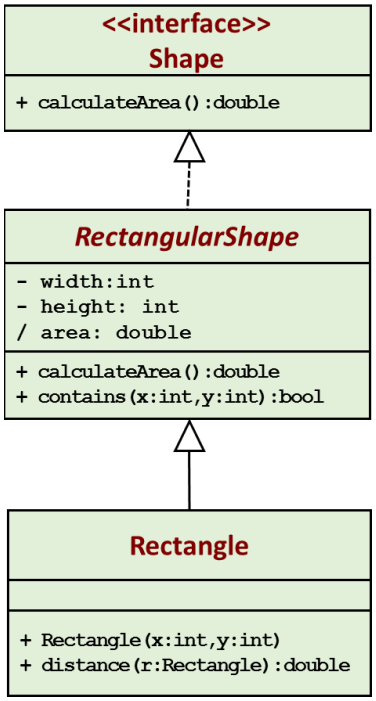
Creational Patterns

UML Revisited



Generalization

Aggregation: “is part of”

Composition: “is entirely made of”

Dependency: “uses”

Creational Patterns to be Covered

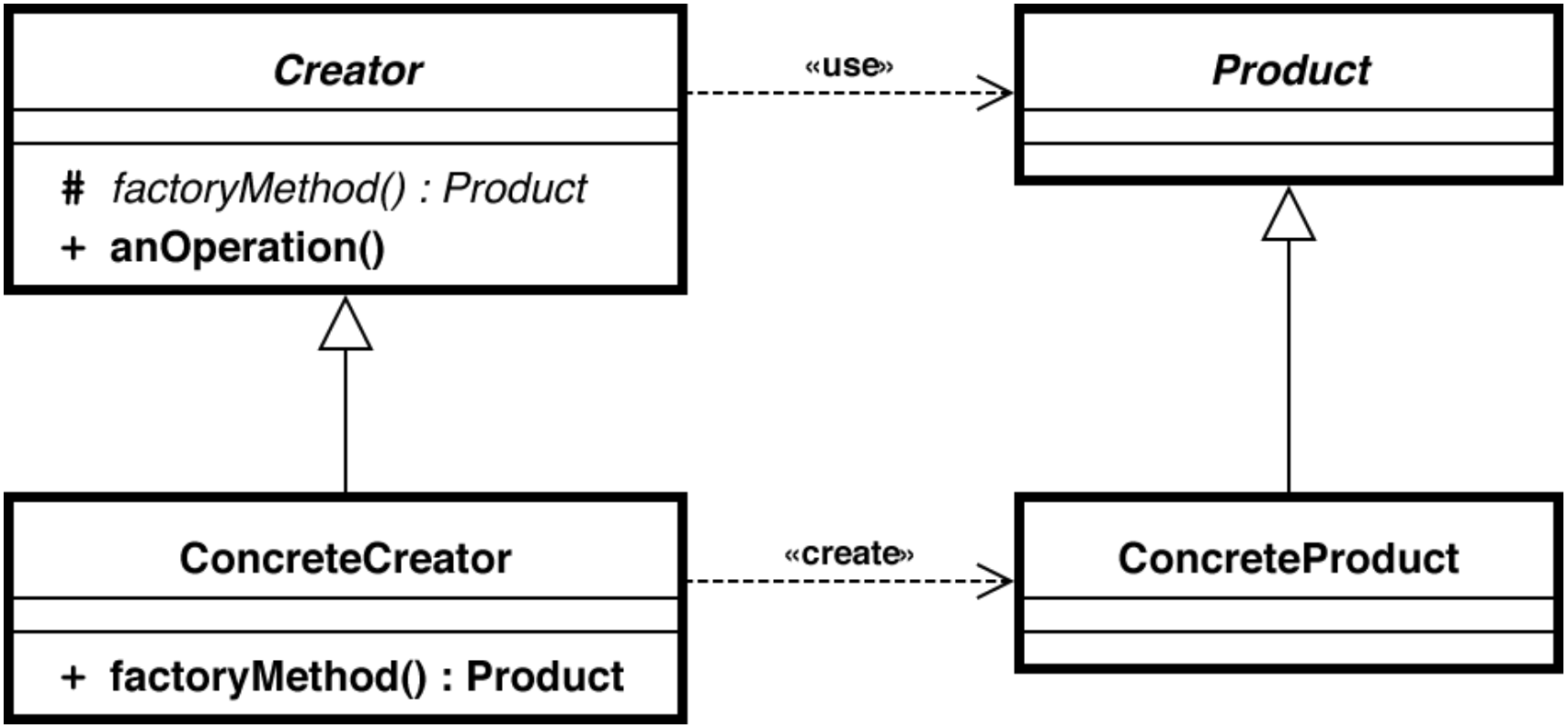
- Factory method
- Abstract factory
- Builder
- Singleton

Examples taken from:
https://www.tutorialspoint.com/design_pattern/
https://www.javatpoint.com/design-patterns-in-java

