BMS INSTITUTE OF TECHNOLOGY & MANAGEMENT

Yelahanka, Bengaluru-560064

Department of MCA

RECORD OF PRACTICAL WORK

Name: SRINIVAS D A

USN: 1TD19MCA20

Semester: 4th sem MCA

Subject: Object Oriented Modeling And Design Lab[18MCA49]

BMS Institute of Technology & Management

Doddaballapur Road, Avalahalli, Yelahanka, Bengaluru – 560 064

LABORATORY CERTIFICATE

This is to Certify that Mr. / Ms. **SRINIVAS D A** has Satisfactory completed the course of experiments in Practical Object Oriented Modeling And Design[18MCA49] Prescribed by the Visvesvaraya Technological University for 4th Semester MCA course in the Laboratory of the college in the year 2021-2022.

Head of the Department **Staff In charge of the Batch**

Date: ……

Name of Candidate: Srinivas D A

Roll No: 1TD19MCA20

USN: 1TD19MCA20

Marks	
Maximum	Obtained

Signature of the Candidate

Content

BMS Institute of Technology & Management

Yelahanka, Bengaluru-560064

Department of MCA

Name: SRINIVAS D A

USN: 1TD19MCA20

Semester: 4th sem MCA

List	Maximum Marks	Obtained Marks
Test-1	**20**	
Test-2	**20**	
Average of Two tests	**20**	
Assignment Marks	**20**	
Total	**40**	

Signature of the staff in charge with Date: _____

Introduction to Rational Rose

Using Rose:

Creating a Use Case diagram:

1. In the Window on the left side click on the "+" (plus) sign next to "Use Case View"
2. In the same window, double click on the icon next to "Main". This will open a blank Use Case diagram called "Main"

Adding an Actor

1. To create an Actor, click on the stick figure icon located on the toolbar in the middle of the screen. Then click on the diagram to place the Actor.
2. Open the Actor's specification by right clicking on the Actor (in the diagram) and select "Specification" from the pop-up menu. Fill in applicable fields. The specification can also be brought up by double clicking on the actor.

Example: Add an Actor called "Elevator Rider"

Adding a Use Case

1. In the toolbar, click on the oval shaped icon then click on the open diagram to place the use case.
2. Bring up the specification and fill in appropriate fields.

Example: Add a Use Case called "Ride Elevator"

Adding an Association

1. Click on the association icon (a solid line with no arrowheads). Then, click on the actor first then the use case. An association (line) will be drawn between the two.

NOTE: Adding additional buttons to your toolbar:

Using the Main Menu Bar (top of the screen) follow these steps:

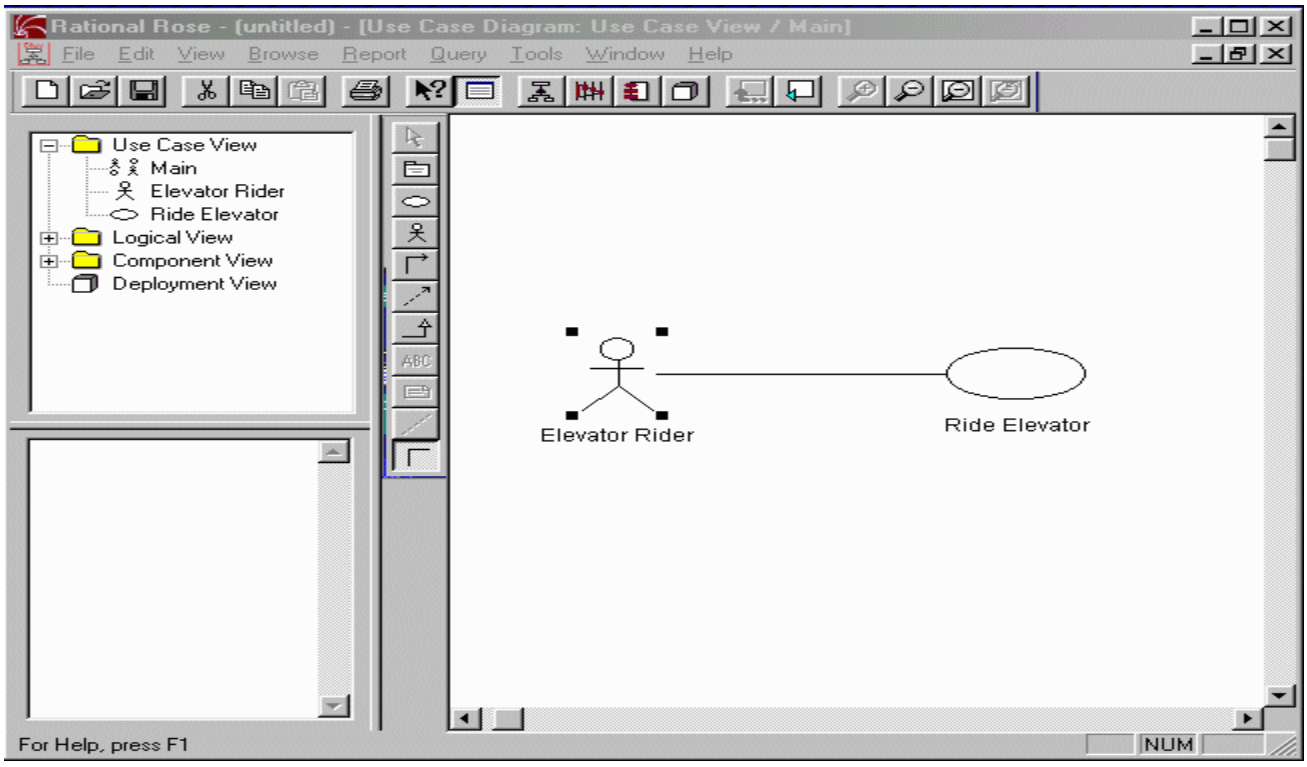
a. Click on "Tools"
b. Click on "Options"

c. Click on the "Toolbars" tab Note that different diagrams have different toolbars associated with them. You can add/remove buttons for each type of tools bar.

d. Click on the button with "..." in it next to the name of the diagram toolbar you want to alter.

e. The window that appears will allow you to add/remove buttons.

See Diagram below for what the Use Case in the above example looks like.



Creating a Class Diagram

1. In the left window, click on the "+" sign next to "Logical View"
2. Double click on the icon called "Main" underneath "Logical View". This opens the diagram called "Main"

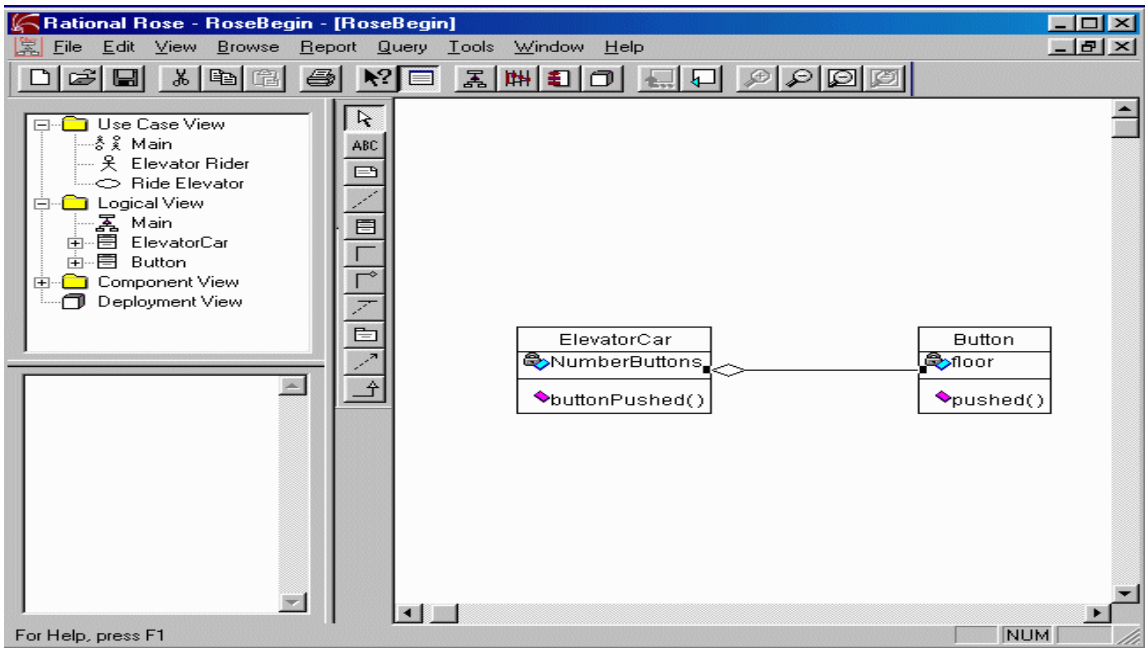
Adding a class to the diagram

1. To add a class, click on the icon that looks like a rectangle divided into3 pieces.
2. Click somewhere in the diagram on the right to place the class. Type in a class name (Ex: ElevatorCar). Note you do not have to open a specification box to name the class (or any object) when you first place it in the diagram. As soon as you add the object, just type in a name.
3. Adding attributes to the Class
    a. Double click on the class to open the specification.
    b. Click on the "Attributes" tab.
    c. Right click in the column titled "Name" and select "Insert".
    d. Add an attribute (Ex: NumberButtons). Hit <return>
    e. Double click on the newly added attribute to open the specification for the attribute.
    f. Fill in additional information for the attribute here.
4. Adding operations
    a. Double click on the class to open the specification.
    b. Click on the "Operations" tab.
    c. Right click in the column "Signature" and select "Insert".
    d. Add a new operation (Ex: buttonPushed). Hit <return>.
    e. Double click on the operation (method) name to open the specification.
    f. Add additional detail here.

Example for making a Class Diagram:

Go ahead and add a class called "ElevatorCar" with the above attribute and operation. Also, add a class called "Button", with an attribute "floor" and operation "pushed". Also, add an aggregation between the two classes. In the toolbar, click on the icon that has a diamond shaped head at one end of a solid line. Then click on the "Button" object in the diagram, then click on the "ElevatorCar" object in the diagram.

The result should look like the diagram below:



Adding a Collaboration Diagram

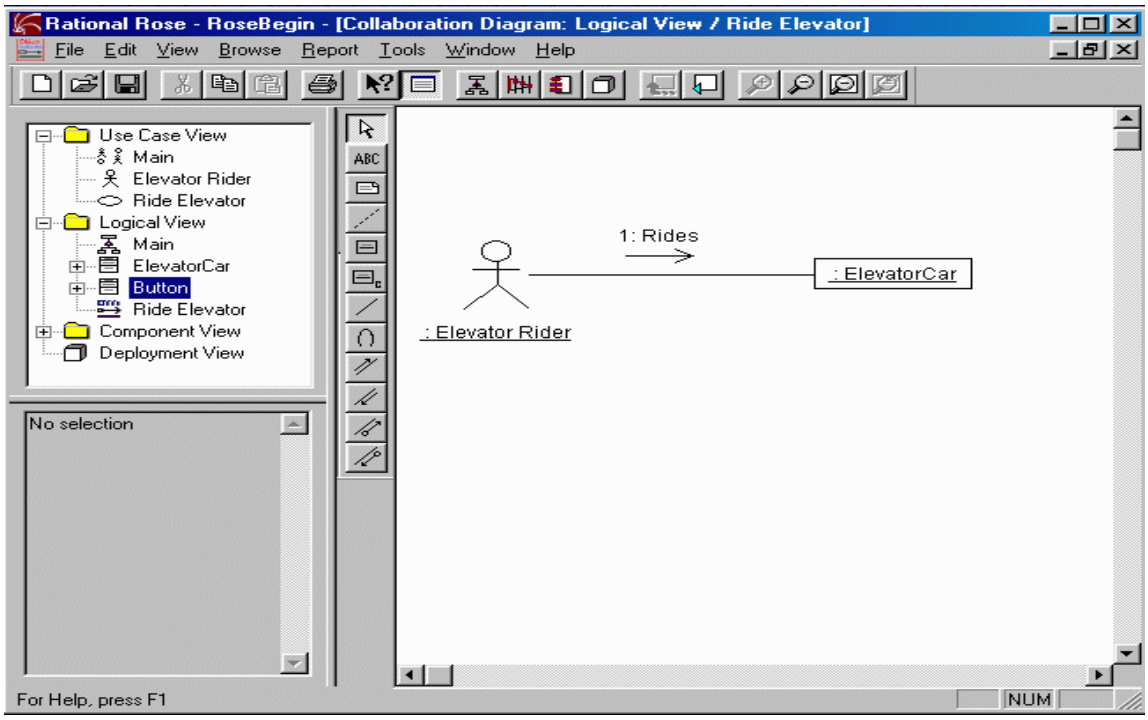
1. In the left window, right click once on "Logical View".
2. Select "New", then "Collaboration Diagram".
3. A new Collaboration Diagram will appear; name it. (Ex: Ride Elevator)
4. Double click on the diagram icon next to the name to open it.
5. In the left window, left-click once on the object to be added to the diagram and drag it onto the newly created Collaboration diagram.
6. Use the other icons in the toolbar to create links between actors, classes, etc.

Example:

1. Click on the Actor "Elevator Rider" and drag and drop the object onto the diagram.
2. Drag and drop the "ElevatorCar" class onto the diagram.
3. Add the object "ElevatorCar" to the Collaboration diagram in the same manner.
4. Click on the Object Link icon (solid line).
5. Then click on the actor (in the diagram) and then then the "ElevatorCar" object (in the diagram).

6. Now, add a link message this way:
    a. Click on the "Link Message" icon (solid line with arrow pointing to the right above the solid line).
    b. Click on the just created object link.

See diagram for what the collaboration diagram looks like:

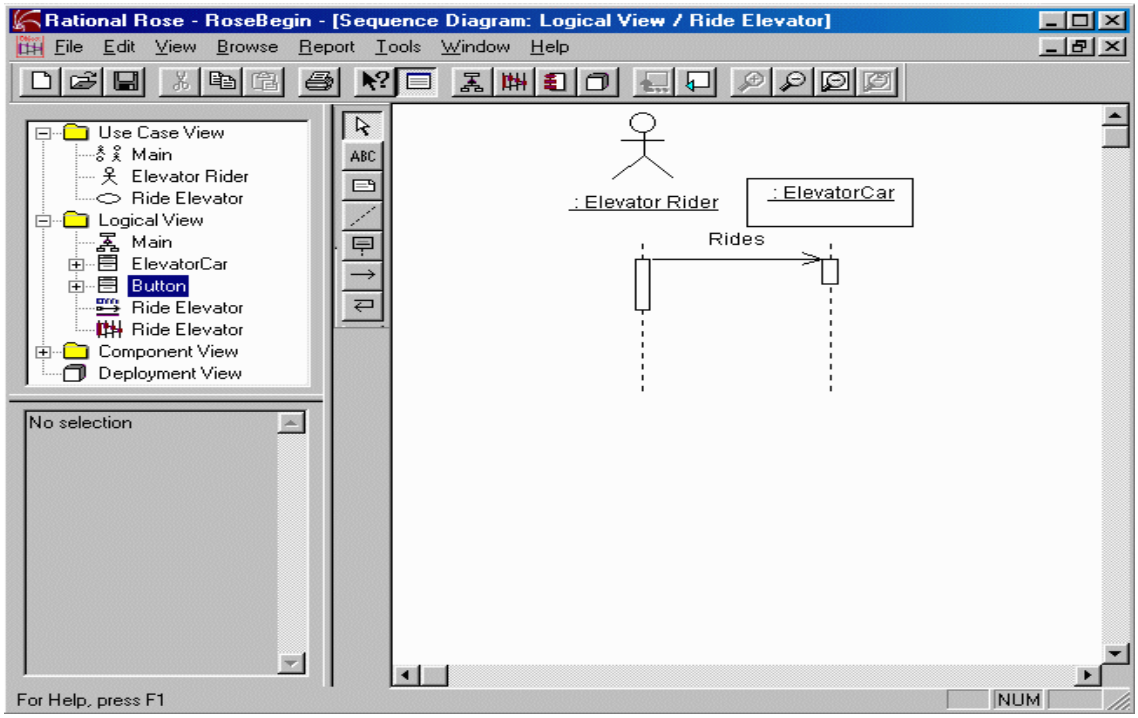


Creating a Sequence Diagram.

1. Make sure the Collaboration diagram is the active diagram. (If you've been doing the examples in order, it should already be active)
2. Hit the "F5" key (the single key labeled 'F5').

Rose generates the Sequence diagram based on the Collaboration diagram. The reverse can be done as well. A Collaboration diagram can be generated from a Sequence diagram by hitting the 'F5' key.

The sequence diagram looks like this:



1. Publisher-Subscriber pattern

Intention:

- Keep contents synchronized.
- Inter component communication.
- Multiple content usages.
- The publisher-subscriber design pattern helps to keep the state of co-operating components synchronized. To achieve this, it enables one way propagation of changes one publisher notifies any number of subscribers about change of its state.

Alias:

- Observer.
- Dependents.

