1. Expert Design Pattern

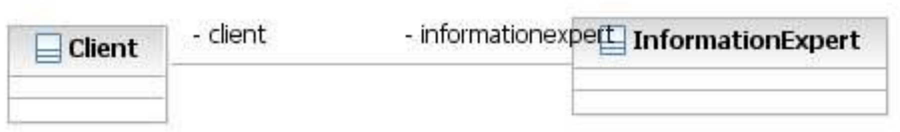
Problem:

What is the most basic principle by which responsibilities are assigned in object-oriented design?

Solution:

Assign a responsibility to the class that has the information necessary to fulfil the responsibility. A Design Model may define hundreds or thousands of software classes, and an application may require hundreds or thousands of responsibilities to be fulfilled. During object design, when the interactions between objects are defined, we make choices about the assignment of responsibilities to software classes. If we've chosen well, systems tend to be easier to understand, maintain, and extend, and our choices afford more opportunity to reuse components in future applications.

Structure:

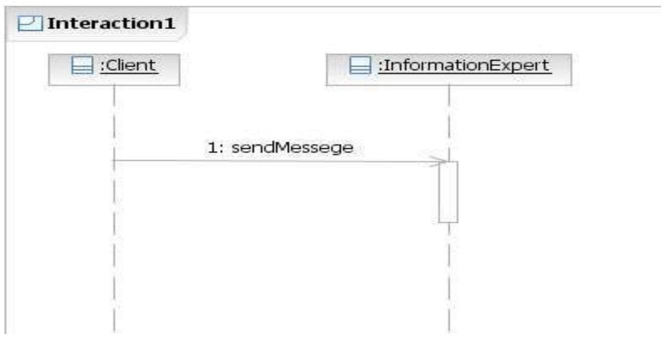


Participants:

Information Expert: This is the class, which has the information to fulfill the responsibility. We assign the responsibility to this class to accomplish the behavior.

Client: The class which will be using the InformationExpert class.

Collaboration:



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.

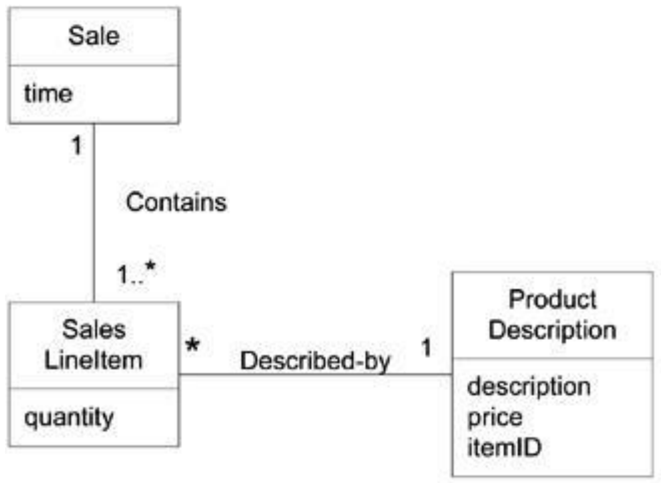
Furthermore, we are creating a commercial POS system that we will sell to different clients with disparate needs in terms of business rule processing. Each client will desire a unique set of logic to execute at certain predictable points in scenarios of using the system, such as when a new sale is initiated 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.

Answer:

Assume we are just starting design work and there is no, or a minimal, Design Model. Therefore, we look to the Domain Model for information experts; perhaps the real-world Sale is one. Then, we add a software class to the Design Model similarly called Sale, and give it the responsibility of knowing its total, expressed with the method named getTotal. This approach supports low representational gap in which the software design of objects appeals to our concepts of how the real domain is organized.

To examine this case in detail, consider the partial Domain Model in following diagram.



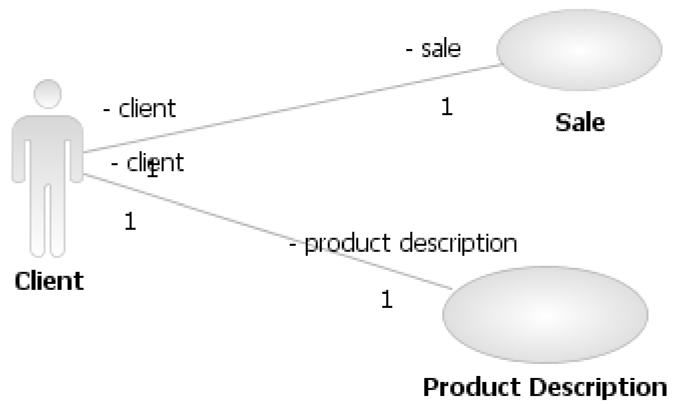
What information do we need to determine the grand total? We need to know about all the SalesLineItem instances of a sale and the sum of their subtotals. A Sale instance contains these; therefore, by the guideline of Information Expert, Sale is a suitable class of object for this responsibility; it is an information expert for the work.

To fulfill the responsibility of knowing and answering its subtotal, a SalesLineItem has to know the product price.
The ProductDescription is an information expert on answering its price; therefore, SalesLineItem sends it a message asking for the product price.

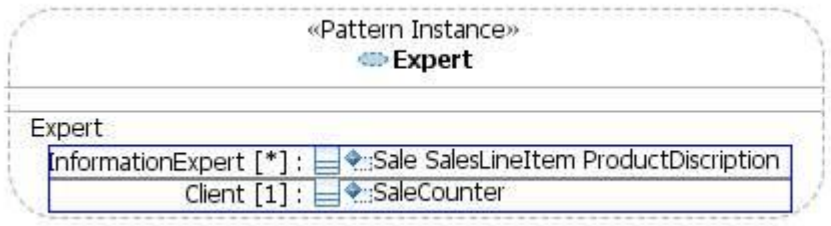
In conclusion, to fulfill the responsibility of knowing and answering the sale's total, we assigned three responsibilities to three design classes of objects as follows.

Design Class	Responsibility
Sale	knows sale total
SalesLineItem	knows line item subtotal
ProductDescription	knows product price

Usecase Diagram:

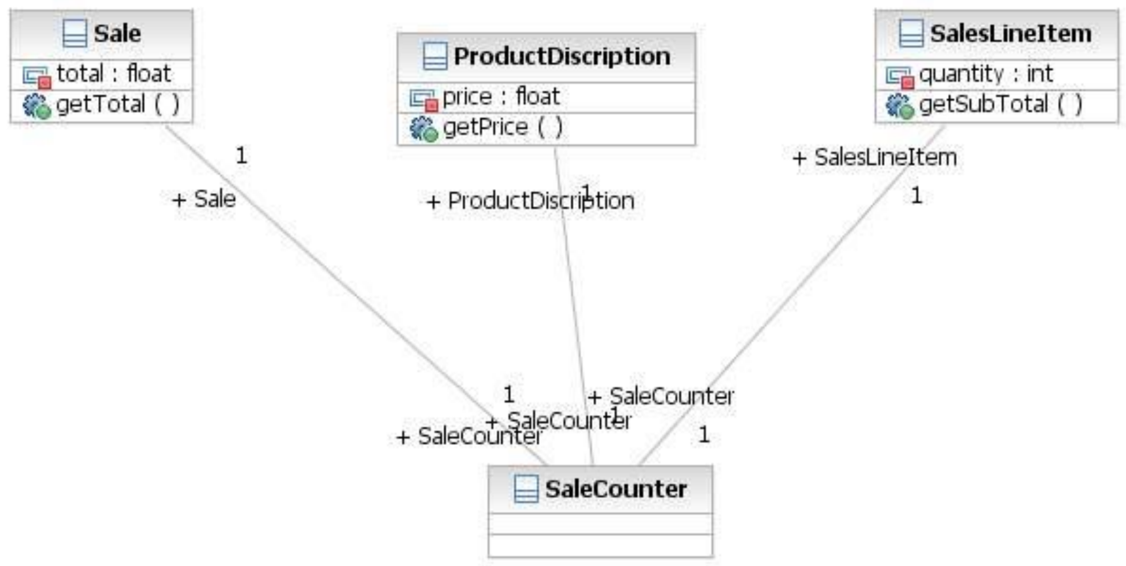


Pattern instance:

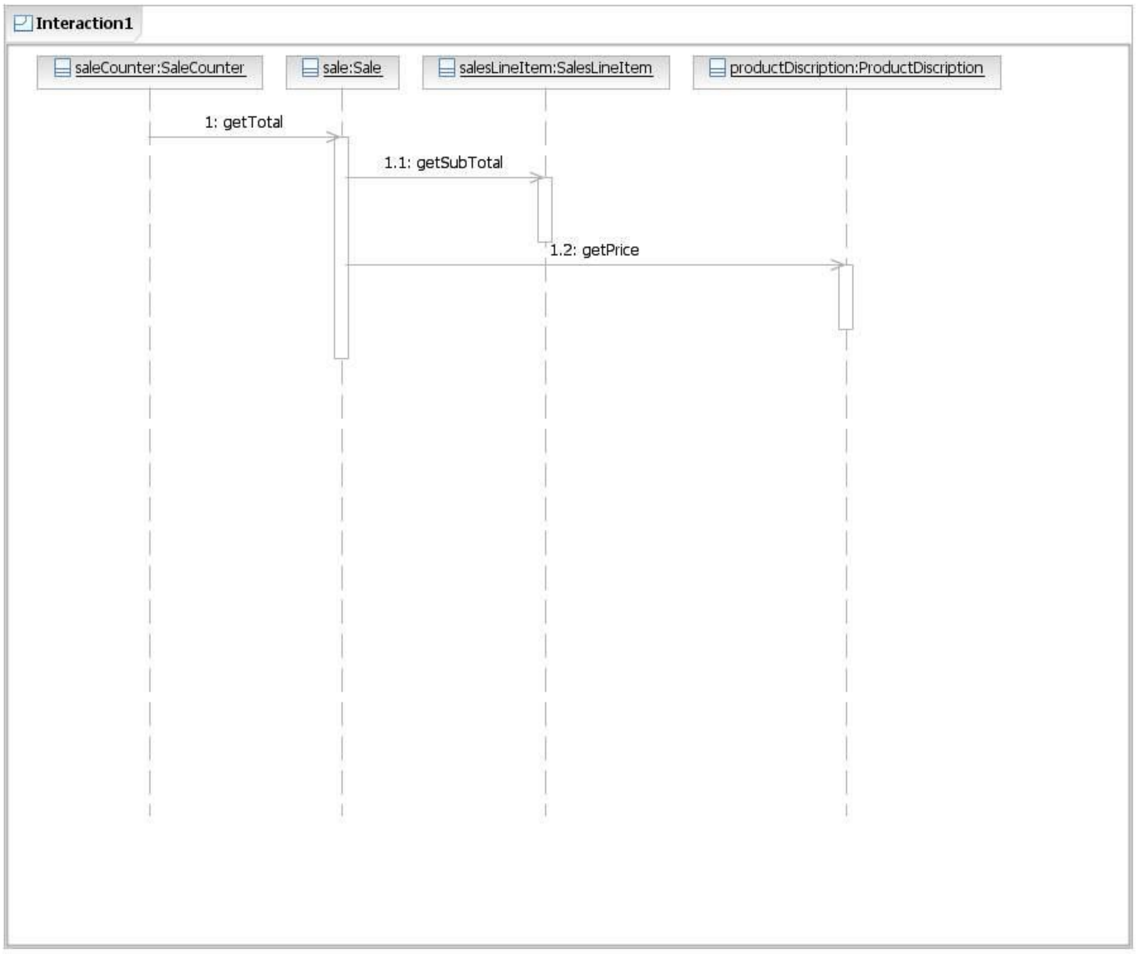


InformationExpert [*] : ::Sale SalesLineItem ProductDiscription
Client [1] : ::SaleCounter

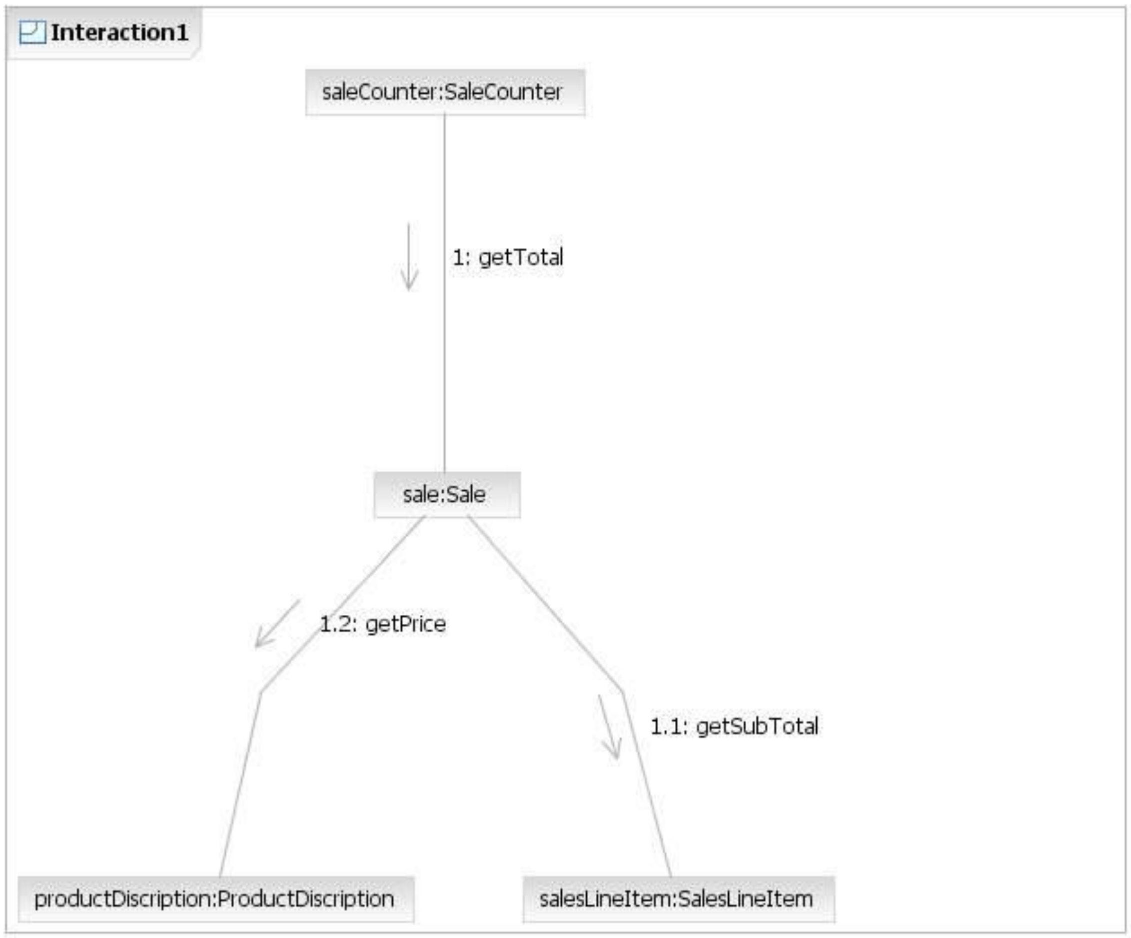
Class diagram:



Sequence Diagram:



Communication diagram:



Java Code:

SaleCounter.java:

```
public class SaleCounter
{
        public Sale Sale;
        public SalesLineItem SalesLineItem;
        public ProductDiscription ProductDiscription;
        public static void main(String []args
        {
                float total;
                Sale s=new Sale();
                total=s.getTotal();
                System.out.println(" Total sale is :" + total);
        }
}
```

Sale.java:

```
public class Sale
{
        public SaleCounter SaleCounter;
        private float total;
        public float getTotal()
        {
                ProductDiscription pd=new ProductDiscription();
                SalesLineItem sli=new SalesLineItem();
                total = (pd.getPrice()* sli.getSubTotal());
                return total;
        }
}
```

SalesLineItem.java:

```
public class SalesLineItem
{
        public SaleCounter SaleCounter;
        private int quantity=100;
        public int getSubTotal()
        {
                return quantity;
        }
}
```

ProductDiscription.java:

```
public class ProductDiscription
{
        public SaleCounter SaleCounter;
-       private float price=43.89f;
        public float getPrice()
        {
                return price;
        }
}
```

Output:

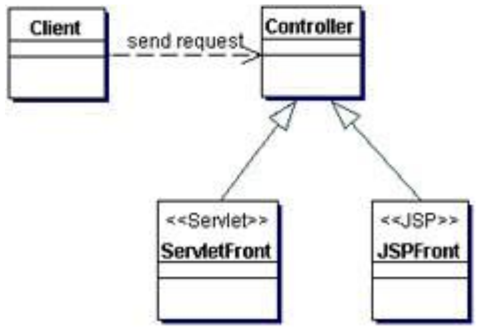
```
Total sale is : 4389.0
```

2. Controller Design Pattern

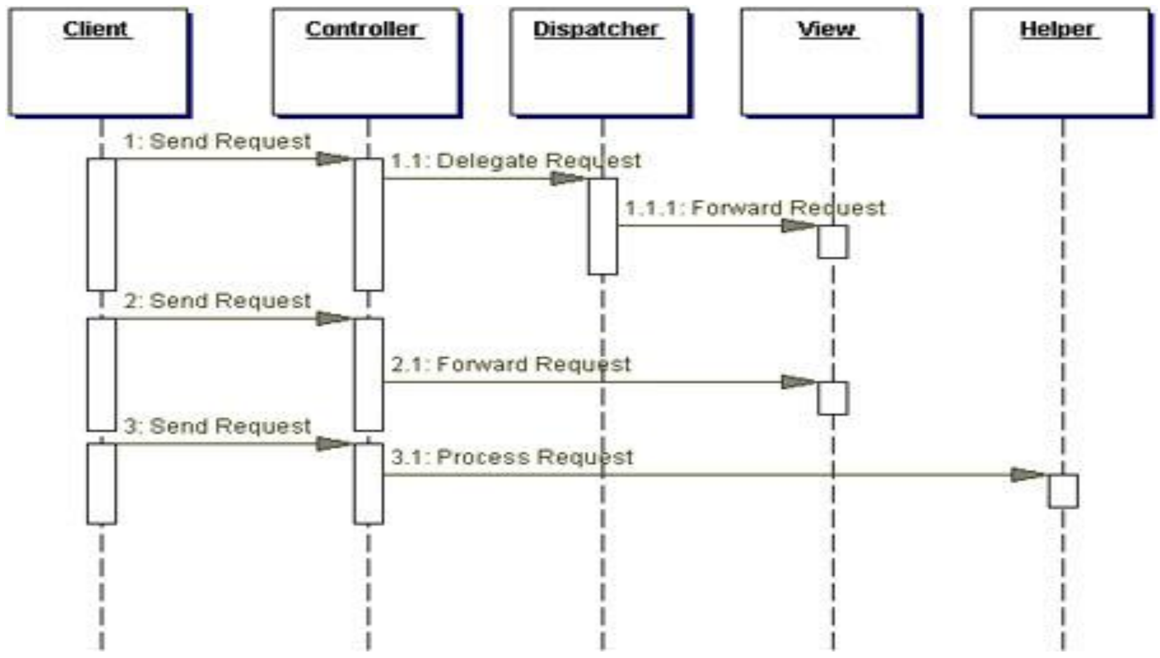
1. Context / Problem:
The presentation-tier request handling mechanism must control and coordinate processing of each user across multiple requests. Such control mechanisms may be managed in either a centralized or decentralized manner.