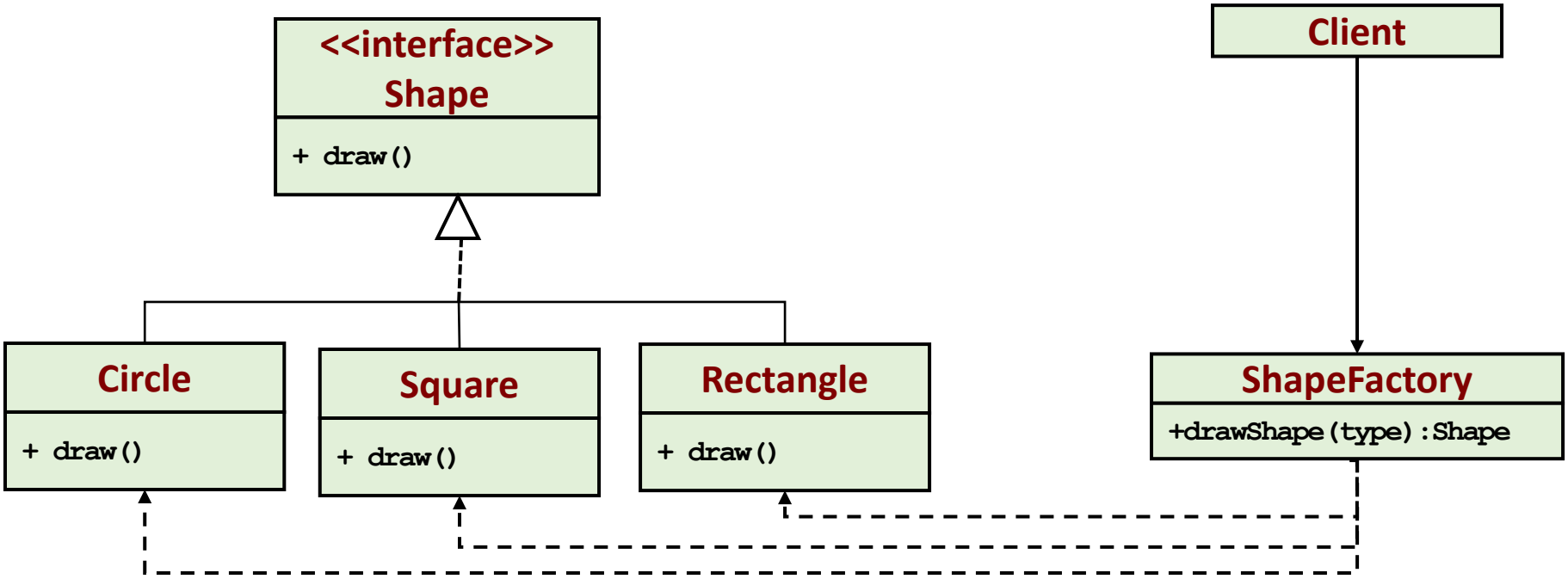
Factory Method

Intent	• Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
Problem	• A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation. • Enable the creator to defer product creation to sub-class.

Solution


Creator
factoryMethod() : Product
+ anOperation()
«use»
Product
ConcreteCreator
+ factoryMethod() : Product
«create»
ConcreteProduct

Example: Drawing



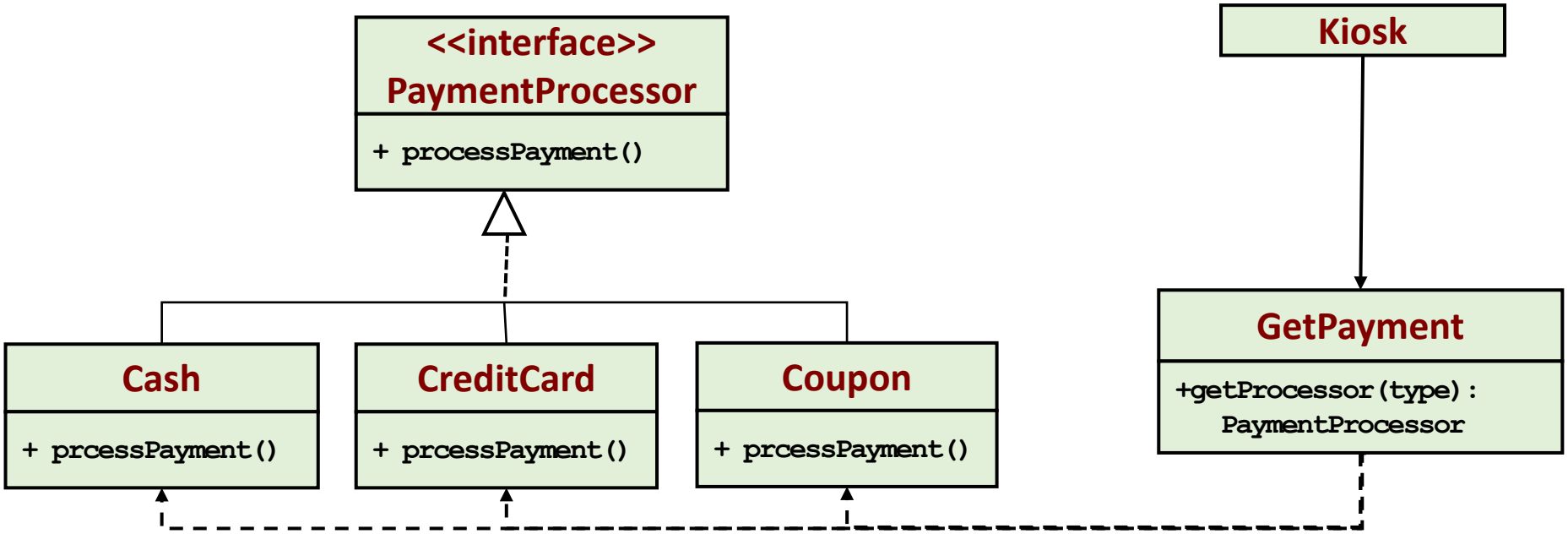
```
public interface Shape {
      void draw();
}
public class Rectangle implements Shape {
     @Override
      public void draw() {
            System.out.println("Inside Rectangle::draw() method.");
           }
 }
```