Problem:

- Data is centralized but many (remote) components depend on this data.
- One or more components must be notified about state changes in a particular component.
- The number and ID's of dependent components are known only in runtime.
- Explicit polling by dependents of new information is not feasible.
- Components should not be tightly coupled when introducing a change propagation mechanism.

Solution:

- Publisher is the main role which is taken by one dedicated component as subject.
- All components dependent on changes is the publisher and its subscribers, also called as observer.
- Publisher maintains a registry of currently subscribed components.
- Subscribers use the interface offered by the publisher.
- Whenever the publisher changes state, its sends a notification to all its subscribers.
- The subscriber intern retrieves the changed data at their description.

Structure:

- Participants:

  -Subject:

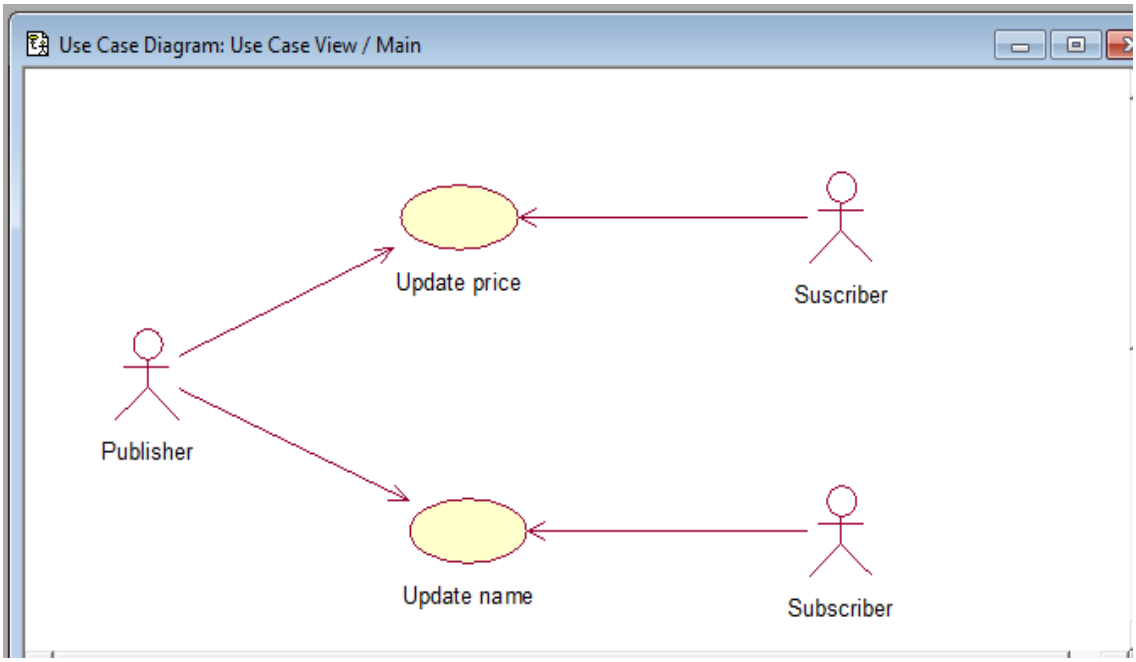
  This class acts as a notification engine. It knows its observers. Any number of observer objects may observer a subject.

  It provides an interface for attaching and detaching observer objects.

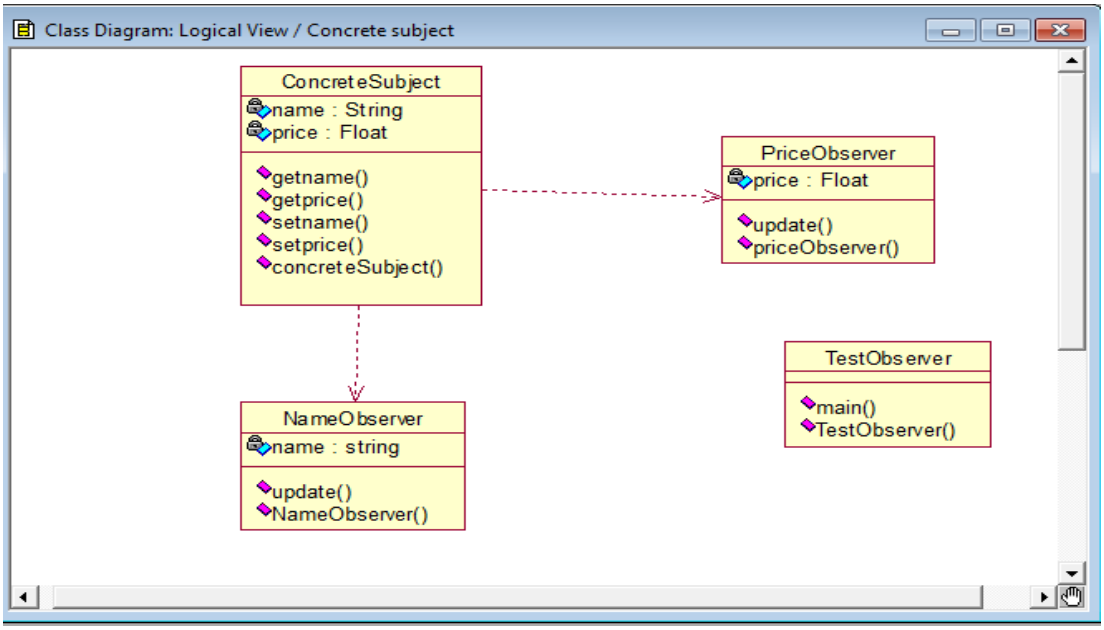
  -Observer:

  This class acts as a receiver of subject notification. Defines an updating interface for objects that should be notified of changes in a subject.

Use Case Diagram:

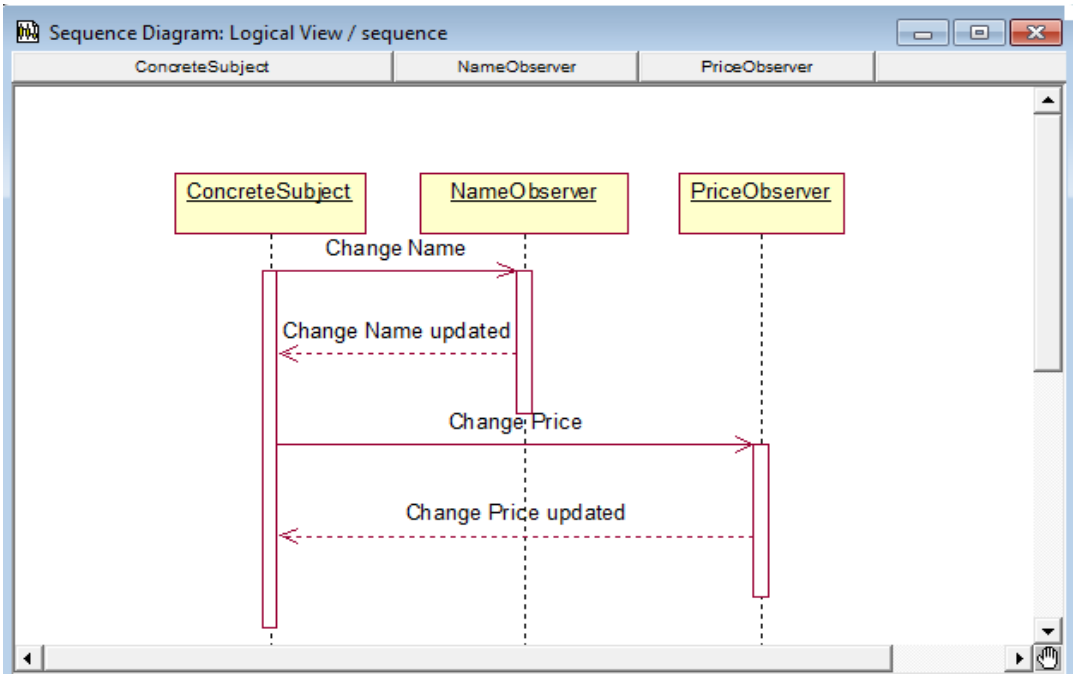


Class Diagram:

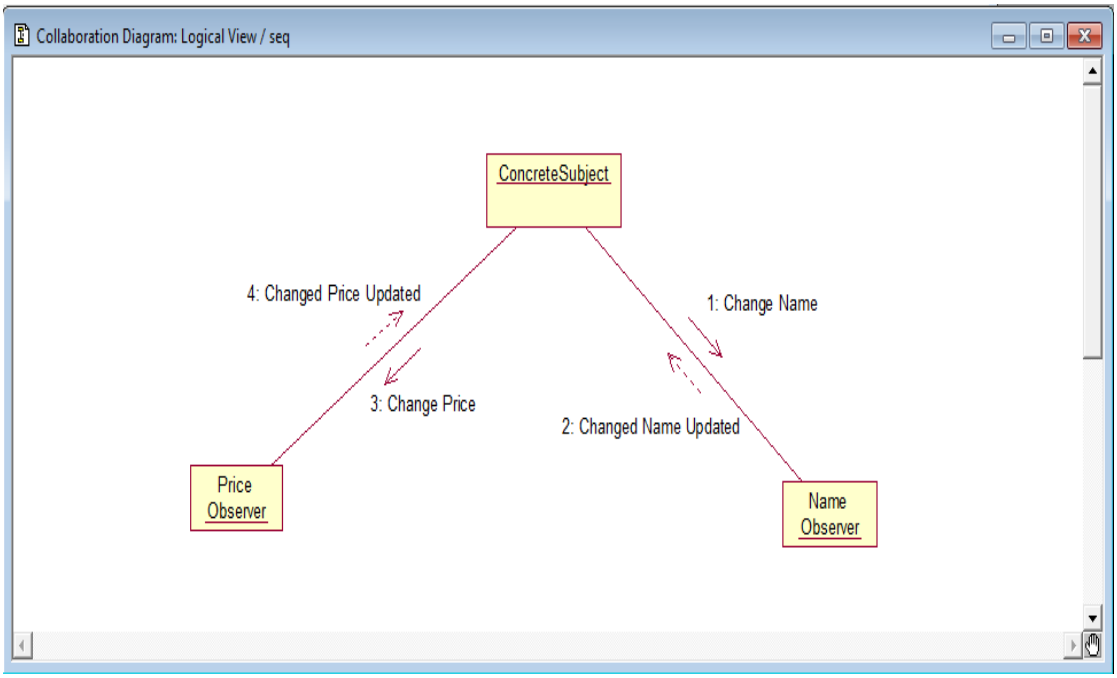


Dynamics:

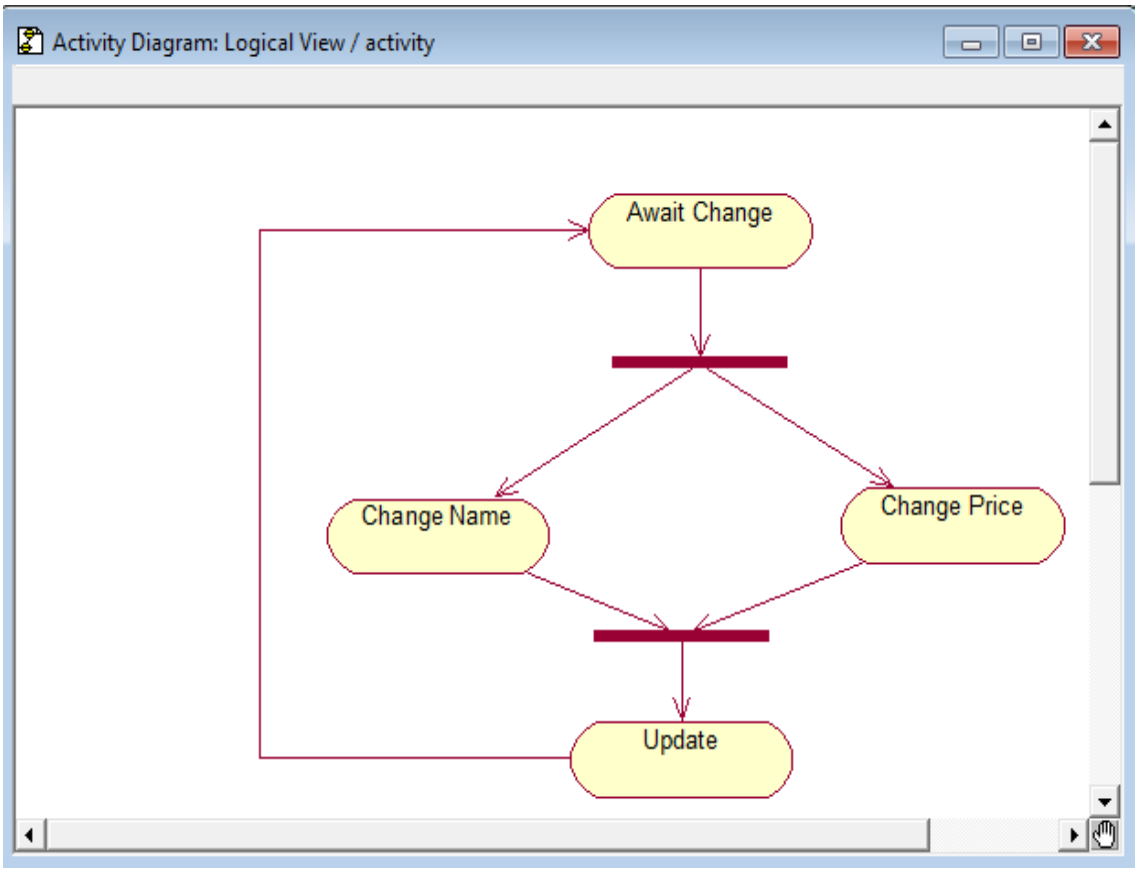
- Sequence diagram:



- Collaboration diagram:



- Activity diagram:



Implementation:

ConcreteSubject.java

```
import java.lang.*;
import java.util.*;
public class ConcreteSubject extends Observable
{
  private String Name;
  private Float Price;
  public ConcreteSubject(String Name,float Price)
  {
        this.Name=Name;
        this.Price=Price;
        System.out.println("\n Concrete Subject Created"+Name+"at"+Price);
  }
  public String getName()
  {
      return Name;
  }
  public float getPrice()
  {
     return Price;
  }
  public void setName(String Name)
```

```
        this.Name=Name;
         setChanged();
         notifyObservers(Name);
  }
  public void setPrice(float Price)
  {
         this.Price=Price;
         setChanged();
         notifyObservers(new Float(Price));
  }
 }
```

NameObserver.java

```
import java.util.Observable;
import java.util.Observer;
public class NameObserver implements Observer
{
  private String Name;
  public NameObserver()
  {
        Name=null;
         System.out.println("\n Name Observer Created! name is:"+Name);
          public void update(Observable obj,Object arg)
     {
```

```
        if(arg instanceof  String)
          {
               Name=(String)arg;
               System.out.println("name observer:"+Name);
          }
          else
          {
              System.out.println("name observer:some other change to subject");
           }
}
 }
```

PriceObserver.java

```
import java.util.Observable;
import java.util.Observer;
public class PriceObserver implements Observer
{
   private float Price;
   public PriceObserver()
   {

          Price=0;
          System.out.println("\n PriceObserver Created!Price is:"+Price);
```

```
  {
      if(arg instanceof Float)
        {
          Price=((Float)arg).floatValue();
          System.out.println("\n PriceObserver:Priece changed to"+Price);
        }
         else
         {
          System.out.println("\n PriceObserver:Priece changed to"+Price);
         }
  }
}
```

TestObserver.java

```
import java.util.*;
public class TestObserver
{
  public static void main(String args[])
  {
 ConcreteSubject s=new ConcreteSubject("corn pops",1.29f);
NameObserver nameobs=new NameObserver();
PriceObserver priceobs=new PriceObserver()
```

```
        s.addObserver(nameobs);
         s.addObserver(priceobs);
         s.setName("frostyed flakes");
         s.setPrice(4.57f);
         s.setPrice(9.22f);
         s.setName("sugar crispies");
  }
}
```

Outputs:

```
C:\Java>java TestObserver
Concrete Subject Createdcorn popsat:1.29
Name Observer Created! name is:null
 PriceObserver Created!Price is:0.0
 PriceObserver:Priece changed to0.0
 name observer:frostyed flakes
 PriceObserver:Priece changed to4.57
 name observer:some other change to subject
 PriceObserver:Priece changed to9.22
 name observer:some other change to subject
 PriceObserver:Priece changed to9.22
 name observer:sugar crispies
```

Variants:

- Gate Keeper: apply to distributed system.
- Event channel: OMG in its event service specification.
- Strong decoupled publishers and subscriber.

Consequences:

- **Advantages:**
  - Changeability
  - Extensibility
  - Support distributed system
- **Disadvantages:**
  - Efficiency
  - complexit

2. Command Processor Pattern

Intention:

- System management.
- System need to handle collections of objects.
- For example, events from other systems that needs to be interpreted and scheduled.
- A command processor component manages requests as separate objects, schedules their execution, and provides additional services such as the string 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 function, such as scheduling or undo.

Problem:

- Application needs large set of features.
- Need a solution that is well-structured for mapping its interface to its internal functionality.
- Need to implement pop-up menus, keyboard shortcuts, or external control of application via a scripting language.
- Provide different ways for different users to work with an application.
- Enhancements of the application should break existing code.
- Services such as undo should be implemented consistently for all requests.