2. 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.

3. Structure:



4. Participants and Responsibilities:
The sequence diagram representing the Front Controller pattern. It depicts how the controller handles a request.



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.

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.

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

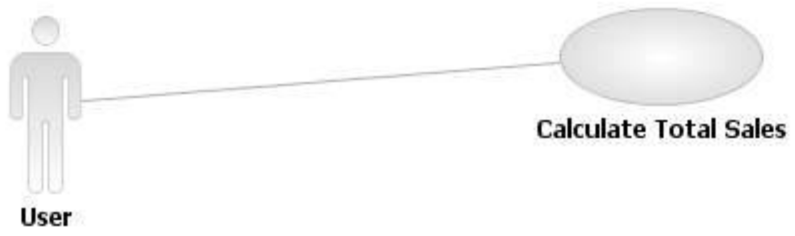
Example:
A Point Of Sale (POS) system is a computerized application used (in part) to record sales and handle payments; it is typically used in a retail store. It includes hardware components such as a computer and bar code scanner, and software to run the system. It interfaces to various service applications,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

Now from information expert pattern we know that Sale class will be calculating the total sale at any given point of time.

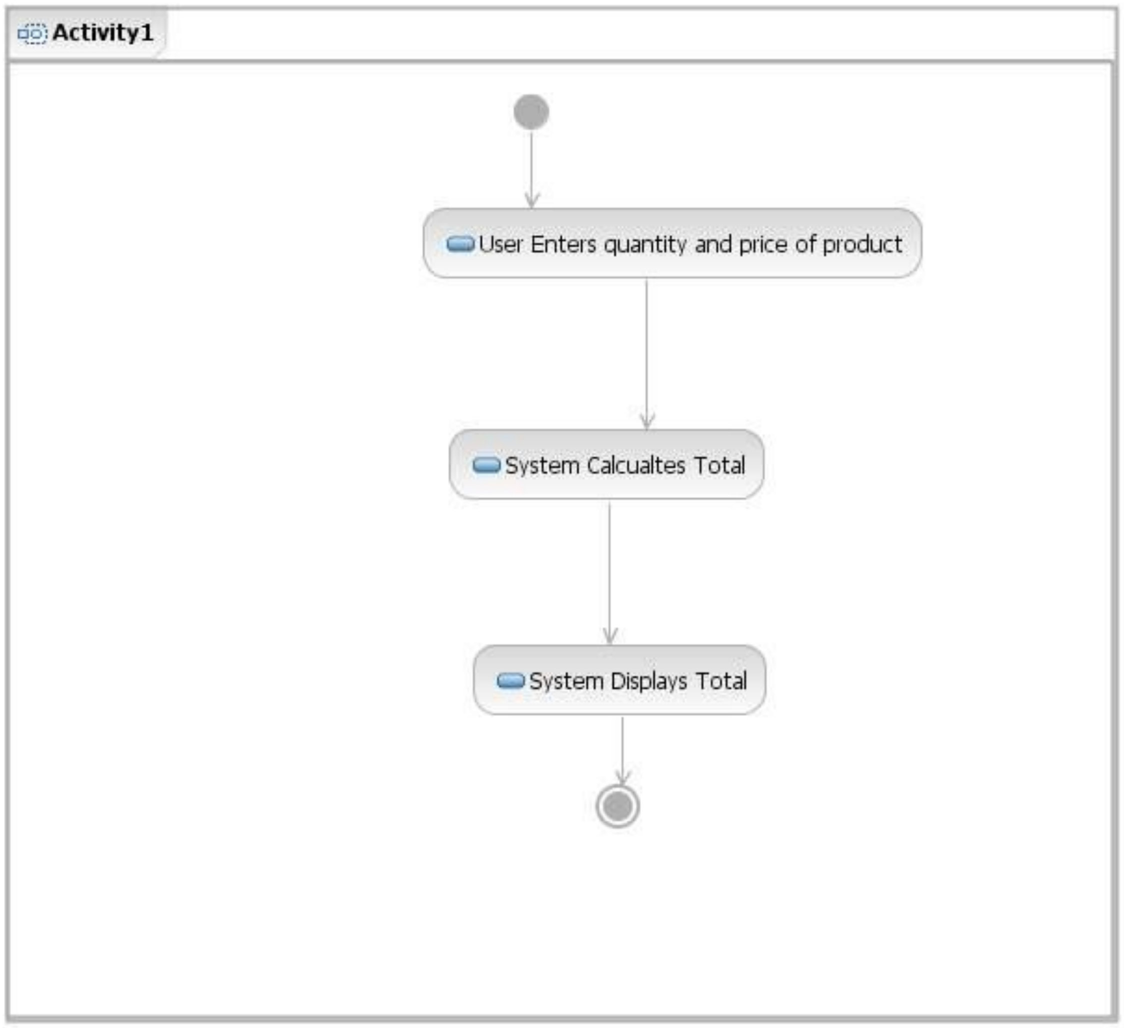
Now the Controller Pattern suggests, that events coming from UI layer should not be directly accessing the expert classes. There should be a Controller class to control these events.

In this example, we will use a SaleController class that will be receiving the events from UI and forward it to the Sale class.

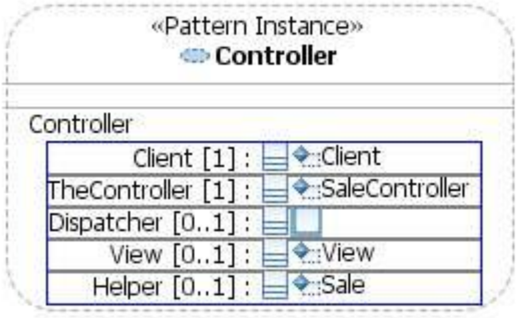
Use case diagram:



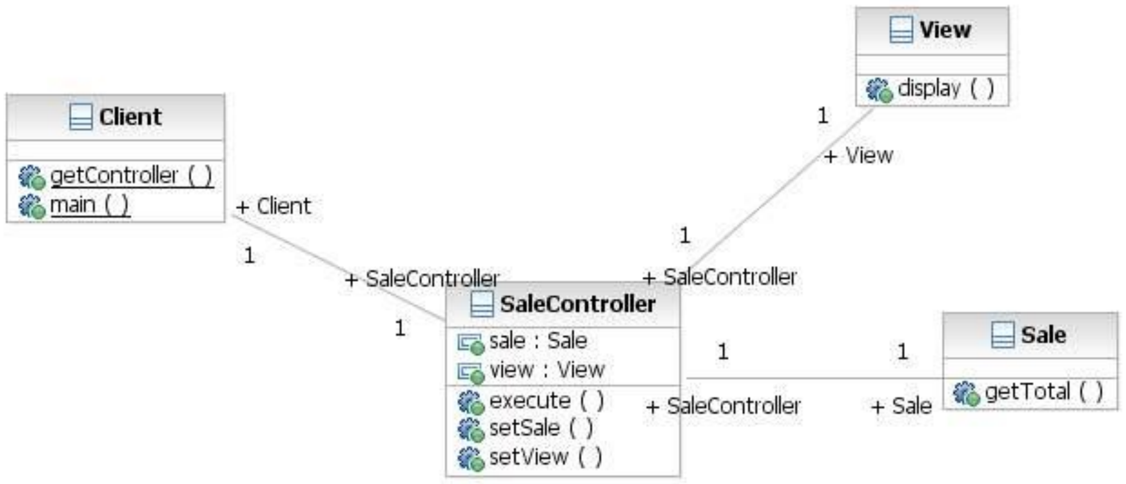
Activity Diagram:



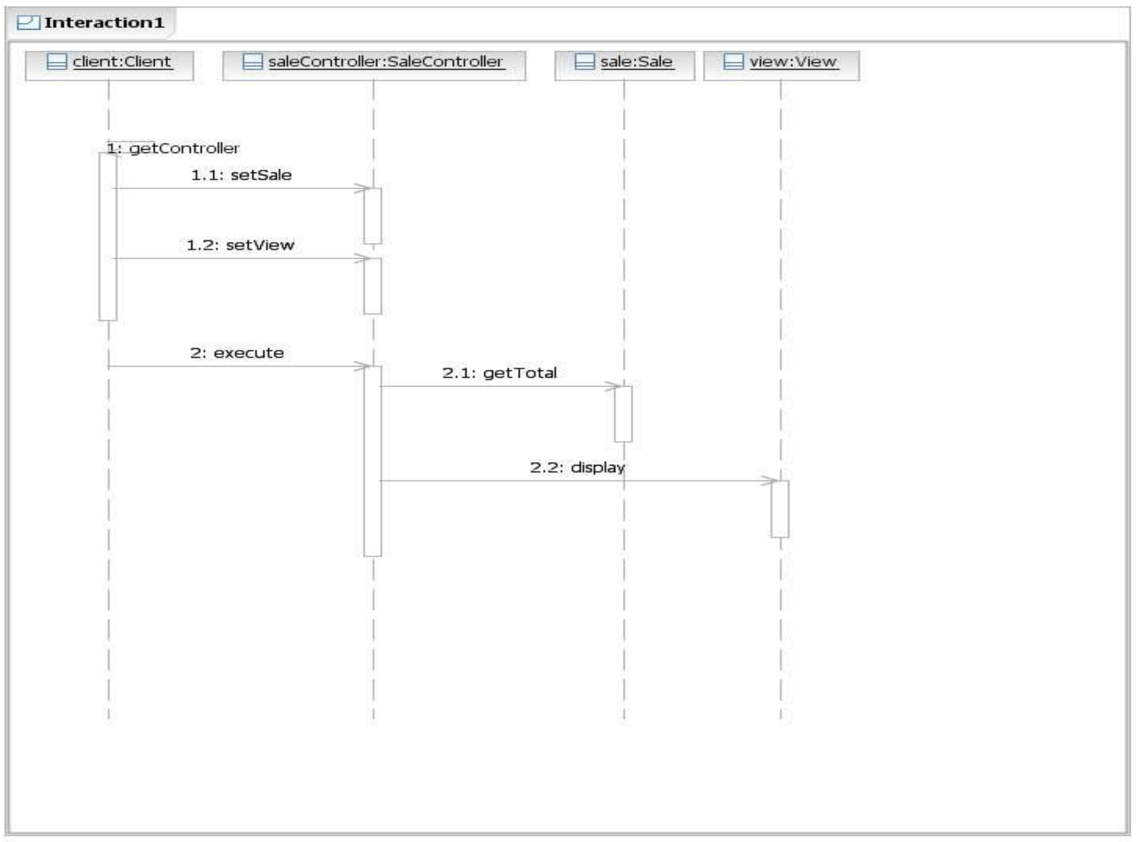
Pattern Instance:



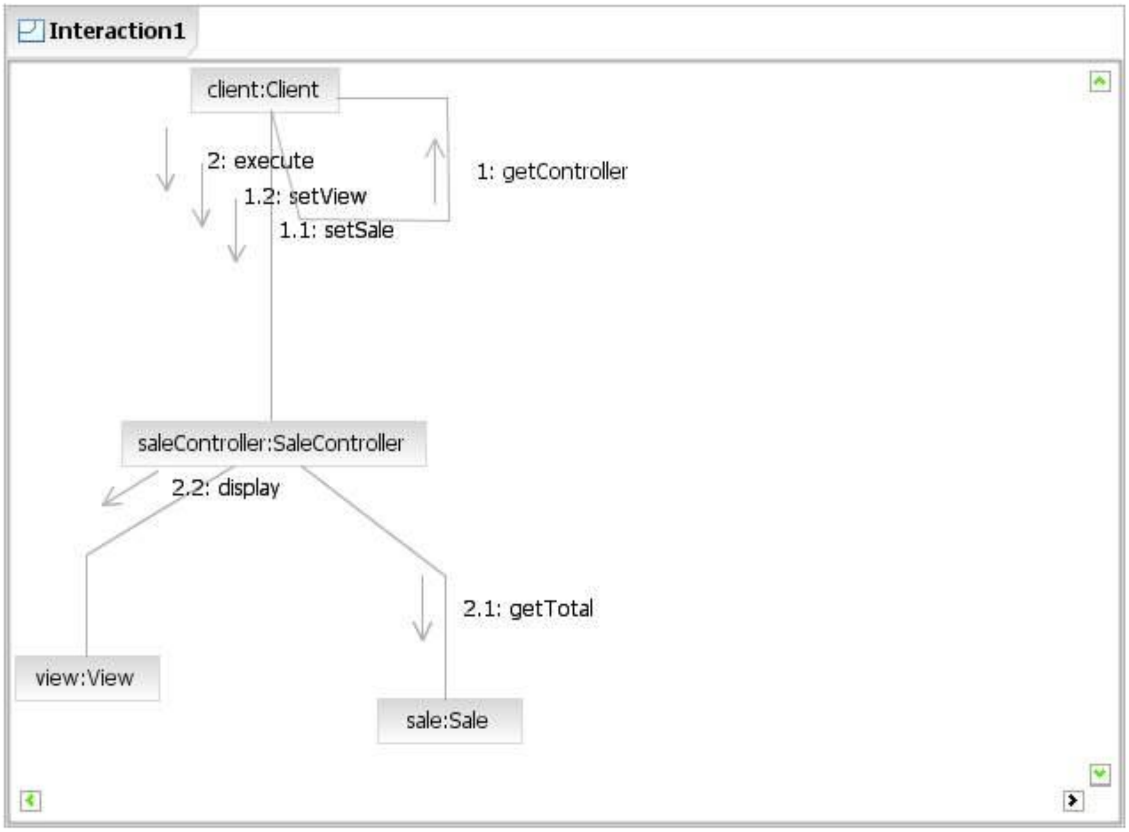
Class Diagram:



Sequence Diagram:



Communication Diagram:



Java Code:

Sale.java:

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

View.java:

```
public class View
{
        public SaleController SaleController;
        public void display(float total)
        {
                System.out.println("The Total Sale is: " +  total);
        }
}
```

SaleController.java:

```
public class SaleController
{
        public Client Client;
        public Sale sale;
        public View view;
        public void execute(int quantity,float price)
        {
                float result = sale.getTotal(quantity, price);
                view.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 Qunatity: ");
		int quantity= scr.nextInt();
		System.out.println("Enter the Price: ");
		float price=scr.nextFloat();
		tc.execute(quantity, price);
	}
}
```

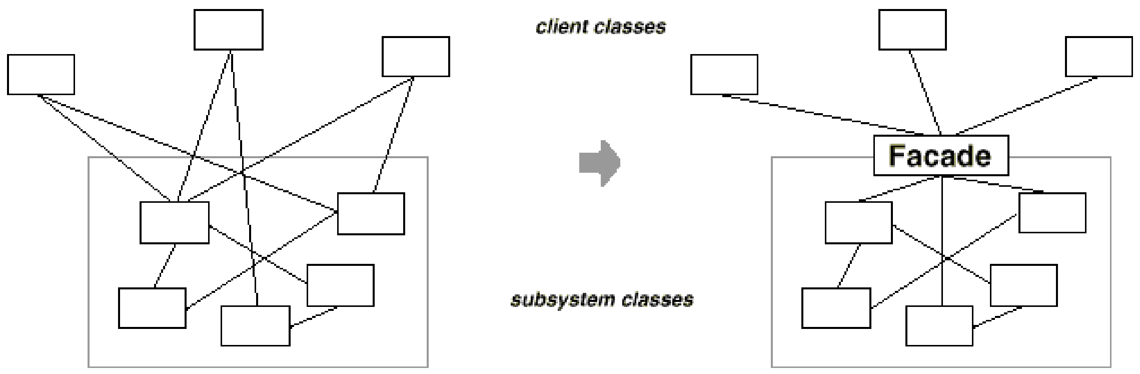
Output:-

```
Enter the quantity: 20
Enter the Price: 10
The Total Sale is: 200
```

3. Façade Design Pattern

<u>Intent</u>: The Façade Pattern provides a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

<u>Motivation</u>:
1) Structuring a system into subsystems helps reduce complexity
2) Subsystems are groups of classes, or groups of classes and other subsystems
3) The interface exposed by the classes in a subsystem or set of subsystems can become quite complex
4) One way to reduce this complexity is to introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem



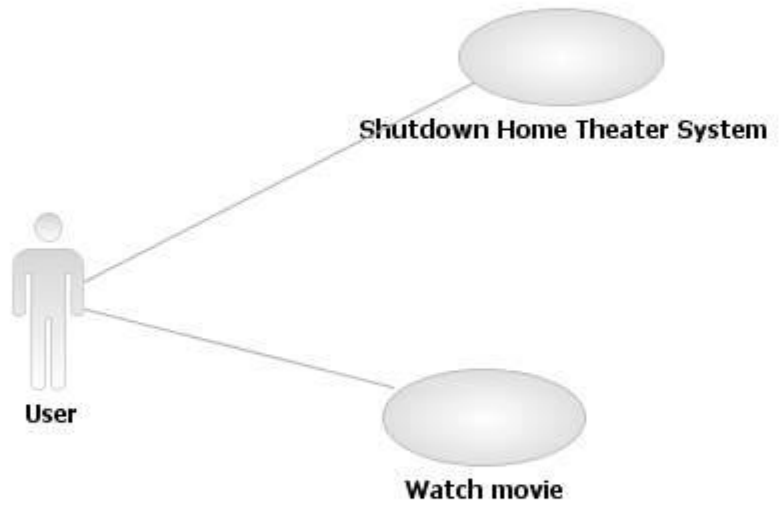
<u>Applicability</u>:
Use the Facade pattern:
1) To provide a simple interface to a complex subsystem. This interface is good enough for most clients; more sophisticated clients can look beyond the facade.
2) To decouple the classes of the subsystem from its clients and other subsystems, thereby promoting subsystem independence and portability

<u>Example:</u>
You have to design a Home Theater System. The home theater system will consist of different components like Projector, to project the movie on screen, DVD Player to play the movie, Amplifier to control the sound effects.

