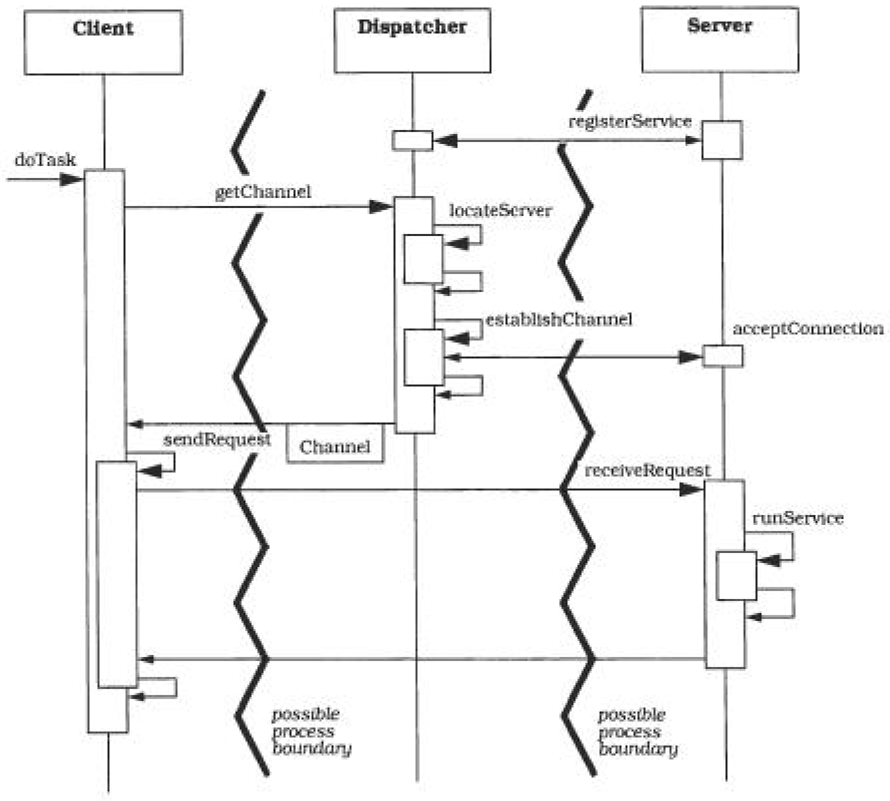


Class Dispatcher	*Collaborators*
Responsibility • Establishes communication channels between clients and servers. • Locates servers. • (Un-)Registers servers. • Maintains a map of server locations.	• Client • Server

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

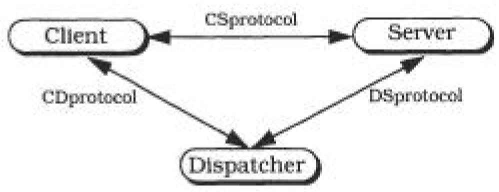


Dynamics A typical scenario for the Client-Dispatcher-Server design pattern includes the following phases:
A server registers itself with the dispatcher component.
At a later time, a client asks the dispatcher for a communication channel to a specified server.
The dispatcher looks up the server that 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 Typica1 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) Write a program in JAVA to implement the Publisher-Subscriber pattern . [6]

WeatherData.java:

```
import java.util.Observable;
public class WeatherData extends Observable
{
        public static void main(String a[])
        {
        WeatherData weatherData= new WeatherData();
        CurrentDisplay cd=new CurrentDisplay(weatherData);
        ForecastDisplay fd=new ForecastDisplay(weatherData);
        weatherData.SetMeasurements(80f,65f,30.6f);
        weatherData.SetMeasurements(82f,60f,29.5f);
        weatherData.SetMeasurements(78f,90f,29.2f);
        }
        private float temperature;
        public float getTemperature()
        {
                return temperature;
        }
        public void setTemperature(float theTemperature)
        {
                temperature = theTemperature;
        }
        private float pressure;
        public float getPressure()
        {
                return pressure;
        }
        public void setPressure(float thePressure)
        {
                pressure = thePressure;
        }
        private float humidity;
        public float getHumidity()
        {
                return humidity;
        }
        public void setHumidity(float theHumidity)
        {
                humidity = theHumidity;
        }
        public WeatherData()
        {
        }
        public void SetMeasurements(float f, float h, float p)
        {
                this.temperature=f;
                this.humidity=h;
                this.pressure=p;
                MeasurementsChanged();
        }
        public void MeasurementsChanged()
        {
                setChanged();
                notifyObservers();
        }
        public void Notify()
        {
        }
}
```