Solution:

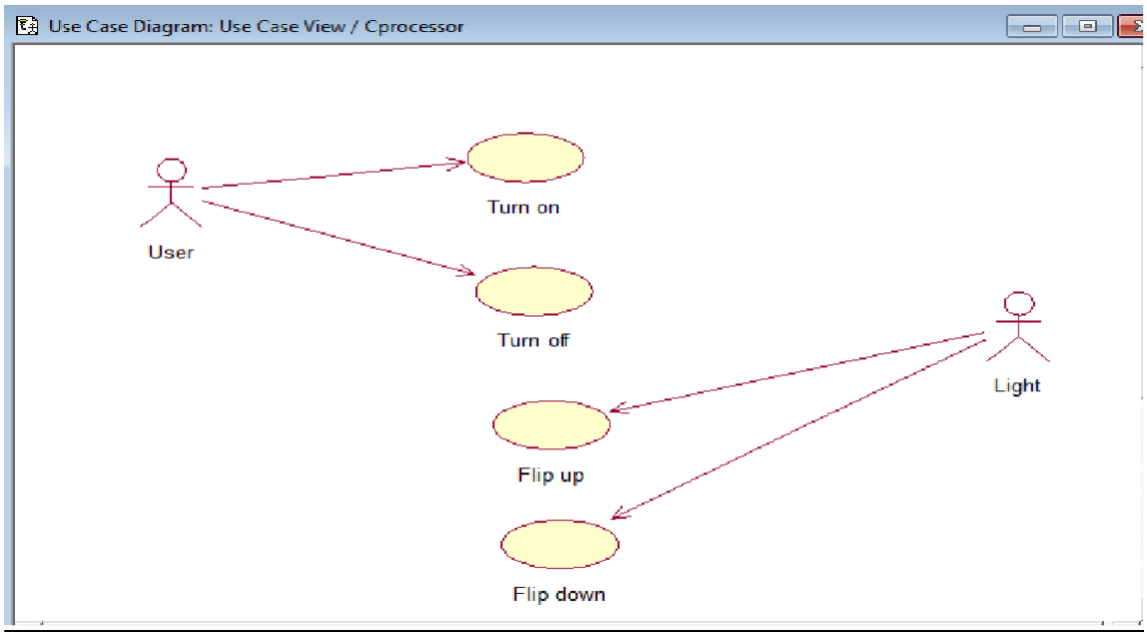
- Use command processor pattern.
- Encapsulate requests in to objects.
- When users call a function, the request is turned into a command objects.
- Command processor component take care of all command objects.
- It schedules the execution, provides undo requires and alter additional services

Structure:

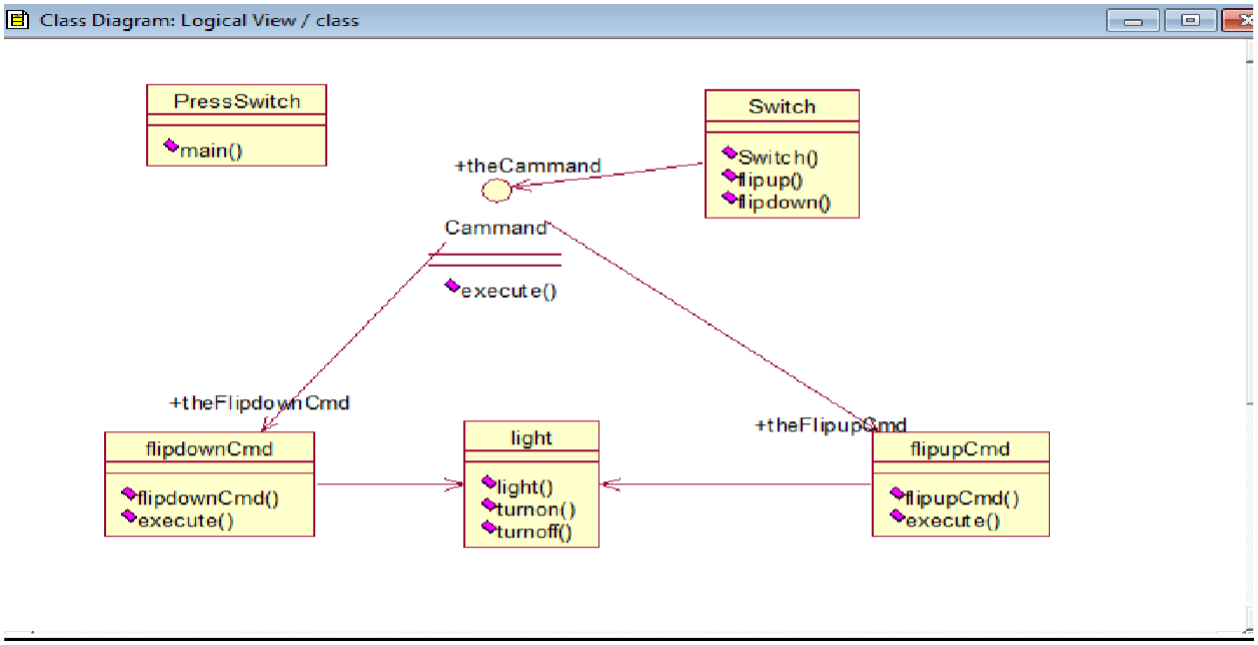
- **participants**

1. Controller: it represents the interface of the application. It accepts requests, and creates the corresponding command objects, which are then derived to command processor for execution.
2. Command Processor: It manages the command objects, schedules them and starts their execution. It is the key components that implements additional services related to the execution commands.
3. It remains independent of specific commands because it only uses the abstract command interface.
4. Supplier: The supplier components provide most of the functionality required to execute concrete commands related commands often share supplier components.

Use Case Diagram:

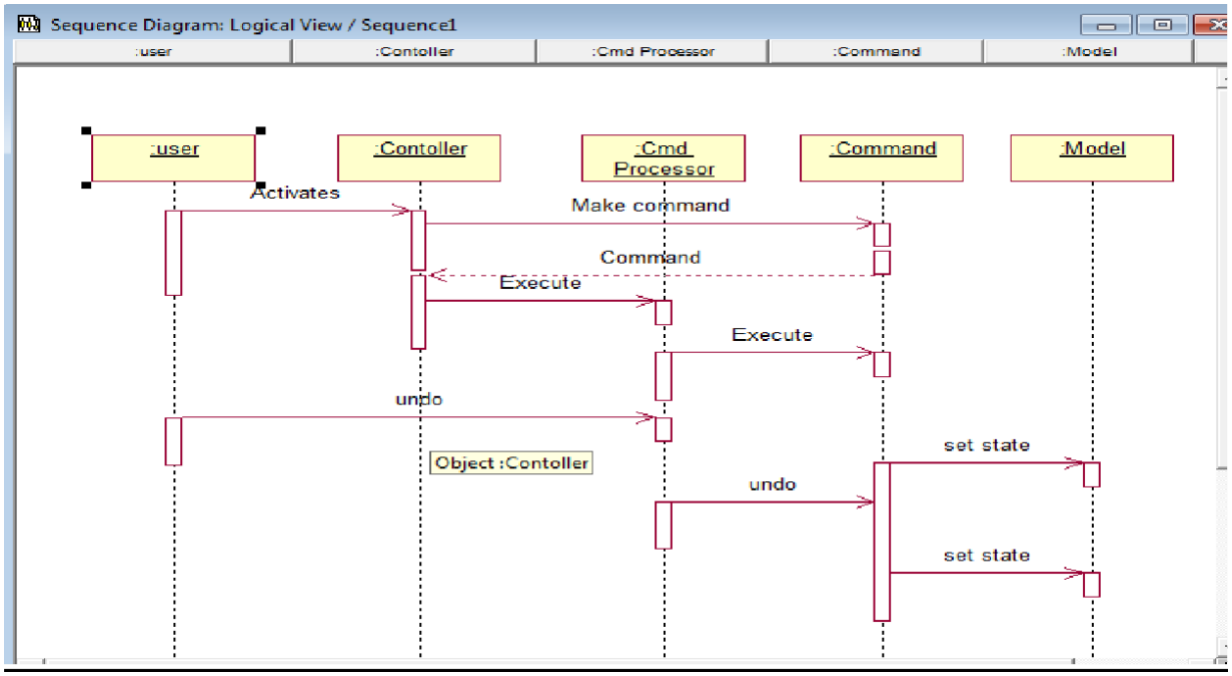


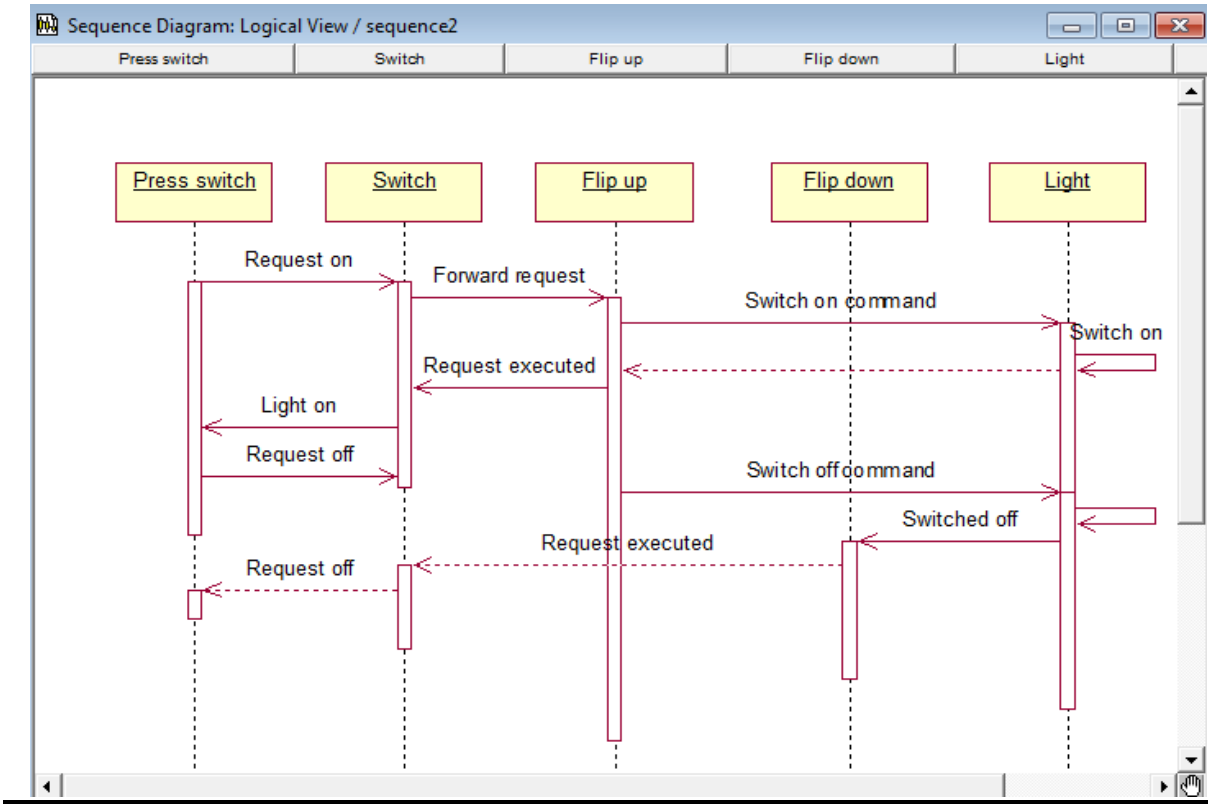
Class Diagram:



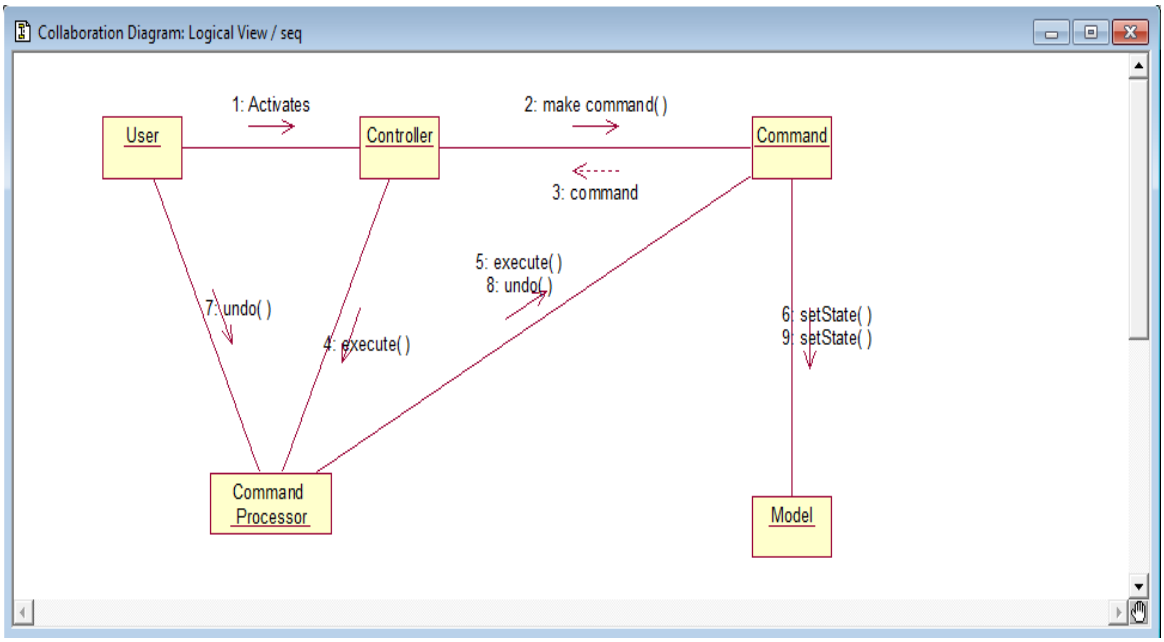
Dynamics:

- Sequence diagram:

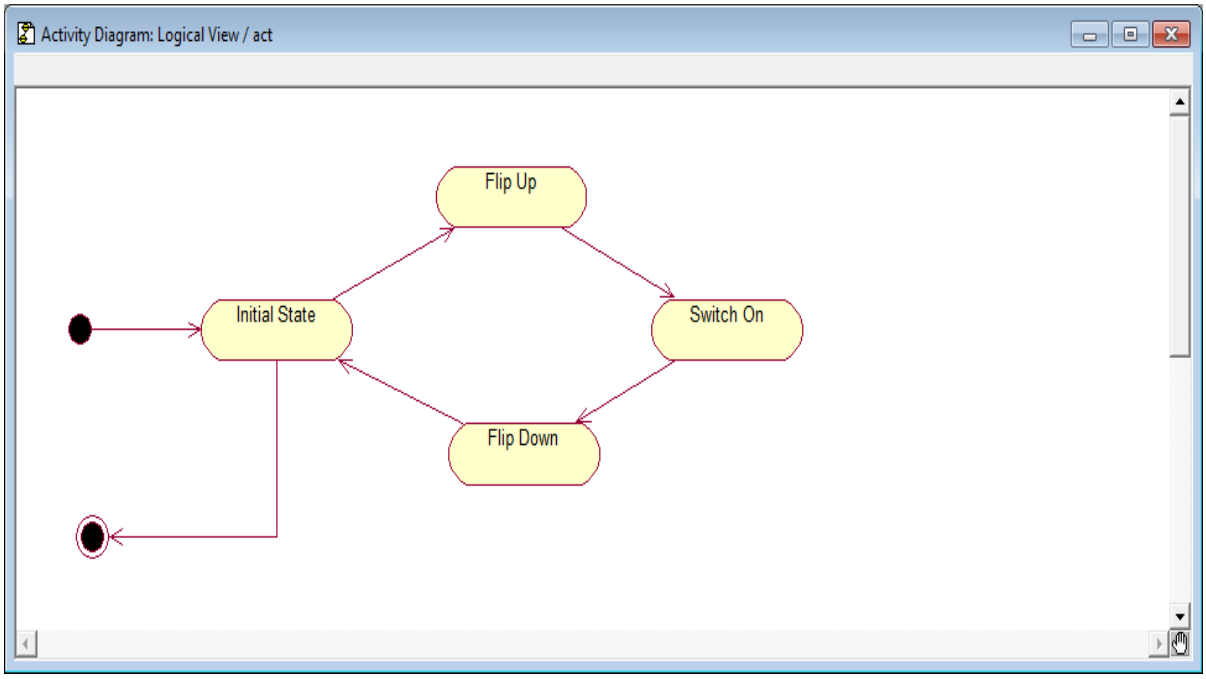




- Collaboration diagram:



Activity diagram:



Implementation:

PrintSwitch.java

```
import java.io.*;

import java.io.*;

import java.lang.*;

public class PressSwitch

{

        public static void main(String args[])

        {

                light lamp=new light();

                Cammand switchUp=new flipupCmd(lamp);

                Cammand switchDown=new flipdownCmd(lamp);

                Switch s=new Switch(switchup,switchdown);
```

```
try
                {
                        if(args[0].equalsIgnoreCase("ON"))
                        s.flipup();
                        else if(args[0].equalsIgnoreCase("OFF"))
                        s.flipdown();
                        else
                        System.out.println("arguments required\n");
                }
                catch(Exception e)
                {
                        System.out.println("Arguments required\n");
                }
        }
}
```

Switch.java

```
public class Switch
{
        private Cammand flipupCmd;
        private Cammand flipdownCmd;
        public Switch(Cammand flipupCmd,Cammand flipdownCmd)
         {
            this.flipupCmd=flipupCmd
```

```
        this.flipdownCmd=flipdownCmd;
    }
    public void flipup()
    {
        flipupCmd.execute();
    }
    public void flipdown()
    {
        flipdownCmd.execute();
    }
}
```

Light.java

```
public class light
{
    public light()
    {
     }
    public void turnOn()
    {
        System.out.println("\n\nthe light is on!\n");
    }
     public void turnOff()
    {
```

```
            System.out.println("\n\nthe light is off!\n");
        }
}
```

Cammand.java

```
import java.io.*;
public interface Cammand
{
        public void execute();
}
```

flipupCmd.java

```
public class flipupCmd implements Cammand
{
        public light theLight;
        public flipupCmd(Light Light)
        {
                this.theLight=Light;
        }
        public void execute()
        {
                theLight.turnOn();
```

```
    }
}
```

flipdownCmd.java

```
public class flipdownCmd implements Cammand
{
        public light theLight;
        public flipdownCmd(Light Light)
        {
                this.theLight=Light;
        }
        public void execute()
        {
                theLight.turnOff();
        }
}
```

Output:

```
C:\Java>java PressSwitch on
the light is on!
```

Variants:

- Speed controller functionality. The role of the controller can be distributed over several components.