Similar implementation for Square and Circle

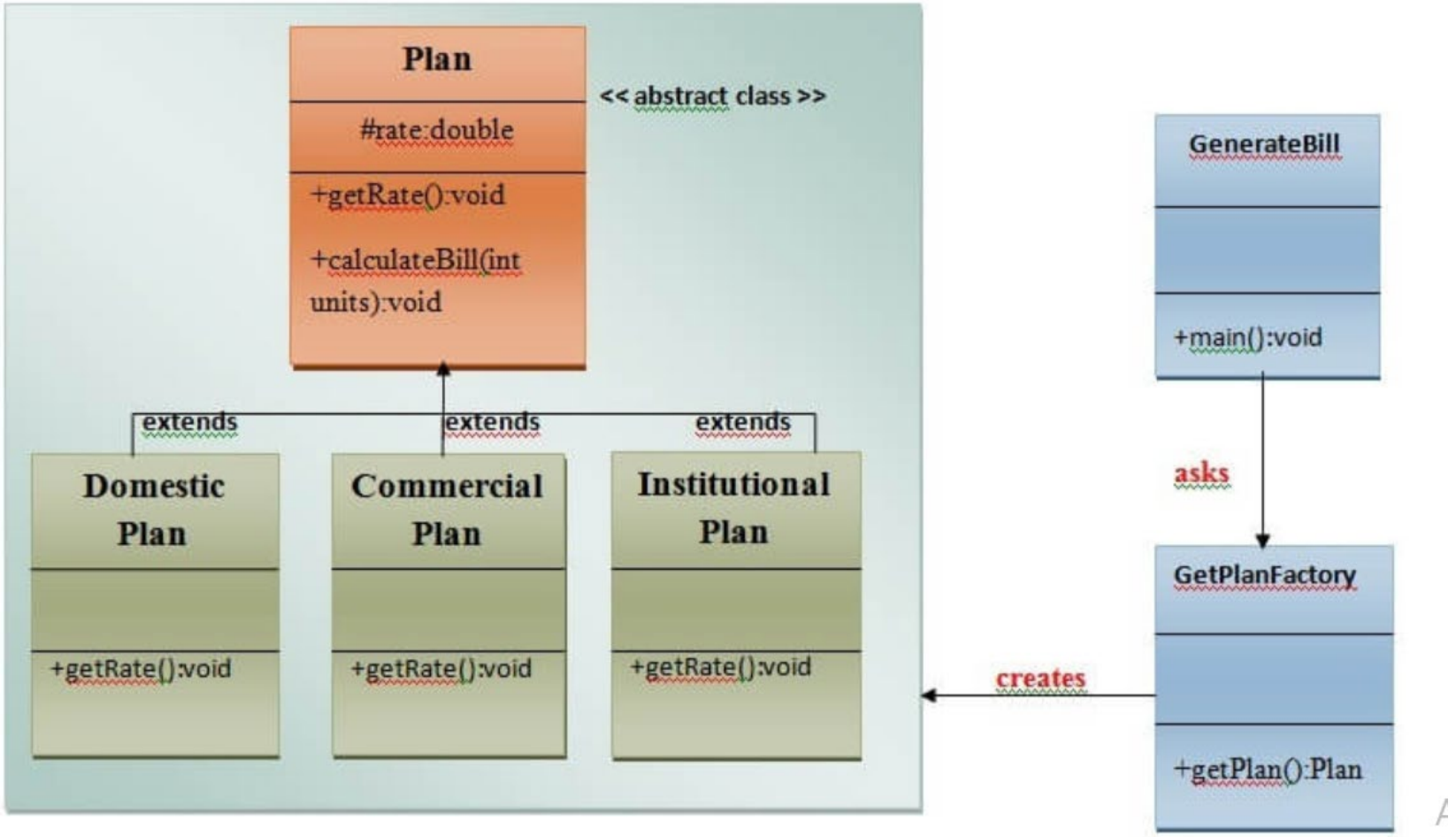
```
public class ShapeFactory {

   //use getShape method to get object of type shape
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();
      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;
   }
}
```

```
public class FactoryPatternDemo {

   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();

      //get an object of Circle and call its draw method.
      Shape shape1 = shapeFactory.getShape("CIRCLE");

      //call draw method of Circle
      shape1.draw();

      //get an object of Rectangle and call its draw method.
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

      //call draw method of Rectangle
      shape2.draw();
```

Example: Kiosk



Factory


Plan
<< abstract class >>
#rate:double
+getRate():void
+calculateBill(int units):void
extends
extends
extends
Domestic Plan
+getRate():void
Commercial Plan
+getRate():void
Institutional Plan
+getRate():void
creates
GenerateBill
+main():void
asks
GetPlanFactory
+getPlan():Plan

```java
import java.io.*;
abstract class Plan{
        protected double rate;
        abstract void getRate();

        public void calculateBill(int units){
            System.out.println(units*rate);
         }
}//end of Plan class.
```

Step 2: Create the concrete classes that extends Plan

```java
class  DomesticPlan extends Plan{
       //@override
        public void getRate(){
            rate=3.50;
       }
  }//end of DomesticPlan class.
```

```java
class  CommercialPlan extends Plan{
  //@override
   public void getRate(){
       rate=7.50;
  }
/end of CommercialPlan class.
```

```java
class  InstitutionalPlan extends Plan{
  //@override
   public void getRate(){
       rate=5.50;
  }
/end of InstitutionalPlan class.
```

```
class GetPlanFactory{

  //use getPlan method to get object of type Plan
     public Plan getPlan(String planType){
        if(planType == null){
         return null;
        }
       if(planType.equalsIgnoreCase("DOMESTICPLAN")) {
            return new DomesticPlan();
          }
        else if(planType.equalsIgnoreCase("COMMERCIALPLAN")){
            return new CommercialPlan();
         }
       else if(planType.equalsIgnoreCase("INSTITUTIONALPLAN")) {
            return new InstitutionalPlan();
       }
    return null;
  }
}//end of GetPlanFactory class.
```

```
import java.io.*;
class GenerateBill{
   public static void main(String args[])throws IOException{
     GetPlanFactory planFactory = new GetPlanFactory();

     System.out.print("Enter the name of plan for which the bill will be generated: ");
     BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

     String planName=br.readLine();
     System.out.print("Enter the number of units for bill will be calculated: ");
     int units=Integer.parseInt(br.readLine());

     Plan p = planFactory.getPlan(planName);
     //call getRate() method and calculateBill()method of DomesticPaln.

      System.out.print("Bill amount for "+planName+" of  "+units+" units is: ");
         p.getRate();
         p.calculateBill(units);
          }
   }//end of GenerateBill class.
```