CurrentDisplay.java:

```
import java.util.Observable;
import java.util.Observer;
public class CurrentDisplay implements Observer
{
        private float temperature;
        public float getTemperature()
        {
                return temperature;
        }
        public void setTemperature(float theTemperature)
        {
                temperature = theTemperature;
        }
        private float pressure;
        public float getPressure()
        {
                return pressure;
        }
        public void setPressure(float thePressure)
        {
                pressure = thePressure;
        }
        private float humidity;
        public float getHumidity()
        {
                return humidity;
        }
        public void setHumidity(float theHumidity)
        {
                humidity = theHumidity;
        }
        private Object observable;
        public CurrentDisplay(Observable observable)
        {
                this.observable=observable;
                observable.addObserver(this);
        }
        public Object getObservable()
        {
                return observable;
        }
        public void setObservable(Object theObservable)
        {
                observable = theObservable;
        }
        public CurrentDisplay()
        {
        }
        public void Update(Observable obs,Object arg)
        {
                WeatherData wd=(WeatherData)obs;
                this.temperature=wd.getTemperature();
```

```
		this.humidity=wd.getHumidity();
		this.pressure=wd.getPressure();
	Display();
	}
	public void Display()
	{
		System.out.println("Current COndition"+temperature+"Fand"+humidity+"%humdity");
	}
	public void update(Observable o, Object arg)
	{
		WeatherData wd=(WeatherData)o;
		this.temperature=wd.getTemperature();
		this.humidity=wd.getHumidity();
		this.pressure=wd.getPressure();
		Display();
	}
}
```

ForecastDisplay.java:

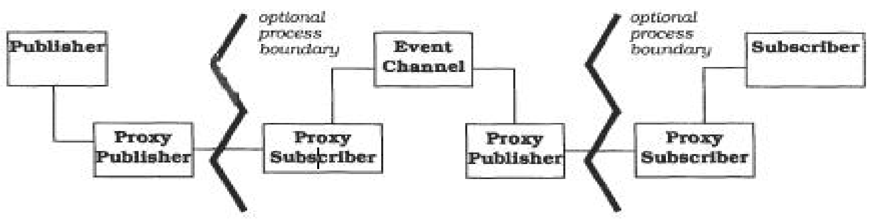
```
import java.util.Observable;
import java.util.Observer;
public class ForecastDisplay implements Observer
{
	private float CurrentPressure=30.52f;
	Observable observable;
	public ForecastDisplay(Observable observable)
	{
		this.observable=observable;
		observable.addObserver(this);
	}
	public float getCurrentPressure()
	{
		return CurrentPressure;
	}

	public void setCurrentPressure(float theCurrentPressure)
	{
		CurrentPressure = theCurrentPressure;
	}
private float LastPressure;
	public float getLastPressure()
	 {
		return LastPressure;
	}
	public void setLastPressure(float theLastPressure)
	{
		LastPressure = theLastPressure;
	}
	public void Update(Observable o, Object arg)
	{
		if(o instanceof WeatherData)
		{
			WeatherData wd=(WeatherData)o;
```