Known users:

- ET++
- MacApp use the pattern to do and undo
- WRKJOBJS use the pattern to do job scheduling.

Consequences:

Advantages:

- Flexibility in the way requests are activated.
- Flexibility in the number and functionality of requests.
- Programming execution related services testability at application level.
- Concurrency.

Disadvantages:

- Efficiency loss.
- Potential for an excessive number of command classes.
- Complexity in airing command parameters.

3.Forwarder –Receiver Pattern

Intention:

- The Forwarder–Receiver design pattern provides transparent interprocess communication for software system with peer to peer interaction model. It introduces forwarders and receivers to decouple peers from the underlying communication mechanism.

Context:

- Distributed peers collaborate to solve a particular problem.
- A peer may act as a client, requesting services, as server, providing services or both

Problem:

- A common way to build distributed applications is to make use of available low-level mechanism for IPC such as TCP/IP, sockets or message queues.
- These are provided by almost all operating systems, and are very efficient when compared to higher-level mechanism such as remote procedure calls.
- These low-level mechanisms often introduce dependencies on the underlying operating system and network protocols.
- The system should allow the exchangeability of the communication mechanisms.
- The cooperation of component follows a peer to peer model, in which a sender only needs to know the names of its receivers.
- The communication between peers should not have a major impact on performance.

Solution:

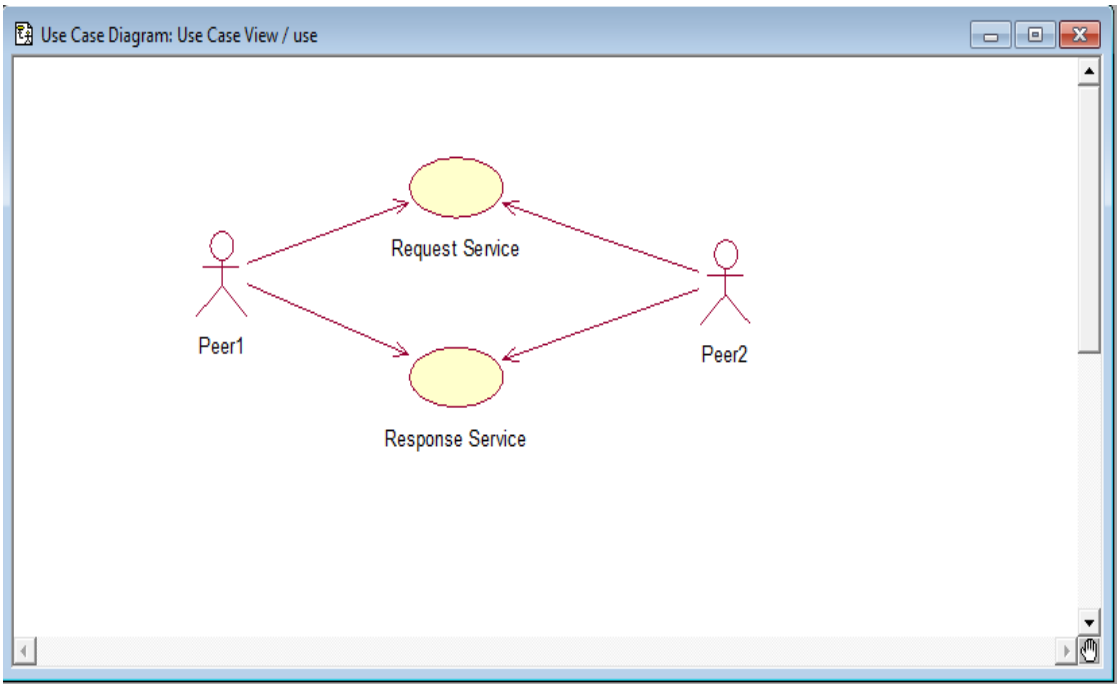
- Distributed peers collaborate to solve a particular problem.
- A peer may act as client, requesting services, as server, providing services or both.
- The details of the underlying IPC mechanism for sending or receiving messages are hidden from the peers by encapsulating all such functionality by the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

Structure:

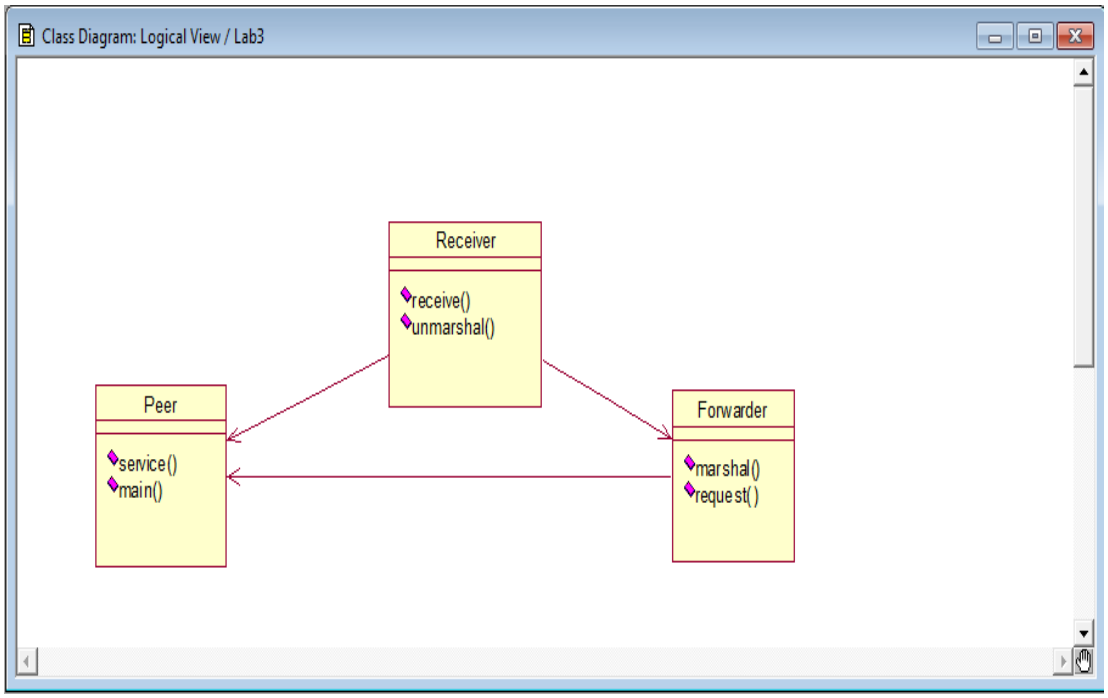
- Participant:

  - Peers: They are responsible for application tasks. They need to communicate with other peers to carry out their tasks. They use forwarder to send messages to other peers, and a receiver to receive messages from other peers. Such messages are either requests or responses.
  - Forwarder: It sends messages across process boundaries. It provides a general interface that is an abstraction of a particular IPC mechanism and includes functionality marshalling and delivery of messages. It also contains a mapping from names to physical addresses.
  - Receiver: It is responsible for receiving messages. It provides a general interface that is an abstraction of a particular IPC mechanism. It also includes functionality for receiving and unmarshalling messages.

Use Case Diagram:

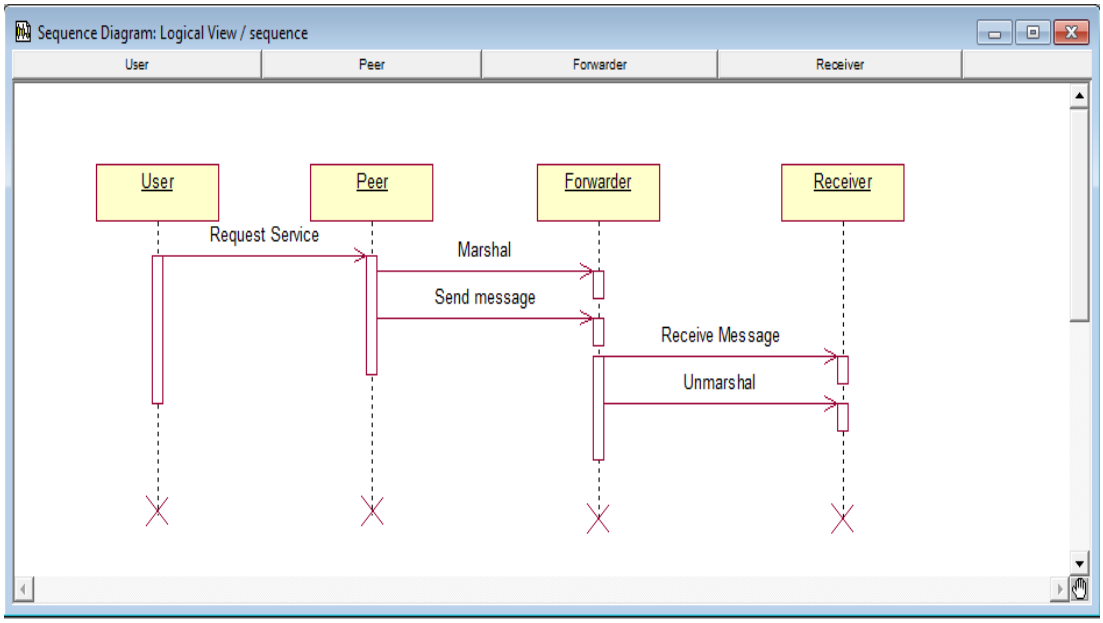


Class Diagram:

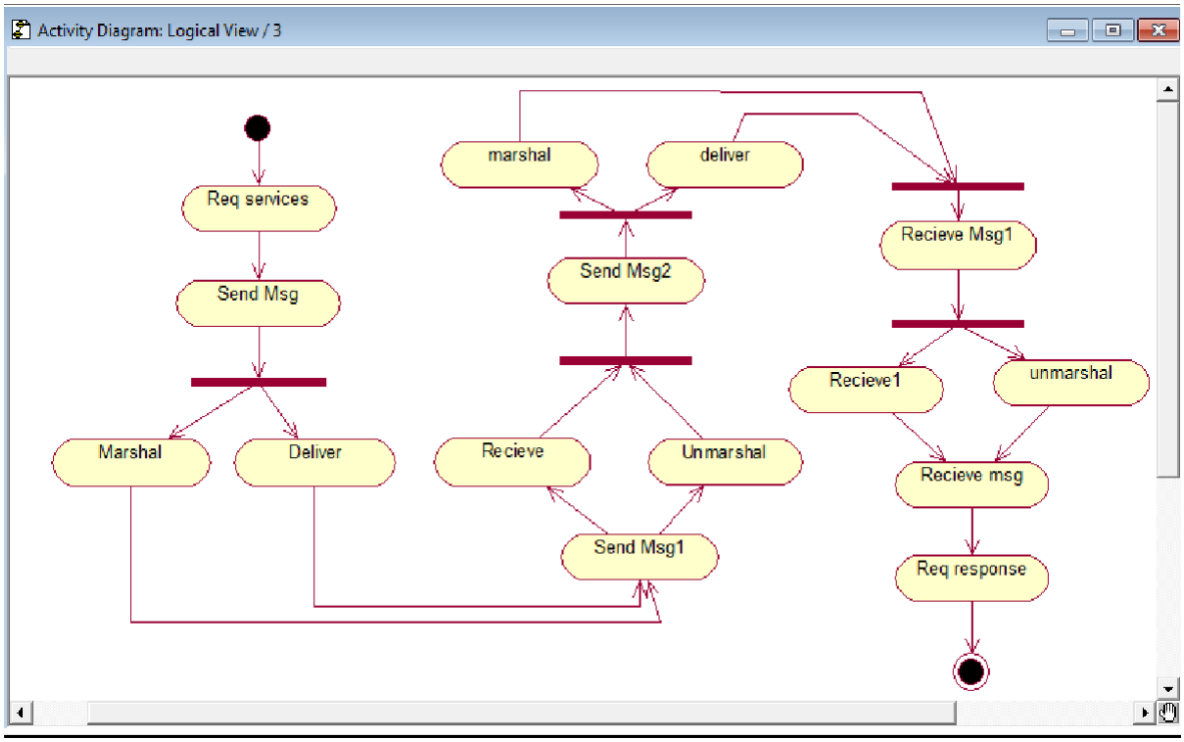


Dynamics:

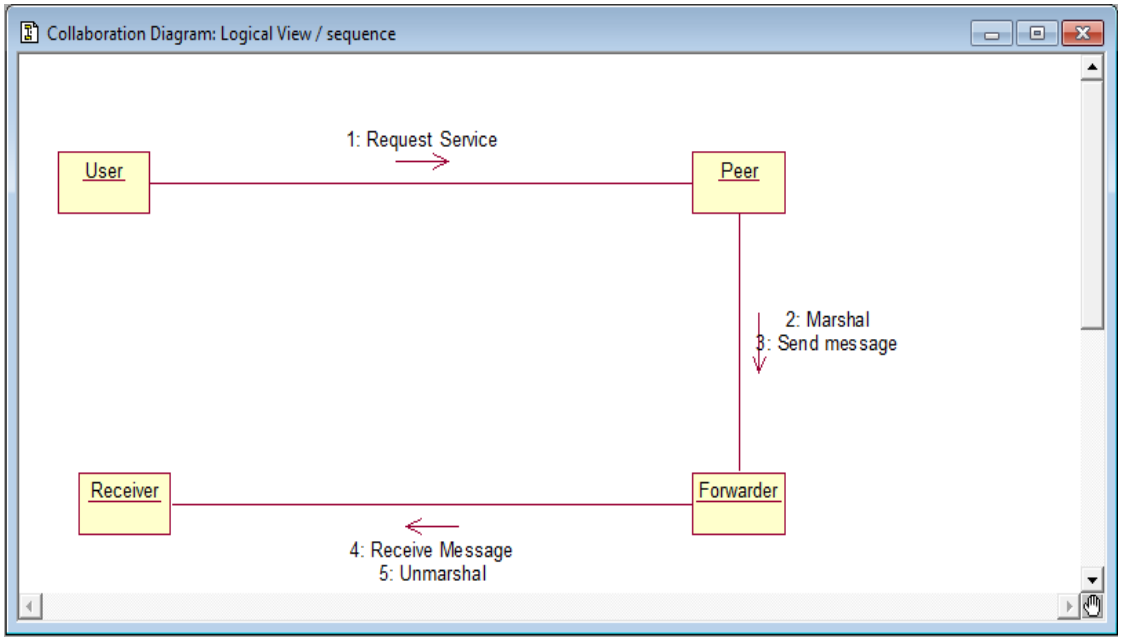
- Sequence diagram:



Activity diagram



Collaboration diagram:



Implementation:

Reciever.java

```
public class Reciever
{
  public void recieve()
  {
    System.out.println("message received");
  }
  public void unmarshal()
  {
    System.out.println("unmarshal done");
        receive();
  } }
```

Forwarder.java

```
import java.lang.*;
import java.util.*;
public class Forwarder
{
   public Reciever r=new Reciever();
   public void marshal()
   {
      System.out.println("marshal done");
```

```
        r.unmarshal();
    }   }
```

Peer.java

```
public class Peer
{
    public Reciever theReciever;
    public Forwarder forword=new Forwarder();

    public void service()
    {
        System.out.println("sending message");
                forword.marshal();
    }
    public static void main(String args[])
    {
     new Peer().service();
    }
}
```

Output:

```
sending message
marshal done
unmarshal done
message received
```

Variants:

- Forwarder-Receiver without name-to-address mapping.

Known user:

- TASC
- REBOOT
- ATM_-P

Consequences:

Advantages:

- Efficient IPC.
- Encapsulation of IPC facilities.

