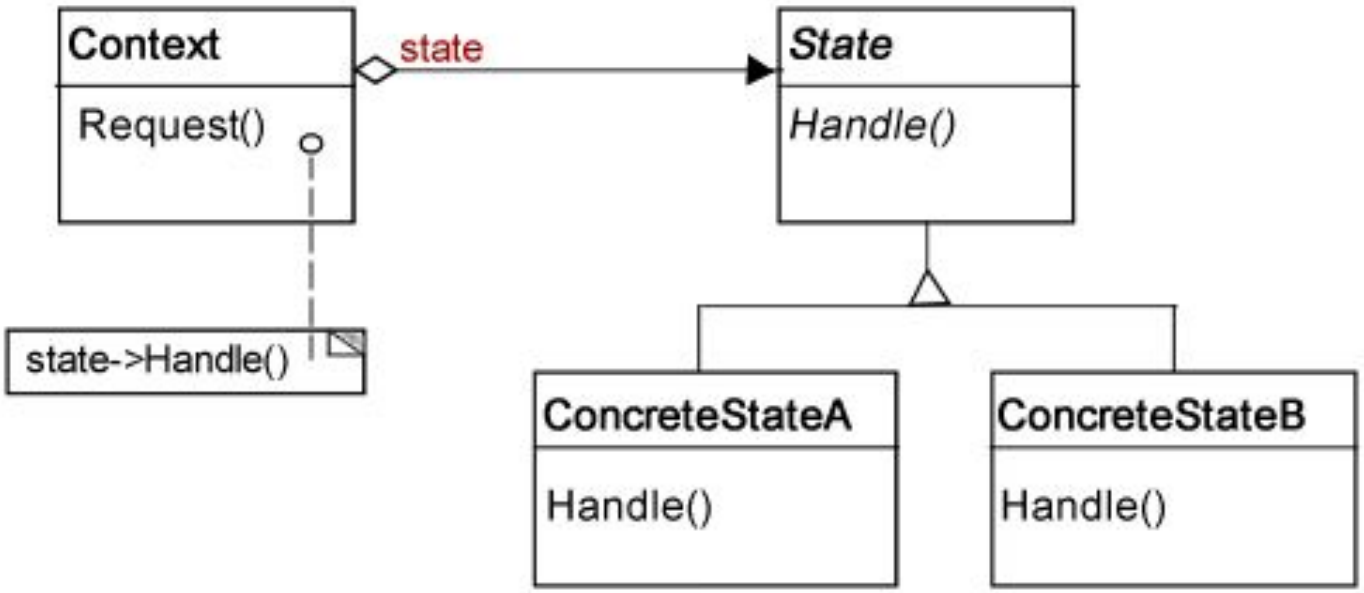
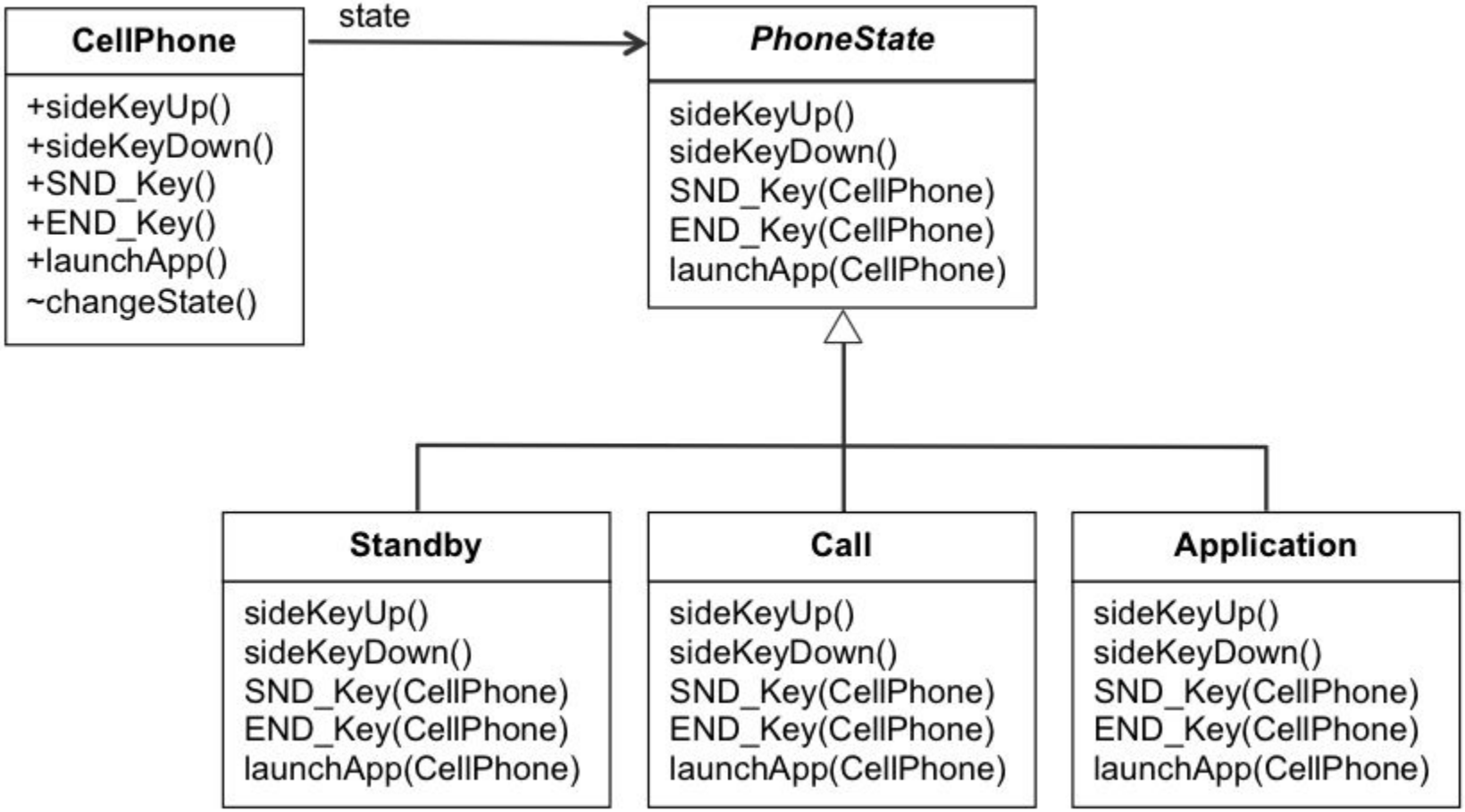
State

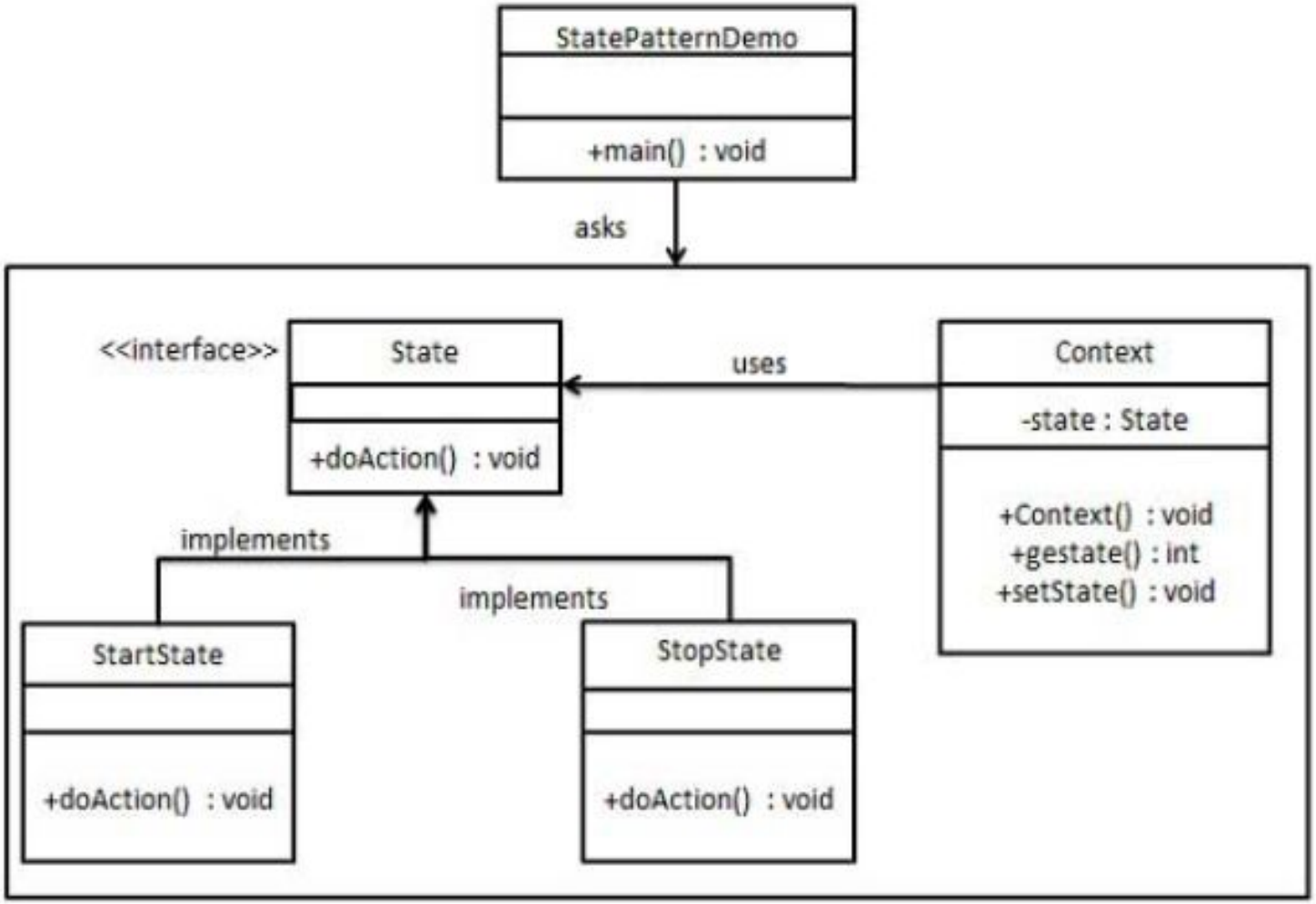
Intent	• Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
Problem	• A monolithic object's behavior is a function of its state, and it must change its behavior at run-time depending on that state.