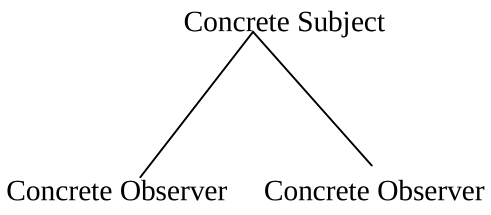
```
                    LastPressure=CurrentPressure;
                    CurrentPressure=wd.getPressure();
                    Display();
               }
          }
          public void Display()
          {
               System.out.println("Weather Forecadt");
               if(CurrentPressure>LastPressure)
                    System.out.println("Weather Conditions Are Improving.");
               else if(CurrentPressure==LastPressure)
                    System.out.println("Weather Conditions Are The Same.");
               else
                    System.out.println("Weather Out Of Control.");
          }
          public void update(Observable o, Object arg)
           {
               if(o instanceof WeatherData)
               {
                    WeatherData wd=(WeatherData)o;
                    LastPressure=CurrentPressure;
                    CurrentPressure=wd.getPressure();
                    Display();
               }
          }
}
```

4(b).Discuss structure and variants of the Publisher-Subscriber design pattern? [4]

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.



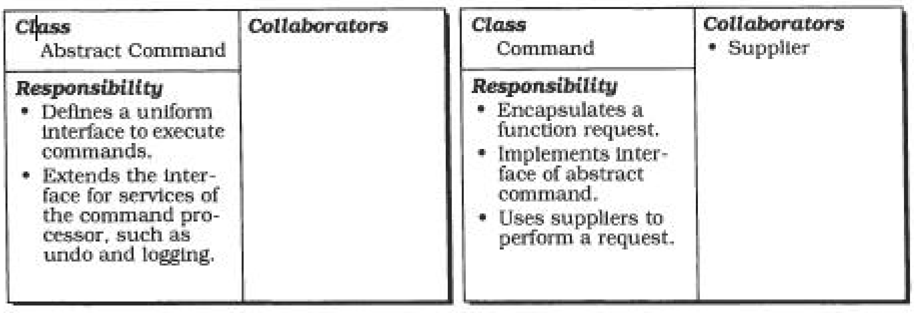
STRUCTURE:



In the push model, the publisher sends all changed data when it notifies the subscribers. The subscribers have no choice about if and when they want to retrieve the data-they just get it.
In the pull model, the publisher only sends minimal information when sending a change notification-the subscribers are responsible for retrieving the data they need.
Many variations are possible in the middle ground between these two extremes.
The push model has a very rigid dynamic behavior, whereas the pull model offers more flexibility, at the expense of a higher number of messages between publisher and subscribers.
Generally, the push model Is a better choice when the subscribers need the published information most of the time.
The pull model is used when only the individual subscribers can decide if and when they need a specific piece of information

5 (a) Explain the structure and implementation of command processor with a diagram. [10]

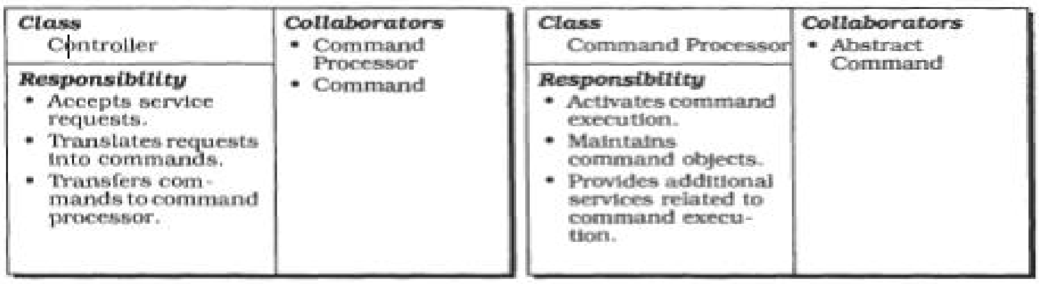
Structure
The abstract command component defines the interface of all command objects. As a minimum this interface consists of a procedure to execute a command. The additional services implemented by the command processor require further interface procedures for all command objects. The abstract command 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.	