Disadvantages:

- No support for flexible reconfiguration of components.
- Complexity.

4. Client dispatcher server pattern

Intention:

- The client-dispatcher-server provides transparent inter process communication when the distributions of component not known at compile time and many vary at run time.
- The client dispatcher server pattern introduces an intermediate layer between clients and server that is the dispatcher component.
- It provides location transparency by means of name service, and hides the details of establishment of the communication connection between clients and servers.

Alias:

- Visitor pattern.

Context:

- A software system integrating set of distributed server over a network.

Problem:

- Core functionality of the components should be separated from details of communication mechanism.
- Clients are unaware of server location. This helps us to change the location of the server dynamically providing resilience to network or server failure.

Solution:

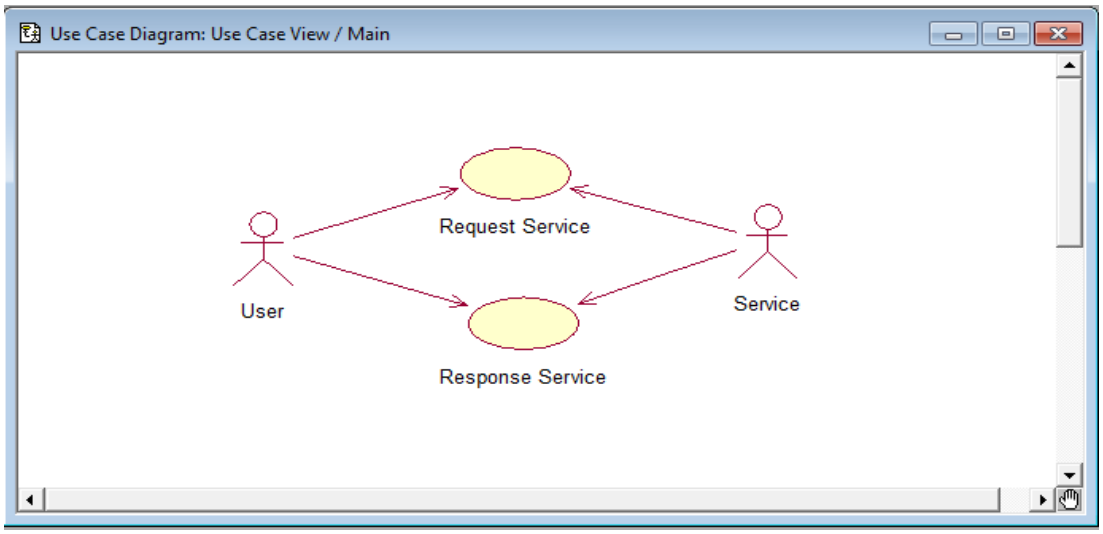
- Provide a dispatcher component to act as an intermediate layer between clients and server.
- The dispatcher implements a names service that allow clients to refer to server by names, with which the servers are registered with dispatcher, instead of physical locations this providing location transparency.
- In addition the dispatcher is responsible for establishing the communication channel between a client and server.

Structure:

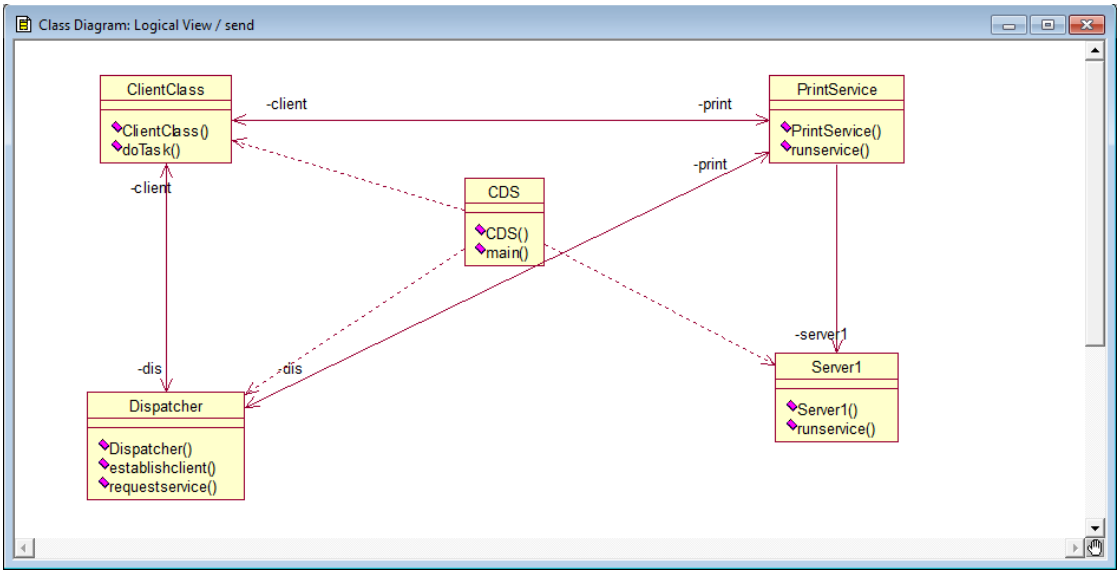
- Participants:

1. Client:
    - It performs domain specific tasks.
    - It sends request for services through the communication channel established by the dispatcher and receives response to the request from the server through the same channel.
2. Server:
    - It registers itself with dispatcher by its name and address.
    - A server component can be located on the same computer as client or may be reachable via network.
3. Dispatcher:
    - It offers functionality for establishing communication channel between clients and servers.
    - It establishes the communication links to the server using available communication mechanism and returns communication handle to the client.
    - It resisters or unregisters servers.

Use case Diagram:

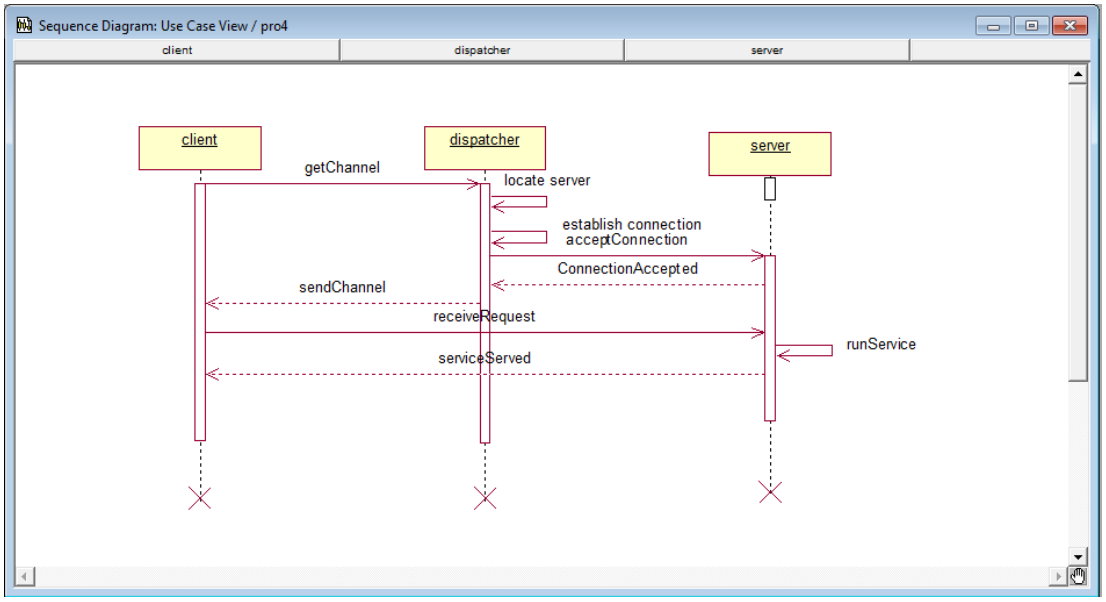


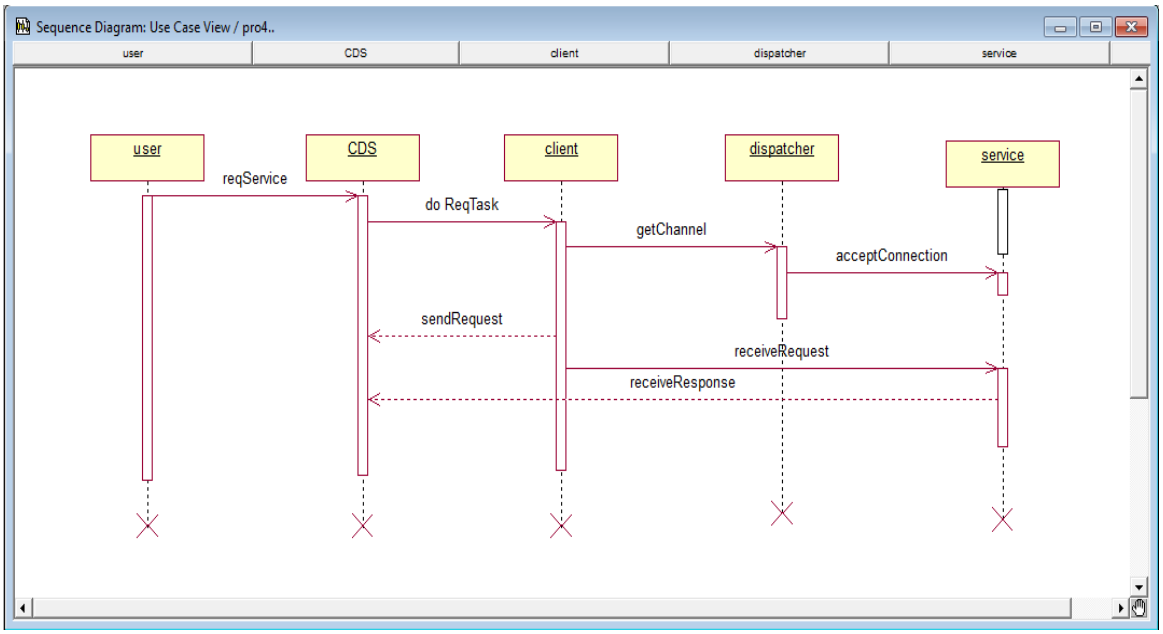
Class Diagram:



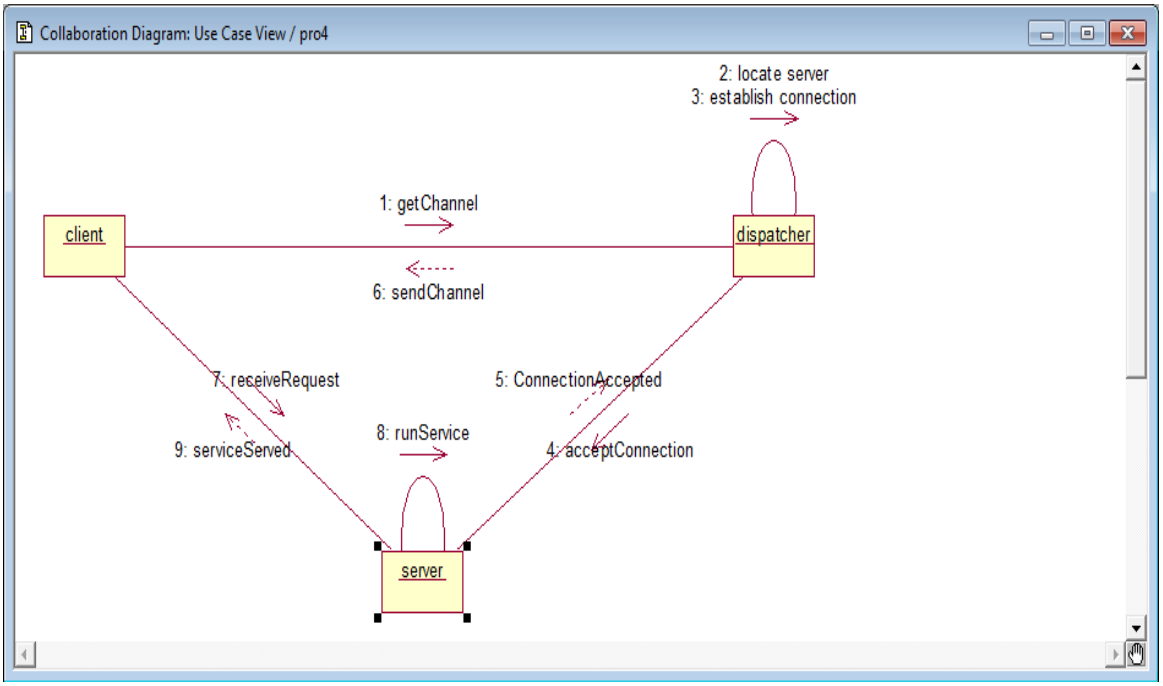
Dynamics:

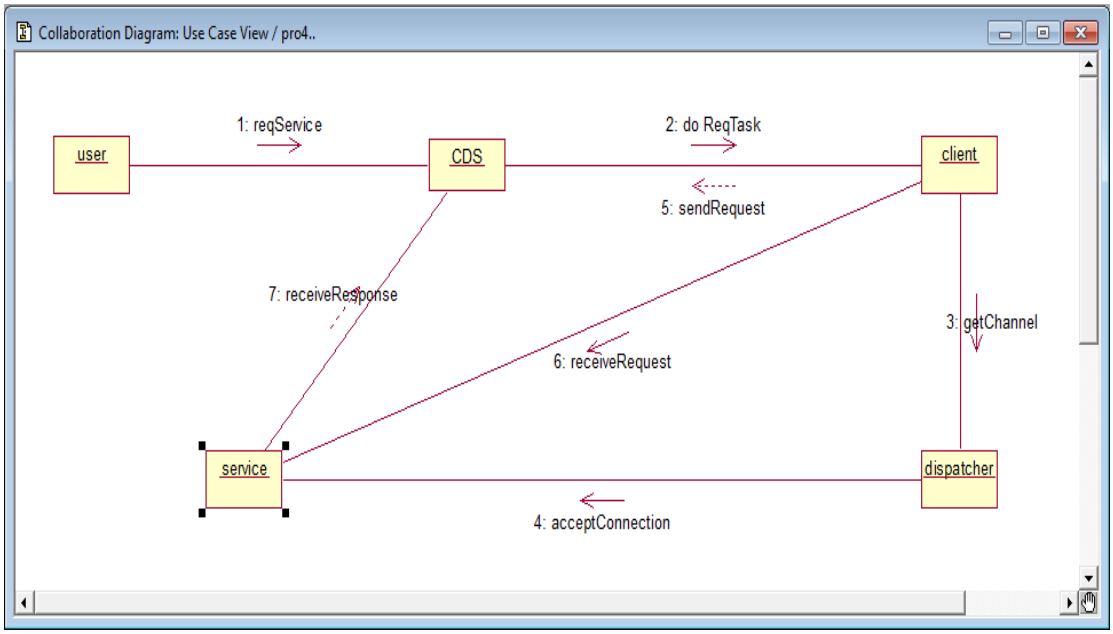
- Sequence diagram:



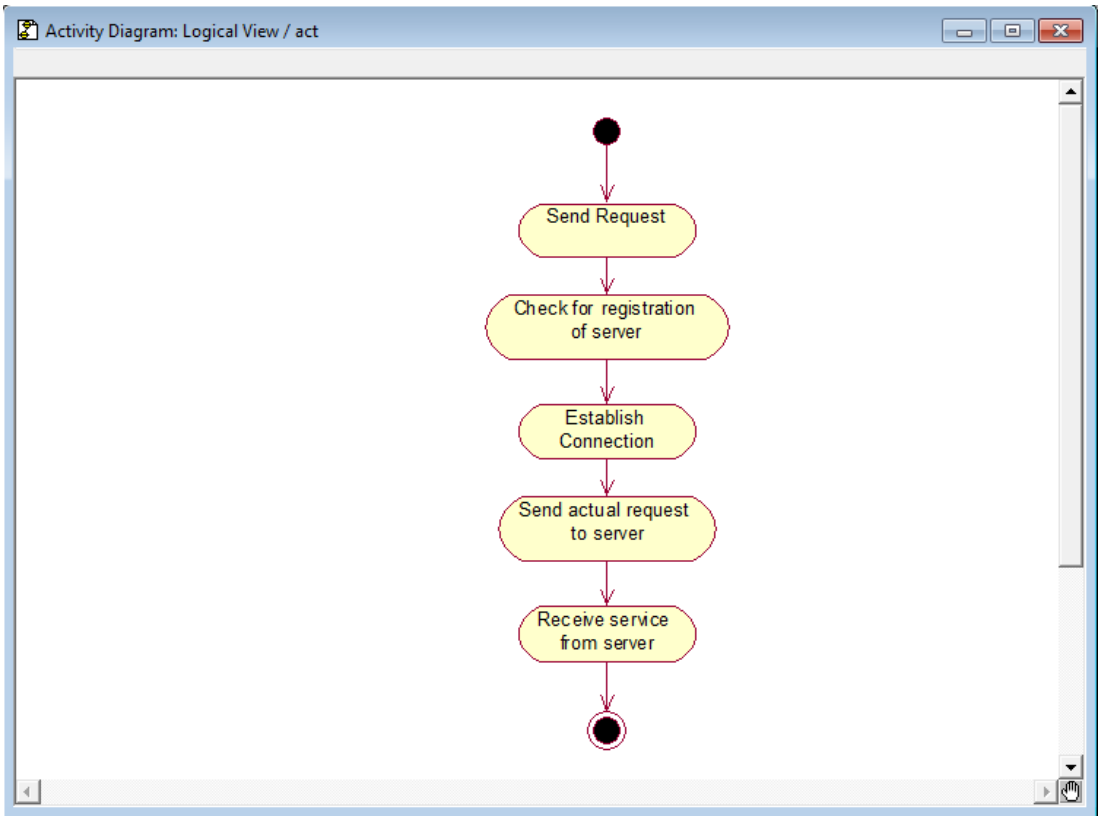


Collaboration diagram:





Activity diagram:



Implementation:

Cds.java

```
import java.io.*;
import java.lang.*;
public class Cds
{
      public static void main(String [] args)
     {
          Clientclass c=new Clientclass();
           c.doTask();
     }
}
```

Server1.java

```
import java.io.*;
public class Server1
{
Public void server1() {
   }
  public void runservice()
   {
     System.out.println("\n Service provided!!!");
   }
}
```

Clientclass.java:

```
import java.io.*;
public class Clientclass
{    private Printserver print;
     private Dispatcher dis=new Dispatcher();
     public Printserver thePrintserver;
     public Clientclass()
    {
     }
     public void dotask()
    {
            dis.request_channel();
            dis.establish_channel();
  }
}
```

Dispatcher.java

```
import java.io.*;
public class Dispatcher
{
  private Clientclass client;
  private Printserver print=new Printserver();
  public Dispatcher()
```

```
  {
  }
  public void establishclient()
  {
    System.out.println("\n\n Established!!!");
     print.runservice();
  }
  public void request_channel()
  {
     System.out.println("\n\n Registerd!!!!!");
  }
}
```

Printserver.java

```
import java.io.*;
public class Printserver
{
  private Server1 server1=new Server1();
  private Clientclass client;
  private Dispatcher dis;
  public Printserver()
  {
  }
```

```
public void runservice()  {

 server1.runservice();

} }
```

Output :

Registerd!!!!!