Solution


Context
Request()
state
State
Handle()
state->Handle()
ConcreteStateA
Handle()
ConcreteStateB
Handle()

Example


CellPhone
+sideKeyUp()
+sideKeyDown()
+SND_Key()
+END_Key()
+launchApp()
~changeState()
state
PhoneState
sideKeyUp()
sideKeyDown()
SND_Key(CellPhone)
END_Key(CellPhone)
launchApp(CellPhone)
Standby
sideKeyUp()
sideKeyDown()
SND_Key(CellPhone)
END_Key(CellPhone)
launchApp(CellPhone)
Call
sideKeyUp()
sideKeyDown()
SND_Key(CellPhone)
END_Key(CellPhone)
launchApp(CellPhone)
Application
sideKeyUp()
sideKeyDown()
SND_Key(CellPhone)
END_Key(CellPhone)
launchApp(CellPhone)




```
public interface State {
   public void doAction(Context context);
}
```

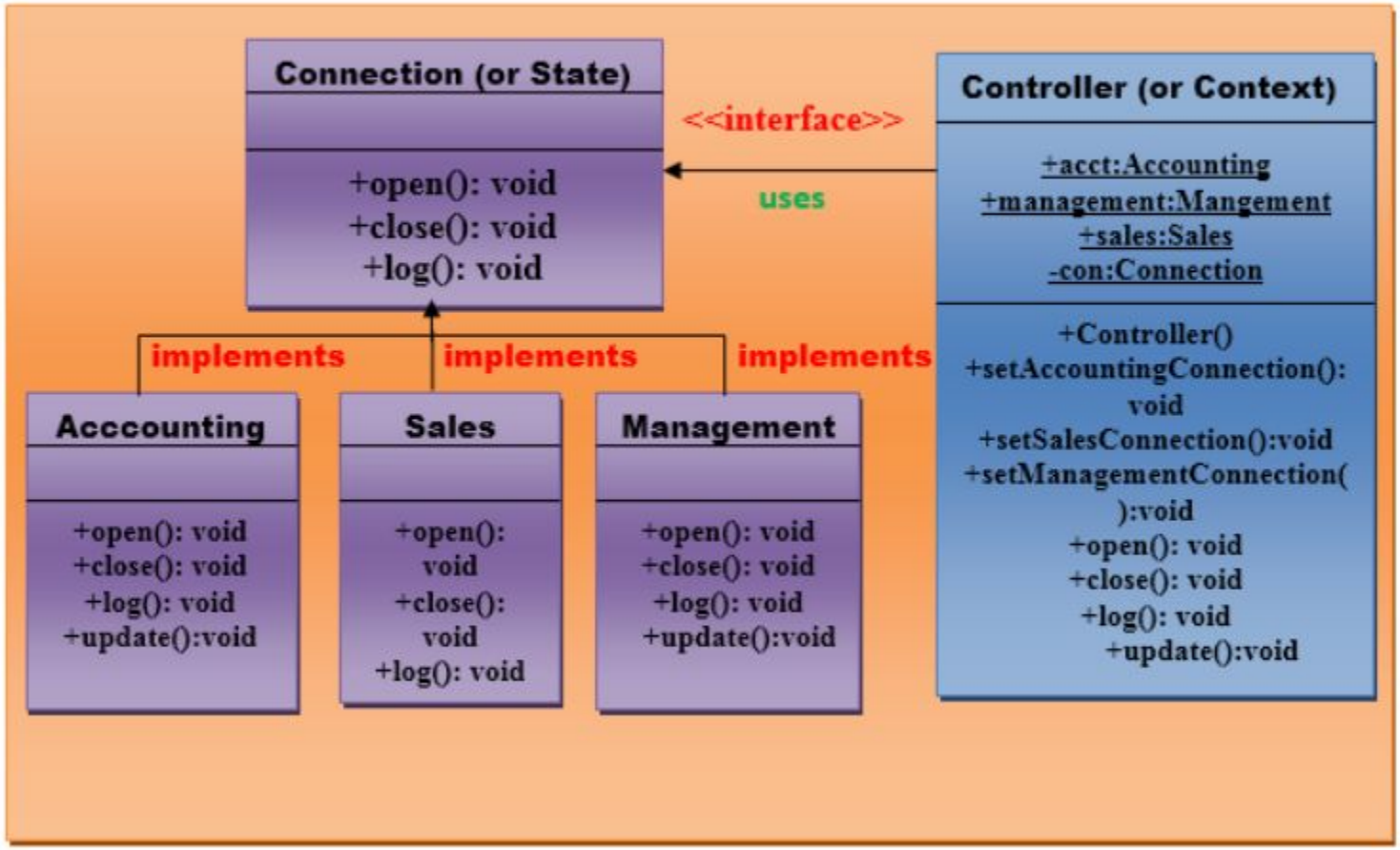
```java
public class StartState implements State {

   public void doAction(Context context) {
      System.out.println("Player is in start state");
      context.setState(this);
   }

   public String toString(){
      return "Start State";
   }
}
```

```java
public class StopState implements State {

   public void doAction(Context context) {
      System.out.println("Player is in stop state");
      context.setState(this);
   }

   public String toString(){
      return "Stop State";
   }
}
```

```java
public class Context {
   private State state;

   public Context(){
      state = null;
   }

   public void setState(State state){
      this.state = state;
   }

   public State getState(){
      return state;
   }
}
```

```java
public class StatePatternDemo {
   public static void main(String[] args) {
      Context context = new Context();

      StartState startState = new StartState();
      startState.doAction(context);

      System.out.println(context.getState().toString());

      StopState stopState = new StopState();
      stopState.doAction(context);

      System.out.println(context.getState().toString());
   }
}
```

UML for State Pattern:



Create a *Connection* interface that will provide the connection to the Controller class.

```
//This is an interface.

public interface Connection {

      public void open();
      public void close();
      public void log();
      public void update();
}// End of the Connection interface.
```

Create an *Accounting* class that will implement to the Connection interface.

```
//This is a class.
public class Accounting implements Connection {

      @Override
      public void open() {
        System.out.println("open database for accounting");
      }
      @Override
      public void close() {
        System.out.println("close the database");
      }

      @Override
      public void log() {
        System.out.println("log activities");
      }

      @Override
      public void update() {
         System.out.println("Accounting has been updated");
      }
}// End of the Accounting class.
```

```java
//This is a class.
public class Sales implements Connection {

    @Override
    public void open() {
        System.out.println("open database for sales");
    }
    @Override
    public void close() {
        System.out.println("close the database");
    }

    @Override
    public void log() {
        System.out.println("log activities");
    }

    @Override
    public void update() {
        System.out.println("Sales has been updated");
    }

}// End of the Sales class.
```

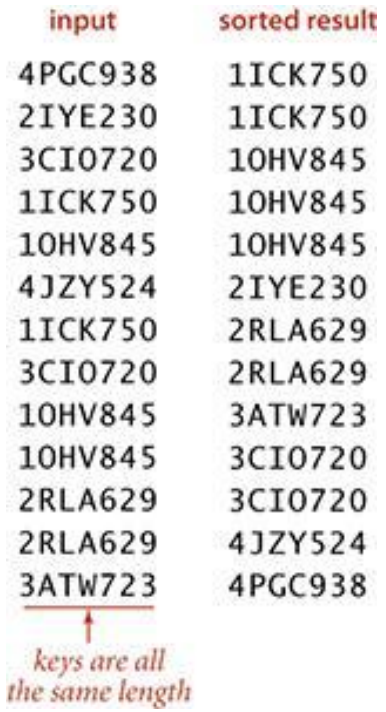
```java
public class Controller {

    public static Accounting acct;
    public static Sales sales;
    public static Management management;

    private static Connection con;

    Controller() {
        acct = new Accounting();
        sales = new Sales();
        management = new Management();
    }

    public void setAccountingConnection() {
        con = acct;
    }
    public void setSalesConnection() {
        con  = sales;
    }
    public void setManagementConnection() {
        con  = management;
    }
    public void open() {
        con .open();
    }
    public void close() {
        con .close();
    }
```

```
public class StatePatternDemo {

    Controller controller;
    StatePatternDemo(String con) {
      controller = new Controller();
      //the following trigger should be made by the user
      if(con.equalsIgnoreCase("management"))
        controller.setManagementConnection();
      if(con.equalsIgnoreCase("sales"))
        controller.setSalesConnection();
      if(con.equalsIgnoreCase("accounting"))
            controller.setAccountingConnection();
      controller.open();
      controller.log();
      controller.close();
      controller.update();
    }


    public static void main(String args[]) {

       new StatePatternDemo(args[0]);

    }

}// End of the StatePatternDemo class.
```