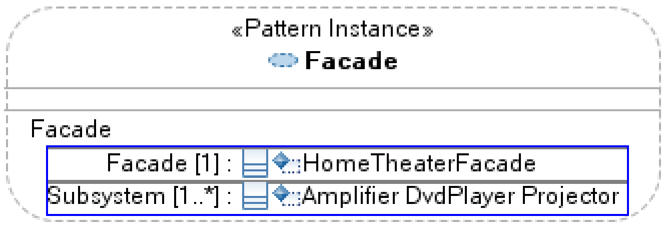
For watching a movie the user starts switching on each component one by one and enjoys the movie. After watching the movie he/she will have to switch off each device.

But now the user is fed up of this process and don’t want to start switching on each device by himself. He wants to control all the devices using single remote so that when he wants to watch a movie he can use this remote to start watching movie and in the end he can close all the devices by one command using same remote.

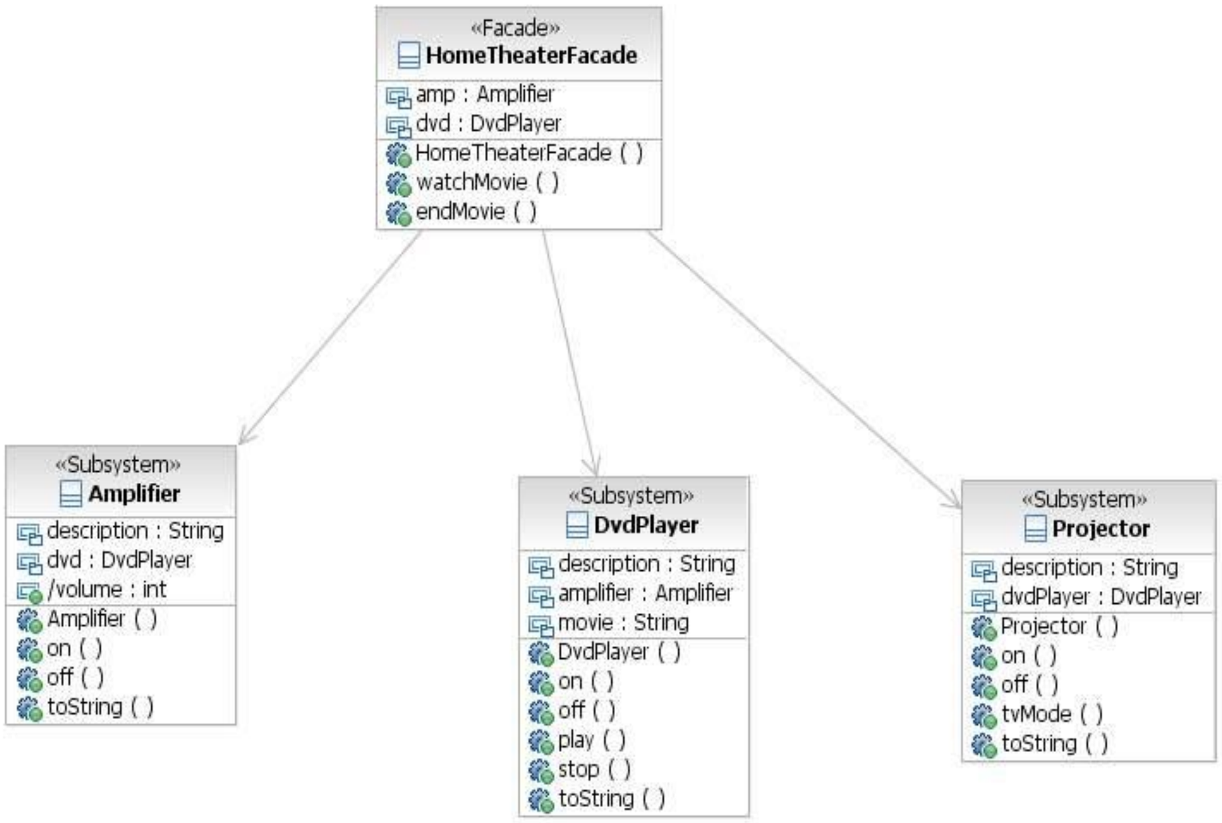
Usecase Diagram:



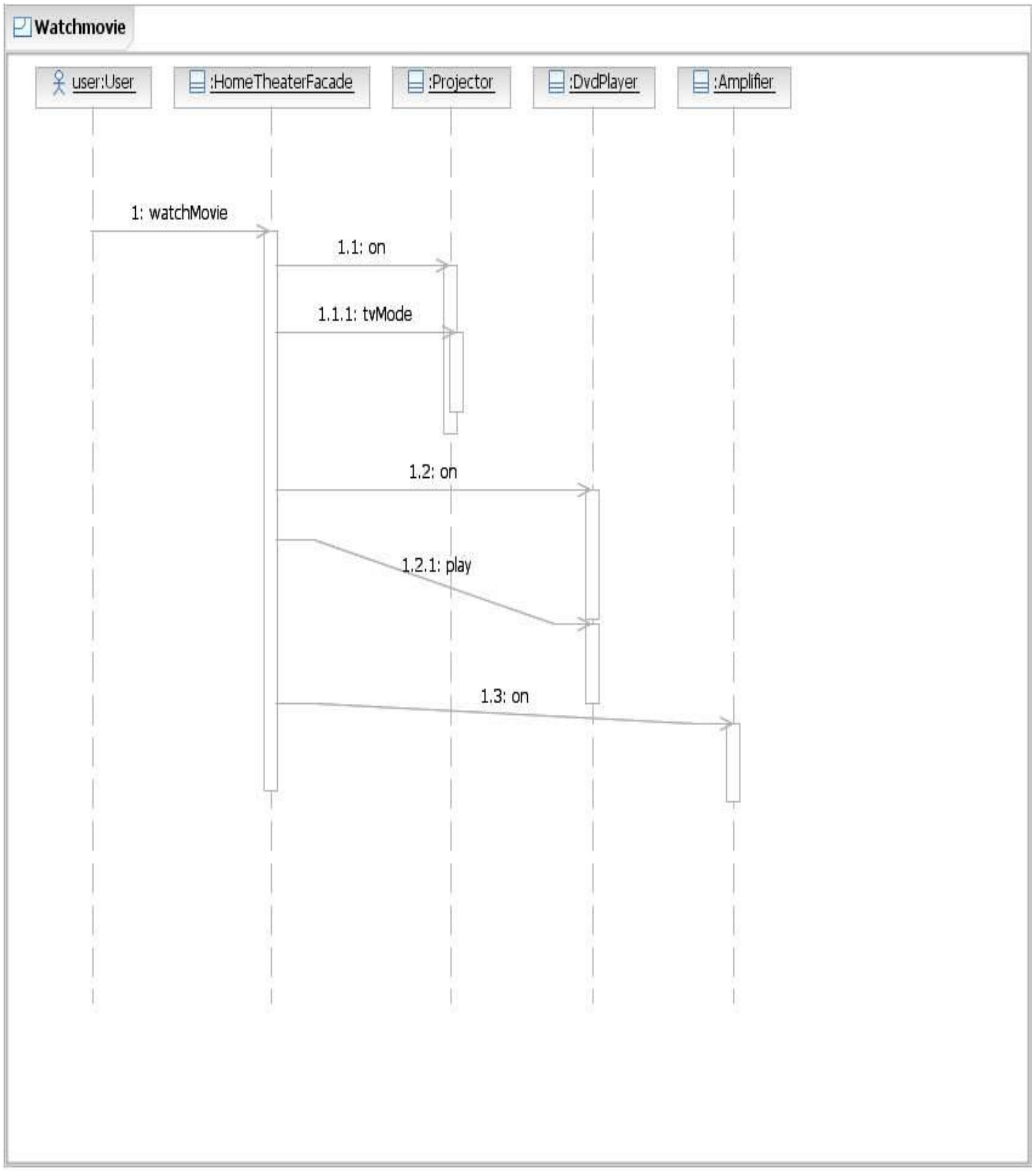
Pattern Instance:


«Pattern Instance»
Facade
Facade
Facade [1] : HomeTheaterFacade
Subsystem [1..*] : Amplifier DvdPlayer Projector