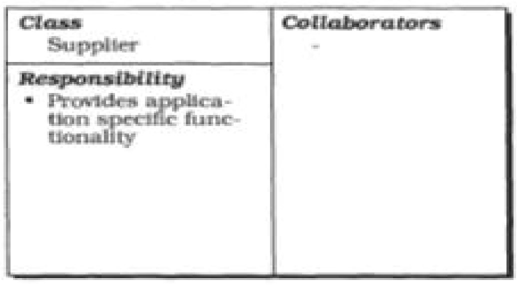
Class Command	***Collaborators*** • Supplier
Responsibility • Encapsulates a function request. • Implements interface of abstract command. • Uses suppliers to perform a request.	

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.



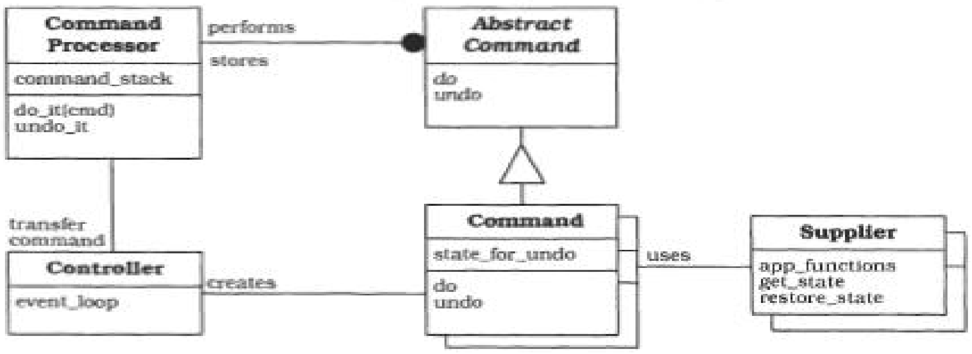
Class Controller	Collaborators • Command Processor • Command
Responsibility • Accepts service requests. • Translates requests into commands. • Transfers commands to command processor.	

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	

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.

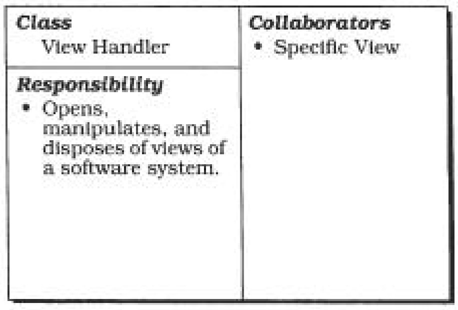


Implementation

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.

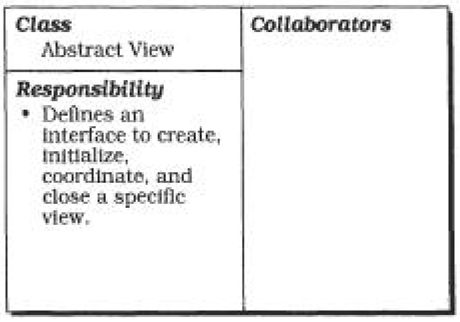
6 (a) Write in detail about the structure and dynamics of View Handler. [10]

Structure The ***view*** *handler* is the central component of this pattern. It is responsible for opening new views, and clients can specify the view they want. The view handler instantiates the corresponding view component, takes care of its correct initialization, and asks the new view to display itself. If the requested view is open already, the view handler brings this open view to the foreground. If the requested view is open , the view handler tells the view to display itself full size.
The view handler also offers functions for closing views, both individual ones and all currently-open views, as is needed when quitting the application. The main responsibility of the view handler, however, is to offer view management services.
.