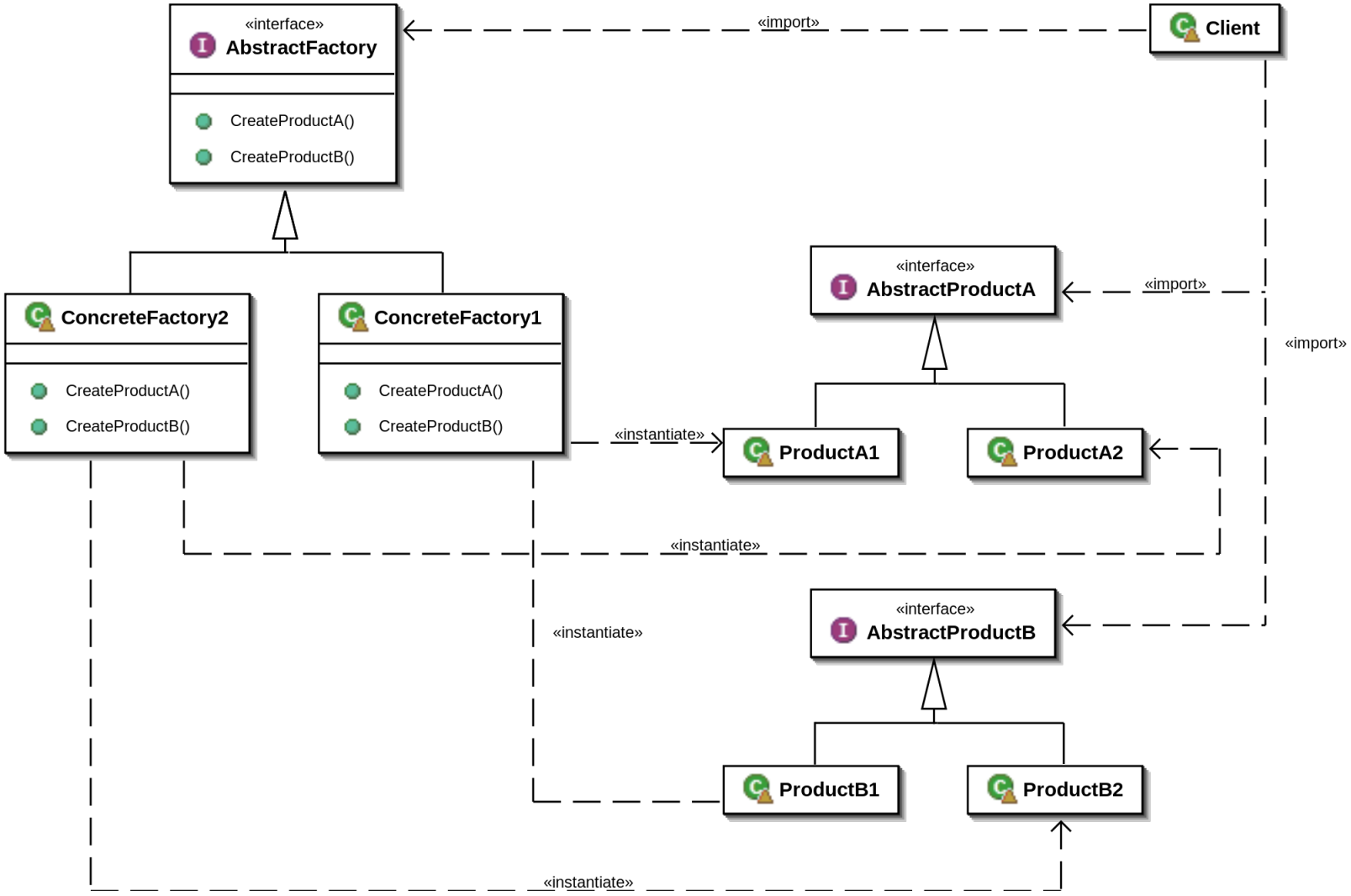
Consequences

- Factory design pattern provides approach to code for interface rather than implementation.
- Factory pattern removes the instantiation of actual implementation classes from client code. Factory pattern makes our code more robust, less coupled and easy to extend. For example, we can easily change lower class implementation because client program is unaware of this.
- Factory pattern provides abstraction between implementation and client classes through inheritance.