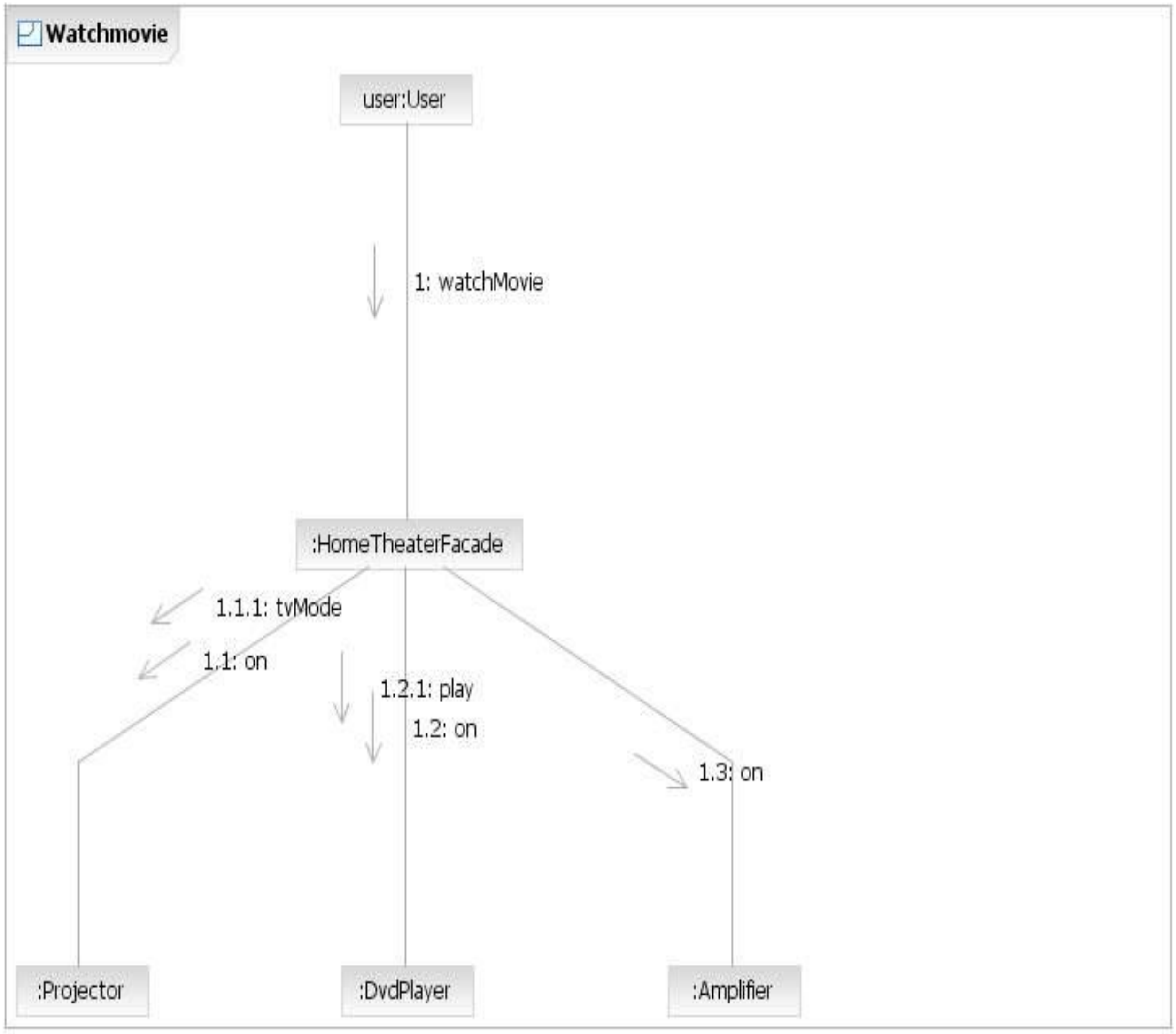
Class Diagram:



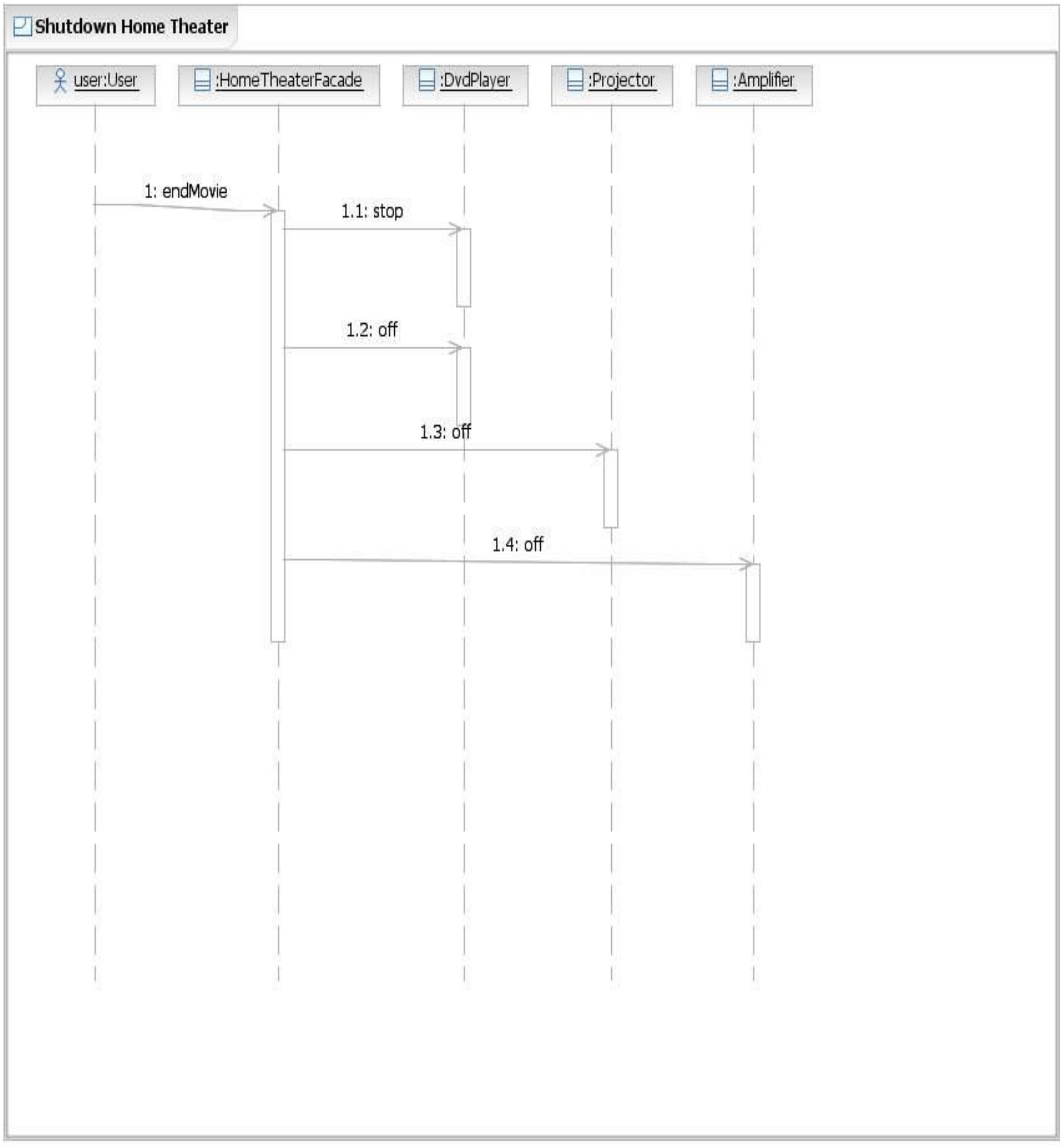
Sequence Diagram for Watch movie:



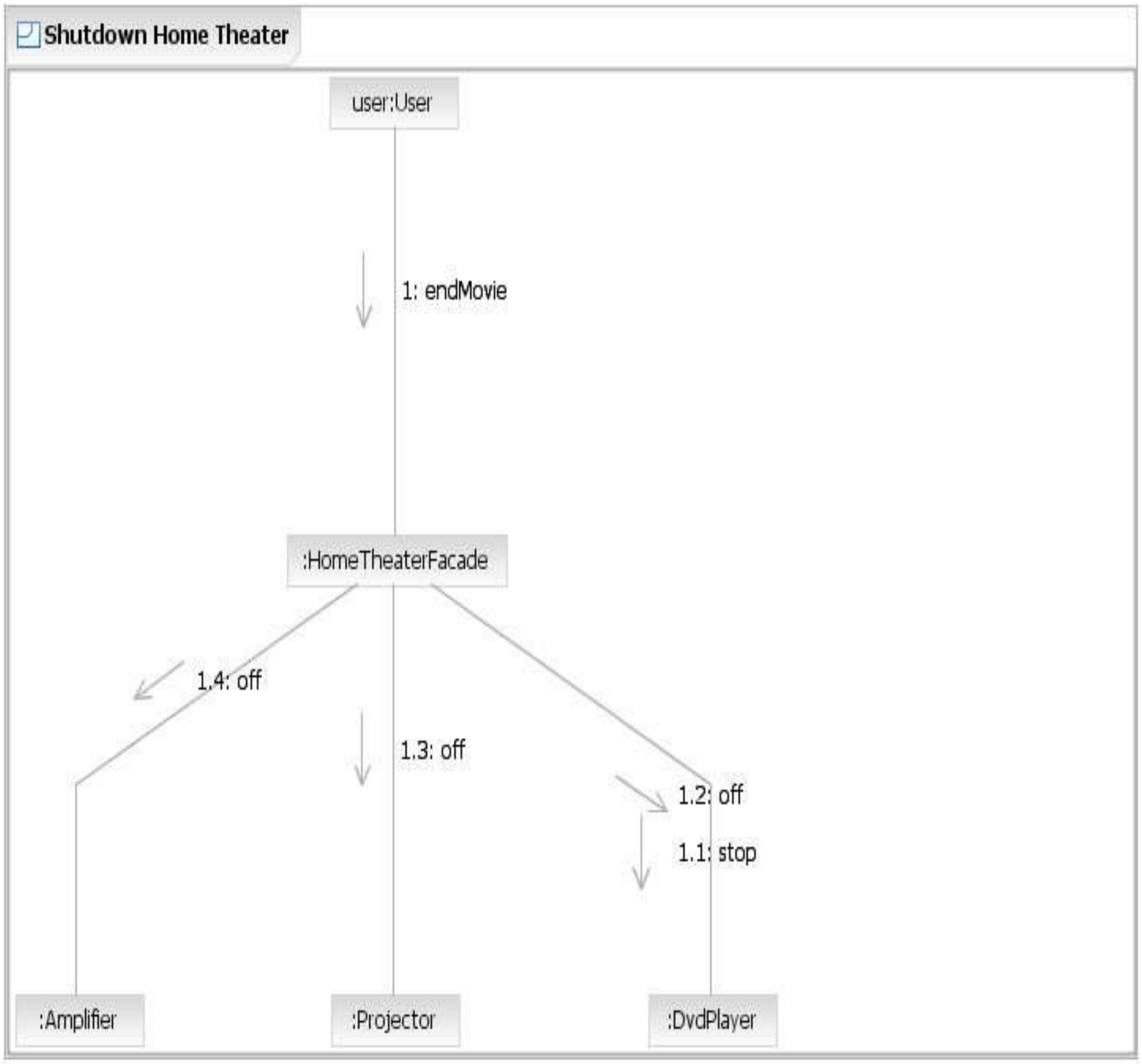
Communication Diagram for Watch Movie:



Sequence diagram for Shutdown home Theater System:



Communication diagram for Shutdown home Theater System:



Java Code :

Amplifier.java:

```
public class Amplifier
{
	String description;
	DvdPlayer dvd;
	public Amplifier(String description)
	{
		this.description = description;
	}
	public void on()
	{
		System.out.println(description + " on");
	}
	public void off()
	{
		System.out.println(description + " off");
	}
	public String toString()
	{
		return description;
	}
}
```

DvdPlayer.java

```
public class DvdPlayer
{
	String description;
	Amplifier amplifier;
	String movie;
	public DvdPlayer(String description, Amplifier amplifier)
	{
		this.description = description;
		this.amplifier = amplifier;
	}
	public void on()
	{
		System.out.println(description + " on");
	}
	public void off()
	{
		System.out.println(description + " off");
	}
	public void play(String movie)
	{
		this.movie = movie;
		System.out.println(description + " playing \"" + movie + "\"");
	}
	public void stop()
	{
		System.out.println(description + " stopped \"" + movie + "\"");
	}
	public String toString()
	{
		return description;
	}
}
```

<u>Projector.java:</u>

```
public class Projector
{
	String description;
	DvdPlayer dvdPlayer;
	public Projector(String description, DvdPlayer dvdPlayer)
	{
		this.description = description;
		this.dvdPlayer = dvdPlayer;
	}
	public void on()
	{
		System.out.println(description + " on");
	}
	public void off()
	{
		System.out.println(description + " off");
	}
	public void tvMode()
	{
		System.out.println(description + " in tv mode (4x3 aspect ratio)");
	}
	public String toString()
	{
	   return description;
	}
}
```

HomeTheaterFacade.java

```java
public class HomeTheaterFacade
{
        Amplifier amp;
        DvdPlayer dvd;
        Projector projector;
        public HomeTheaterFacade(Amplifier amp, DvdPlayer dvd, Projector projector)
        {
                this.amp = amp;
                this.dvd = dvd;
                this.projector = projector;
        }
        public void watchMovie(String movie)
        {
                System.out.println("Get ready to watch a movie...");
                projector.on();
                amp.on();
                dvd.on();
                dvd.play(movie);
        }
        public void endMovie()
        {
                System.out.println("Shutting movie theater down...");
                projector.off();
                amp.off();
                dvd.stop();
                dvd.off();
        }
}
```

HomeTheaterTestDrive.java:

```java
public class HomeTheaterTestDrive
{
        public static void main(String[] args)
        {
                Amplifier amp = new Amplifier("Top-O-Line Amplifier");
                DvdPlayer dvd = new DvdPlayer("Top-O-Line DVD Player", amp);
                Projector projector = new Projector("Top-O-Line Projector", dvd);
                HomeTheaterFacade homeTheater = new  HomeTheaterFacade(amp, dvd, projector);
                homeTheater.watchMovie("Write your own movie name !!");
                homeTheater.endMovie();
        }
}
```

Output:-

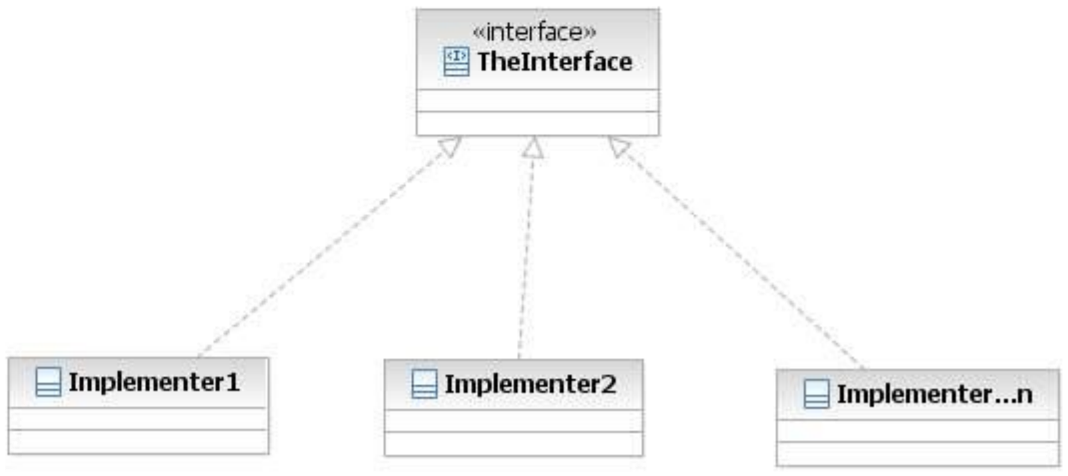
```
Get ready to watch a movie...
Top-O-Line Projector on
Top-O-Line Amplifier on
Top-O-Line DVD Player on
Top-O-Line DVD Player playing "Write your own movie name !!"
Shutting movie theater down...
Top-O-Line Projector off
Top-O-Line Amplifier off
Top-O-Line DVD Player stopped "Write your own movie name !!"
Top-O-Line DVD Player off
```

4. Polymorphism Design Pattern

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

Solution:
When related alternatives or behaviors vary by type (class), assign responsibility for the behavior using polymorphic operations to the types for which the behavior varies.

Structure:



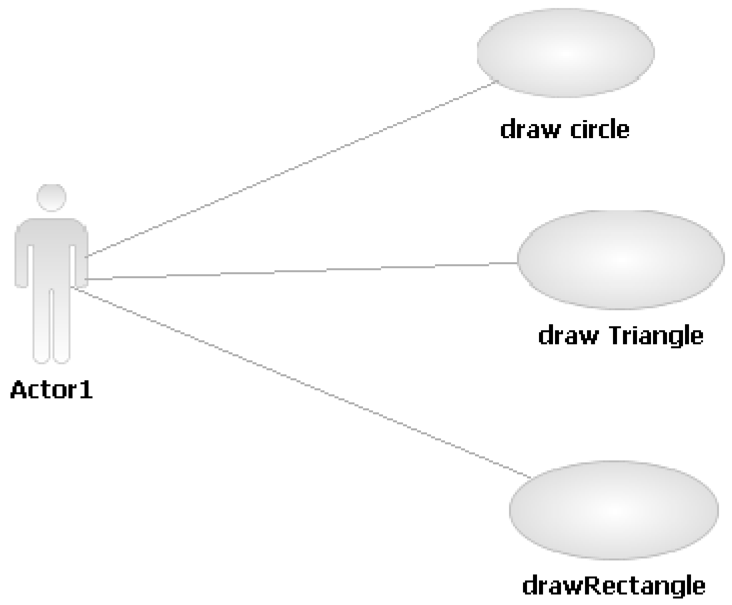
Participants:

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

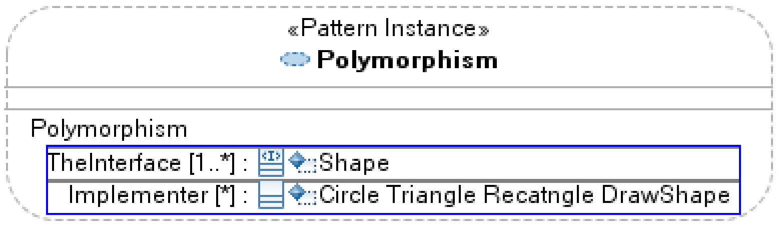
Implementer:
The classes will implement the operations from the interface as per the polymorphic nature.

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.