Consequences

- Localizes the state specific behavior
- Makes state transitions explicit

Motivation: Strategy

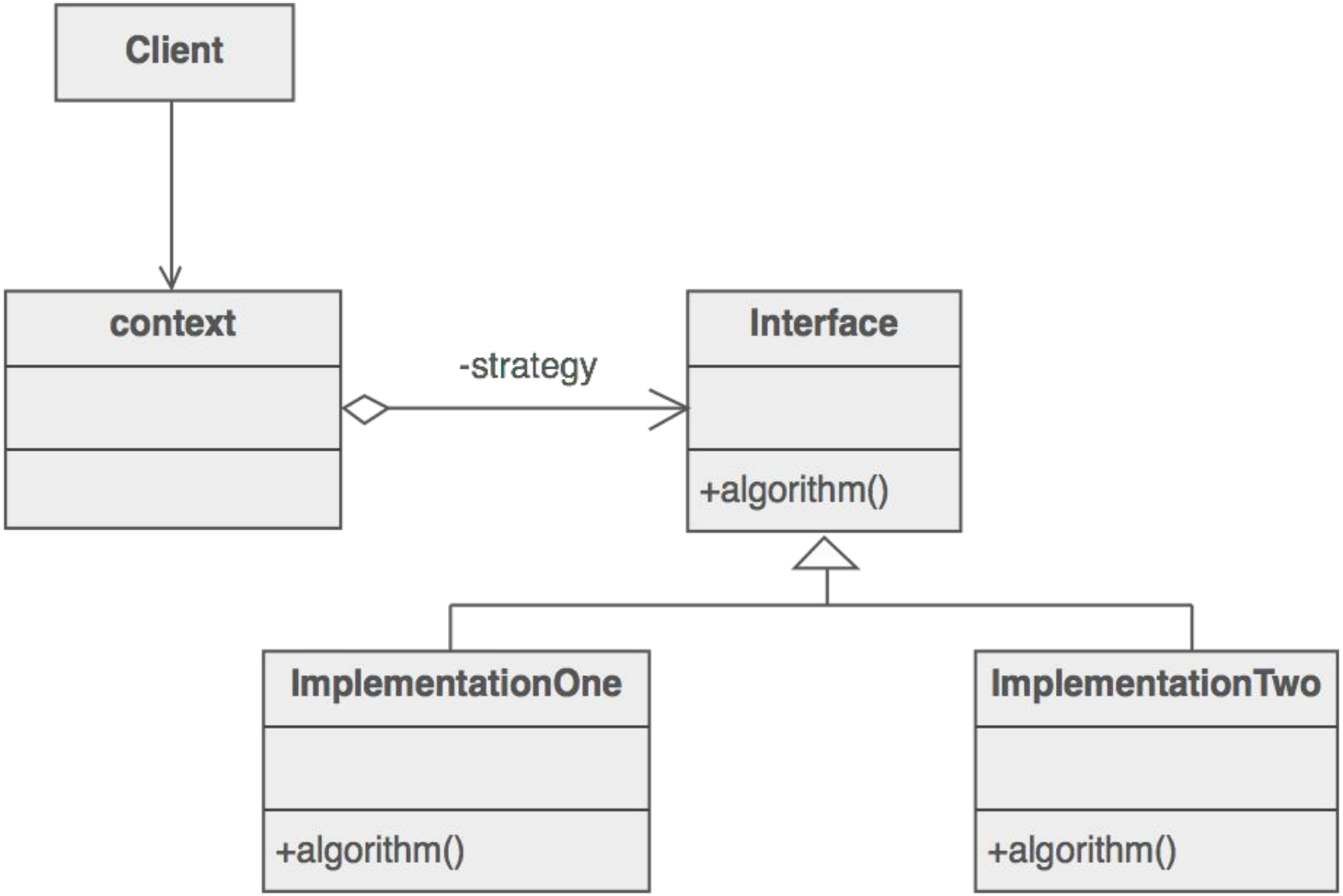


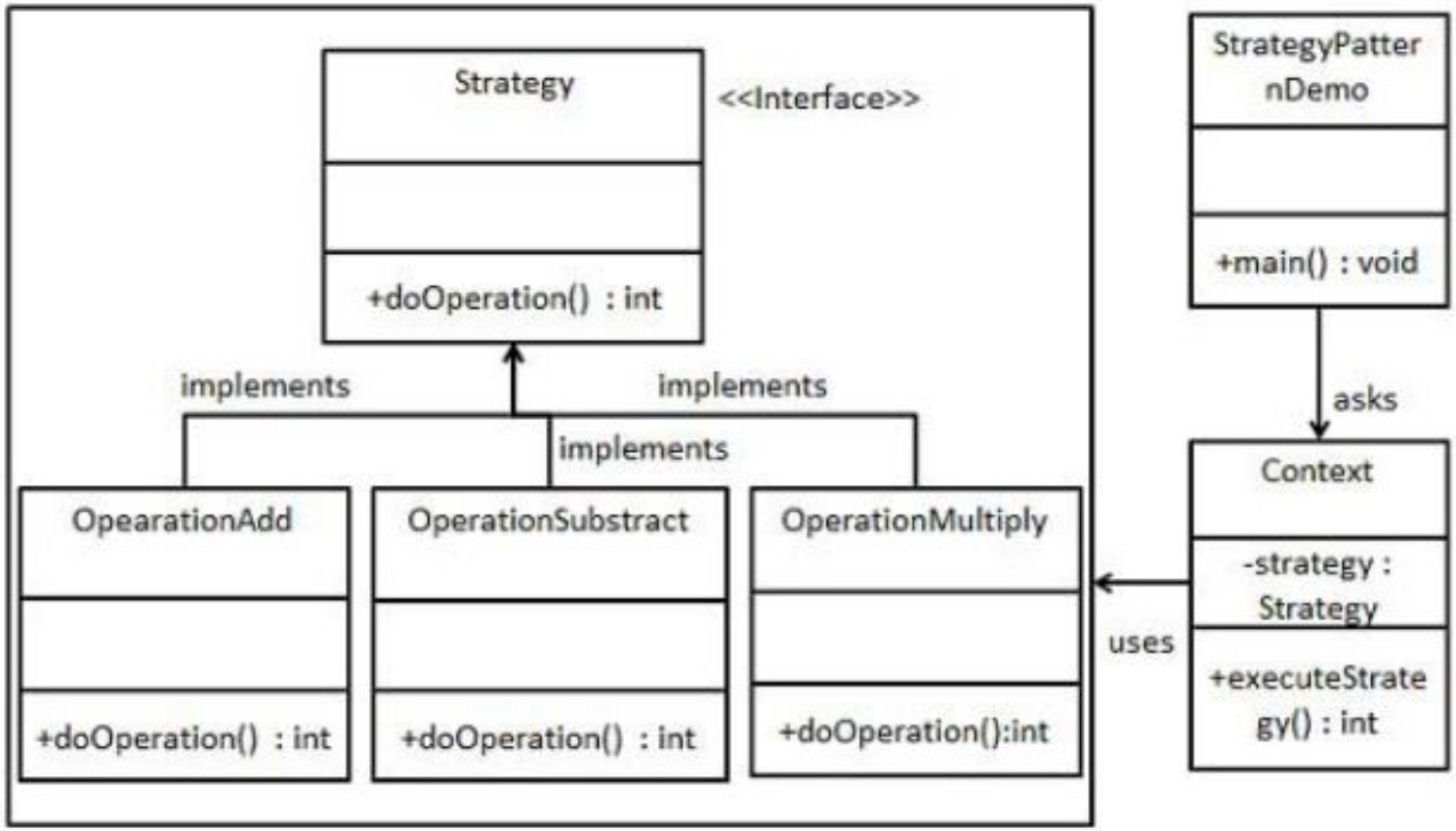
- Quick sort
- Merge sort
- Insertion sort
- Bubble sort
- Radix sort
- Heap sort
- Bucket sort
- ..

Strategy

Intent	• Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
Problem	• Capture the abstraction in an interface, bury implementation details in derived classes.

Solution


Client
context
-strategy
Interface
+algorithm()
ImplementationOne
+algorithm()
ImplementationTwo
+algorithm()




```
public interface Strategy {
   public int doOperation(int num1, int num2);
}
```

OperationAdd.java

```
public class OperationAdd implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 + num2;
   }
}
```

OperationSubstract.java

```
public class OperationSubstract implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 - num2;
   }
}
```

