Particulars of experiments to be performed

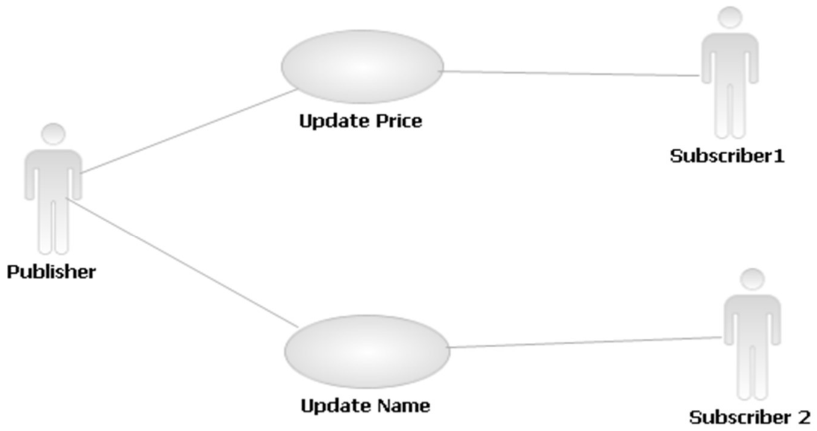
Contents

1. Publisher- Subscriber Pattern

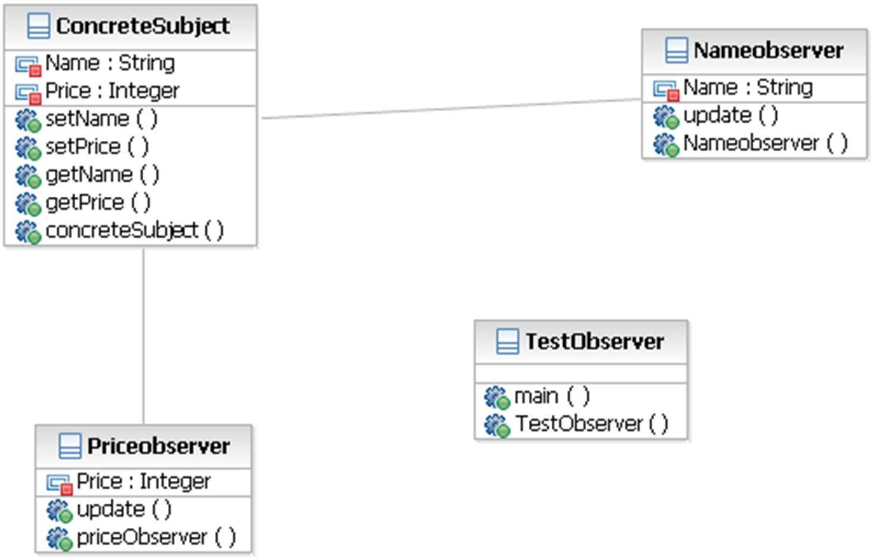
Objective:

- 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.
- The Publisher-Subscriber design pattern promotes loose coupling, flexibility, scalability, and maintainability in a software system by providing a means for components to communicate without being tightly bound to each other. It enhances the overall modularity and adaptability of the system.

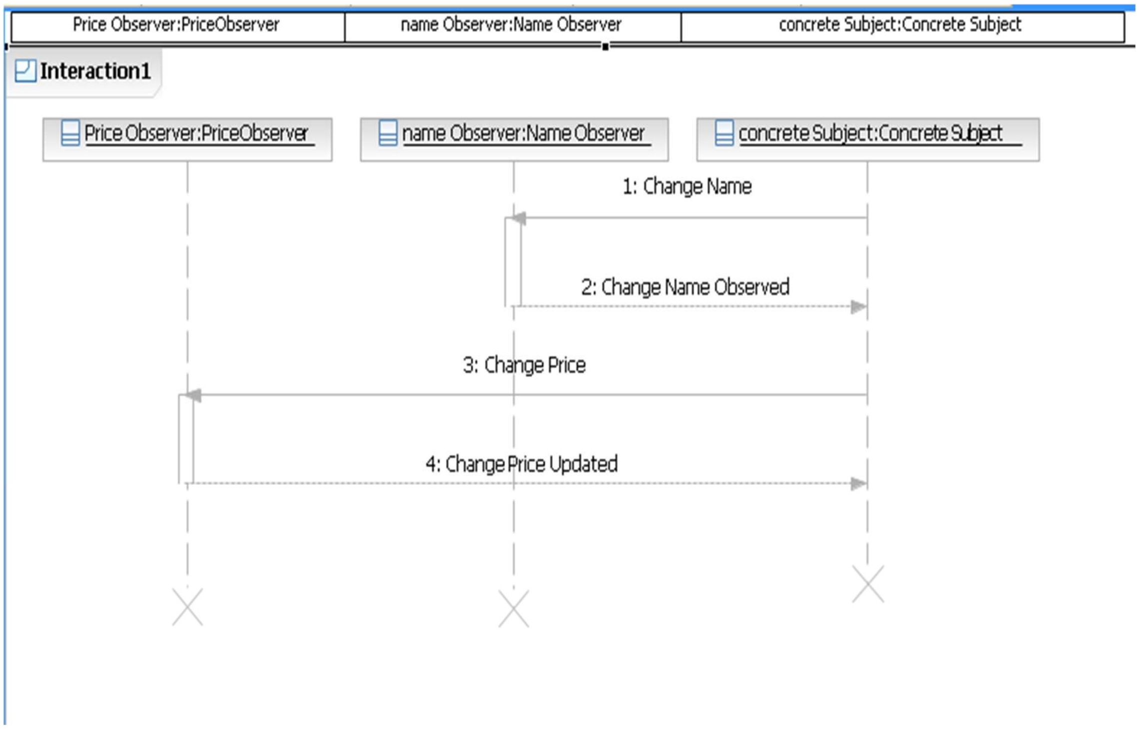
1. Use Case Diagram:



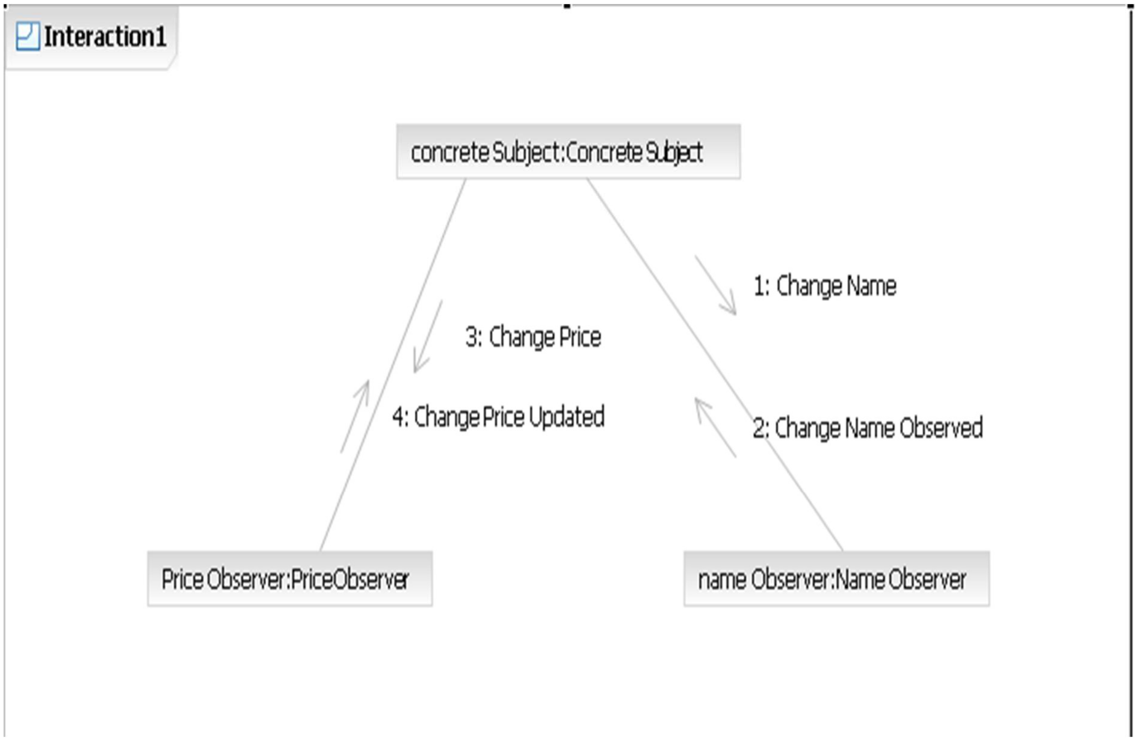
2. Class Diagram



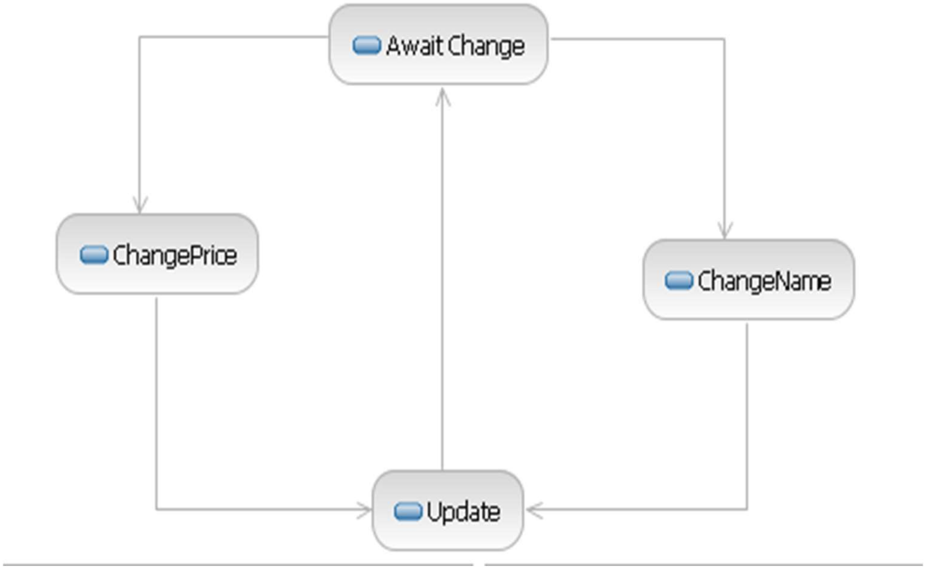
3. Sequence Diagram


Price Observer:PriceObserver
name Observer:Name Observer
concrete Subject:Concrete Subject
Interaction1
Price Observer:PriceObserver
name Observer:Name Observer
concrete Subject:Concrete Subject
1: Change Name
2: Change Name Observed
3: Change Price
4: Change Price Updated

4. Collaboration Diagram


Interaction1
concrete Subject:Concrete Subject
1: Change Name
3: Change Price
4: Change Price Updated
2: Change Name Observed
Price Observer:PriceObserver
name Observer:Name Observer