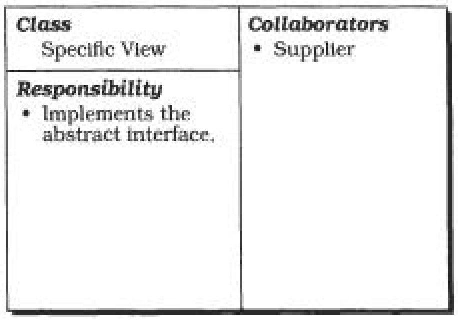


Class View Handler	***Collaborators***
Responsibility	• Specific View
• Opens, manipulates, and disposes of views of a software system.	

An additional responsibility of the view handler is coordination. There may be dependencies between views such as occurs. If a user modifies one view of the document, it may be necessary to update the others in a predefined order.
An abstract view component defines an interface that is common to all views. The view handler uses this interface for creating, coordinating, and closing views. The platform underlying the system uses the interface to execute user events,.
Specific view components are derived from the abstract view and implement its interface. In addition, each view implements its own display function. This retrieves data from the view's suppliers, prepares this data for display, and presents them to the user. The display function is called when opening or updating a view.

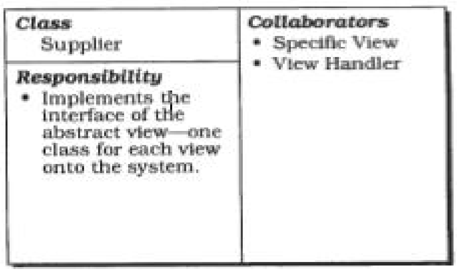


Class Abstract View	***Collaborators***
Responsibility	
• Defines an interface to create, initialize, coordinate, and close a specific view.	

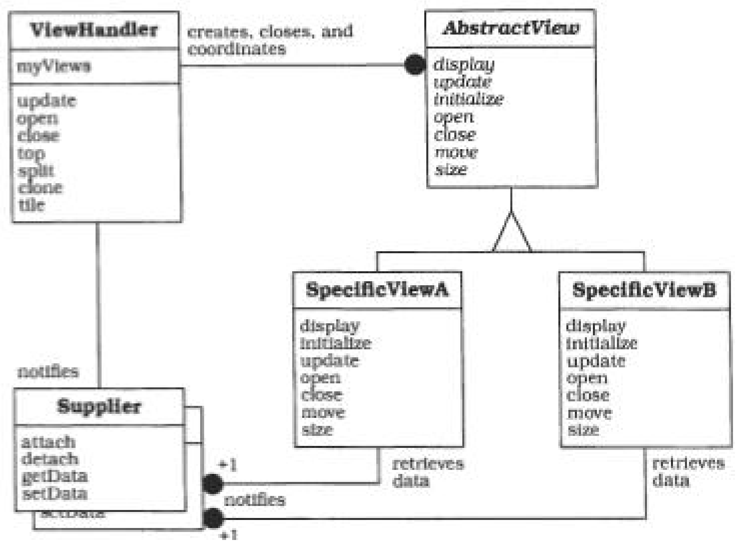


Class Specific View	***Collaborators***
Responsibility	• Supplier
• Implements the abstract interface.	

Supplier components provide the data that is displayed by view components. Suppliers offer an interface that allows clients--such as views-to retrieve and change data. They notify dependent components about changes to their internal state. Such dependent components are individual views or, in the case where the view handler organizes updates, the view handler itself.



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



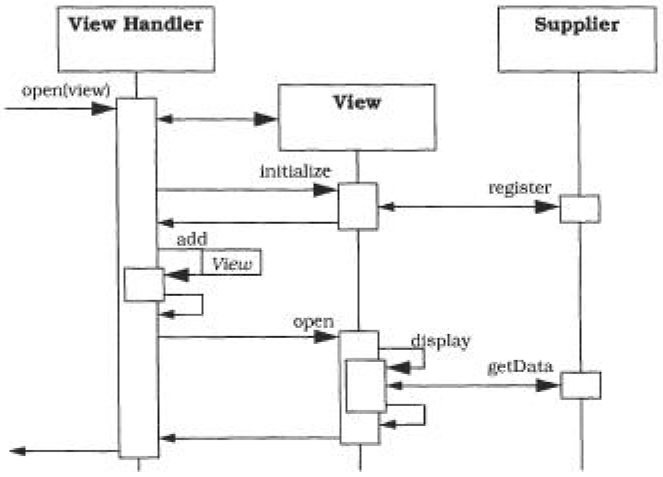
Dynamics We select two scenarios to illustrate the behavior of the View Handler pattern: view creation and tiling. Both scenarios assume that each view is displayed in its own window.