Established!!!

Service provided!!!

Variants:

- Distributed dispatcher.
- Client dispatcher server with communication managed by clients.
- Client dispatcher server with heterogeneous communication.

Known Users:

- Sun's implementation of RPC(Remote procedure calls).
- OMG CORBA.

Consequences:

Advantages:

- Exchangeability of server.
- Location migration transparency.
- Reconfiguration.
- Fault tolerance.

5.Proxy pattern

Intention:

- The Proxy design pattern makes client in compliment communicate with representative rather than component itself.
- Introducing set of placeholder can serve many purposes including enhance efficiency, easier access and protection from unauthorized access.

Alias:

- Decoder pattern

Context:

- A client needs access to services of another component direct access is technically possible but may not be best approach.

Problem:

- It is often appropriate to access component direct. We want to hard-code its physical location into clients and directed and unrestricted access to component may be inefficient and unsure.
- Additional control mechanism is needed.
- Access the component should be runtime efficient, cost effective and say for both client and component.
- Accessing component should be transparent and simple for client.
- Client should be well managed for possible performance or financial penalty for accessing remote client.

Solution:

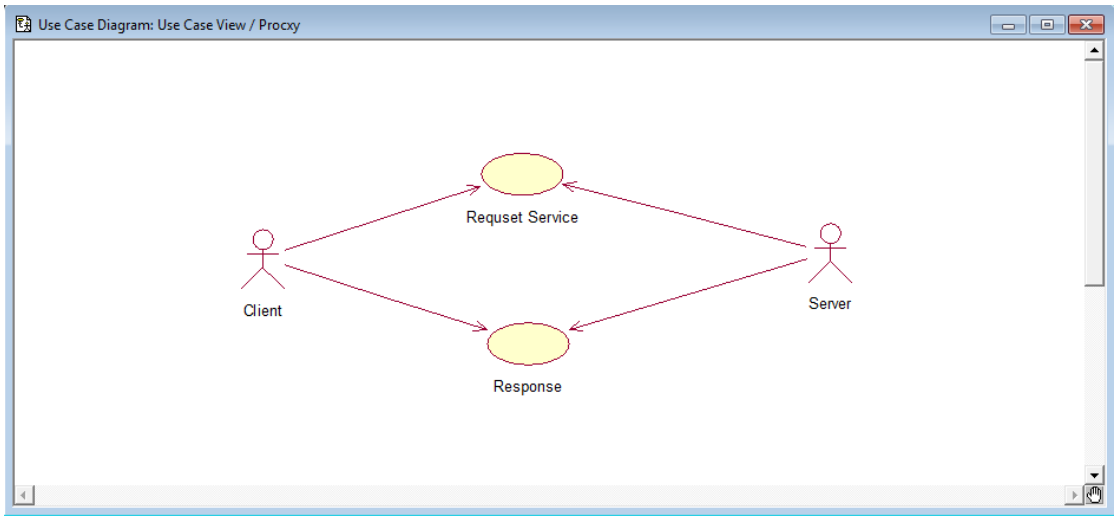
- Let client communicate with representative rather than component list
- This representative, called proxy, offer interface of component but performs additional pre- and post-processing such as control checking and making read-only copies of the original(server).

Structure:

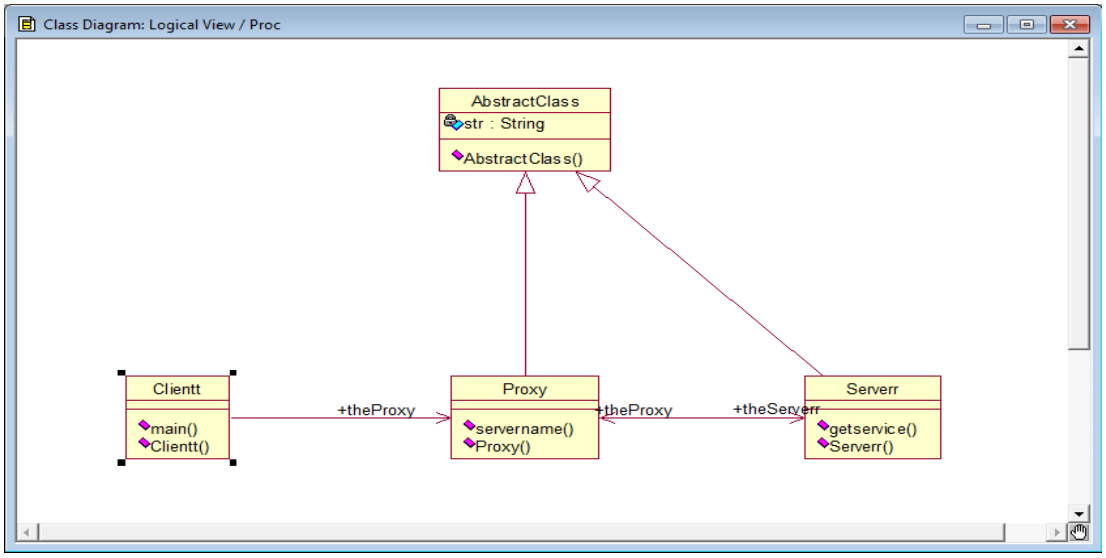
- Participants:

  - Client: Uses the interface provided by the proxy should request particular services
  - Proxy: Provide the interface of the original to client ensure the same efficient and correct access to original.
  - Server: Implement particular services.

Use Case Diagram:

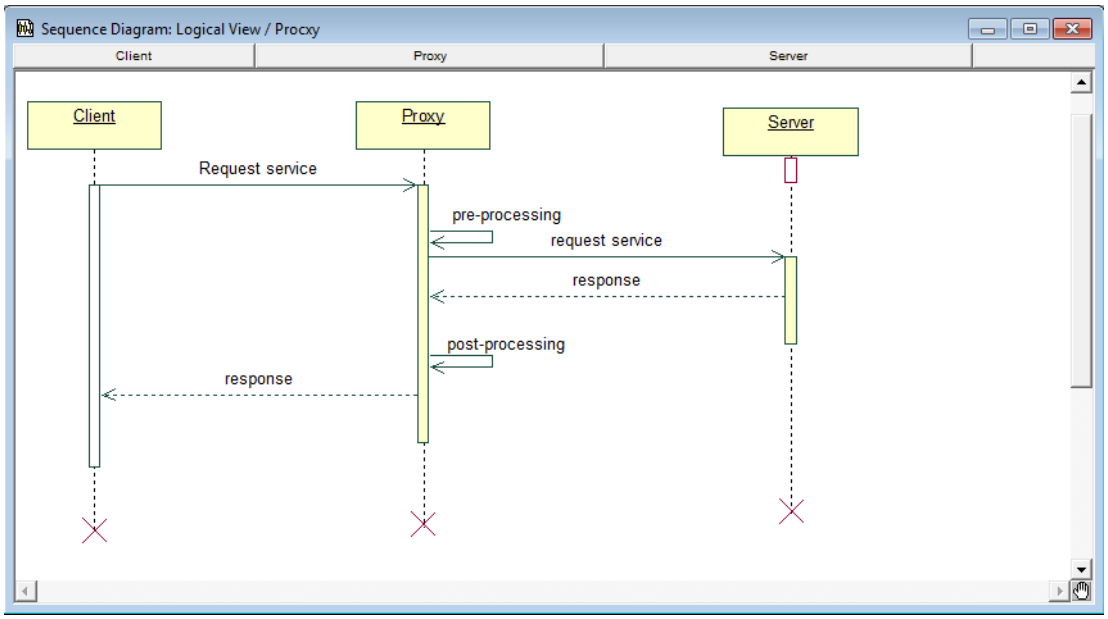


Class Diagram:

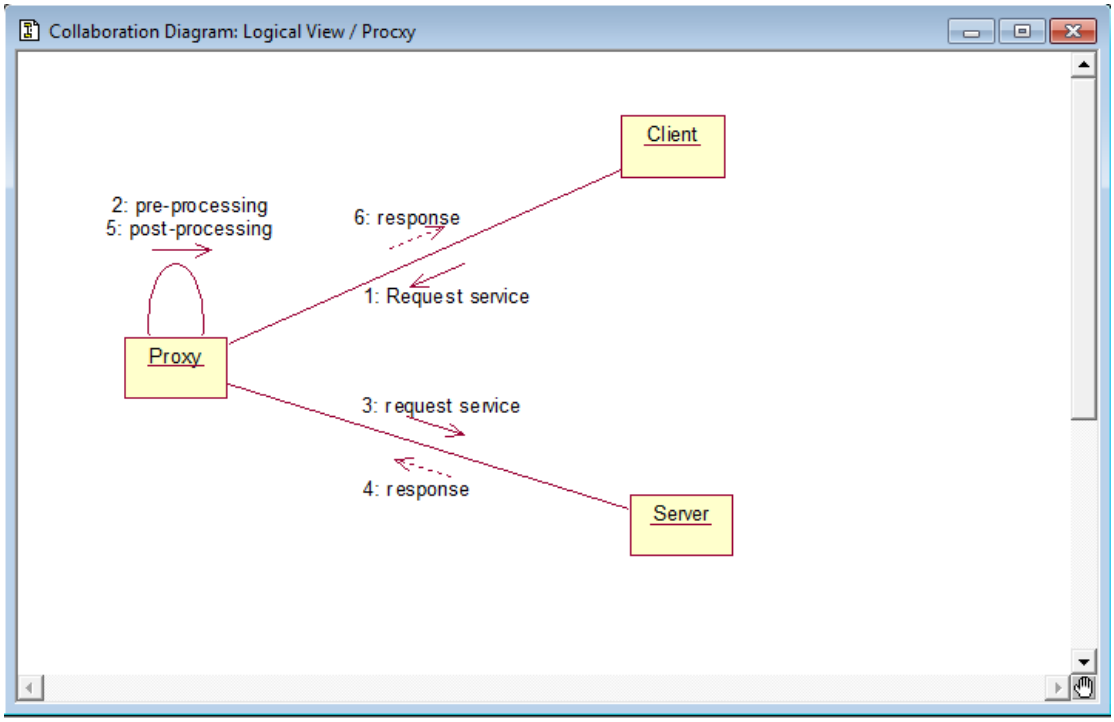


Dynamics:

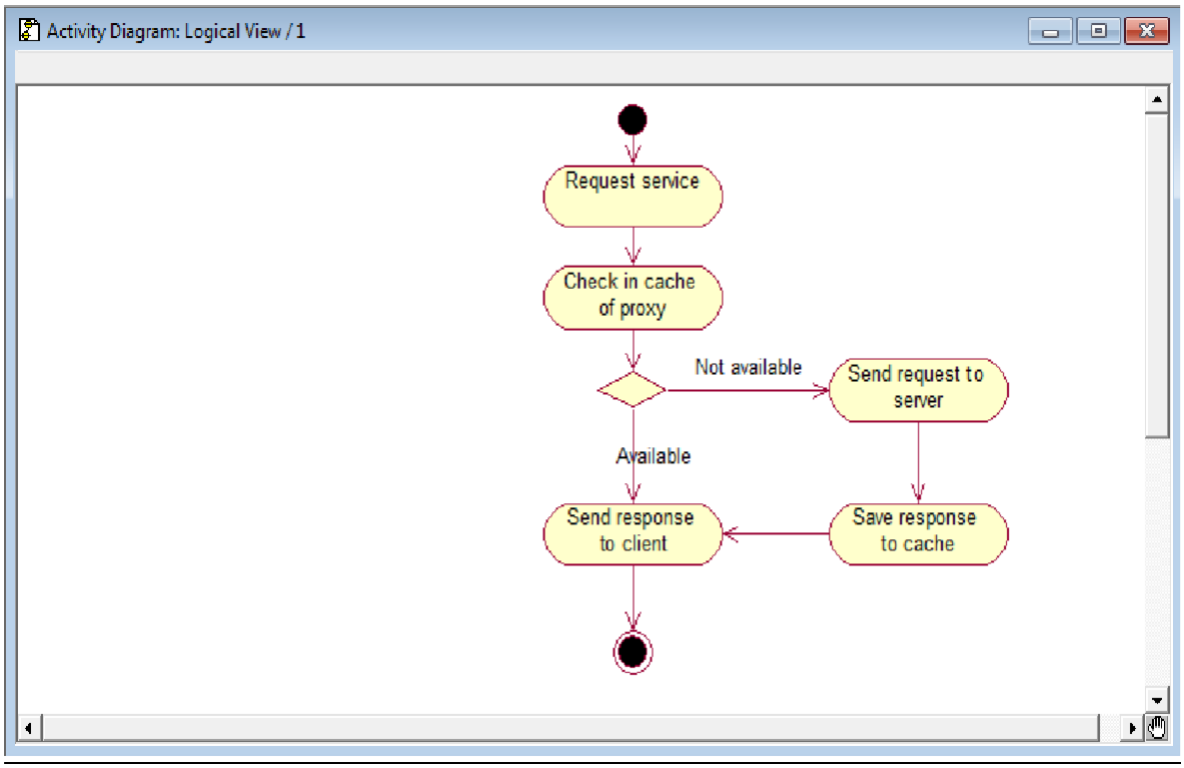
- Sequence diagram:



- Collaboration diagram:



- Activity diagram:



Implementation:

Abstractclass.java

```
import java.io.*;

import java.lang.*;

public class AbstractClass

{

public String str;

public AbstractClass() throws IOException

{

str="";

}

}
```

Server.java

```
import java.io.*;
import java.lang.*;
public class Server extends AbstractClass
{
public Proxy theproxy;
public Server() throws IOException
{
}
public String getServer() throws IOException
{
System.out.println("server initialize");
str="hello";
return str;
}
}
```

Proxy.java

```
import java.io.*;
import java.lang.*;
 public class Proxy extends AbstractClass
 {
 public Server theserver;
 public AbstractClass theabclass;
```

```
public Client theclient;
public Proxy() throws IOException
{
}
public void ServerName() throws IOException
{
System.out.println("\nPROXY\n");
if(str=="")
{
System.out.println("\ntrying to get connected");
theserver=new Server();
str=theserver.getServer();
}
else
{
System.out.println("proxy you.... ");
System.out.println("proxy says"+str);
}
}
}
```

Client.java

```
import java.io.*;
import java.lang.*;
public class Client
{
    public Proxy theproxy;
   public static void main(String args[]) throws IOException
  {
        Proxy mproxy=new Proxy();
       System.out.println("first time establishing connection\n");
       mproxy.ServerName();
       System.out.println("second time");
       mproxy.ServerName();
   } }
```

Output:

```
first time establishing connection
PROXY
trying to get connected
server initialize
second time
PROXY
proxy you.....
proxy sayshello
```

Variants:

1. Remote proxy
2. Protection proxy
3. Cache proxy
4. Synchronization proxy
5. Counting proxy
6. Virtual proxy
7. Firewall proxy

Known Users:

- NEXTSTEP
- OMG-CORBA
- WORLD WIDE WEB proxy
- OLE

Consequences

Advantages:

- Enhance efficiency and lower cost.
- Decoupling client from location of server component.
- Separation of house keeping code from functionality.