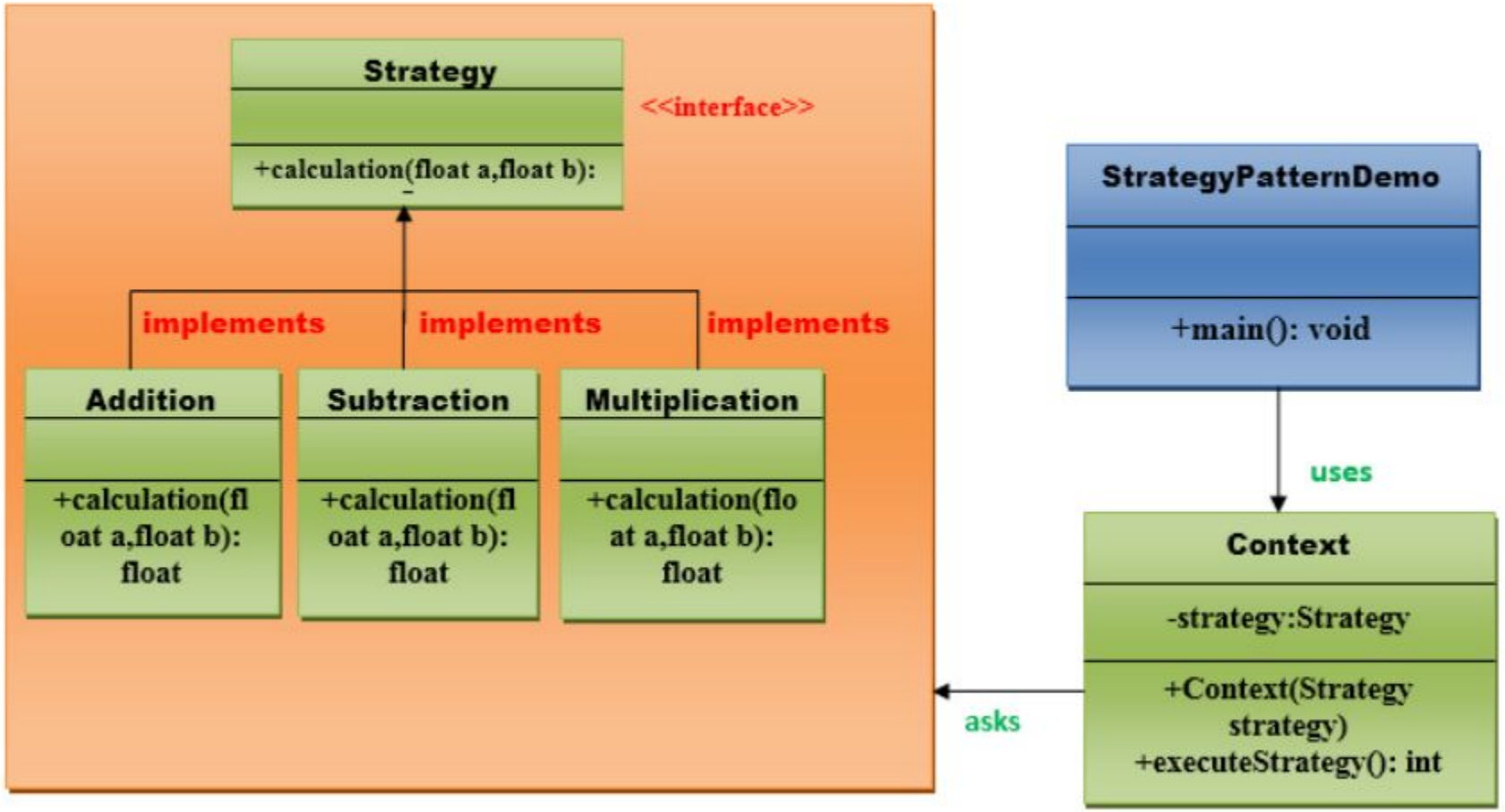
OperationMultiply.java

```
public class OperationMultiply implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 * num2;
   }
}
```

```java
public class Context {
   private Strategy strategy;

   public Context(Strategy strategy){
      this.strategy = strategy;
   }

   public int executeStrategy(int num1, int num2){
      return strategy.doOperation(num1, num2);
   }
}
```

```java
public class StrategyPatternDemo {
   public static void main(String[] args) {
      Context context = new Context(new OperationAdd());
      System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationSubstract());
      System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationMultiply());
      System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
   }
}
```

UML for Strategy Pattern:



Create a *Strategy* interface.

```
//This is an interface.

public interface Strategy {

    public float calculation(float a, float b);

}// End of the Strategy interface.
```

Create a *Subtraction* class that will implement Startegy interface.

```
//This is a class.
public class Subtraction  implements Strategy{

    @Override
    public float calculation(float a, float b) {
        return a-b;
    }

}// End of the Subtraction class.
```

Step 2:

Create a *Addition* class that will implement Startegy in

```
//This is a class.
public class Addition implements Strategy{

    @Override
    public float calculation(float a, float b) {
        return a+b;
    }

}// End of the Addition class.
```

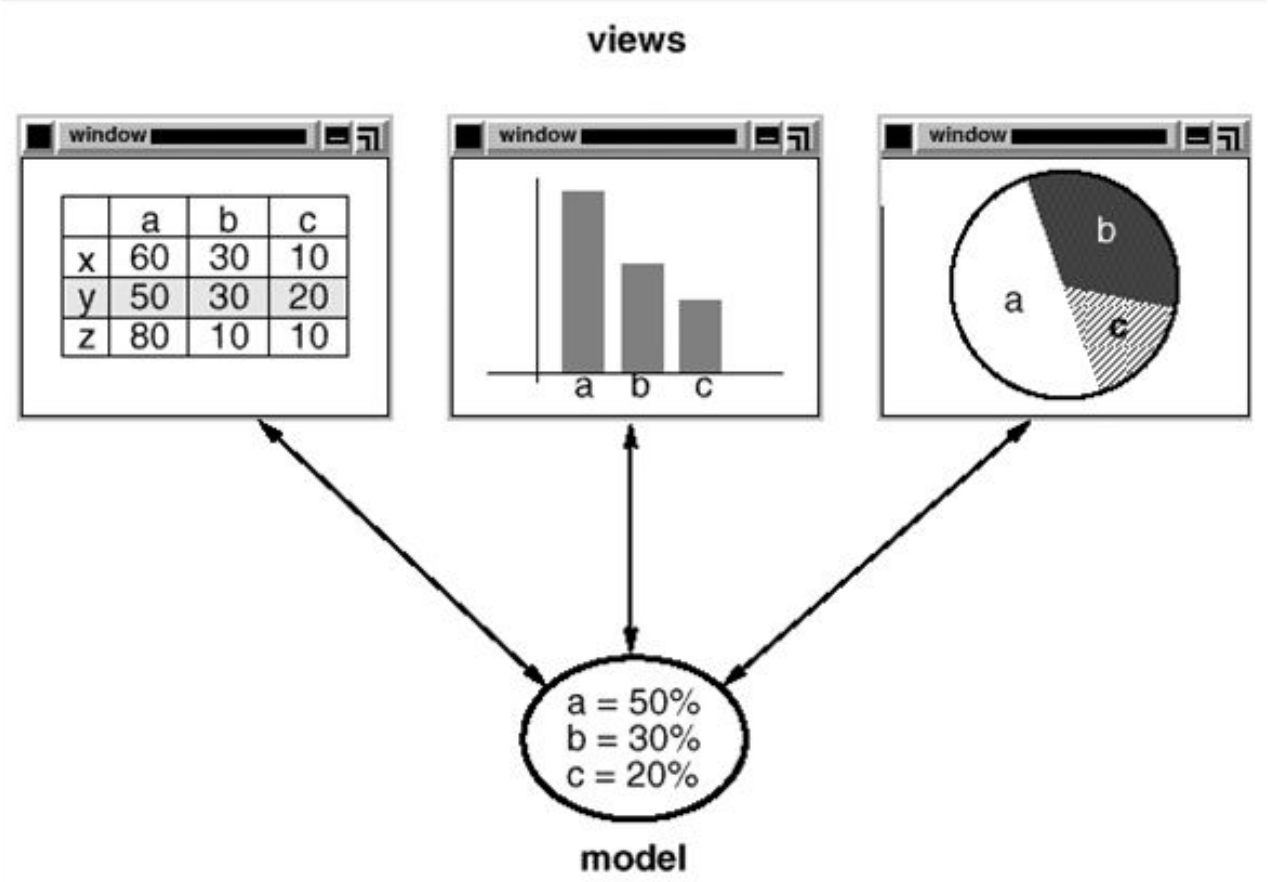
```
public class Context {

    private Strategy strategy;

    public Context(Strategy strategy){
      this.strategy = strategy;
    }

    public float executeStrategy(float num1, float num2){
      return strategy.calculation(num1, num2);
    }
}// End of the Context class.
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class StrategyPatternDemo {

   public static void main(String[] args) throws NumberFormatException, IOException {

      BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
      System.out.print("Enter the first value: ");
      float value1=Float.parseFloat(br.readLine());
      System.out.print("Enter the second value: ");
      float value2=Float.parseFloat(br.readLine());
      Context context = new Context(new Addition());
      System.out.println("Addition = " + context.executeStrategy(value1, value2));

      context = new Context(new Subtraction());
      System.out.println("Subtraction = " + context.executeStrategy(value1, value2));

      context = new Context(new Multiplication());
      System.out.println("Multiplication = " + context.executeStrategy(value1, value2));
   }

}// End of the StrategyPatternDemo class.
```

Consequences

- Families of related algorithms
- Eliminate conditional statements
- Client must be aware of different strategies