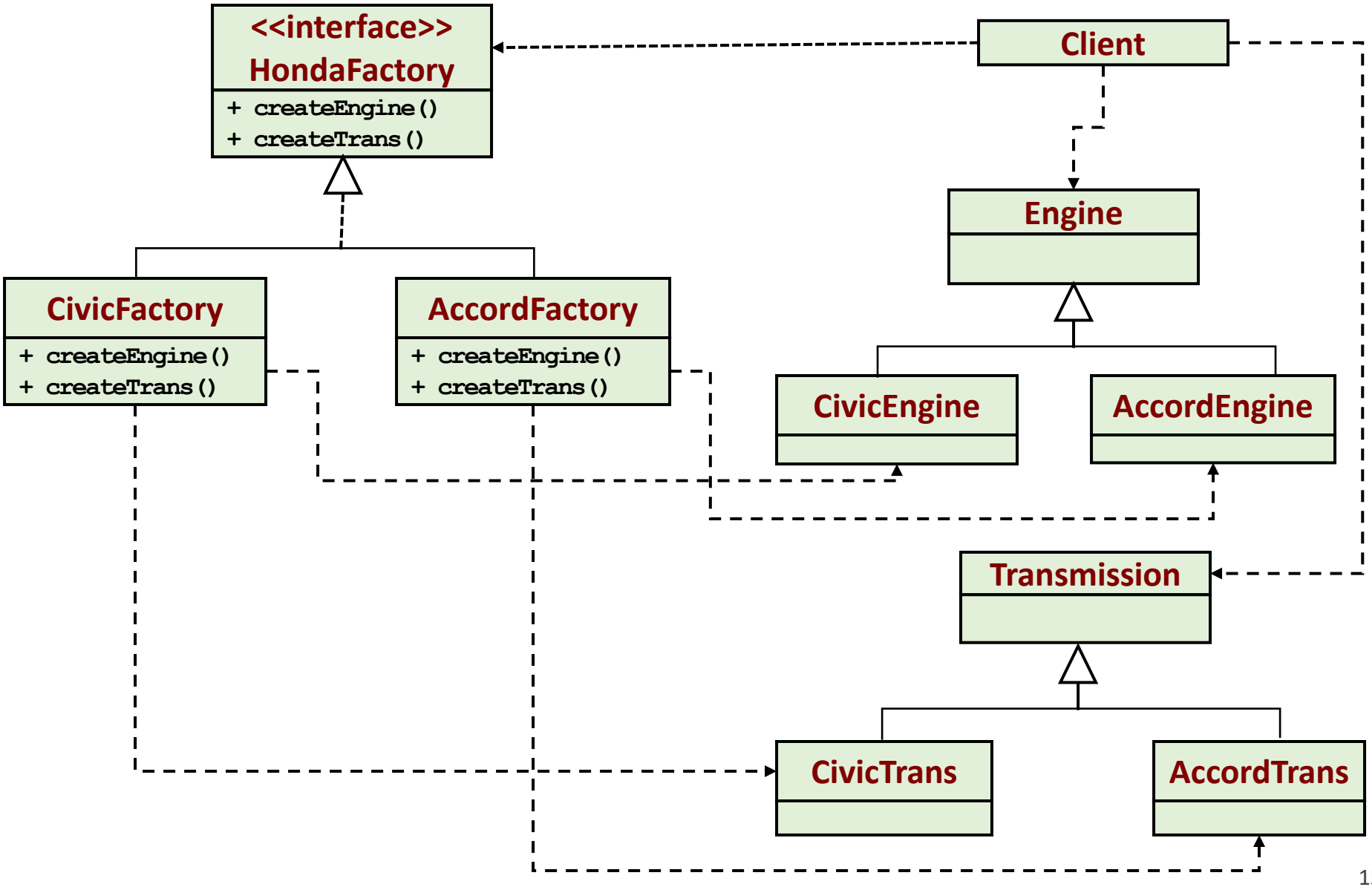
Abstract Factory

Intent	• Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
Problem	• A portable application needs to encapsulate platform dependencies • Consider an application that support multiple look and feels. • An application need to work with multiple types of DBMS

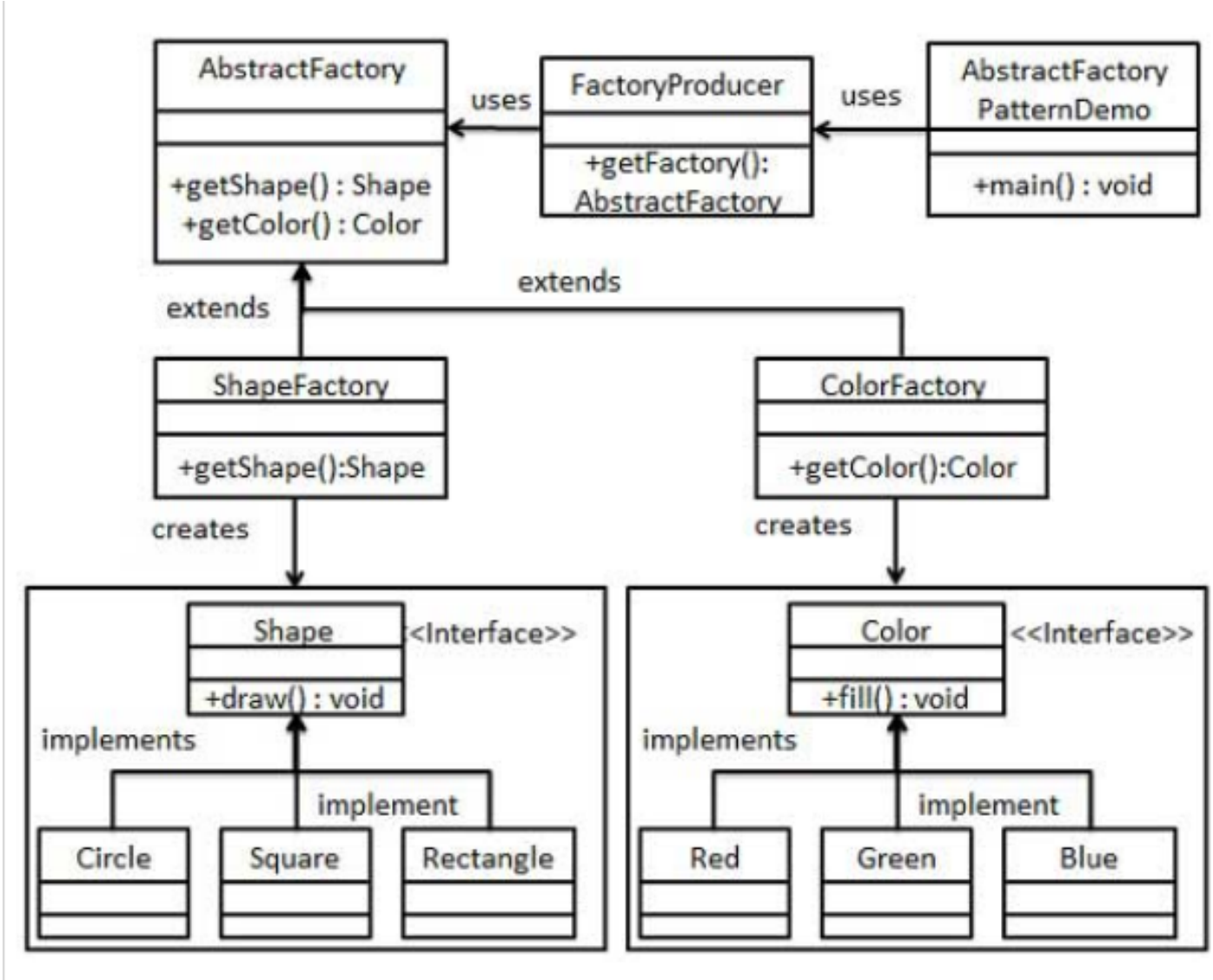
Solution


«interface»
AbstractFactory
CreateProductA()
CreateProductB()
«import»
Client
ConcreteFactory2
CreateProductA()
CreateProductB()
ConcreteFactory1
CreateProductA()
CreateProductB()
«interface»
AbstractProductA
«import»
«import»
«instantiate»
ProductA1
ProductA2
«instantiate»
«instantiate»
«interface»
AbstractProductB
ProductB1
ProductB2
«instantiate»