Disadvantages:

- Less efficiency due to indirection.
- Overkill via sophisticated skill.
- Caching, coding and demanding.

6. POLYMORPHISM

Problem:

How handle alternatives based on type? How to create pluggable software components?

Solution:

When related alternatives or behaviours vary by type (class), assign responsibility for the behaviour using polymorphic operations to the types for which the behaviour varies.

Corollary:

Do not test for the type of an object and use conditional logic to perform varying alternatives based on type. Alternatives based on type Conditional variation are a fundamental theme in programs. If a program is designed using if-then-else or case statement conditional logic, then if a new variation arises, it requires modification of the case logic often in many places. This approach makes it difficult to easily extend a program with new variations because changes tend to be required in several places wherever the conditional logic exists.

Participants:

The Interface: This interface will provide the behaviour which varies according to the class type. All classes implementing this interface will write the method accordingly.

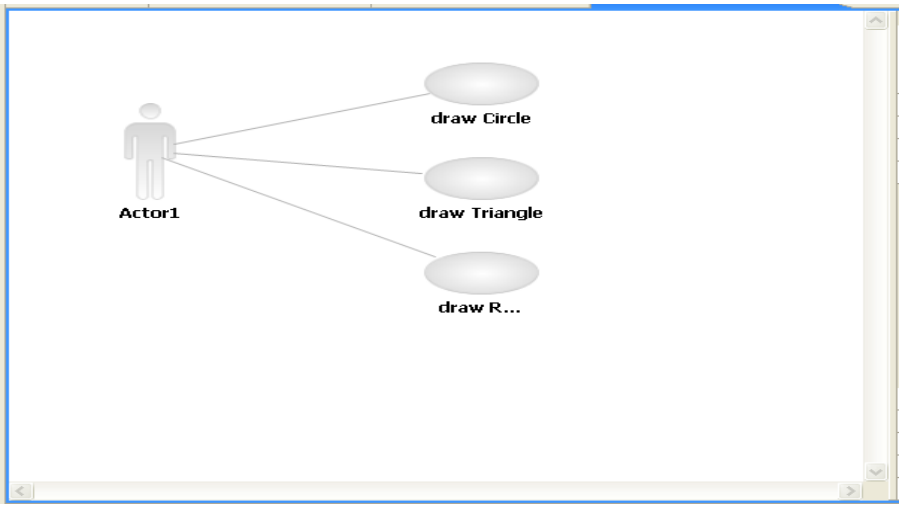
Implementer:

The classes will implement the operations from the interface as per the polymorphic nature.

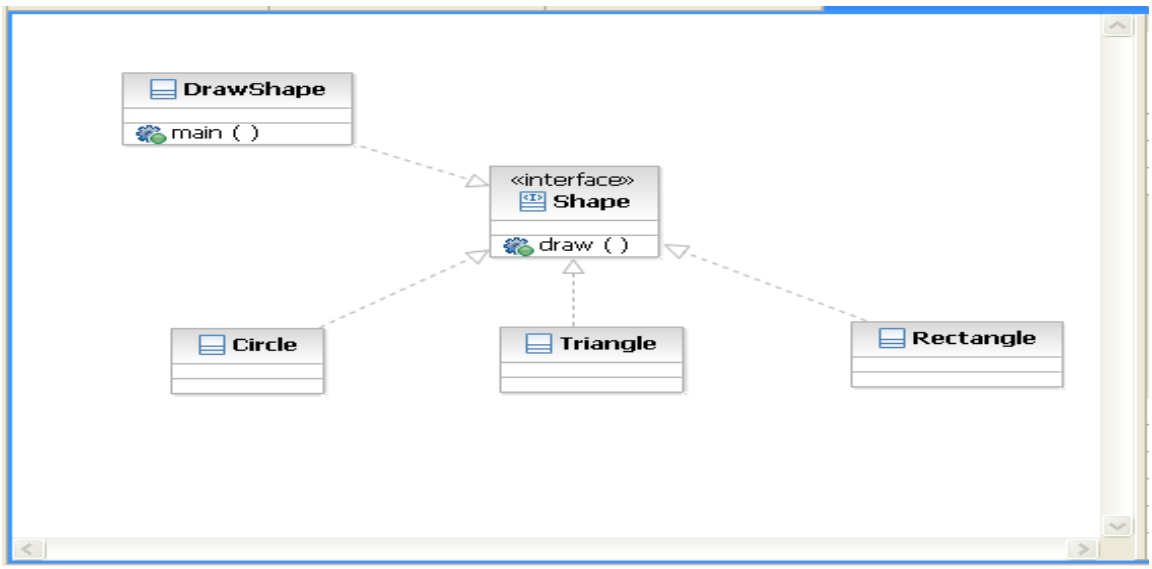
Example:

We have to develop an application which will draw different shapes. The user will use this application to draw shapes like Circle, Rectangle, triangle. At a later stage we can have some more shapes be added to the application.

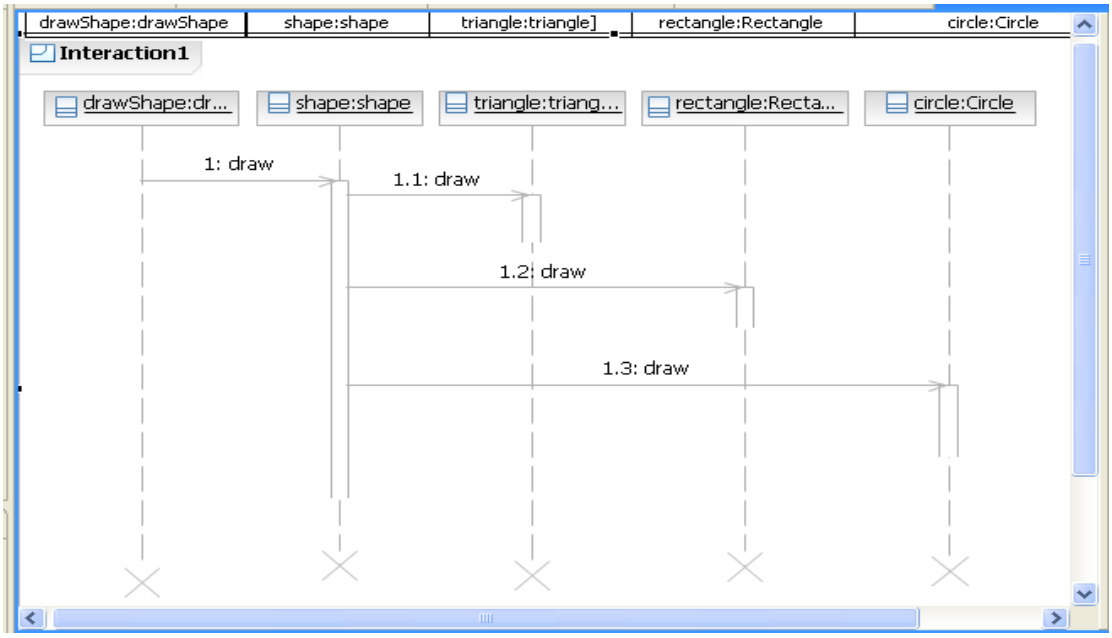
Use-Case Diagram



Class Diagram:

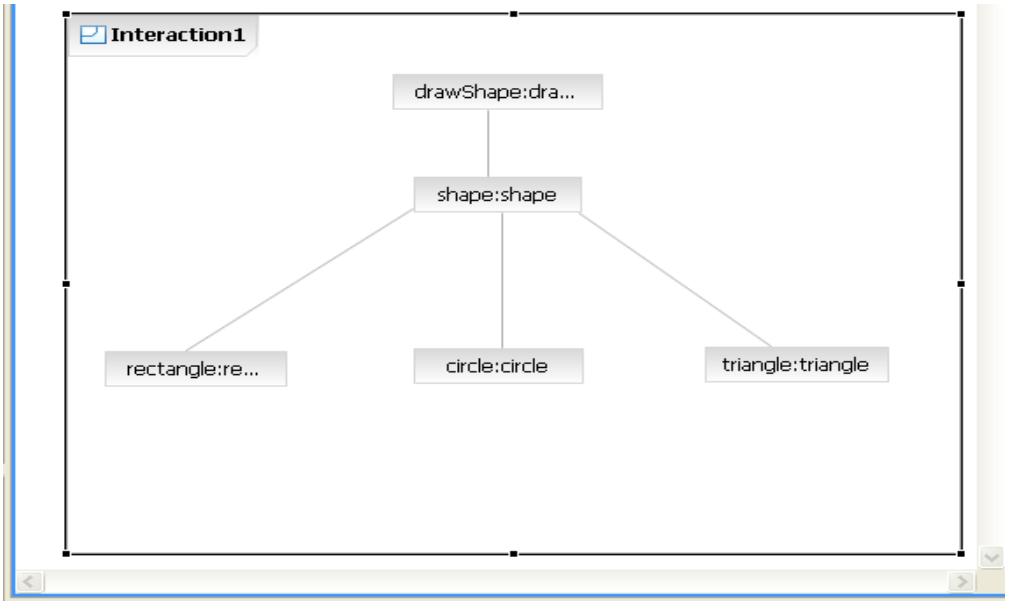


Sequence Diagram



Collaboration Diagram

Implementation



Circle.java

```
public class Circle implements Shape
{
        public void draw()
        {
                System.out.println("This is a circle");
        }
}
```

Rectangle.java

```
public class Rectangle implements Shape
 {
        public void draw()
       {
                System.out.println("This is a Rectangle");
        }
}
```

Triangle.java

```
public class Triangle implements Shape
 {
        public void draw()
        {
                System.out.println("this is a triangle");
        }
}
```

Shape.java

```
public interface Shape
{
        public void draw();
}
```

DrawShape.java

```
import java.util.Scanner;
public class DrawShape
{
        public static void main(String args[])
        {
                System.out.println("Please enter option draw 1. circle 2. triangle 3.rectangle");
                Scanner sin=new Scanner(System.in);
                int opt;
                Shape shape=null;
                opt=sin.nextInt();
                switch(opt)
                {
                case 1: shape=new Circle();
                                break;
                case 2: shape= new Triangle();
                                break;
                case 3: shape=new Rectangle();
                                break;
                default: System.out.println("Invalid option");
                        System.exit(0);
```

```
                }
                shape.draw();
        }

}
```

OUTPUT:

```
Please enter option to draw
1. circle 2. triangle 3.rectangle
1
This is a circle

Please enter option to draw
1. circle 2. triangle 3.rectangle
2
This is a triangle

Please enter option to draw
1. circle 2. triangle 3.rectangle
   3    This is a Rectangle
```

7. WHOLE-PART PATTERN

Intention:

- 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.

Alias:

- Composite Pattern.

Context:

- Implementing aggregate objects.

Problem:

- A complex should either be decomposed into smaller objects, or composed of existing objects, to support reusability, changeability and the recombination of recombination of the constituent objects in other types of aggregate.
- Clients should see the aggregate object as an atomic object that does allow any direct access to its constituent parts.

Solution:

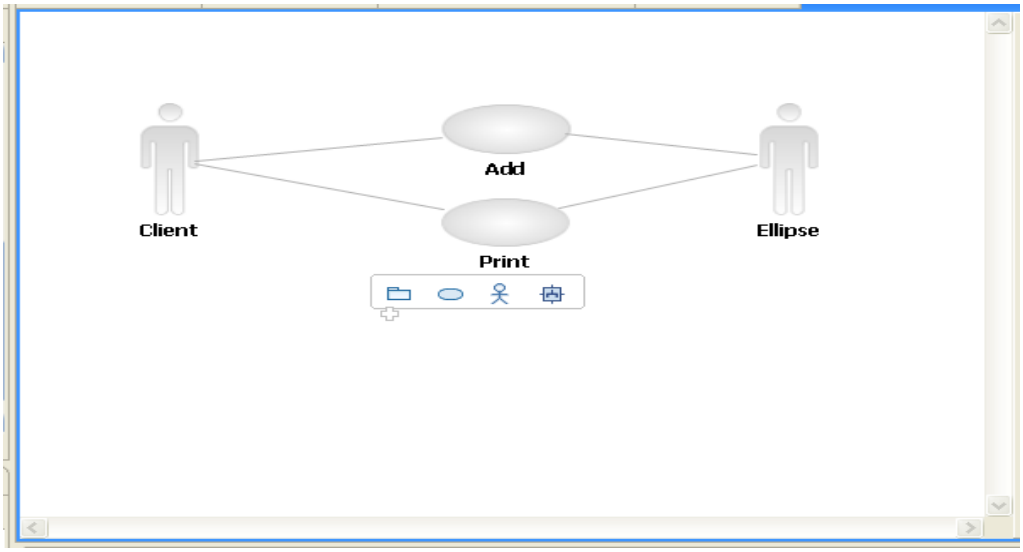
- An assembly parts relationship, which differentiates between a product and its parts or subassemblies. All parts are tightly integrated according to the internal structure of the assembly. The amount and type subassembly are predefined and does not vary.
- A container-contents relationship, in which the aggregated object represents a container.
- The collection-members relationship, which helps to group similar objects-such as 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 are treated equally.

Structures:

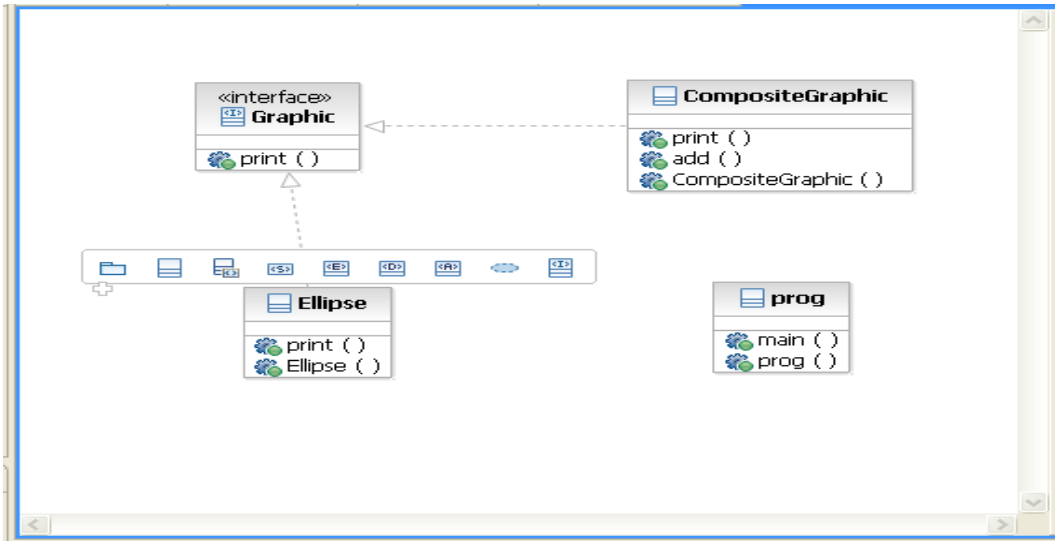
Participant:

- Whole: 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.
- Part: 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 result to the client.