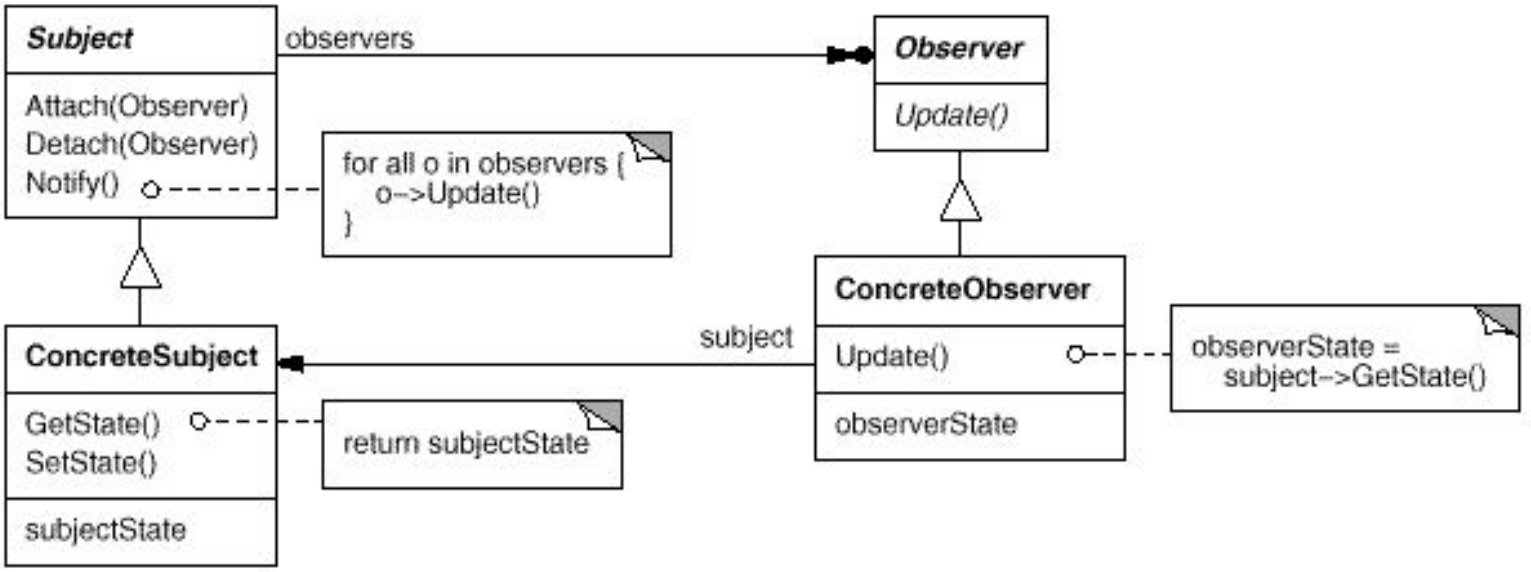
Motivation: Observer

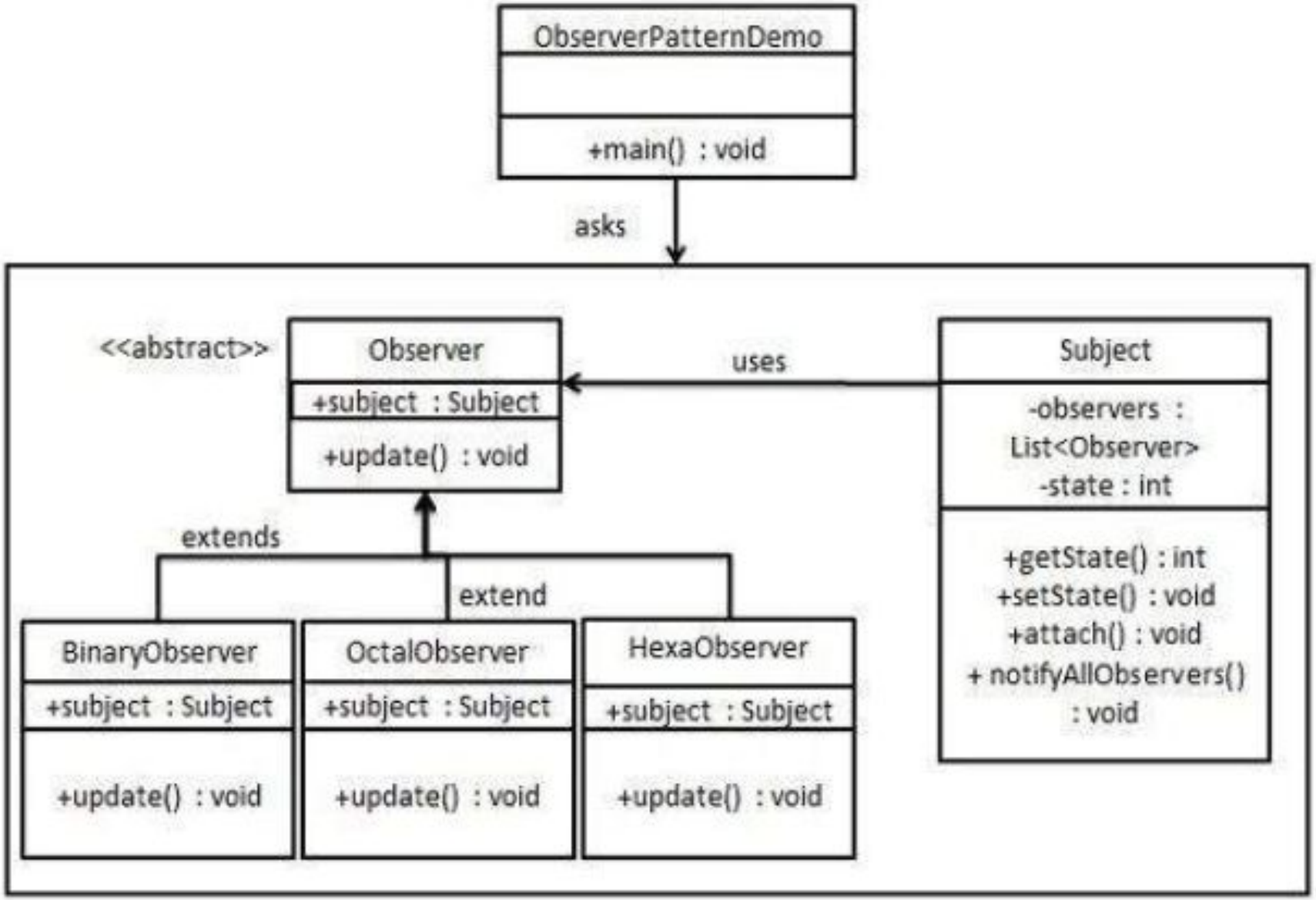


Observer

Intent	• Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Problem	• Encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.

Solution


Subject
observers
Attach(Observer)
Detach(Observer)
Notify()
for all o in observers {
o->Update()
}
Observer
Update()
ConcreteObserver
Update()
observerState
observerState =
subject->GetState()
subject
ConcreteSubject
GetState()
SetState()
subjectState
return subjectState


ObserverPatternDemo
+main() : void
asks
<<abstract>>
Observer
+subject : Subject
+update() : void
uses
Subject
-observers : List<Observer>
-state : int
+getState() : int
+setState() : void
+attach() : void
+ notifyAllObservers() : void
extends
extend
BinaryObserver
+subject : Subject
+update() : void
OctalObserver
+subject : Subject
+update() : void
HexaObserver
+subject : Subject
+update() : void

```java
import java.util.ArrayList;
import java.util.List;

public class Subject {

   private List<Observer> observers = new ArrayList<Observer>();
   private int state;

   public int getState() {
      return state;
   }

   public void setState(int state) {
      this.state = state;
      notifyAllObservers();
   }

   public void attach(Observer observer){
      observers.add(observer);
   }

   public void notifyAllObservers(){
      for (Observer observer : observers) {
         observer.update();
      }
   }
}
```

```java
public abstract class Observer {
   protected Subject subject;
   public abstract void update();
}
```

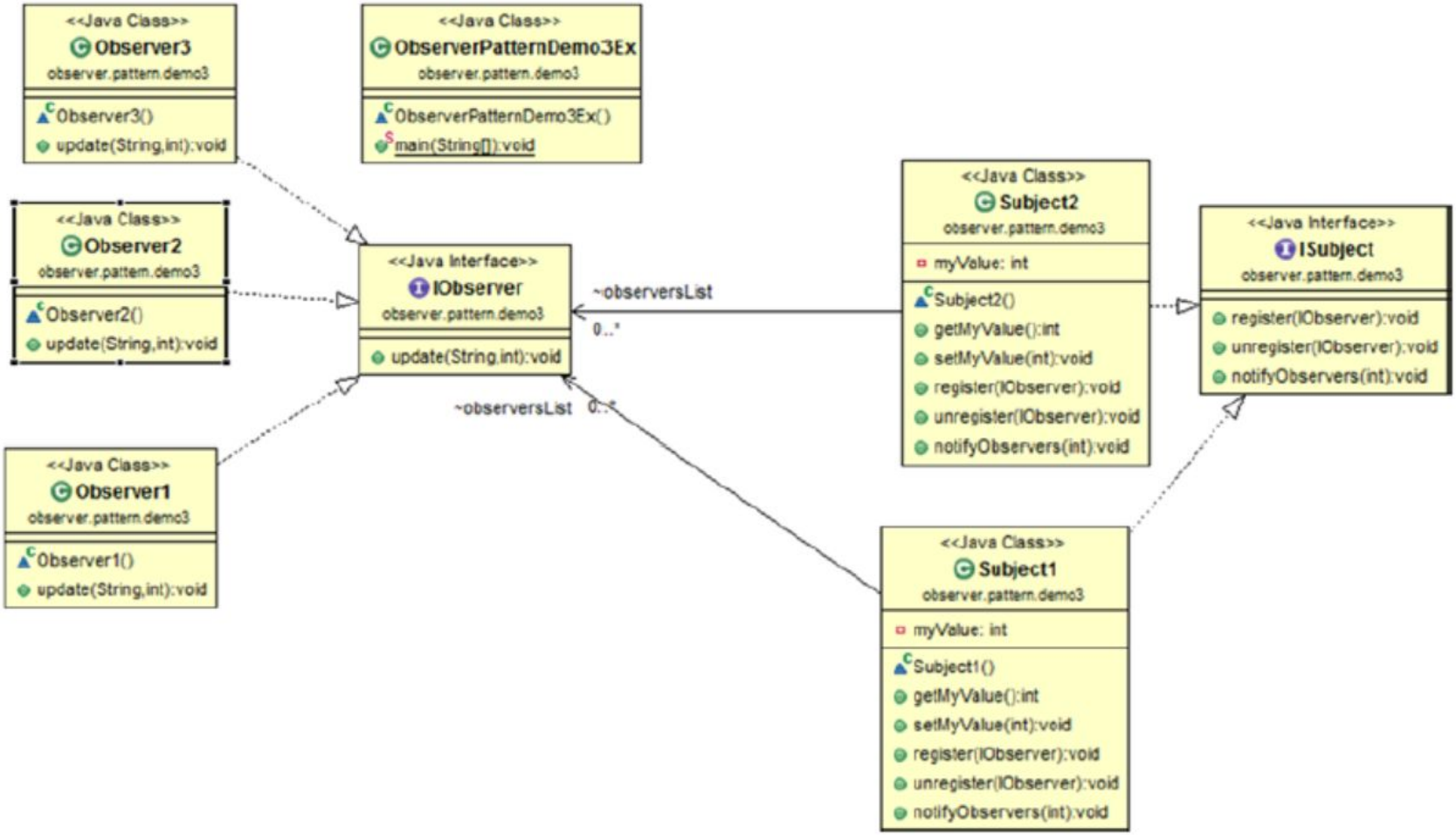
```java
public class ObserverPatternDemo {
   public static void main(String[] args) {
      Subject subject = new Subject();

      new HexaObserver(subject);
      new OctalObserver(subject);
      new BinaryObserver(subject);

      System.out.println("First state change: 15");
      subject.setState(15);
      System.out.println("Second state change: 10");
      subject.setState(10);
   }
}
```

```java
public class BinaryObserver extends Observer{

   public BinaryObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update() {
      System.out.println( "Binary String: " + Integer.toBinaryString( subject.gets
   }
}
```