Design Patterns

Scenario I 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 view handler 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

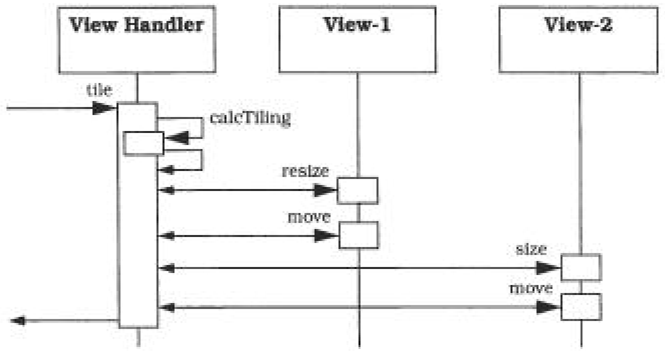
The view handler adds the new view to its internal list of open views. The view handler calls the view to display itself. The view opens a new window, retrieves data from its supplier, prepares this data for display, and presents it to the user.



Scenario II illustrates how the view handler organizes the tiling of views. For simplicity, we assume that 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 view handler. 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.

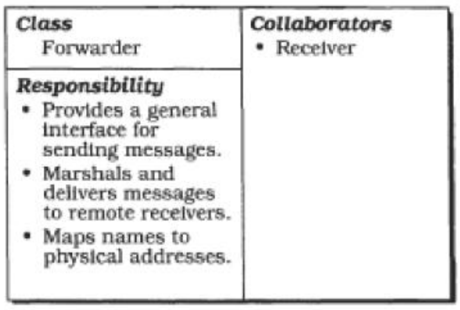


7(a) Define communication design pattern ? [2]

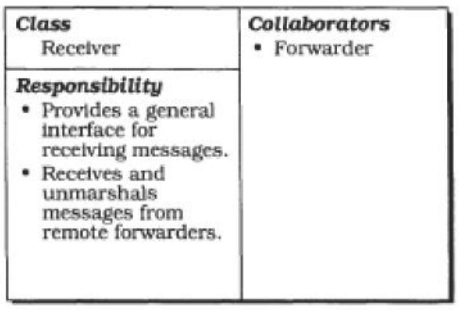
Communication. Patterns in this category help to organize communication between components. Two patterns address issues of inter-process communication: the Forwarder-Receiver pattern deals with peer-to-peer communication, while the Client Dispatcher-Server pattern describes location-transparent communication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

(b) Describe the various components of Forwarder Receiver design pattern in the structure along with the Dynamics. [8]

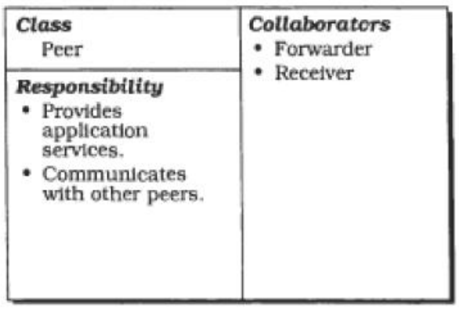
Peer components are responsible for application tasks. To carry out their tasks peers need to communicate with other peers. These may be located in a different process, or even on a different machine. Each peer knows the names of the remote peers with which it needs to communicate.
Forwarder components 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 un marshaling messages.