5. 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");
             }
```

```
    }
}
```

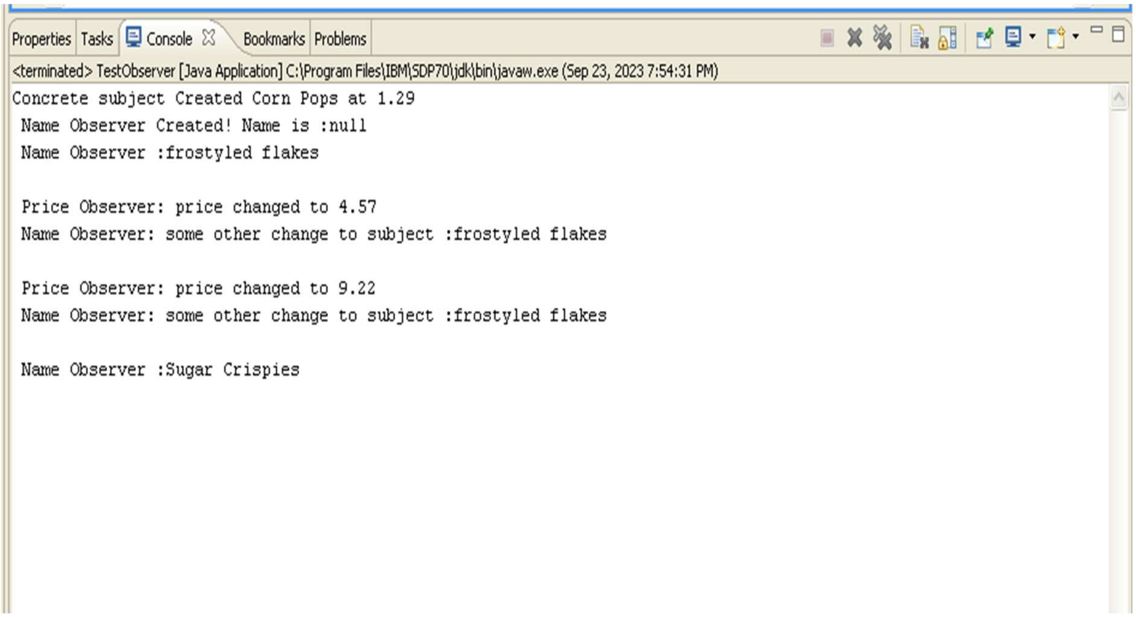
PriceObserver.java

```
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);
  }
  public void update(Observable obj,Object arg)
  {
       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");
  }
}
```

Output



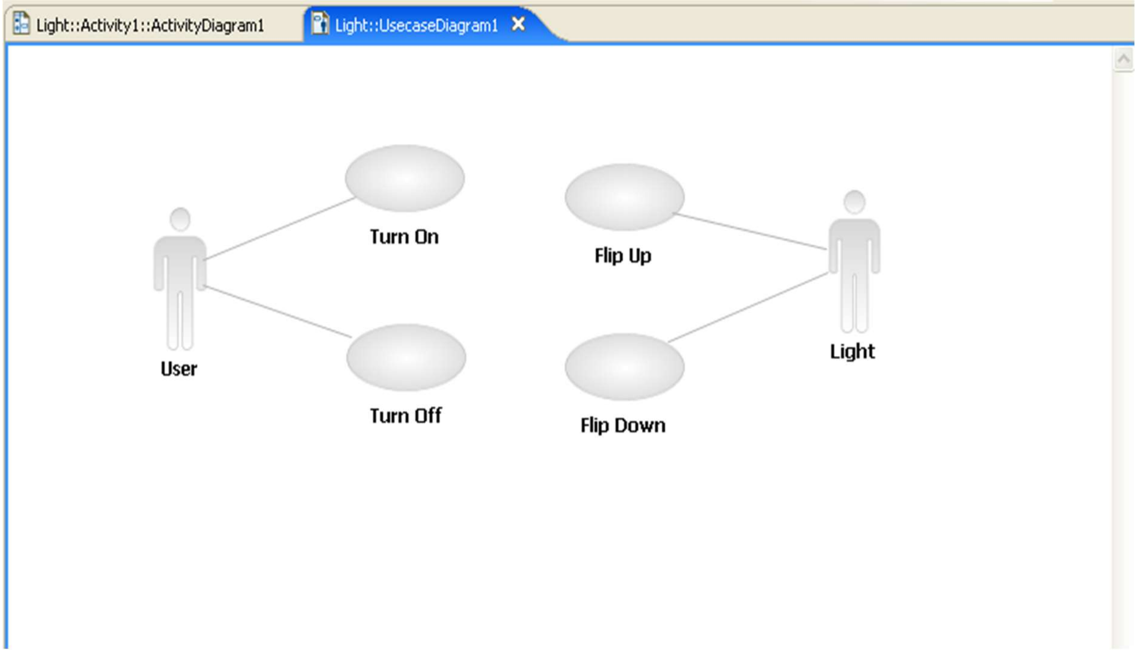
```
<terminated> TestObserver [Java Application] C:\Program Files\IBM\SDP70\jdk\bin\javaw.exe (Sep 23, 2023 7:54:31 PM)
Concrete subject Created Corn Pops at 1.29
 Name Observer Created! Name is :null
 Name Observer :frostyled flakes

 Price Observer: price changed to 4.57
 Name Observer: some other change to subject :frostyled flakes

 Price Observer: price changed to 9.22
 Name Observer: some other change to subject :frostyled flakes

 Name Observer :Sugar Crispies
```

2. Command Processor Pattern

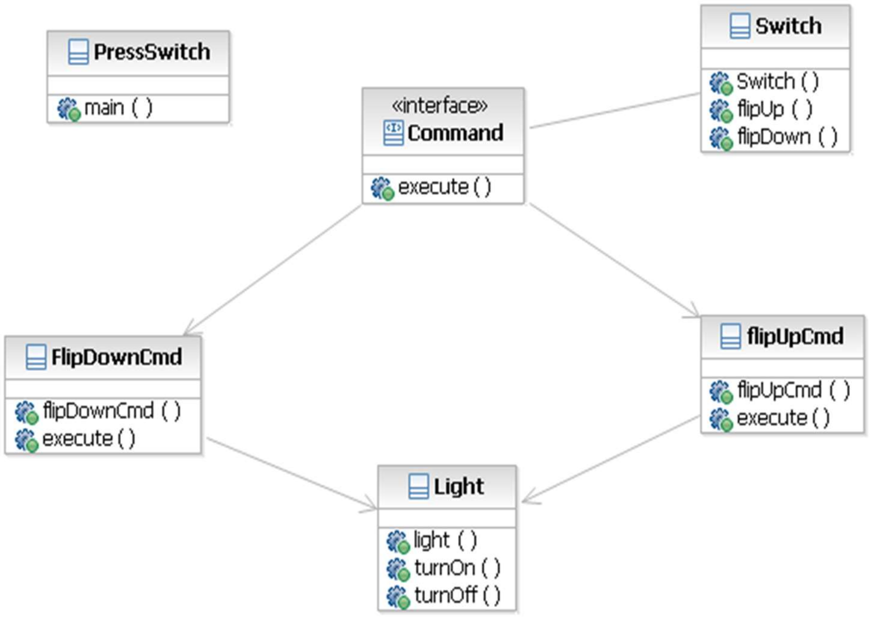
Objective

- System management.
- System needs 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.

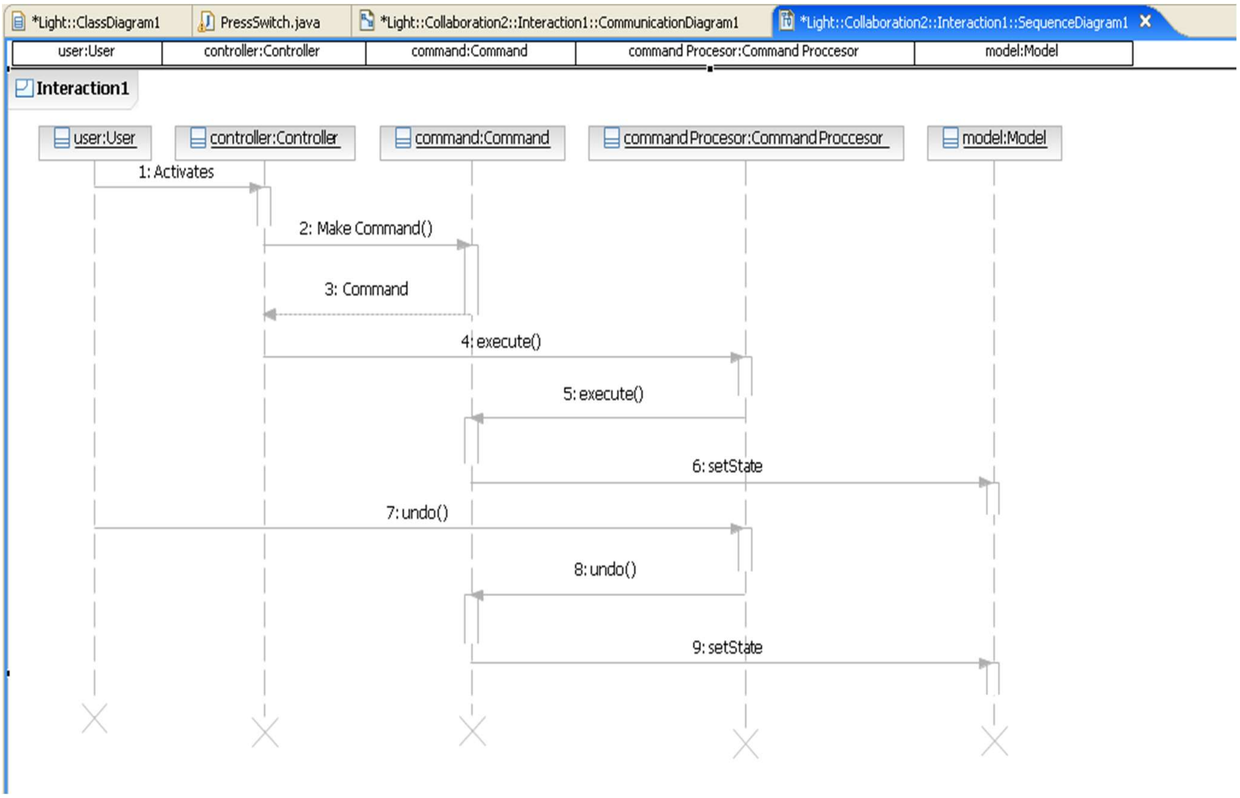
1. Use Case Diagram



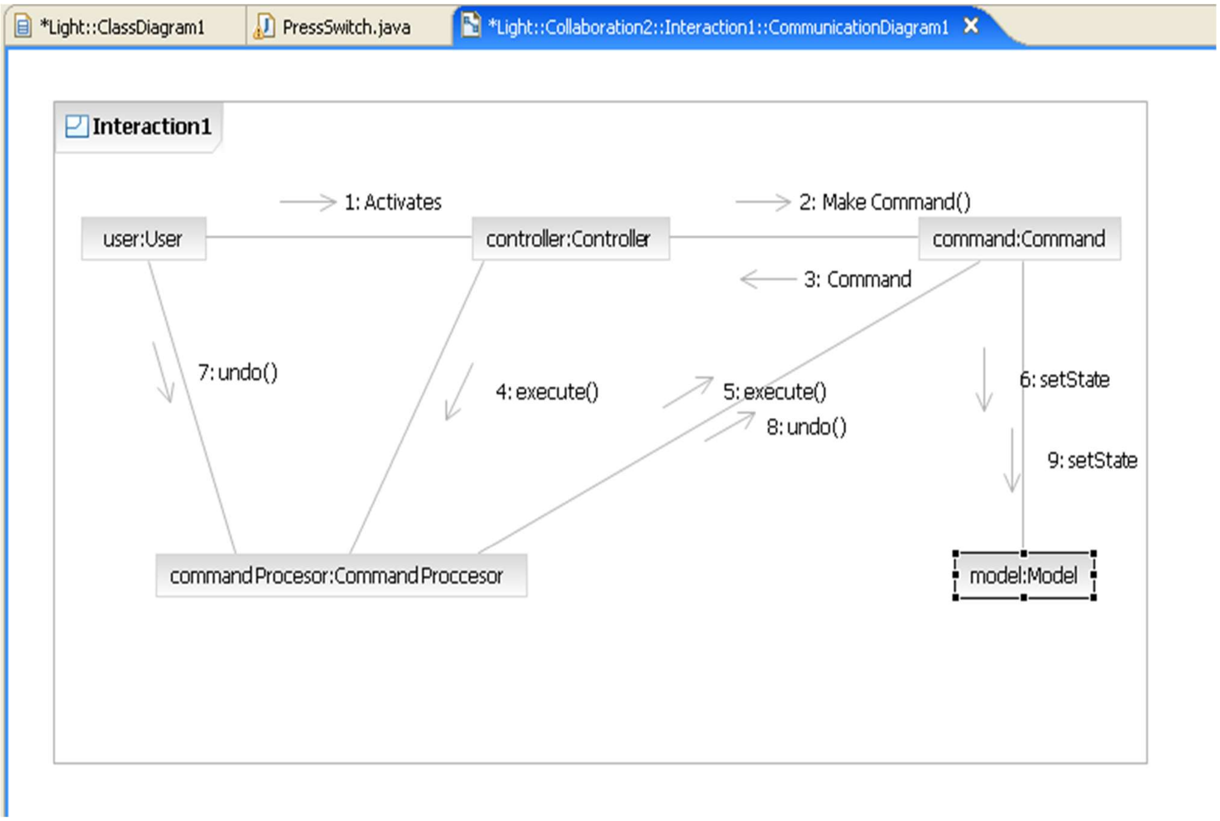
2. Class Diagram



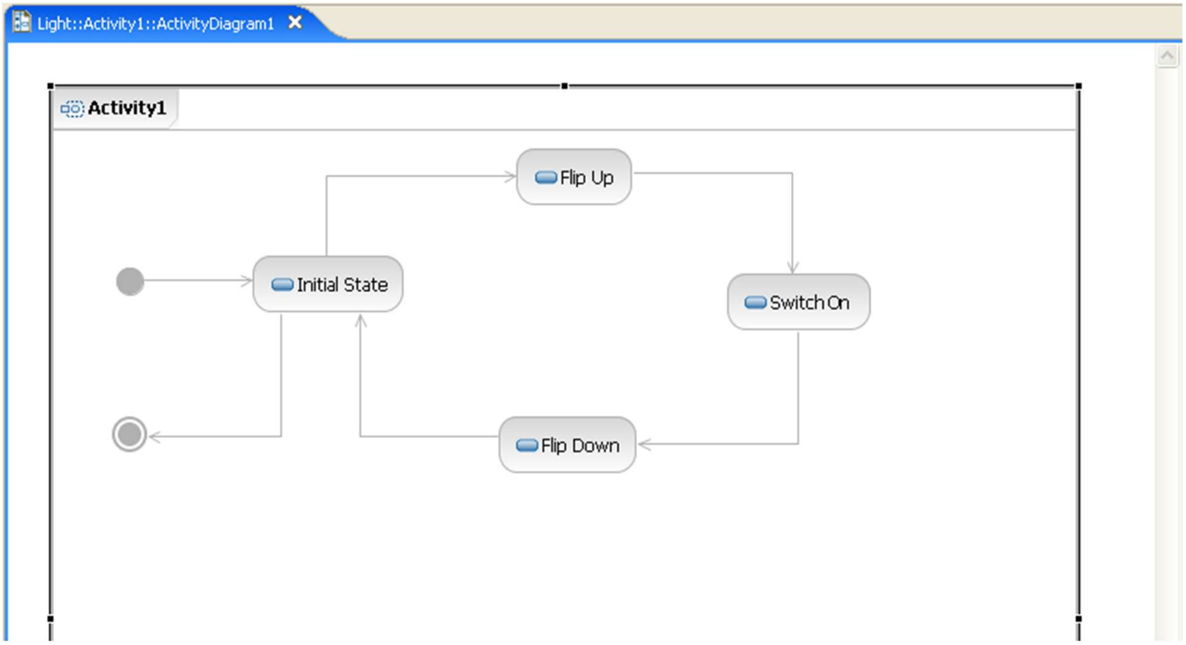
3. Sequence Diagram



4. Collaboration Diagram



5. 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();
}
```