Consequences

- Decoupling subject and observer
- Support broadcast communication

Many observers, many subjects

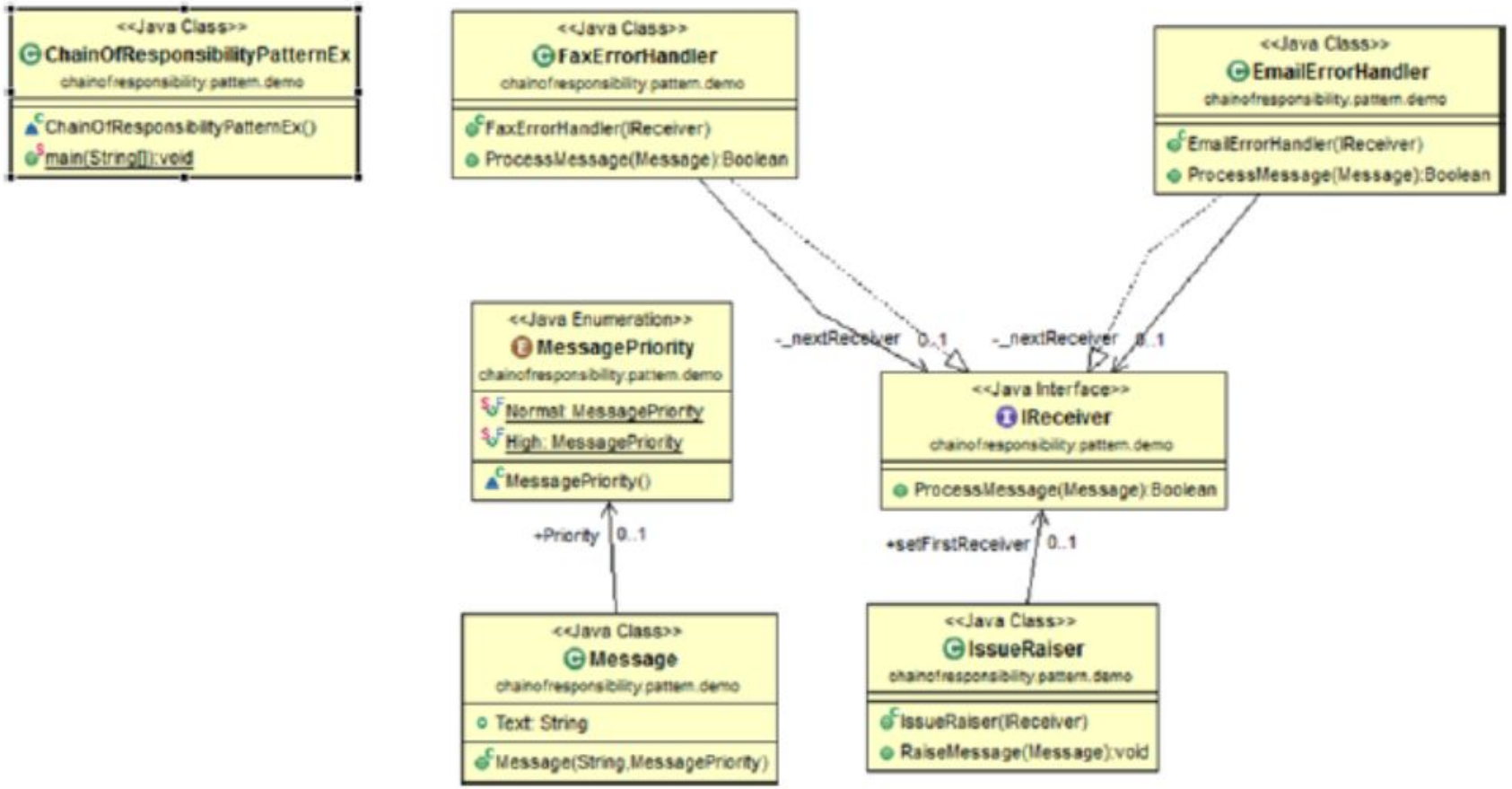


```
class Observer3 implements IObserver
{
    @Override
            public void update(String s,int i)
            {
                    System.out.println("Observer3 is observing:myValue is changed in
                    "+s+" to :"+i);
            }
}

interface ISubject
{
    void register(IObserver o);
    void unregister(IObserver o);
    void notifyObservers(int i);
}
```

```
class Subject1 implements ISubject
{
    private int myValue;

    public int getMyValue() {
                return myValue;
        }

        public void setMyValue(int myValue) {
                this.myValue = myValue;
                 //Notify observers
                notifyObservers(myValue);
        }

        List<IObserver> observersList=new ArrayList<IObserver>();

        @Override
    public void register(IObserver o)
    {
        observersList.add(o);
    }
        @Override
    public void unregister(IObserver o)
    {
        observersList.remove(o);
    }
    @Override
    public void notifyObservers(int updatedValue)
    {
       for(int i=0;i<observersList.size();i++)
            {
                observersList.get(i).update(this.getClass().getSimpleName(),
                updatedValue);
            }
    }
}
```

```
class ObserverPatternDemo3Ex
{
    public static void main(String[] args)
    {
            System.out.println("*** Observer Pattern Demo3***\n");
        Subject1 sub1 = new Subject1();
        Subject2 sub2 = new Subject2();

        Observer1 ob1 = new Observer1();
        Observer2 ob2 = new Observer2();
        Observer3 ob3 = new Observer3();

      //Observer1 and Observer2 registers to //Subject 1
        sub1.register(ob1);
        sub1.register(ob2);
      //Observer2 and Observer3 registers to //Subject 2
        sub2.register(ob2);
        sub2.register(ob3);
      //Set new value to Subject 1
      //Observer1 and Observer2 get //notification
        sub1.setMyValue(50);
        System.out.println();
      //Set new value to Subject 2
      //Observer2 and Observer3 get //notification
        sub2.setMyValue(250);
        System.out.println();
        //unregister Observer2 from Subject 1
```