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

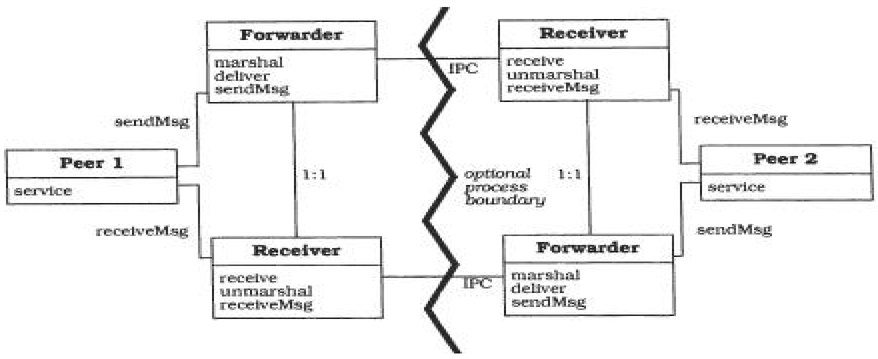


Class Receiver	***Collaborators***
Responsibility	• Forwarder
• Provides a general interface for receiving messages. • Receives and unmarshals messages from remote forwarders.	



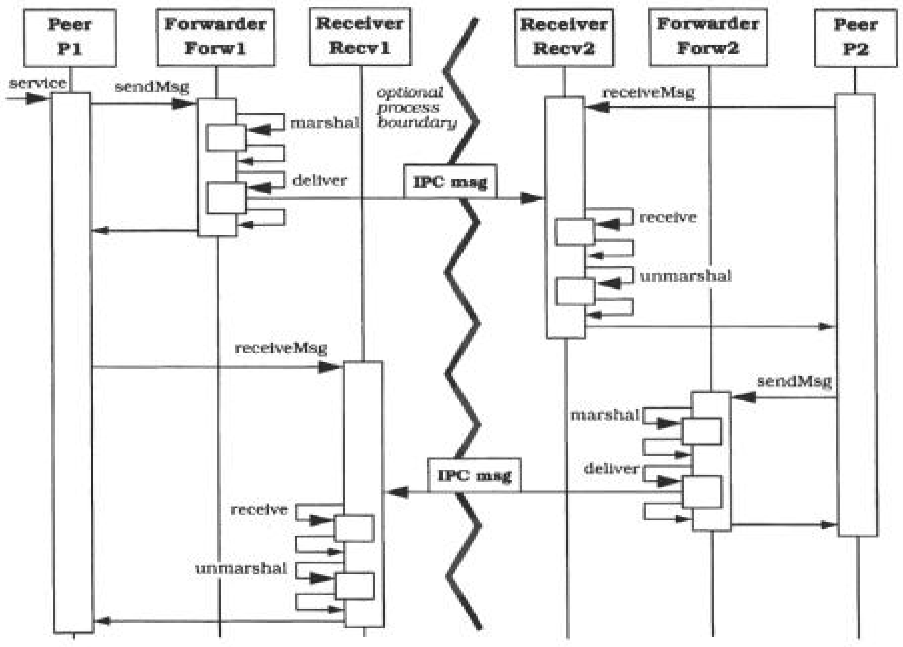
Class Peer	***Collaborators***
Responsibility	• Forwarder • Receiver
• Provides application services. • Communicates with other peers.	

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.



8(a) Write the problem and solution of the Master Slave design pattern. [5]

The ***Master-Slave*** design pattern supports fault tolerance, parallel computation and computational accuracy. A master component distributes work to identical slave components and computes a Anal result from the results these slaves return.

Example The traveling-salesman problem is well-known in graph theory.

Context Partitioning work into semantically-identical sub-tasks.