Usecase diagram:

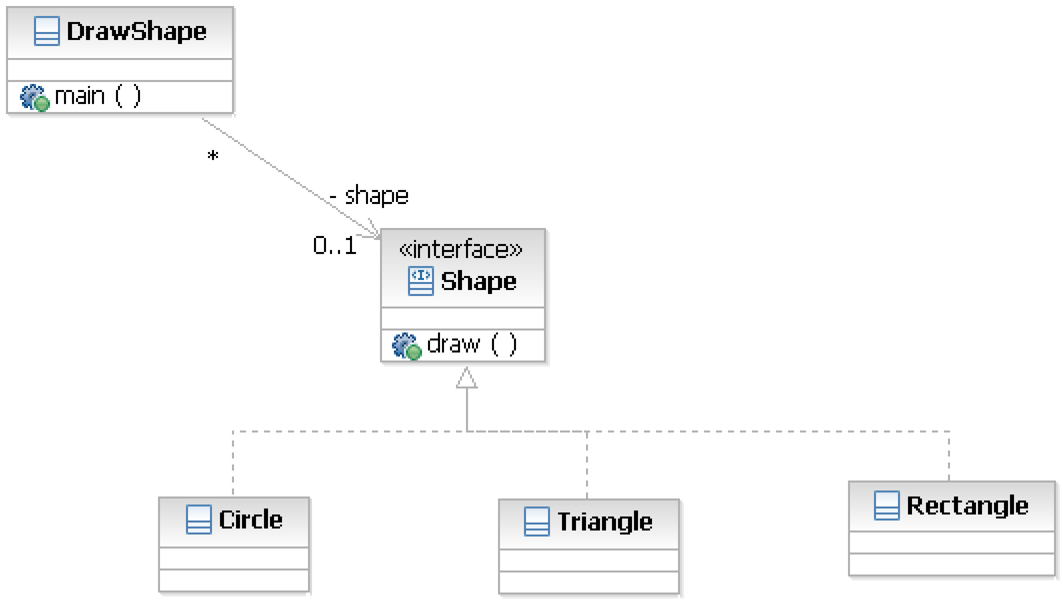


Pattern Instance:

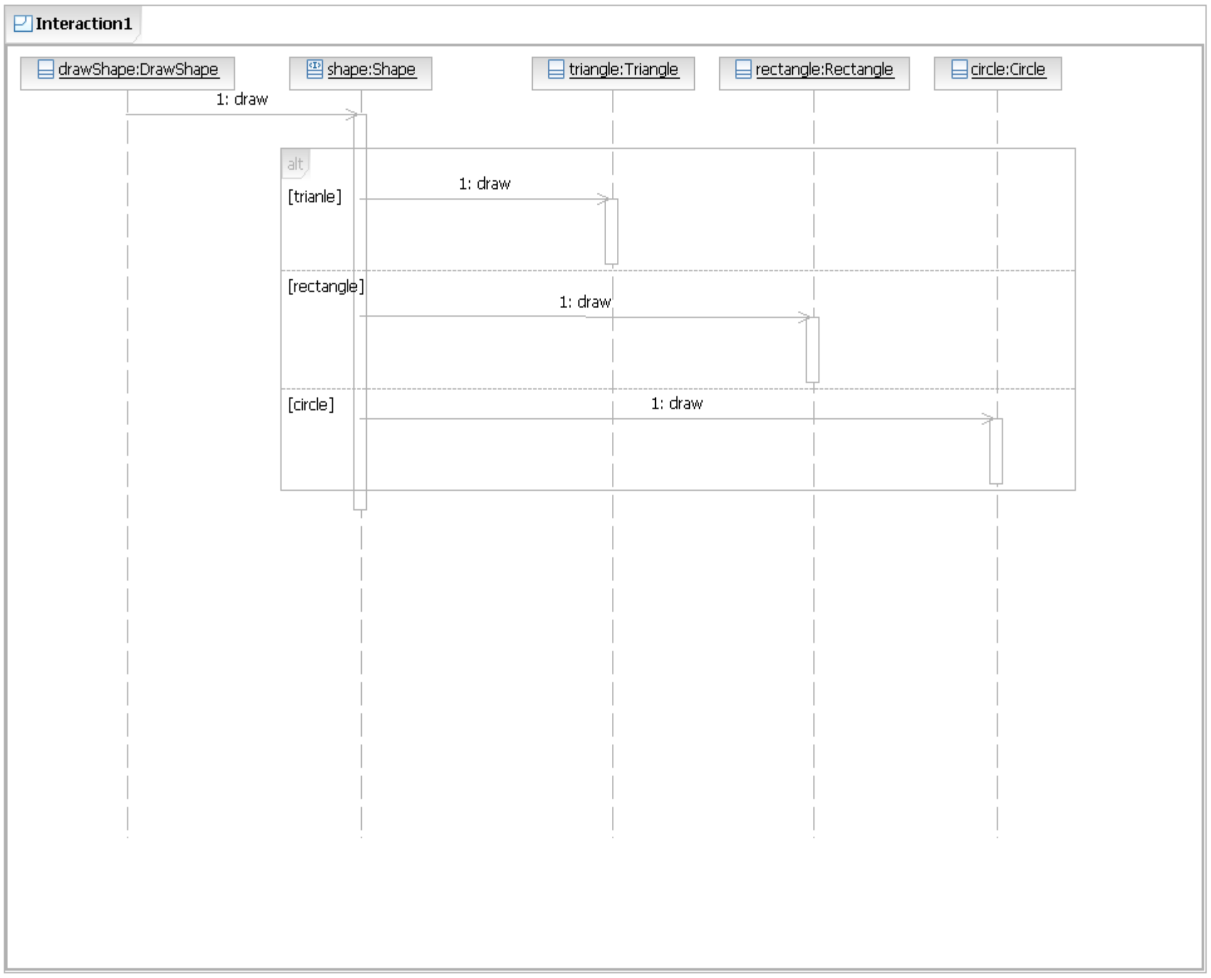


Polymorphism
TheInterface [1..*] : Shape
Implementer [*] : Circle Triangle Recatngle DrawShape

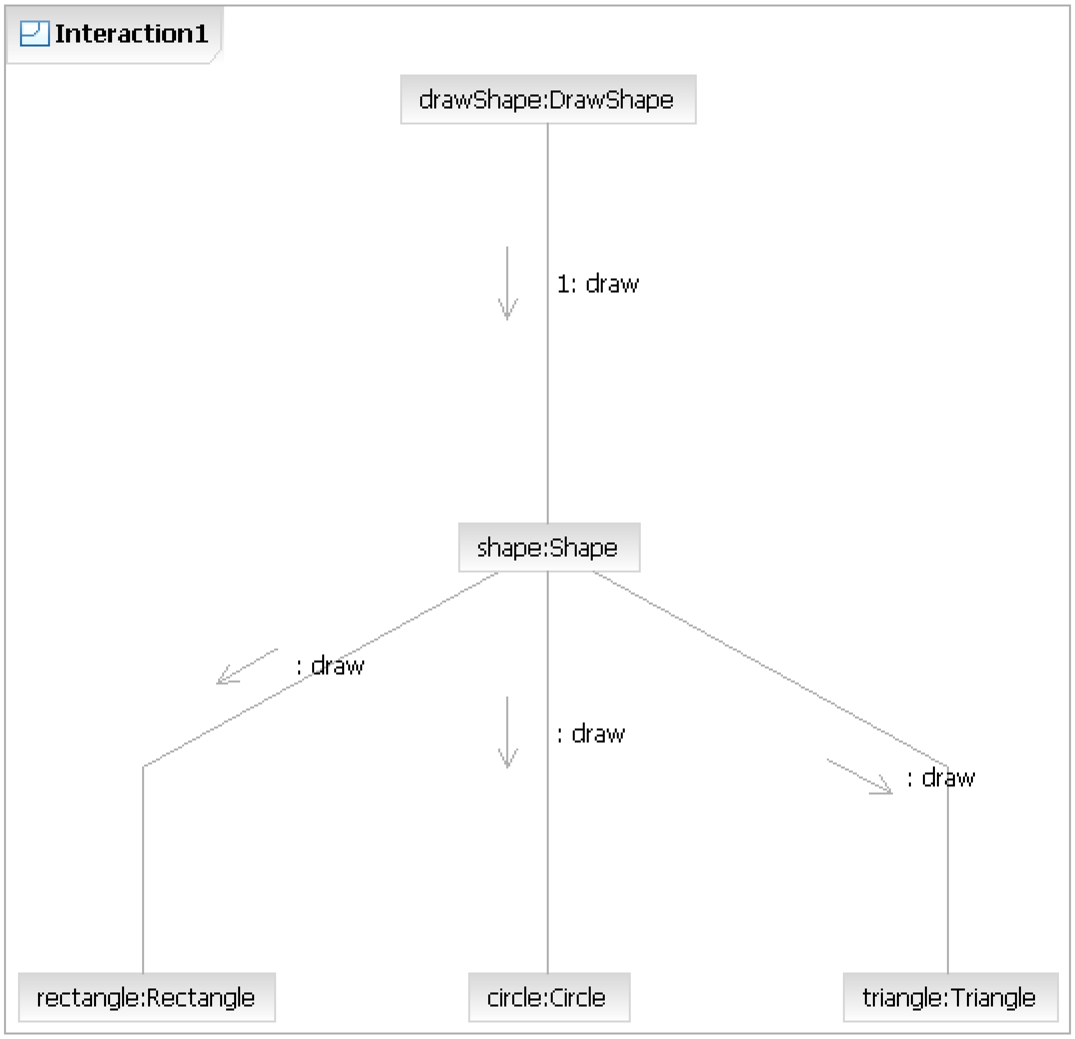
Class diagram:



Sequence diagram:



Communication Diagram:



Java Code:

Shape.java:

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

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");
	}
}
```

<u>DrawShape.java:</u>

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

<u>Output:-</u>

```
Please enter options Draw 1.Circle 2.Rectangle 3.Triangle
1
This is a Circle.
```