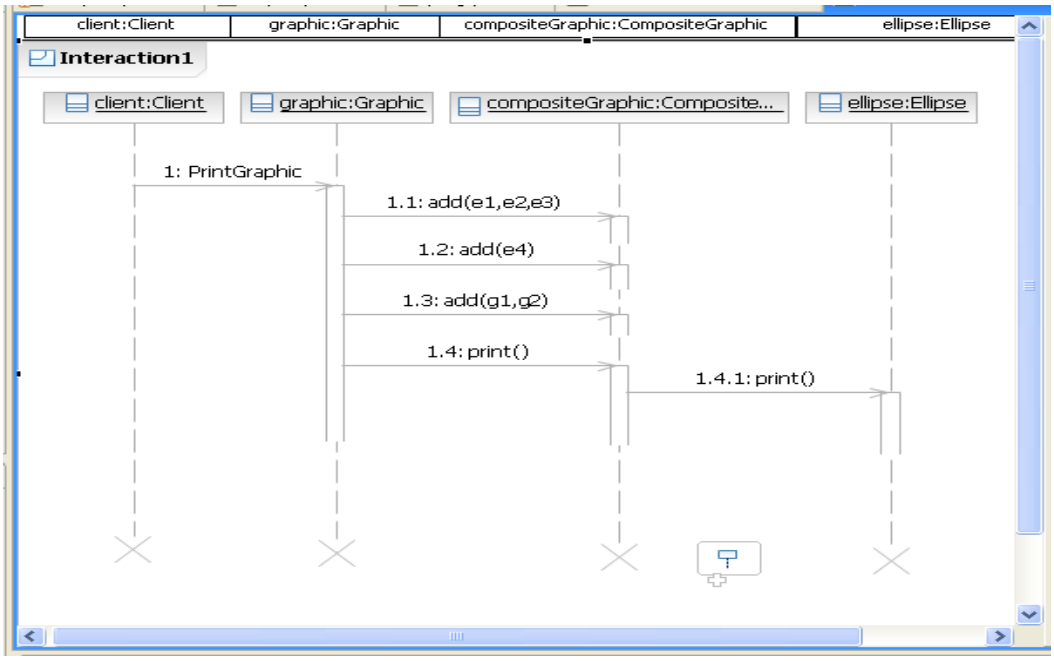
Use-Case Diagram



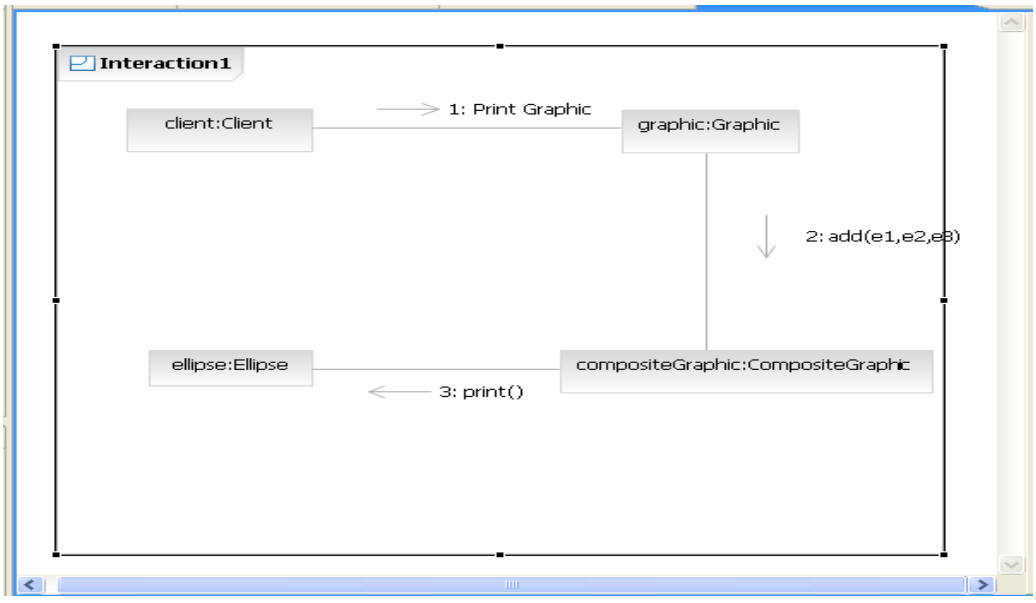
Class Diagram



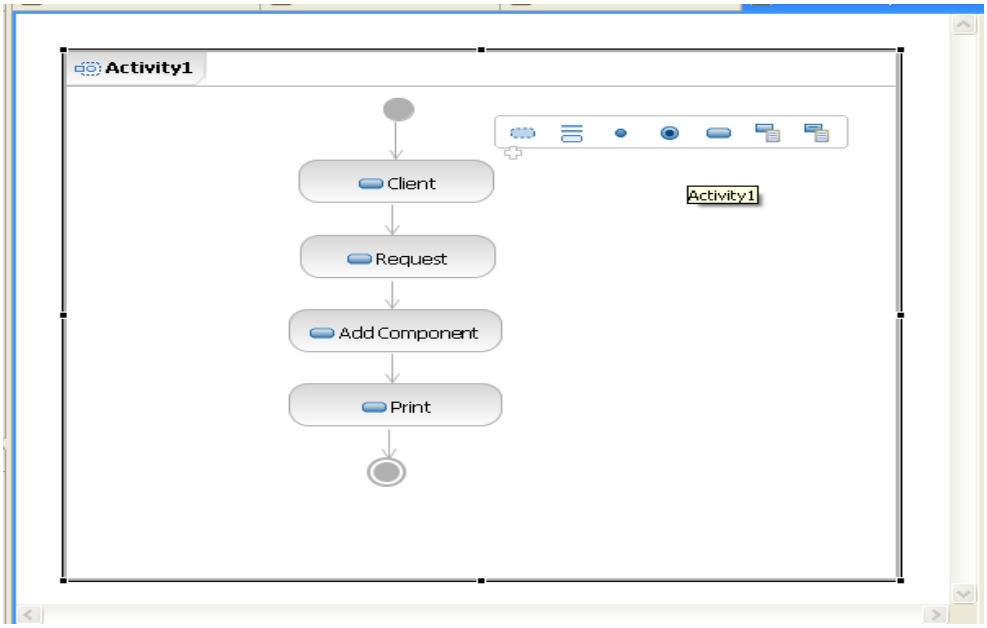
Sequence Diagram



Collaboration Diagram



Activity Diagram



Implementation

CompositeGraphic.java

```
import java.util.List;
import java.util.ArrayList;
public class CompositeGraphic implements Graphic
{
        private List <Graphic> childgraphic = new ArrayList<Graphic>();
        public void print()
        {
                for(Graphic graphic : childgraphic)
                {
                        graphic.print();
                }
        }
        public void add(Graphic graphic)
        {
                childgraphic.add(graphic);
        }
}
```

Ellipse.java

```
public class Ellipse implements Graphic
{
        public void print()
        {
                System.out.println("Ellipse");
        } }
```

Graphic.java

```
import java.util.List;
import java.util.ArrayList;
public interface Graphic
{
        public void print();
}
```

Prog.java

```
public class prog
{
        public static void main(String args[])
        {
                Ellipse e1 = new Ellipse();
                Ellipse e2 = new Ellipse();
                Ellipse e3 = new Ellipse();
                Ellipse e4 = new Ellipse();
                CompositeGraphic g = new CompositeGraphic();
                CompositeGraphic g1 = new CompositeGraphic();
                CompositeGraphic g2 = new CompositeGraphic();
                g1.add(e1);
                g1.add(e2);
                g1.add(e3);
                g2.add(e4);
                g.add(g1);
                g.add(g2);
```

```
        }
}
```

OUTPUT

```
Ellipse
Ellipse
Ellipse
```

8. CONTROLLER PATTERN

Context/Problem

The presentation-tier request handling mechanism must control and coordinate processing of each user across multiple requests. Such control mechanisms may be managed in either a centralized or decentralized manner.

Solution

Use a controller as the initial point of contact for handling a request. The controller manages the handling of the request, including invoking security services such as authentication and authorization, delegating business processing, managing the choice of an appropriate view, handling errors, and managing the selection of content creation strategies.

Controller

The controller is the initial contact point for handling all requests in the system. The controller may delegate to a helper to complete authentication and authorization of a user or to initiate contact retrieval.

Dispatcher : A dispatcher is responsible for view management and navigation, managing the choice of the next view to present to the user, and providing the mechanism for vectoring control to this resource. A dispatcher can be encapsulated within a controller or can be a separate component working in coordination. The dispatcher provides either a static dispatching to the view or a more sophisticated dynamic dispatching mechanism. For example, The dispatcher uses the RequestDispatcher object (supported in the servlet specification) and encapsulates some additional processing.

Helper

A helper is responsible for helping a view or controller complete its processing. Thus, helpers have numerous responsibilities, including gathering data required by the view and storing this intermediate model, in which case the helper is sometimes referred to as a value bean. Additionally, helpers may adapt this data model for use by the view. Helpers can service requests for data from the view by simply providing access to the raw data or by formatting the data as Web content.

For example, a helper may represent a Command object, a delegate , or an XSL Transformer, which is used in combination with a style sheet to adapt and convert the model into the appropriate form.

View : A view represents and displays information to the client. The view retrieves information from a model. Helpers support views by encapsulating and adapting the underlying data model for use in the display

Example:

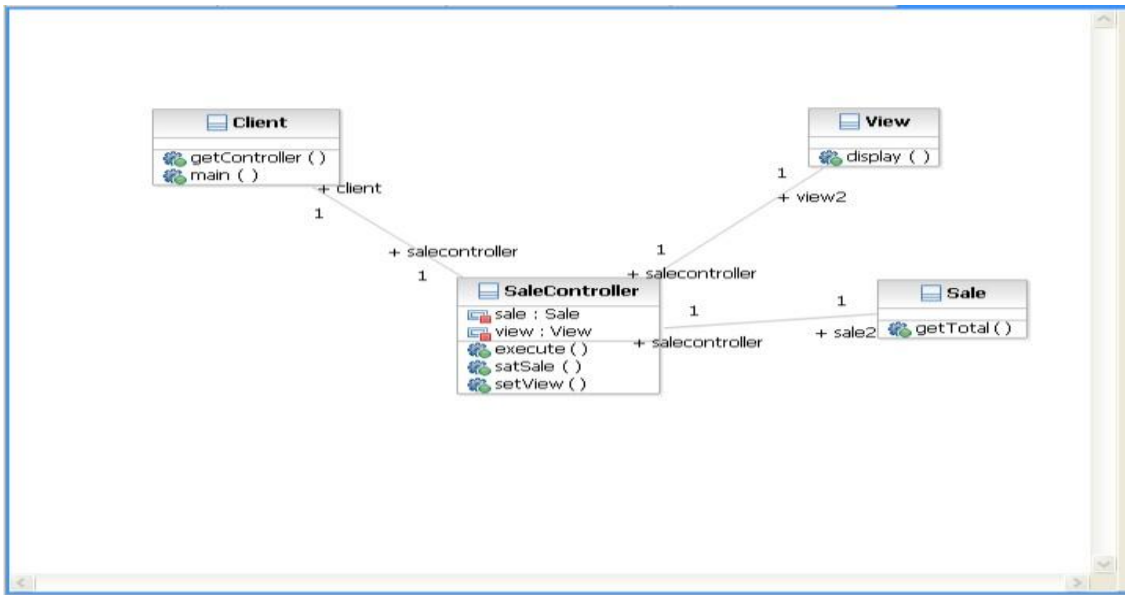
A Point Of Sale (POS) system is a computerized application used (in part) to record sales and handle payments; it is typically used in a retail store. It includes hardware components such as a computer and bar code scanner, and software to run the system. It interfaces to various service applications, such as a third-party tax calculator and inventory control.

Furthermore, we are creating a commercial POS system that we will sell to different clients with disparate needs in terms of business rule processing. Each client will desire a unique set of logic to execute at certain predictable points in scenarios of using the system, such as when a new sale is initiated or when a new line item is added. Therefore, we will need a mechanism to provide this flexibility and customization.

The POS will be calculating the total sales at any given point of time. Now from information expert pattern we know that Sale class will be calculating the total sale at any given point of time. Now the Controller Pattern suggests, that events coming from UI layer should not be directly accessing the expert classes. There should be a Controller class to control these events.

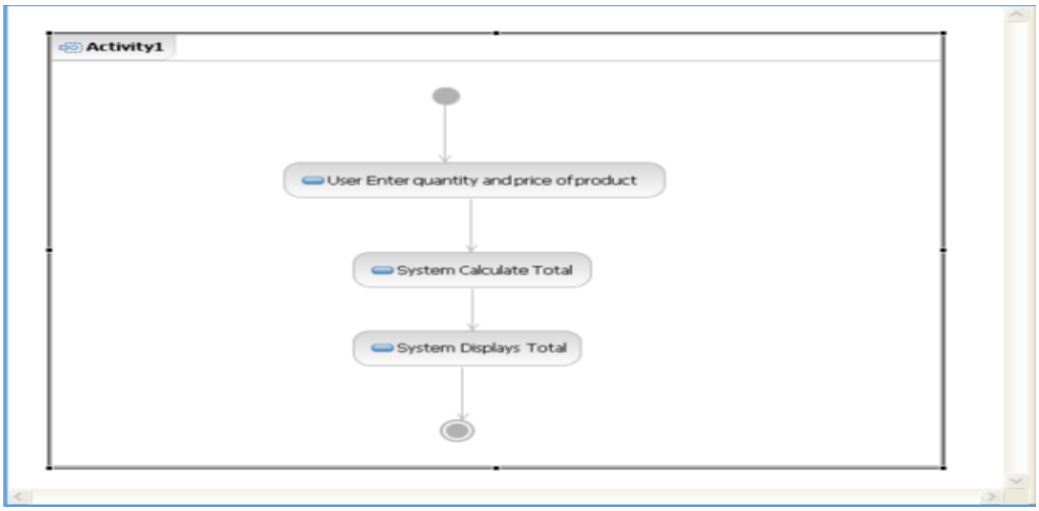
In this example, we will use a Sale Controller class that will be receiving the events from UI and forward it to the Sale class. There will be one View class to decide how the output will be displayed.

Class Diagram

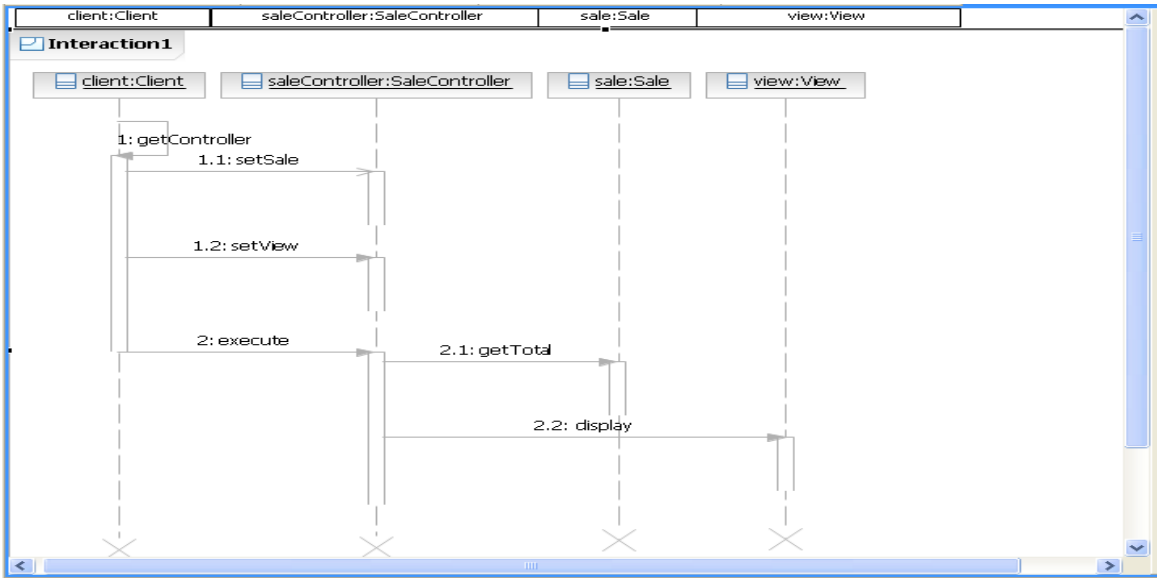


Use Case Diagram

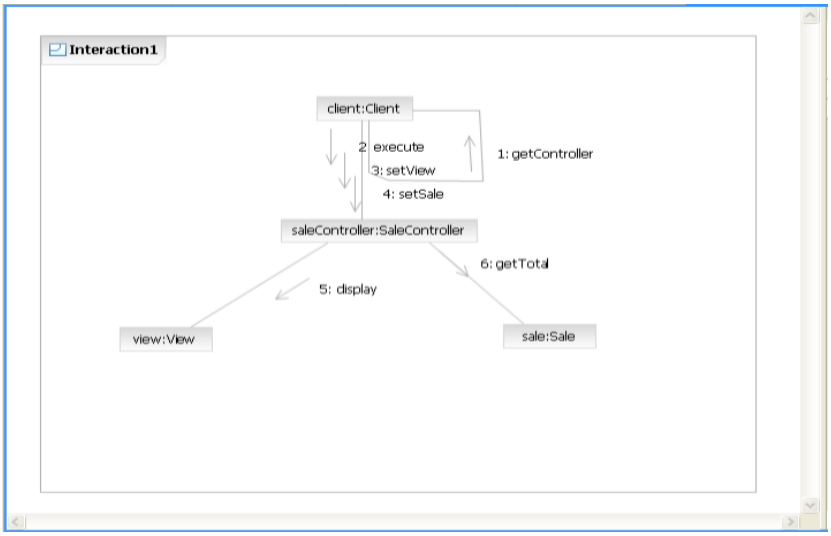
Activity Diagram



Sequence Diagram



Communication Diagram



Implementation

Sale.java

```
public class Sale
{
    //public SaleController SaleController;
    public float getTotal(int quantity, float price)
    {
        return quantity * price;
    }
}
```

View.java

```
public class View
{
    public void display(float total)
    {
```

```
        System.out.println("The total sale is:" +total);
    }
}
```

SaleController.java

```
public class SaleController
{
    public void execute(int quantity, float price)
    {
        Float result = sale.getTotal(quantity , price);
        vew.display(result);
    }
    public void setSale(Sale sale)
    {
        this.sale=sale;
    }
    public void setView(View view)
    {
        this.view=view;
    }
}
```

Client.java

```
import java.util.Scanner;
public class Client
{
    public SaleController SaleController;

    public static SaleController getController()
    {
        SaleController tc = new SaleController();
        Sale s = new Sale();
        View v = new view();
         tc.setSale(s);
         tc.setView(v);
          return tc;
      }
      public static void main(String args[])
      {
          SaleController tc = getController();
          Scanner scr = new Scanner(System.in);
          System.out.println("Enter the quantity:");
          int quantity = scr.nextInt();
          System.out.println("Enter the price");
          Float price = scr.nextFloat();
```

```
            tc.execute(quantity,price);
        }
    }
```

OUTPUT

Enter the quantity : 10

Enter the price : 1000

The Total Sale is : 10000