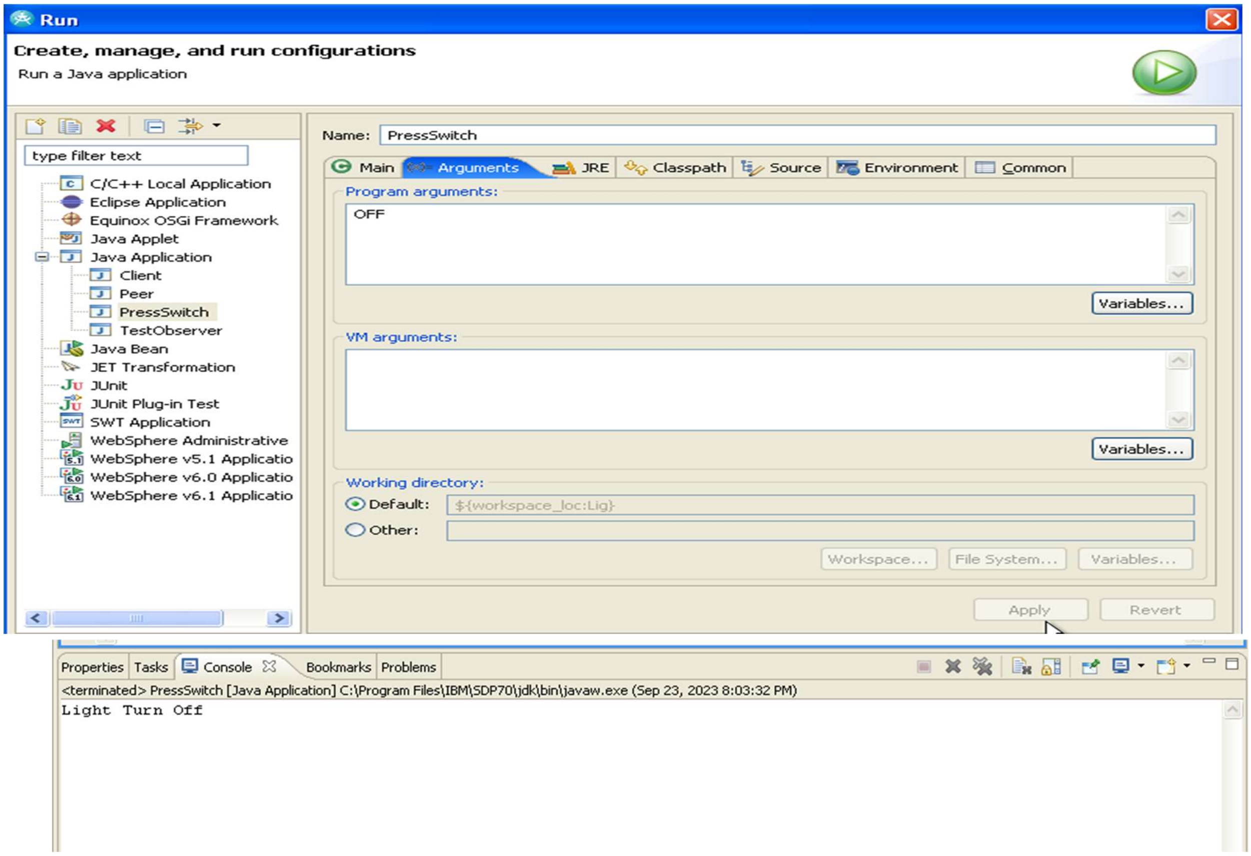
flipupCmd.java

```
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



Run

Create, manage, and run configurations

Run a Java application

Name: PressSwitch

Program arguments:

```
OFF
```

Working directory:

○ Default: ${workspace_loc:Lig}

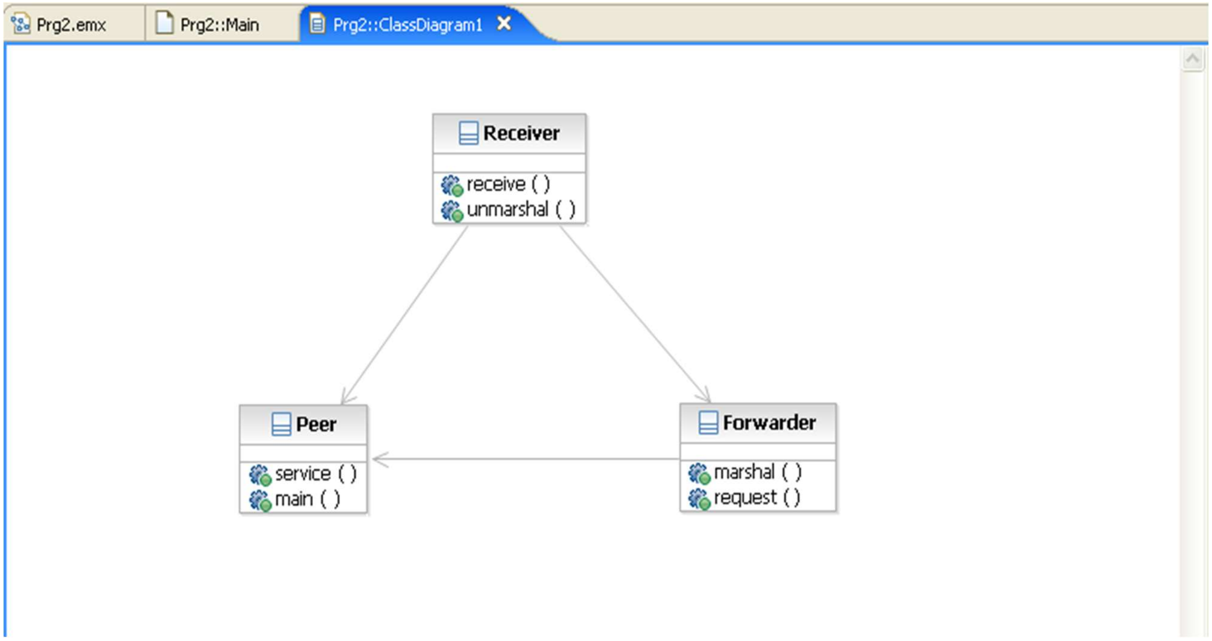
```
<terminated> PressSwitch [Java Application] C:\Program Files\IBM\SDP70\jdk\bin\javaw.exe (Sep 23, 2023 8:03:32 PM)
Light Turn Off
```

3. Forwarder –Receiver Pattern

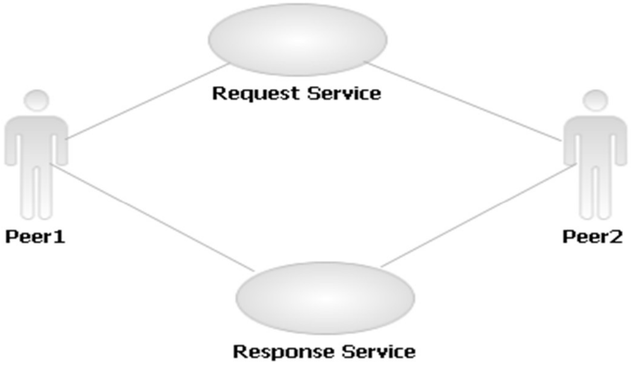
Objectives:

- 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.

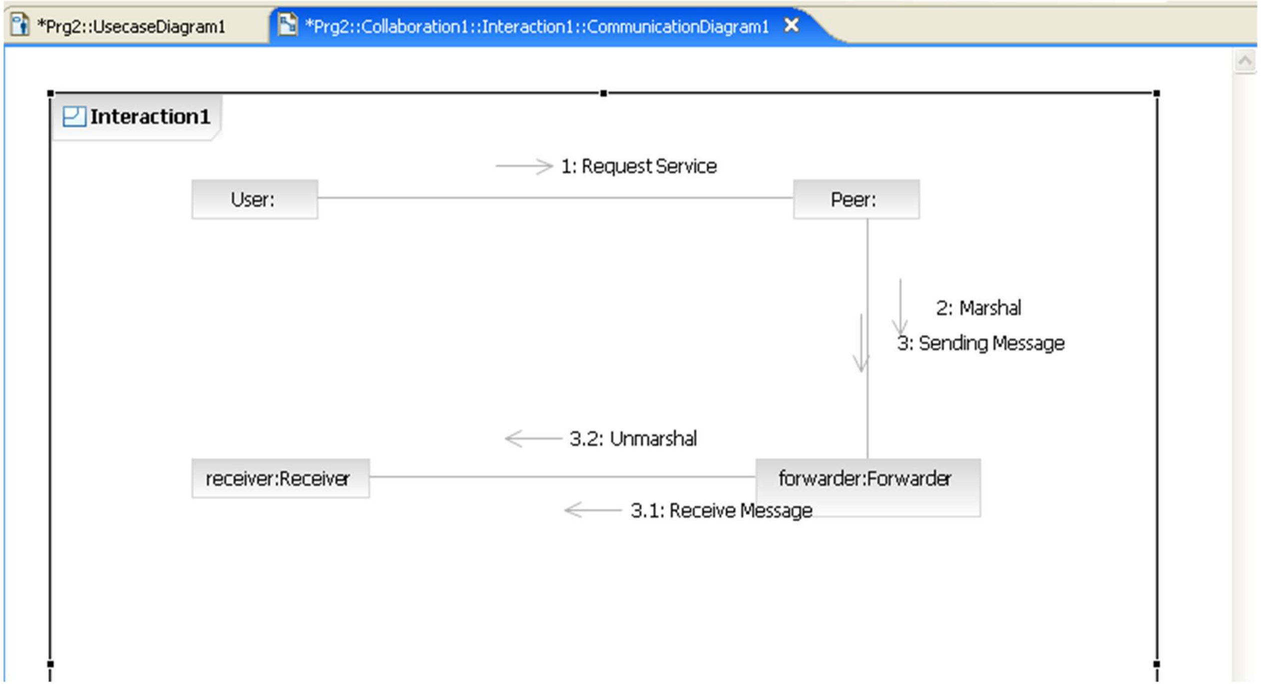
1. Class Diagram



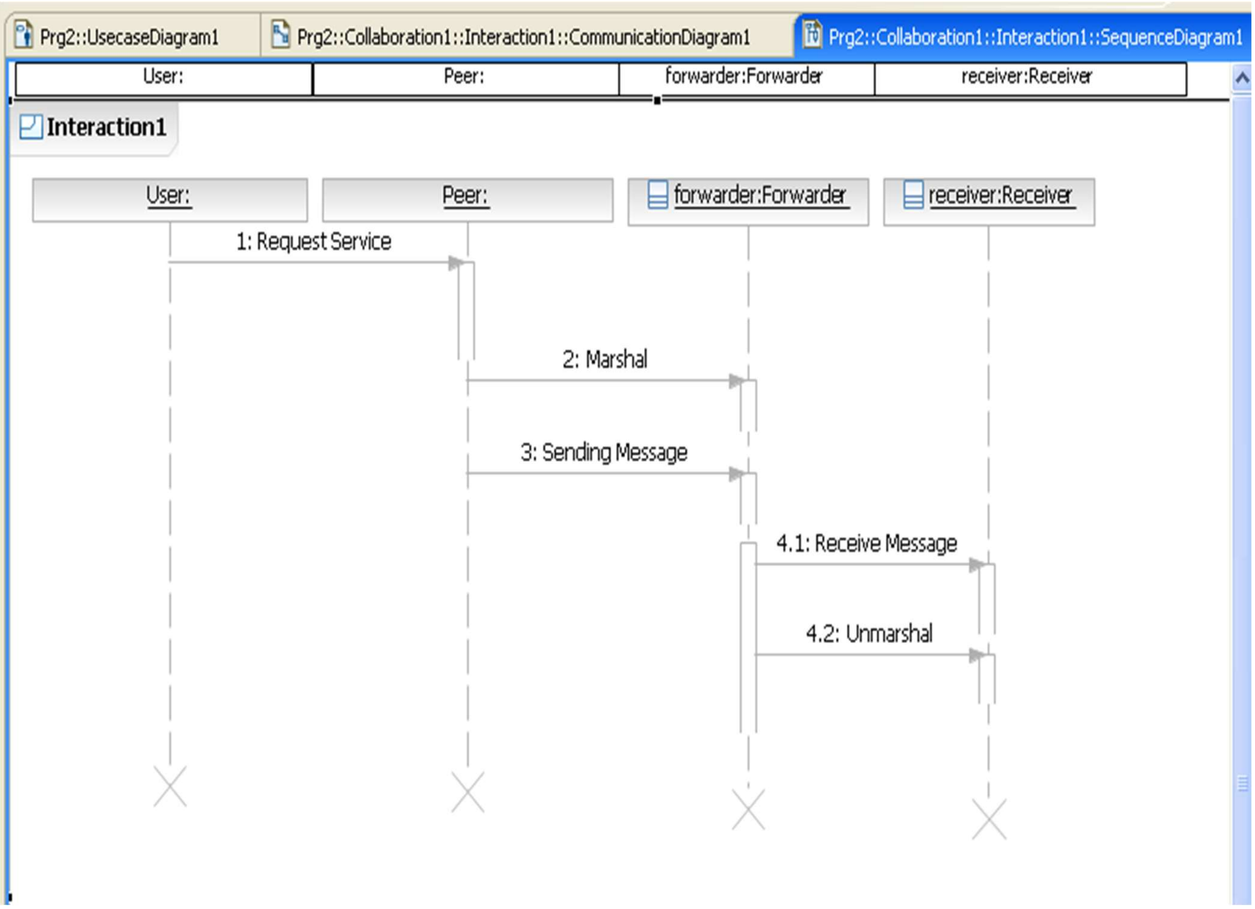
2. Use case Diagram



3. Collaboration Diagram



4. Sequence Diagram



Implementation:

Reciever.java

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

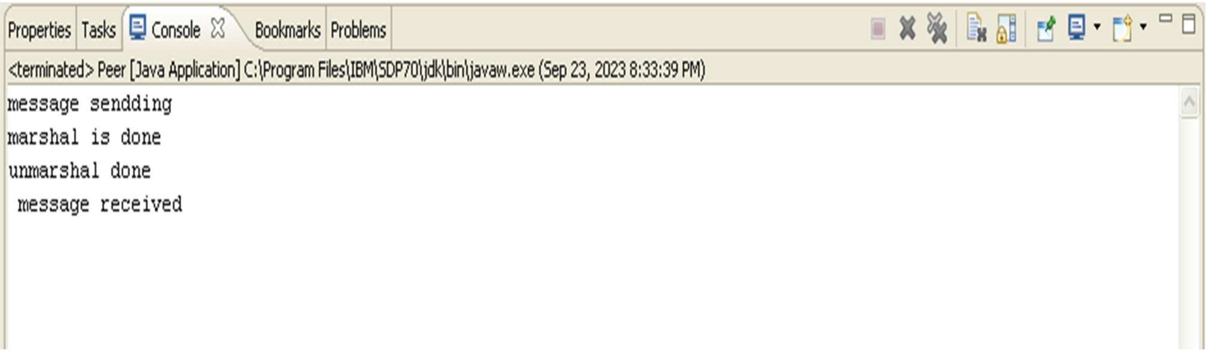
Forwarder.java