Chain of Responsibility

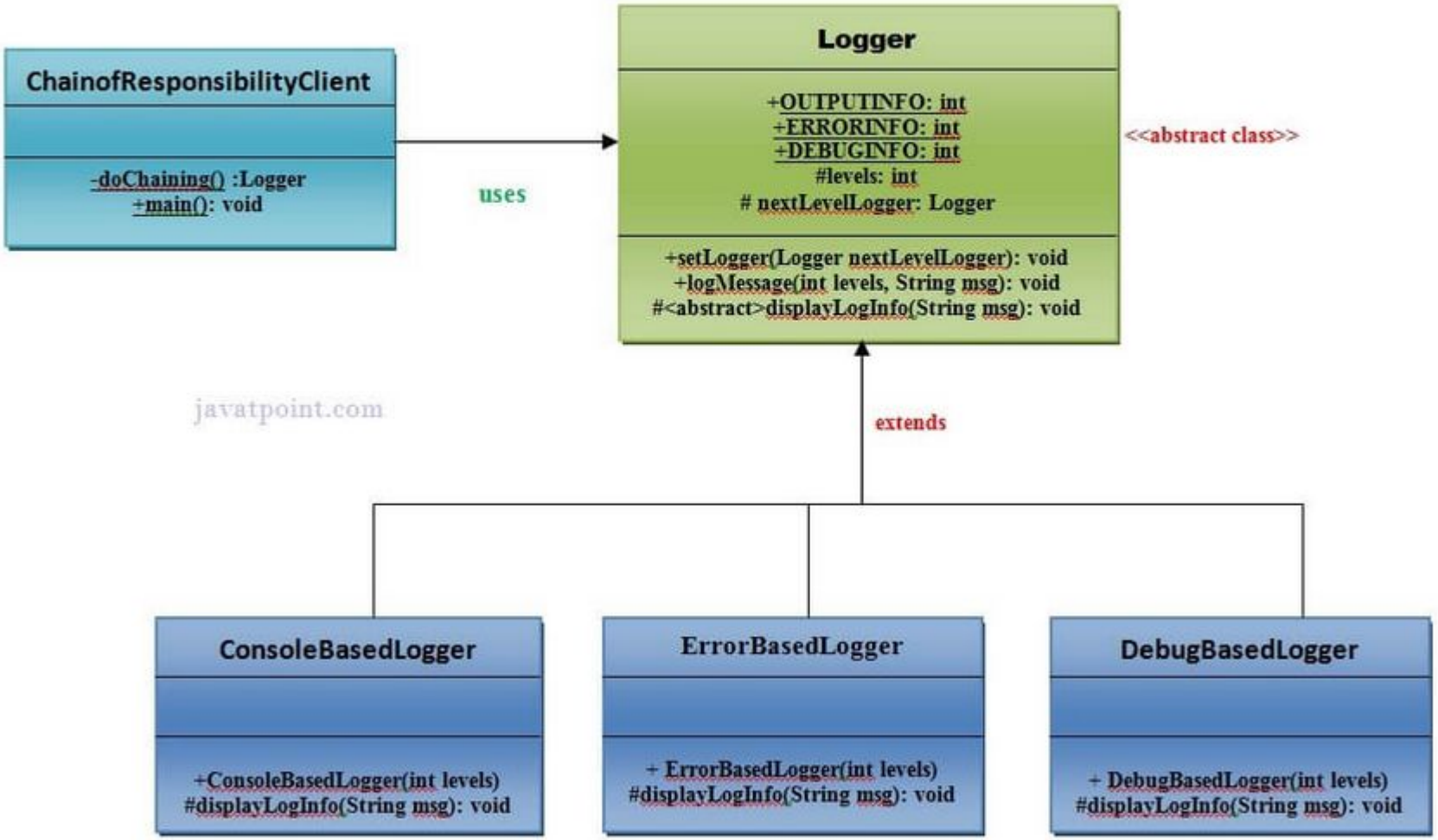


```
interface IReceiver
{
        Boolean ProcessMessage(Message msg);
}
Class IssueRaiser
{
        public IReceiver setFirstReceiver;
        public IssueRaiser(IReceiver firstReceiver)
        {
                this.setFirstReceiver = firstReceiver;
        }
        public void RaiseMessage(Message msg)
        {
                if (setFirstReceiver != null)
                        setFirstReceiver.ProcessMessage(msg);
        }
}
class FaxErrorHandler implements IReceiver
{
        private IReceiver _nextReceiver;
        public FaxErrorHandler(IReceiver nextReceiver)
        {
                _nextReceiver = nextReceiver;
        }
        public Boolean ProcessMessage(Message msg)
        {
                if (msg.Text.contains("Fax"))
                {
                        System.out.println("FaxErrorHandler processed "+  msg.Priority+
                        "priority issue: "+ msg.Text);
                        return true;
                }
                else
                {
                        if (_nextReceiver != null)
                                _nextReceiver.ProcessMessage(msg);
                }
                return false;
        }
}
```

```
class EmailErrorHandler implements IReceiver
{
        private IReceiver _nextReceiver;
        public EmailErrorHandler(IReceiver nextReceiver)
        {
                _nextReceiver = nextReceiver;
        }
        public Boolean ProcessMessage(Message msg)
        {
                if (msg.Text.contains("Email"))
                {
                        System.out.println("EmailErrorHandler processed "+ msg.Priority+
                        "priority issue: "+ msg.Text);
                        return true;
                }
                else
                {
                        if (_nextReceiver != null)
                                _nextReceiver.ProcessMessage(msg);
                }
                return false;
        }
}
class ChainOfResponsibilityPatternEx
{
```

```
class ChainOfResponsibilityPatternEx
{
        public static void main(String[] args)
        {
                System.out.println("***Chain of Responsibility Pattern Demo***\n");
                //Making the chain first: IssueRaiser->FaxErrorhandler->EmailErrorHandler
                IReceiver faxHandler, emailHandler;
                //end of chain
                emailHandler = new EmailErrorHandler(null);
                //fax handler is before email
                faxHandler = new FaxErrorHandler(emailHandler);

                //starting point: raiser will raise issues and set the first handler
                IssueRaiser raiser = new IssueRaiser (faxHandler);

                Message m1 = new Message("Fax is reaching late to the destination",
                MessagePriority.Normal);
                Message m2 = new Message("Email is not going", MessagePriority.High);
                Message m3 = new Message("In Email, BCC field is disabled occasionally",
                MessagePriority.Normal);
                Message m4 = new Message("Fax is not reaching destination",
                MessagePriority.High);

                raiser.RaiseMessage(m1);
                raiser.RaiseMessage(m2);
                raiser.RaiseMessage(m3);
                raiser.RaiseMessage(m4);

        }
}
```

UML for Chain of Responsibility Pattern:



Create a **Logger** abstract class.

```
public abstract class Logger {
   public static int OUTPUTINFO=1;
   public static int ERRORINFO=2;
   public static int DEBUGINFO=3;
   protected int levels;
   protected Logger nextLevelLogger;
   public void setNextLevelLogger(Logger nextLevelLogger) {
      this.nextLevelLogger = nextLevelLogger;
   }
      public void logMessage(int levels, String msg){
      if(this.levels<=levels){
         displayLogInfo(msg);
      }
      if (nextLevelLogger!=null) {
         nextLevelLogger.logMessage(levels, msg);
      }
   }
   protected abstract void displayLogInfo(String msg);
}
```

Create a **ConsoleBasedLogger** class.

File: ConsoleBasedLogger.java

```
public class ConsoleBasedLogger extends Logger {
   public ConsoleBasedLogger(int levels) {
      this.levels=levels;
   }
   @Override
   protected void displayLogInfo(String msg) {
      System.out.println("CONSOLE LOGGER INFO: "+msg);
   }
}
```

Step 3

Create a **DebugBasedLogger** class.

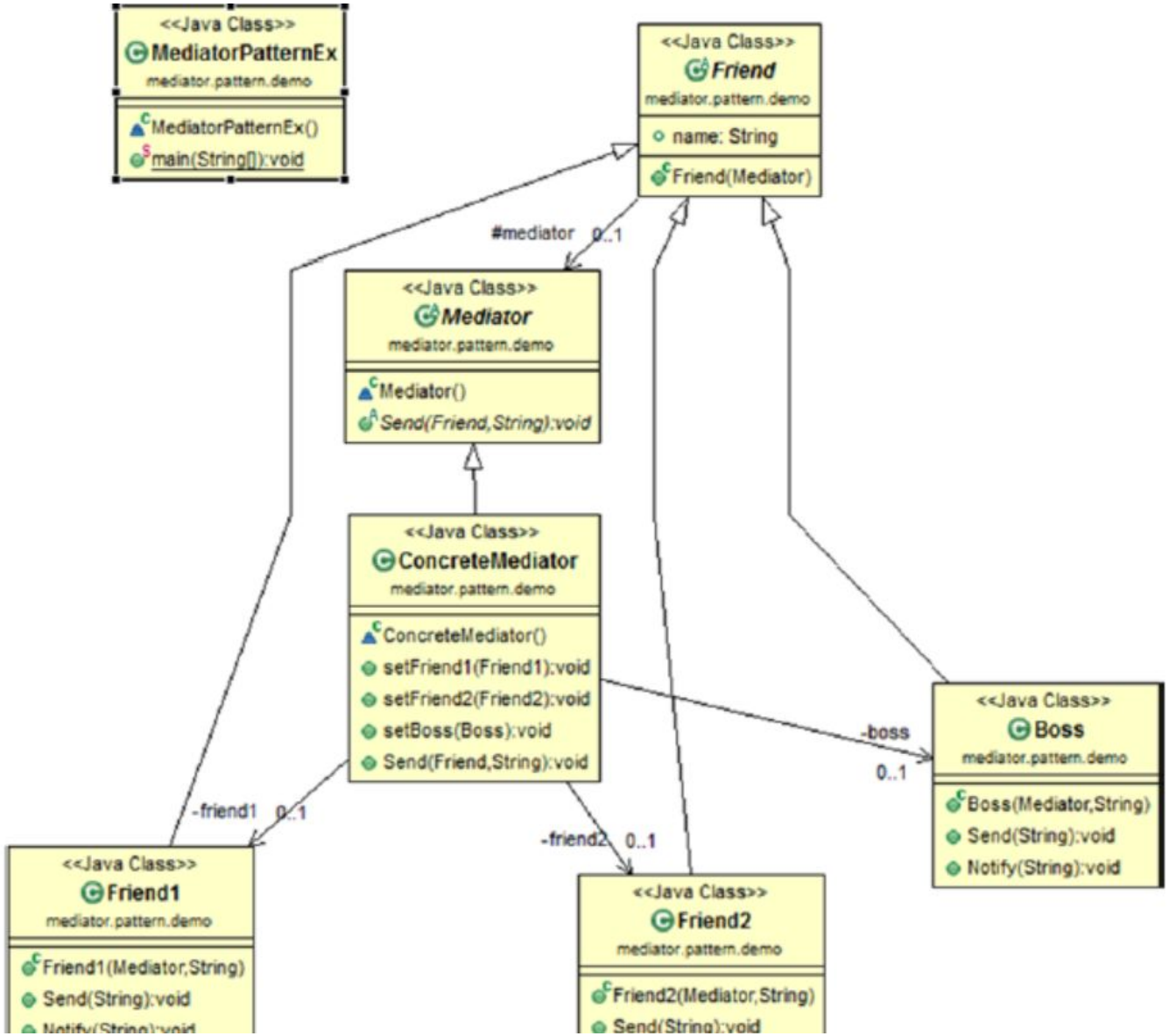
File: DebugBasedLogger.java

```
public class DebugBasedLogger extends Logger {
   public DebugBasedLogger(int levels) {
      this.levels=levels;
   }
   @Override
   protected void displayLogInfo(String msg) {
      System.out.println("DEBUG LOGGER INFO: "+msg);
   }
}// End of the DebugBasedLogger class.
```

File: ChainofResponsibilityClient.java

```
public class ChainofResponsibilityClient {
   private static Logger doChaining(){
        Logger consoleLogger = new ConsoleBasedLogger(Logger.OUTPUTINFO);

        Logger errorLogger = new ErrorBasedLogger(Logger.ERRORINFO);
        consoleLogger.setNextLevelLogger(errorLogger);

        Logger debugLogger = new DebugBasedLogger(Logger.DEBUGINFO);
        errorLogger.setNextLevelLogger(debugLogger);

        return consoleLogger;
        }
        public static void main(String args[]){
        Logger chainLogger= doChaining();

           chainLogger.logMessage(Logger.OUTPUTINFO, "Enter the sequence of values ");
           chainLogger.logMessage(Logger.ERRORINFO, "An error is occured now");
           chainLogger.logMessage(Logger.DEBUGINFO, "This was the error now debugging is compeled");
           }
}
```