Example: Car Factory



https://www.tutorialspoint.com/design_pattern/design_pattern_quick_guide.htm



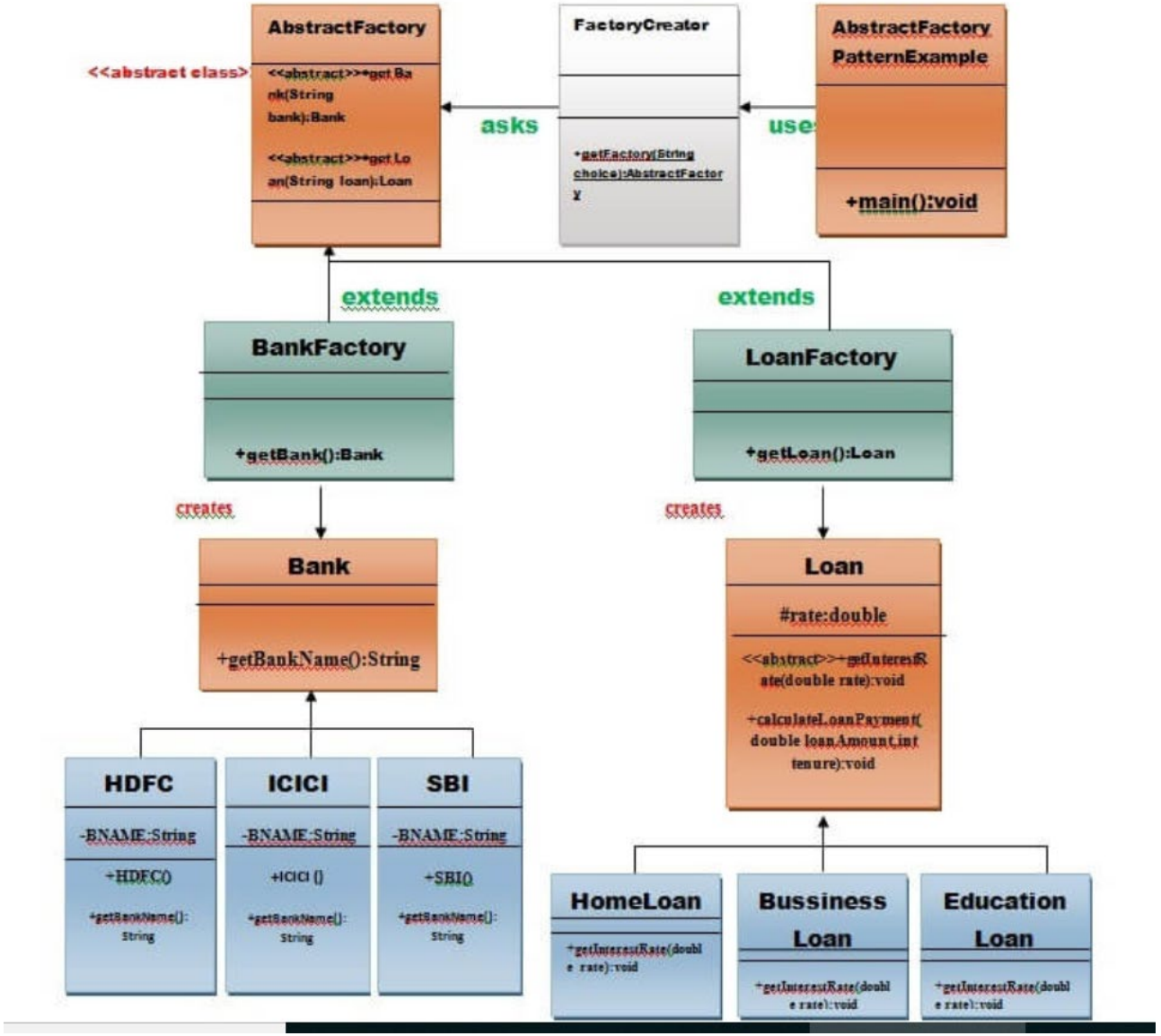
```java
public abstract class AbstractFactory {
   abstract Color getColor(String color);
   abstract Shape getShape(String shape) ;
}
```

```java
public class ShapeFactory extends AbstractFactory {

   @Override
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();
      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;
   }
```

```java
public class FactoryProducer {
   public static AbstractFactory getFactory(String choice){
      if(choice.equalsIgnoreCase("SHAPE")){
         return new ShapeFactory();
      } else if(choice.equalsIgnoreCase("COLOR")){
         return new ColorFactory();
      }
      return null;
   }
}
```

```java
public class AbstractFactoryPatternDemo {
   public static void main(String[] args) {

      //get shape factory
      AbstractFactory shapeFactory = FactoryProducer.getFactory("SHAPE"

      //get an object of Shape Circle
      Shape shape1 = shapeFactory.getShape("CIRCLE");

      //call draw method of Shape Circle
      shape1.draw();

      //get an object of Shape Rectangle
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

      //call draw method of Shape Rectangle
      shape2.draw();
```

Abstract Factory



```
import java.io.*;
interface Bank{
        String getBankName();
}
```

Step 2: Create concrete classes that implement the I

```
class HDFC implements Bank{
        private final String BNAME;
        public HDFC(){
                BNAME="HDFC BANK";
        }
        public String getBankName() {
                return BNAME;
        }
}
```

```
class ICICI implements Bank{
        private final String BNAME;
        ICICI(){
                BNAME="ICICI BANK";
        }
        public String getBankName() {
                return BNAME;
        }
}
```

```
class SBI implements Bank{
        private final String BNAME;
        public SBI(){
                BNAME="SBI BANK";
        }
        public String getBankName(){
                return BNAME;
        }
}
```

```
abstract class Loan{
  protected double rate;
  abstract void getInterestRate(double rate);
  public void calculateLoanPayment(double loanamount, int years)
  {
     /*
          to calculate the monthly loan payment i.e. EMI

          rate=annual interest rate/12*100;
          n=number of monthly installments;
          1year=12 months.
          so, n=years*12;

        */

       double EMI;
       int n;

       n=years*12;
       rate=rate/1200;
       EMI=((rate*Math.pow((1+rate),n))/((Math.pow((1+rate),n))-1))*loanamount;

System.out.println("your monthly EMI is "+ EMI +" for the amount"+loanamount+" you have borrowed");
 }
}// end of the Loan abstract class.
```

```
class HomeLoan extends Loan{
    public void getInterestRate(double r){
        rate=r;
    }
}//End of the HomeLoan class.
```

```
class BussinessLoan extends Loan{
    public void getInterestRate(double r){
        rate=r;
    }

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

```
class EducationLoan extends Loan{
    public void getInterestRate(double r){
      rate=r;
 }
}//End of the EducationLoan class.
```

Step 5: Create an abstract class (i.e AbstractFactor