```
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



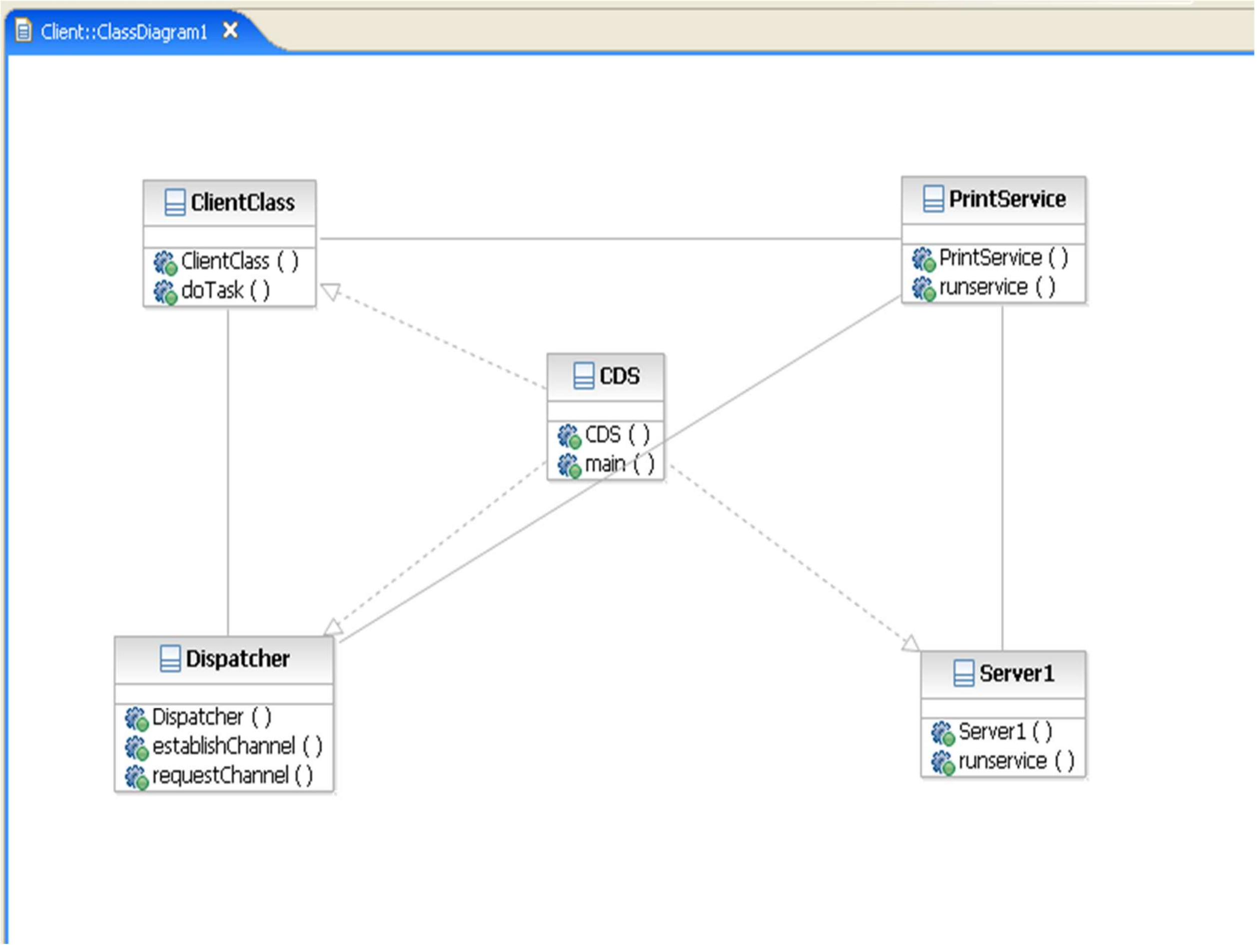
```
<terminated> Peer [Java Application] C:\Program Files\IBM\SDP70\jdk\bin\javaw.exe (Sep 23, 2023 8:33:39 PM)
message sendding
marshal is done
unmarshal done
 message received
```

4. Client dispatcher server pattern

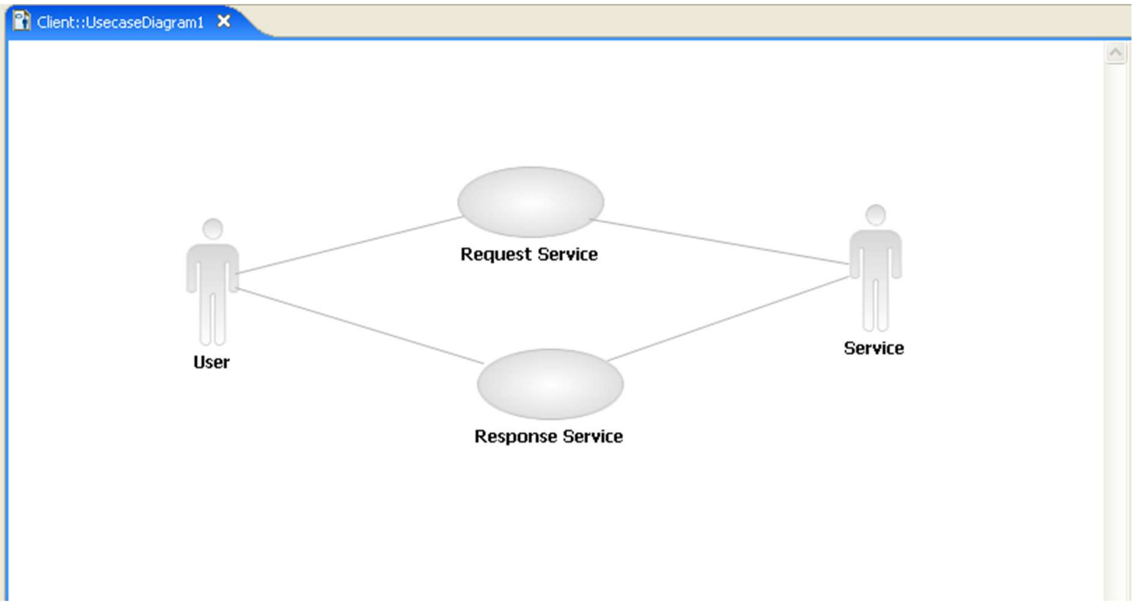
Objectives

- 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.