Problem 'Divide and conquer' is a common principle for solving many problems. Work is partitioned into several equal sub-tasks that are processed independently. The result of the whole calculation is computed from the results provided by each partial process.
Several ***forces*** arise when implementing such a structure:
Clients should not be aware that the calculation is based on the 'divide and conquer' principle.
Neither clients nor the processing of sub-tasks should depend on the algorithms for partitioning work and

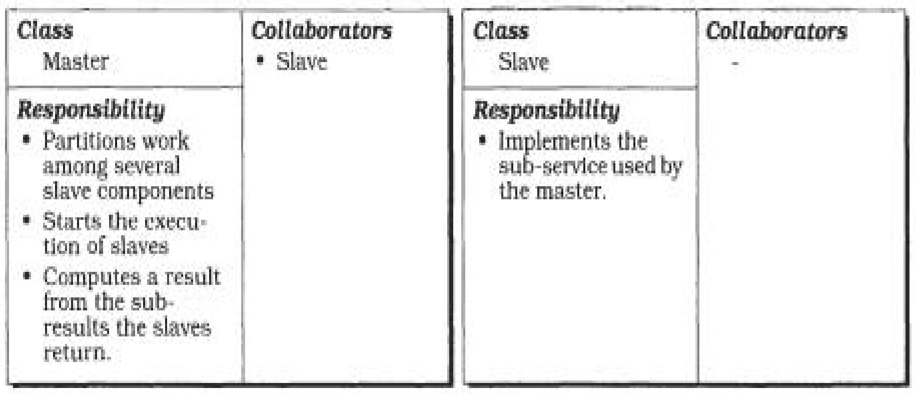
assembling the final result. It can be helpful to use different but semantically-identical implementations for processing sub-tasks, for example to increase computational accuracy. Processing of sub-tasks sometimes needs coordination, for example in simulation applications using the finite element method.

Solution
Introduce a coordination instance between clients of the service and the processing of individual sub-tasks.
A master component divides work into equal sub-tasks, delegates these sub-tasks to several independent but semantically-identical ***slave*** components, and computes a final result from the partial results the slaves return.
This general principle is found in three application areas:
Fault tolerance. The execution of a service is delegated to several replicated implementations. Failure of service executions can be detected and handled.
Parallel computing. A complex task is divided into a fixed number of identical sub-tasks that are executed in parallel. The final result is built with the help of the results obtained from processing these sub-tasks.
Computational accuracy. The execution of a service is delegated to several different implementations. Inaccurate results can be detected and handled.

8(b) Explain the structure of this design pattern [5]

Structure The master component provides a service that can be solved by applying the 'divide and conquer' principle. It offers an interface that allows clients to access this service. Internally, the master implements functions for partitioning work into several equal sub-tasks, starting and controlling their processing, and computing a final result from all the results obtained. The master also maintains references to all slave instances to which it delegates the processing of sub-tasks.

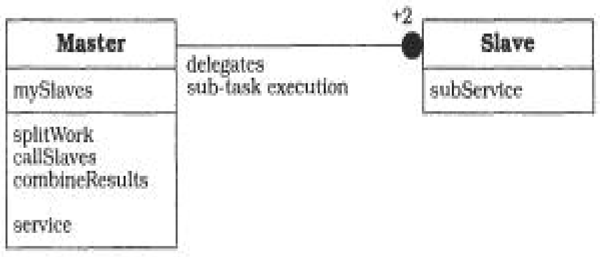
The slave component provides a sub-service that can process the sub-tasks defined by the master. Within a Master-Slave structure, there are at least two instances of the slave component connected to the master.



Class Master	***Collaborators***
Responsibility	• Slave
• Partitions work among several slave components	
• Starts the execution of slaves	
• Computes a result from the sub-results the slaves return.	

Class Slave	***Collaborators***
Responsibility	-
• Implements the sub-service used by the master.	

The structure defined by the Master-Slave pattern is illustrated by the following OMT diagram.


Master
mySlaves
splitWork
callSlaves
combineResults
service
delegates
sub-task execution
+2
Slave
subService