```
abstract class AbstractFactory{
  public abstract Bank getBank(String bank);
  public abstract Loan getLoan(String loan);
}
```

```
class BankFactory extends AbstractFactory{
   public Bank getBank(String bank){
      if(bank == null){
         return null;
      }
      if(bank.equalsIgnoreCase("HDFC")){
         return new HDFC();
      } else if(bank.equalsIgnoreCase("ICICI")){
         return new ICICI();
      } else if(bank.equalsIgnoreCase("SBI")){
         return new SBI();
      }
      return null;
   }
  public Loan getLoan(String loan) {
      return null;
   }
}//End of the BankFactory class.
```

```
class LoanFactory extends AbstractFactory{
        public Bank getBank(String bank){
            return null;
        }

    public Loan getLoan(String loan){
     if(loan == null){
       return null;
     }
     if(loan.equalsIgnoreCase("Home")){
       return new HomeLoan();
     } else if(loan.equalsIgnoreCase("Business")){
       return new BussinessLoan();
     } else if(loan.equalsIgnoreCase("Education")){
       return new EducationLoan();
     }
     return null;
   }

}
```

```
class FactoryCreator {
    public static AbstractFactory getFactory(String choice){
     if(choice.equalsIgnoreCase("Bank")){
       return new BankFactory();
     } else if(choice.equalsIgnoreCase("Loan")){
       return new LoanFactory();
     }
     return null;
   }
}//End of the FactoryCreator.
```

```
import java.io.*;
class AbstractFactoryPatternExample {
    public static void main(String args[])throws IOException {

    BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

    System.out.print("Enter the name of Bank from where you want to take loan amount: ");
    String bankName=br.readLine();

System.out.print("\n");
System.out.print("Enter the type of loan e.g. home loan or business loan or education loan : ");

String loanName=br.readLine();
AbstractFactory bankFactory = FactoryCreator.getFactory("Bank");
Bank b=bankFactory.getBank(bankName);

System.out.print("\n");
System.out.print("Enter the interest rate for "+b.getBankName()+ ": ");

double rate=Double.parseDouble(br.readLine());
System.out.print("\n");
System.out.print("Enter the loan amount you want to take: ");

double loanAmount=Double.parseDouble(br.readLine());
System.out.print("\n");
System.out.print("Enter the number of years to pay your entire loan amount: ");
int years=Integer.parseInt(br.readLine());

System.out.print("\n");
System.out.println("you are taking the loan from "+ b.getBankName());

AbstractFactory loanFactory = FactoryCreator.getFactory("Loan");
        Loan l=loanFactory.getLoan(loanName);
        l.getInterestRate(rate);
        l.calculateLoanPayment(loanAmount,years);
 }
```

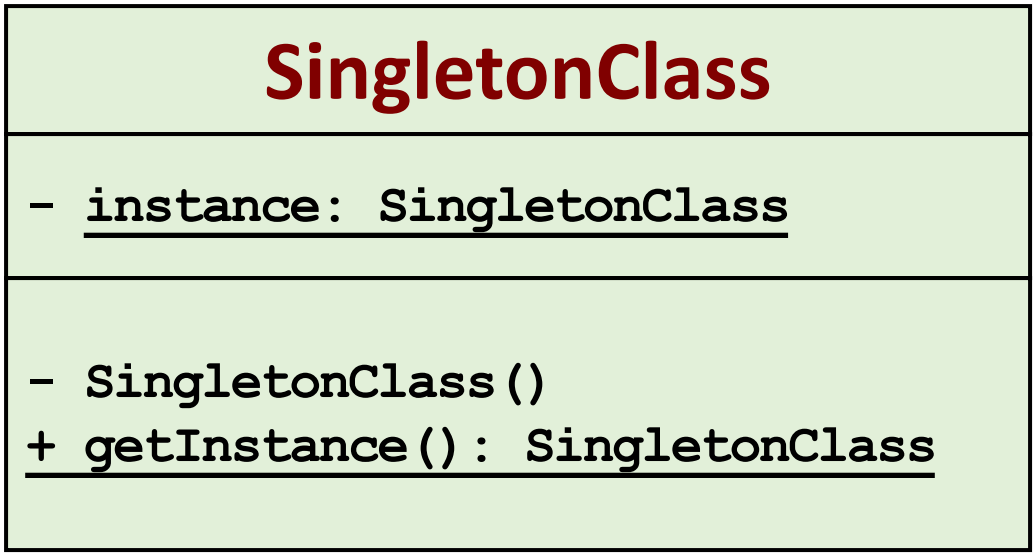
Consequences

- Isolates the concrete class
- Makes exchanging product family easy
- Promotes consistency among products
- Abstract Factory design pattern provides approach to code for interface rather than implementation.
- Abstract Factory pattern is “factory of factories” and can be easily extended to accommodate more products.
- Abstract Factory pattern is robust and avoid conditional logic of Factory pattern.
- Supporting new types of products is difficult

Singleton

Intent	• Ensure a class has only one instance, and provide a global point of access to it. • Encapsulated "just-in-time initialization" or "initialization on first use".
Problem	• Application needs one, and only one, instance of an object. Additionally, lazy initialization and global access are necessary.

Solution


SingletonClass
- instance: SingletonClass
- SingletonClass()
+ getInstance(): SingletonClass

Example: DB Connection Manger