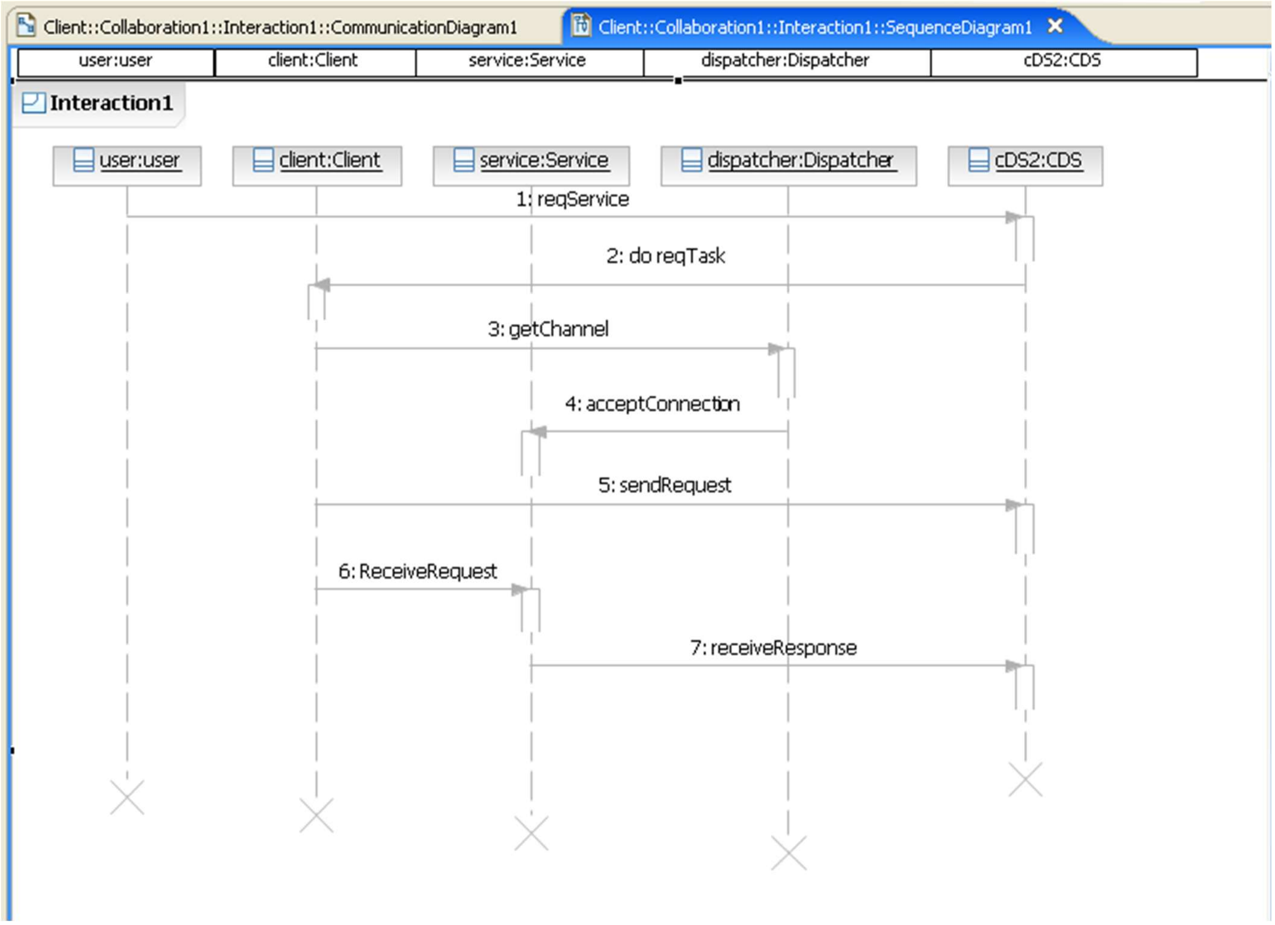
1. Class Diagram



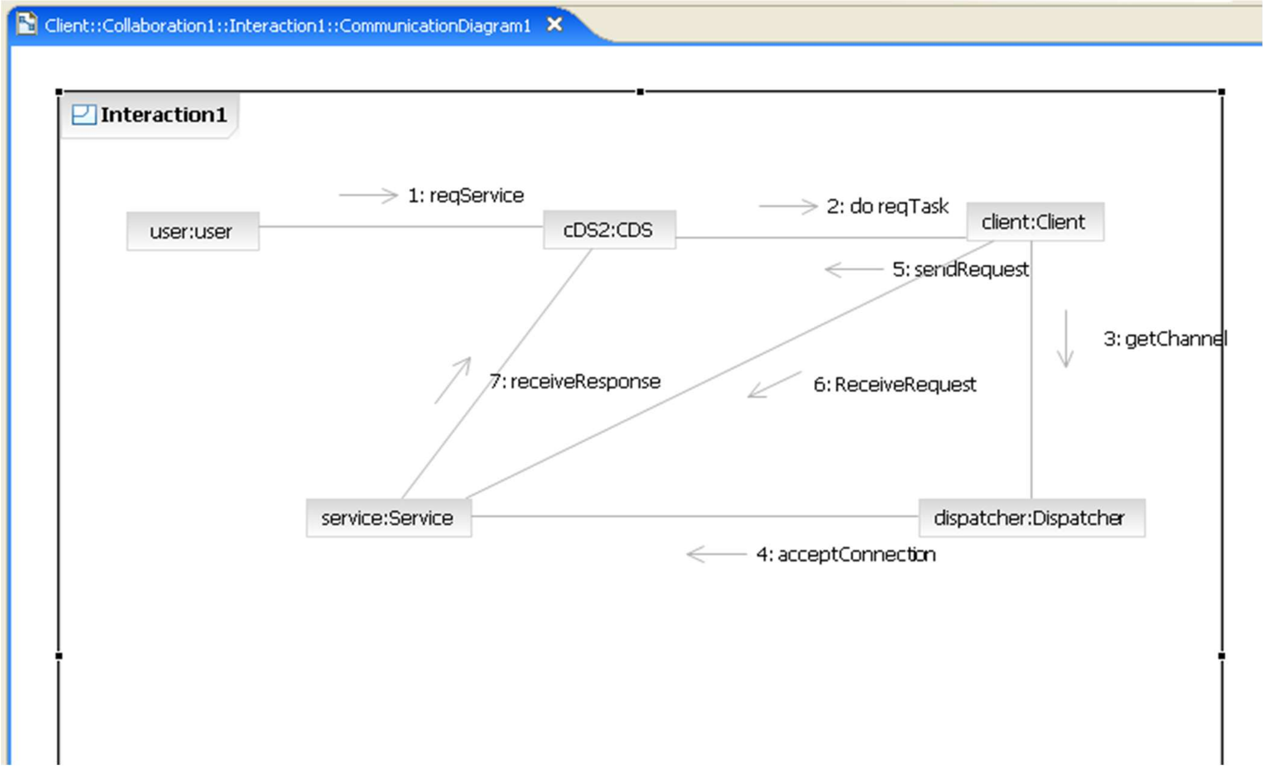
2. Use case Diagram



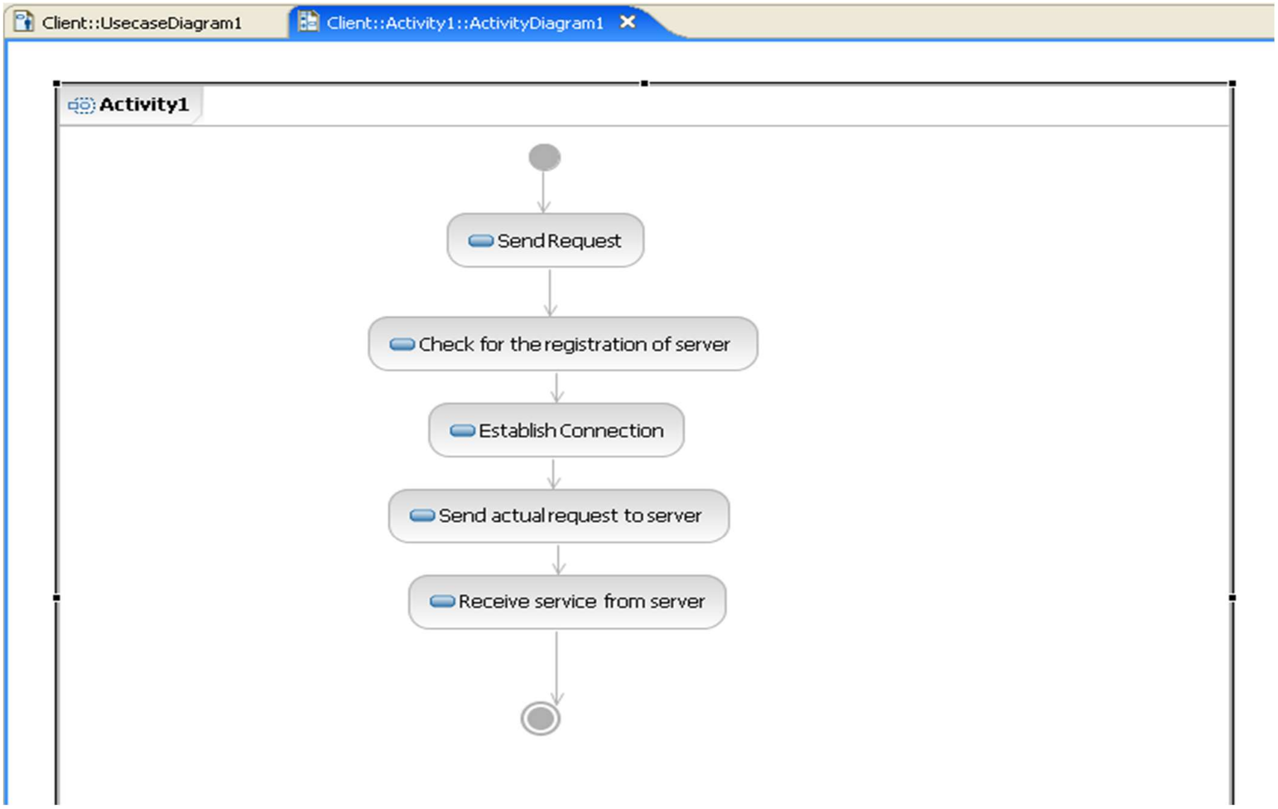
3. Sequence Diagram



4. Collaboration Diagram



5. Activity Diagram


Client::UsecaseDiagram1
Client::Activity1::ActivityDiagram1
Activity1
Send Request
Check for the registration of server
Establish Connection
Send actual request to server
Receive service from server

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

Printserver.java

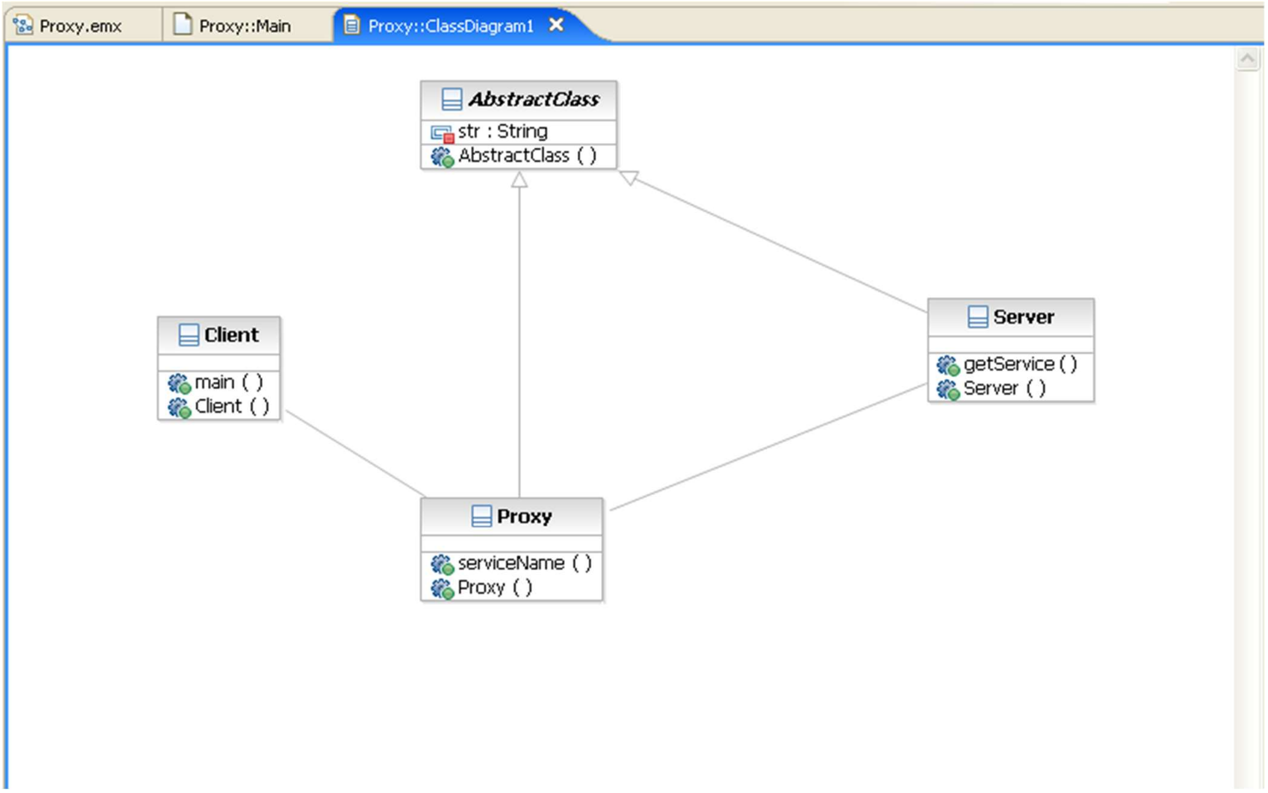
```
import java.io.*;
public class Printserver
{
   private Server1 server1=new Server1();
   private Clientclass client;
   private Dispatcher dis;
   public Printserver()
   {
   }
   public void runservice()
   {
    server1.runservice();
   }
}
```