Mediator


<<Java Class>>
MediatorPatternEx
mediator.pattern.demo
MediatorPatternEx()
main(String[]):void
<<Java Class>>
Friend
mediator.pattern.demo
name: String
Friend(Mediator)
#mediator 0..1
<<Java Class>>
Mediator
mediator.pattern.demo
Mediator()
Send(Friend,String):void
<<Java Class>>
ConcreteMediator
mediator.pattern.demo
ConcreteMediator()
setFriend1(Friend1):void
setFriend2(Friend2):void
setBoss(Boss):void
Send(Friend,String):void
-boss
0..1
<<Java Class>>
Boss
mediator.pattern.demo
Boss(Mediator,String)
Send(String):void
Notify(String):void
-friend1 0..1
-friend2 0..1
<<Java Class>>
Friend1
mediator.pattern.demo
Friend1(Mediator,String)
Send(String):void
<<Java Class>>
Friend2
mediator.pattern.demo
Friend2(Mediator,String)
Send(String):void

```
abstract class Mediator
{
    public abstract void Send(Friend frd, String msg);
}

// ConcreteMediator
class ConcreteMediator extends  Mediator
{
    private Friend1 friend1;
    private Friend2 friend2;
    private Boss boss;

    public void setFriend1(Friend1 friend1) {
            this.friend1 = friend1;
    }

    public void setFriend2(Friend2 friend2) {
            this.friend2 = friend2;
    }

    public void setBoss(Boss boss) {
            this.boss = boss;
    }
```

```
    public void Send(Friend frd,String msg)
    {
        //In all cases, boss is notified
        if (frd == friend1)
        {
            friend2.Notify(msg);
            boss.Notify(friend1.name + " sends message to " + friend2.name);
        }
        if(frd==friend2)
        {
            friend1.Notify(msg);
            boss.Notify(friend2.name + " sends message to " + friend1.name);

        }
        //Boss is sending message to others
        if(frd==boss)
        {
            friend1.Notify(msg);
            friend2.Notify(msg);
        }
    }

}
```

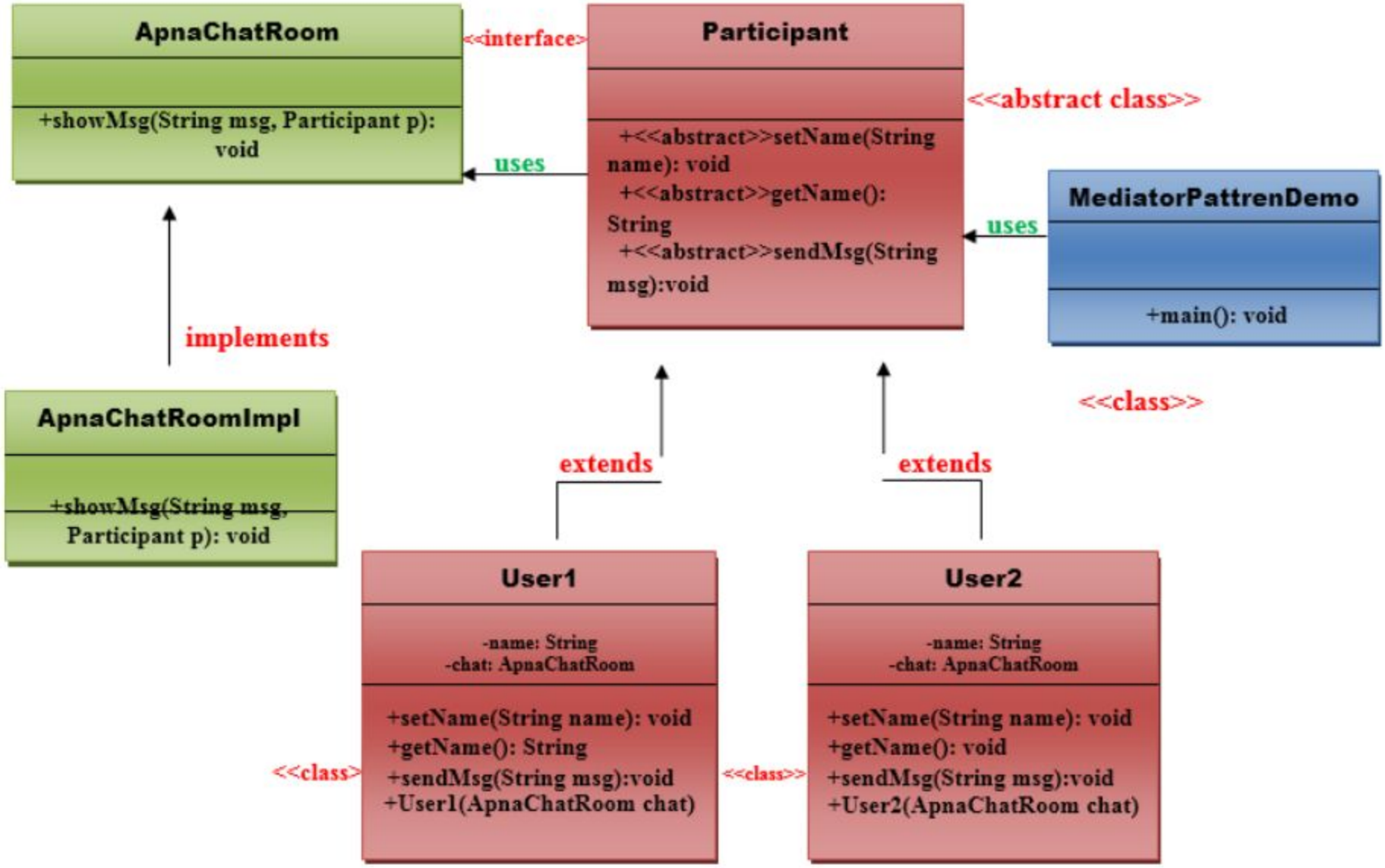
```
abstract class Friend
{
    protected Mediator mediator;
    public String name;

   public Friend(Mediator _mediator)
   {
       mediator = _mediator;
   }
}
```

```
// Friend1-first participant
class Friend1 extends Friend
{
    public Friend1(Mediator mediator,String name)
    {
            super(mediator);
        this.name = name;
    }

    public void Send(String msg)
    {
        mediator.Send(this,msg);
    }

    public void Notify(String msg)
    {
            System.out.println("Amit gets message: "+ msg);
    }
}
```

```
class Boss extends Friend
{
    // Constructor
    public Boss(Mediator mediator,String name)
    {
            super(mediator);
        this.name = name;
    }

    public void Send(String msg)
    {
        mediator.Send(this, msg);
    }

    public void Notify(String msg)
    {
            System.out.println("\nBoss sees message: " + msg);
            System.out.println("");
    }
}
```

```
class MediatorPatternEx
{
    public static void main(String[] args)
    {
            System.out.println("***Mediator Pattern Demo***\n");
            ConcreteMediator m = new ConcreteMediator();

        Friend1 Amit= new Friend1(m,"Amit");
        Friend2 Sohel = new Friend2(m,"Sohel");
        Boss Raghu = new Boss(m,"Raghu");

        m.setFriend1(Amit);
        m.setFriend2(Sohel);
        m.setBoss(Raghu);

        Amit.Send("[Amit here]Good Morrning. Can we discuss the mediator pattern?");
        Sohel.Send("[Sohel here]Good Morning.Yes, we can discuss now.");
        Raghu.Send("\n[Raghu here]:Please get back to work quickly");
    }
}
```

UML for Mediator Pattern:



Create a *ApnaChatRoom* interface.

```
//This is an interface.
public interface ApnaChatRoom {

    public void showMsg(String msg, Participant p);

}// End of the ApnaChatRoom interface.
```

Step 2:

Create a *ApnaChatRoomIml* class that will implement ApnaChatRoom interface and will also use the r
Participant interface.

```
//This is a class.
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;

public class ApnaChatRoomImpl implements ApnaChatRoom{
    //get current date time
    DateFormat dateFormat = new SimpleDateFormat("E dd-MM-yyyy hh:mm a");
    Date date = new Date();
    @Override
    public void showMsg(String msg, Participant p) {

        System.out.println(p.getName()+"'gets message: "+msg);
        System.out.println("\t\t\t\t"+"["+dateFormat.format(date).toString()+"]");
    }
}// End of the ApnaChatRoomImpl class.
```

Create a *Participant* abstract class.

```
//This is an abstract class.
public abstract class Participant {
    public abstract void sendMsg(String msg);
    public abstract void setname(String name);
    public abstract String getName();
}// End of the Participant abstract class.
```

```
public class User1 extends Participant {

    private String name;
    private ApnaChatRoom chat;

    @Override
    public void sendMsg(String msg) {
    chat.showMsg(msg,this);

    }

    @Override
    public void setname(String name) {
        this.name=name;
    }

    @Override
    public String getName() {
        return name;
    }

    public User1(ApnaChatRoom chat){
        this.chat=chat;
    }

}// End of the User1 class.
```

```
public class User2 extends Participant {

    private String name;
    private ApnaChatRoom chat;

    @Override
    public void sendMsg(String msg) {
    this.chat.showMsg(msg,this);

    }

    @Override
    public void setname(String name) {
        this.name=name;
    }

    @Override
    public String getName() {
        return name;
    }

    public User2(ApnaChatRoom chat){
        this.chat=chat;
    }
```

```
public class MediatorPatternDemo {

    public static void main(String args[])
    {

        ApnaChatRoom chat = new ApnaChatRoomImpl();

        User1 u1=new User1(chat);
        u1.setname("Ashwani Rajput");
        u1.sendMsg("Hi Ashwani! how are you?");


        User2 u2=new User2(chat);
        u2.setname("Soono Jaiswal");
        u2.sendMsg("I am Fine ! You tell?");
    }

}// End of the MediatorPatternDemo class.
```