5. Proxy Pattern

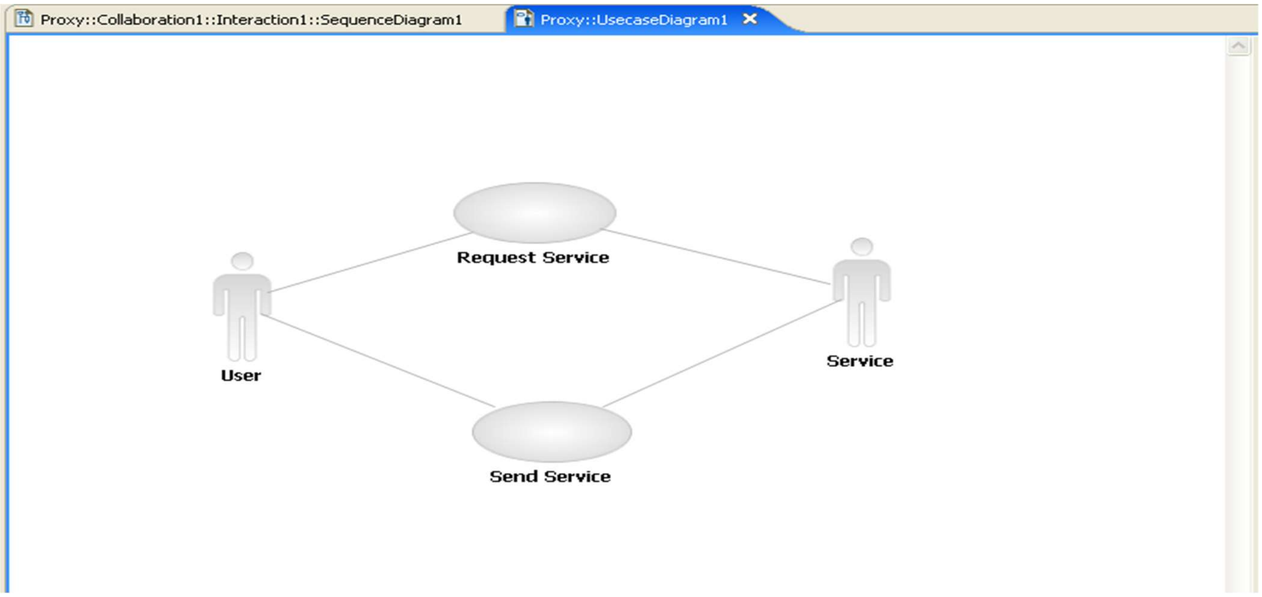
Objective

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

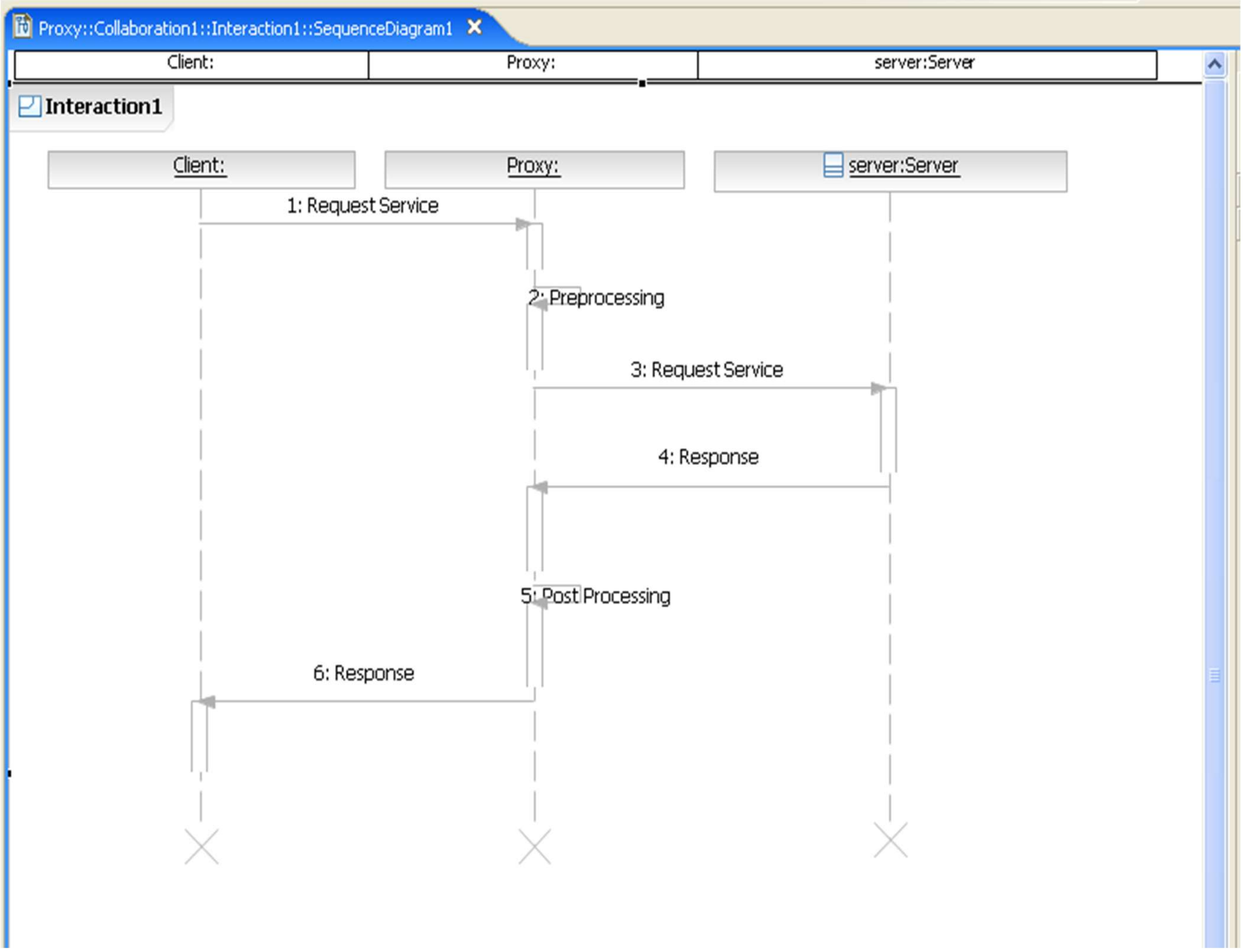
1. Class Diagram



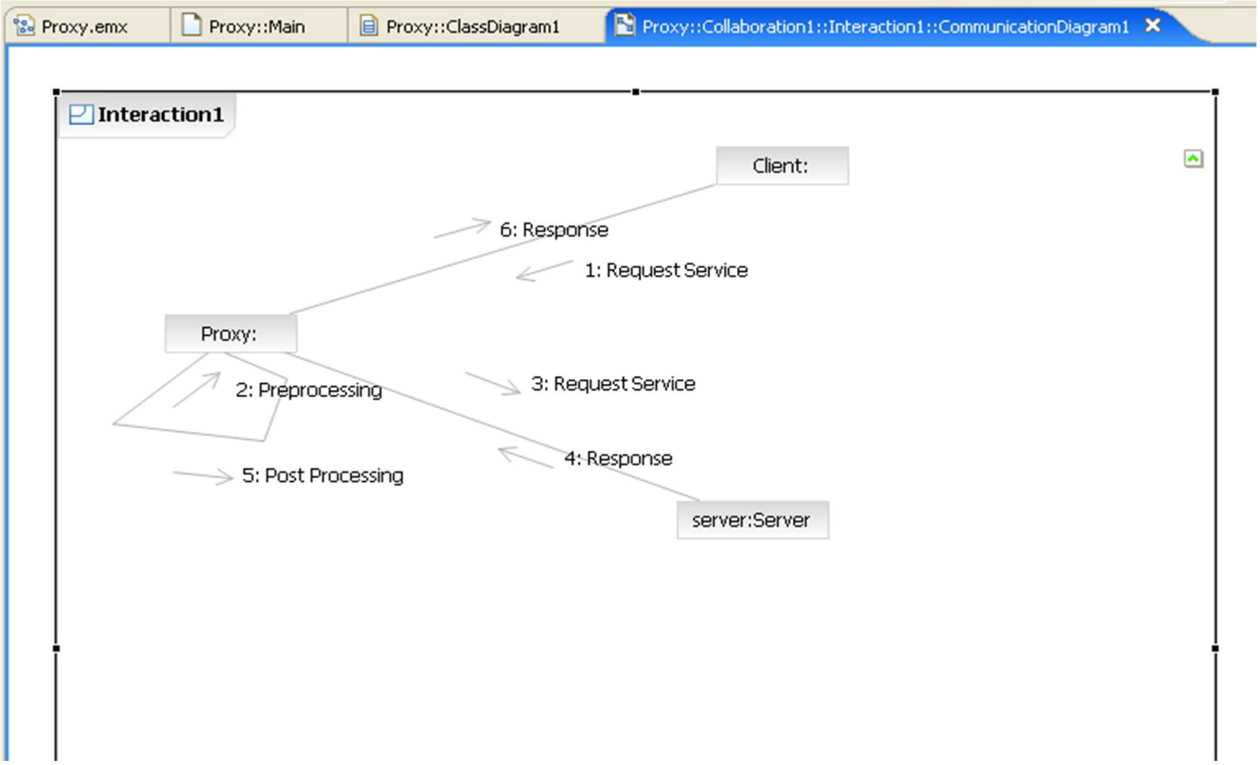
2. Use Case Diagram



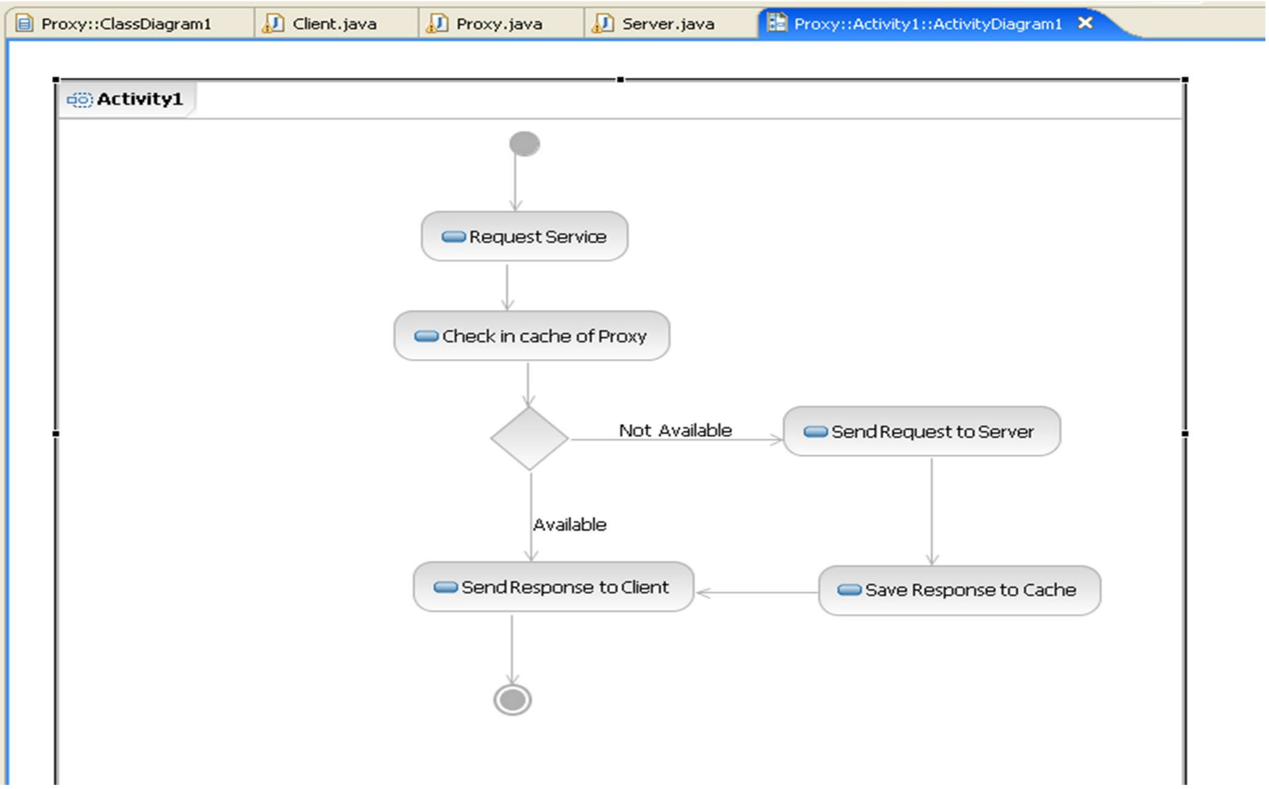
3. Sequence Diagram



4. Collaboration Diagram



5. 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;
}
}
```

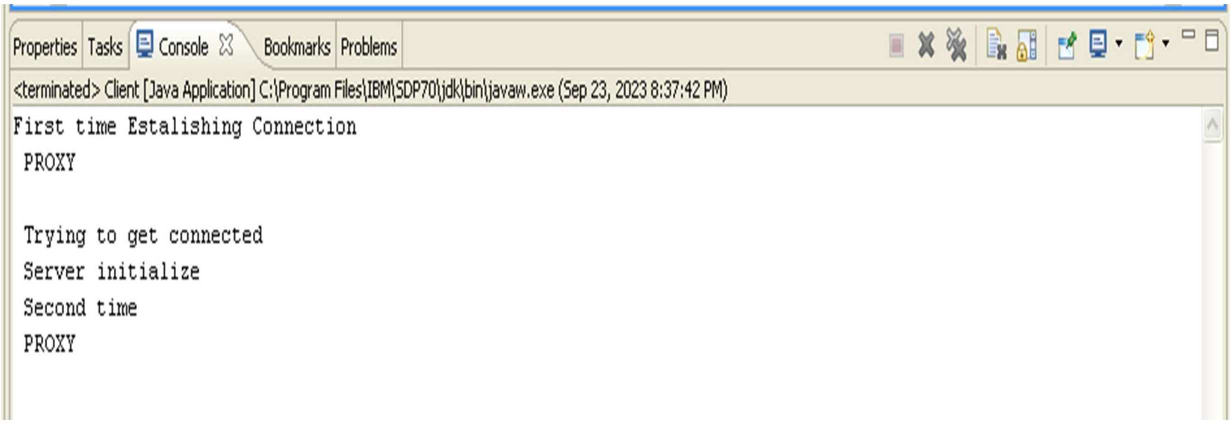
Proxy.java

```
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



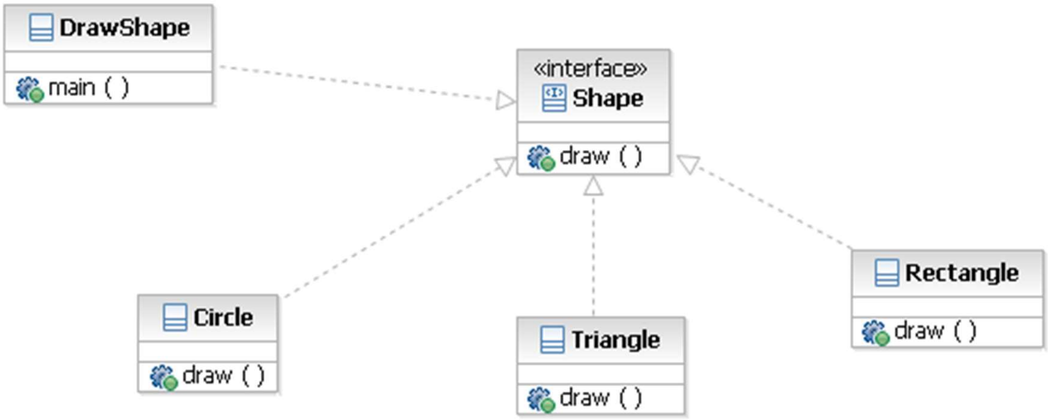
```
<terminated> Client [Java Application] C:\Program Files\IBM\SDP70\jdk\bin\javaw.exe (Sep 23, 2023 8:37:42 PM)
First time Estalishing Connection
 PROXY

 Trying to get connected
 Server initialize
 Second time
 PROXY
```

6.Polymorphism Pattern

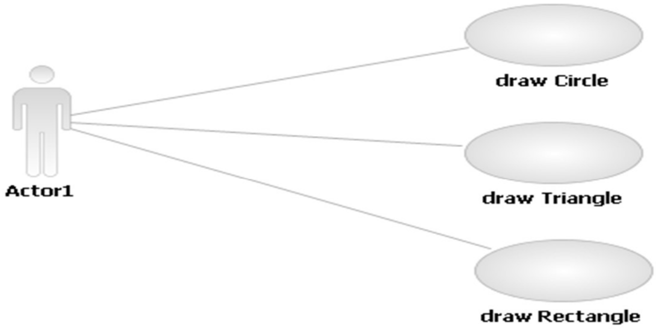
Objective

The objective of using the polymorphism pattern is to create a software design that can accommodate multiple types with varying behaviours in a consistent and extensible manner. This promotes the development of pluggable and adaptable software components, making it easier to manage complexity, support changes, and enhance overall software quality.

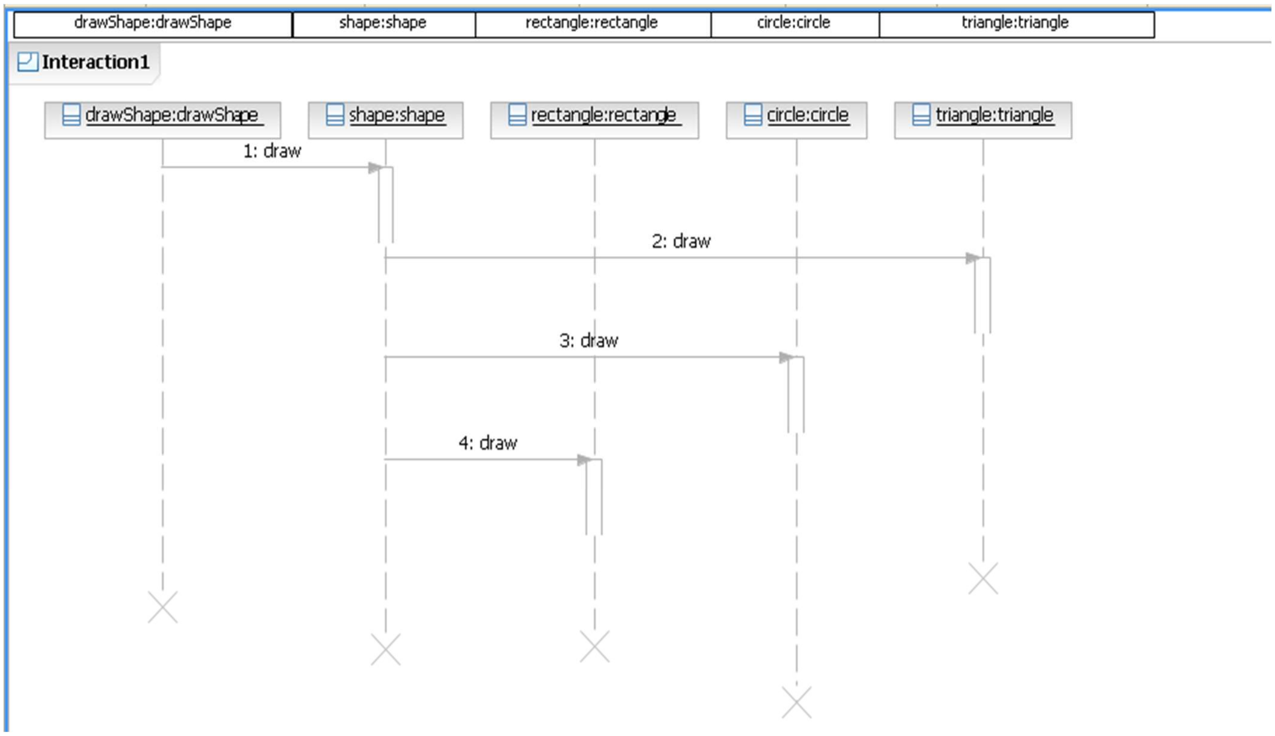
1. **Class Diagram**



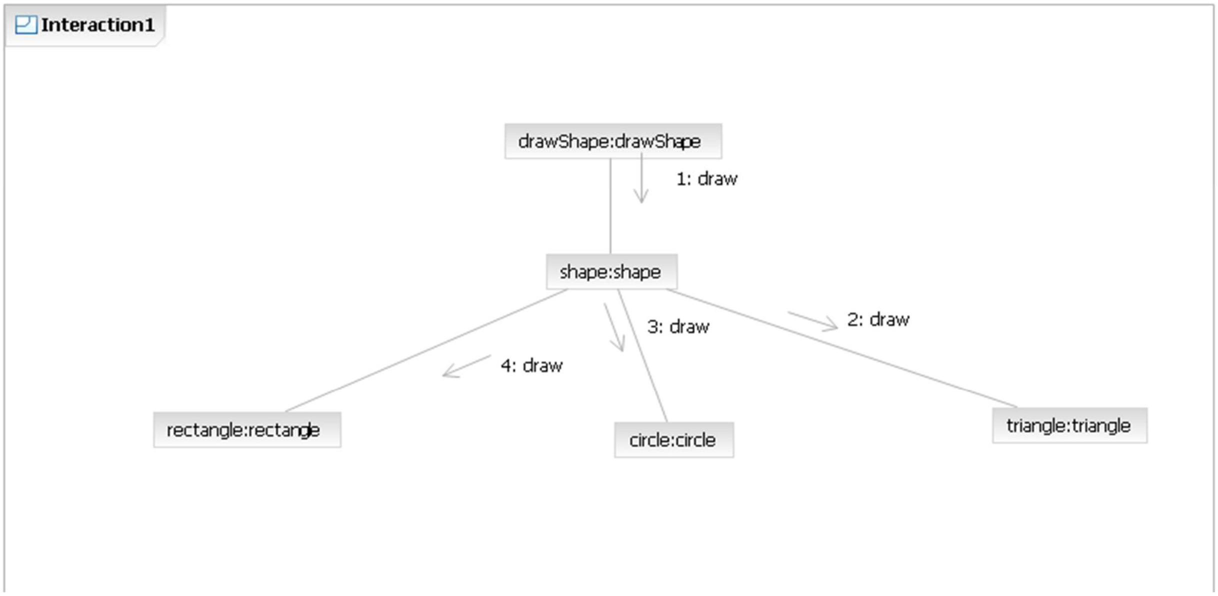
2. **Use Case Diagram**



3. Sequence Diagram



4. Collaboration Diagram



Implementation

Circle.java

```
public class Circle implements Shape
{
         public void draw()
        {
                System.out.println(“Drawing circle");
        }
}
```

Rectangle.java

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

Triangle.java

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

Shape.java

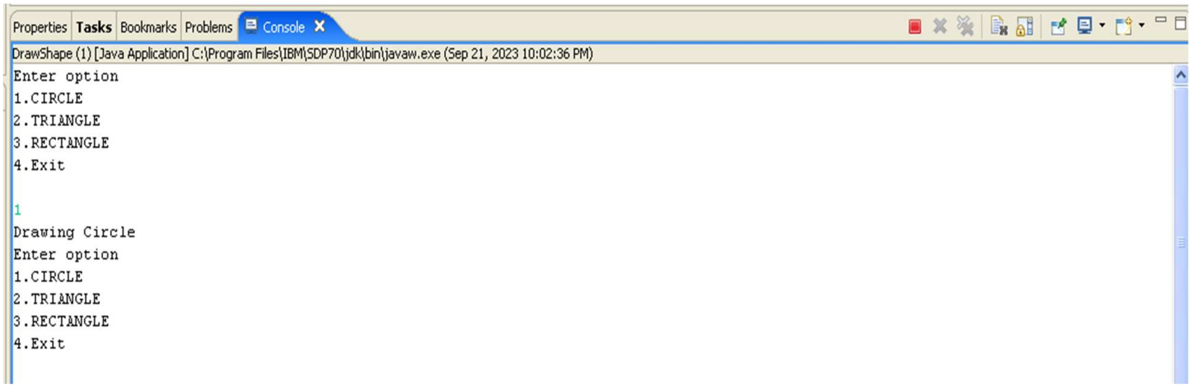
```
public interface Shape
{
        public void draw();
}
```

DrawShape.java

```
import java.util.Scanner;
```

```
public class DrawShape
{
        public static void main(String args[])
        {
                while true {
                        System.out.println("Please enter option draw 1. circle 2. triangle 3. Rectangle 4. Exit");
                        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;
                        case 4: System.exit(0);

                        default: System.out.println("Invalid option");
                        }
                        shape.draw();
                }
        }

}
```

OUTPUT



DrawShape (1) [Java Application] C:\Program Files\IBM\SDP70\jdk\bin\javaw.exe (Sep 21, 2023 10:02:36 PM)

```
Enter option
1.CIRCLE
2.TRIANGLE
3.RECTANGLE
4.Exit

1
Drawing Circle
Enter option
1.CIRCLE
2.TRIANGLE
3.RECTANGLE
4.Exit
```