```
public class DbConnection{

    private static DbConnection instance=null;
    private SQLConnection connection;

    private DbConnection() {
       connection = connectToDatabase(dbUser,dbPassword,dbName);
   }

    public static getDbConnection() {
      if (instance== null )
            instance = new DbConnection() ;

      return instance;
    }
}
```

```
DbConnection connection=DbConnection.getDbConnection();
```

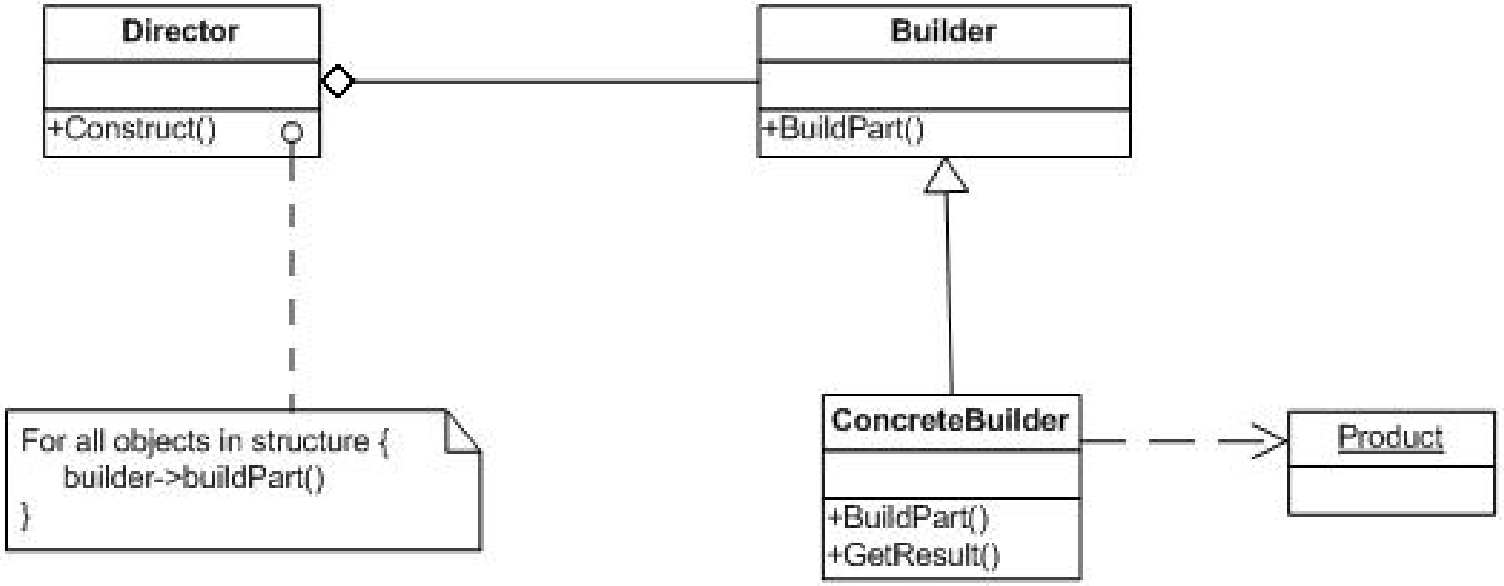
Consequences

- Controlled access to sole instance
- Reduced namespace
- Permits variable number of instances

Builder

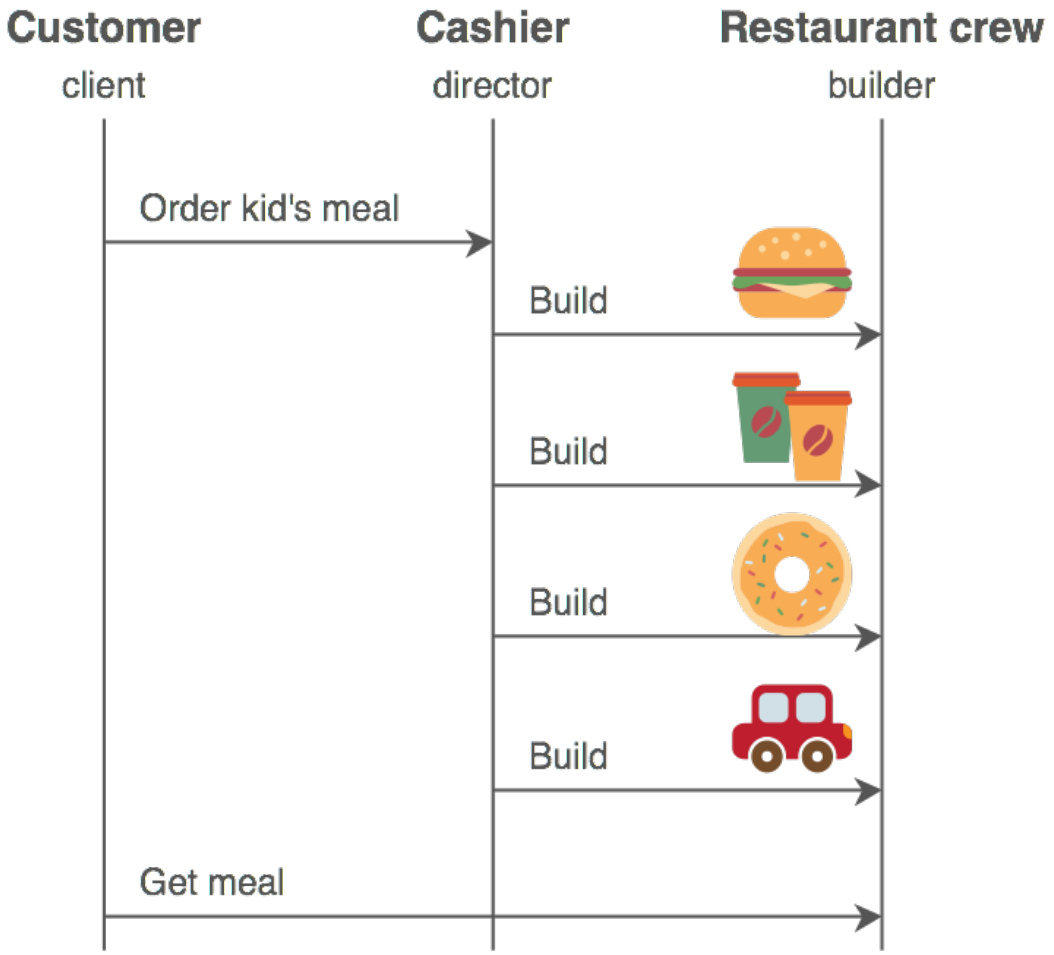
Intent	• Separate the construction of a complex object from its representation so that the same construction process can create different representations. • Parse a complex representation, create one of several targets.
Problem	• An application needs to create the elements of a complex aggregate. The specification for the aggregate exists on secondary storage and one of many representations needs to be built in primary storage.

Solution


Director
+Construct()
Builder
+BuildPart()
For all objects in structure {
builder->buildPart()
}
ConcreteBuilder
+BuildPart()
+GetResult()
Product

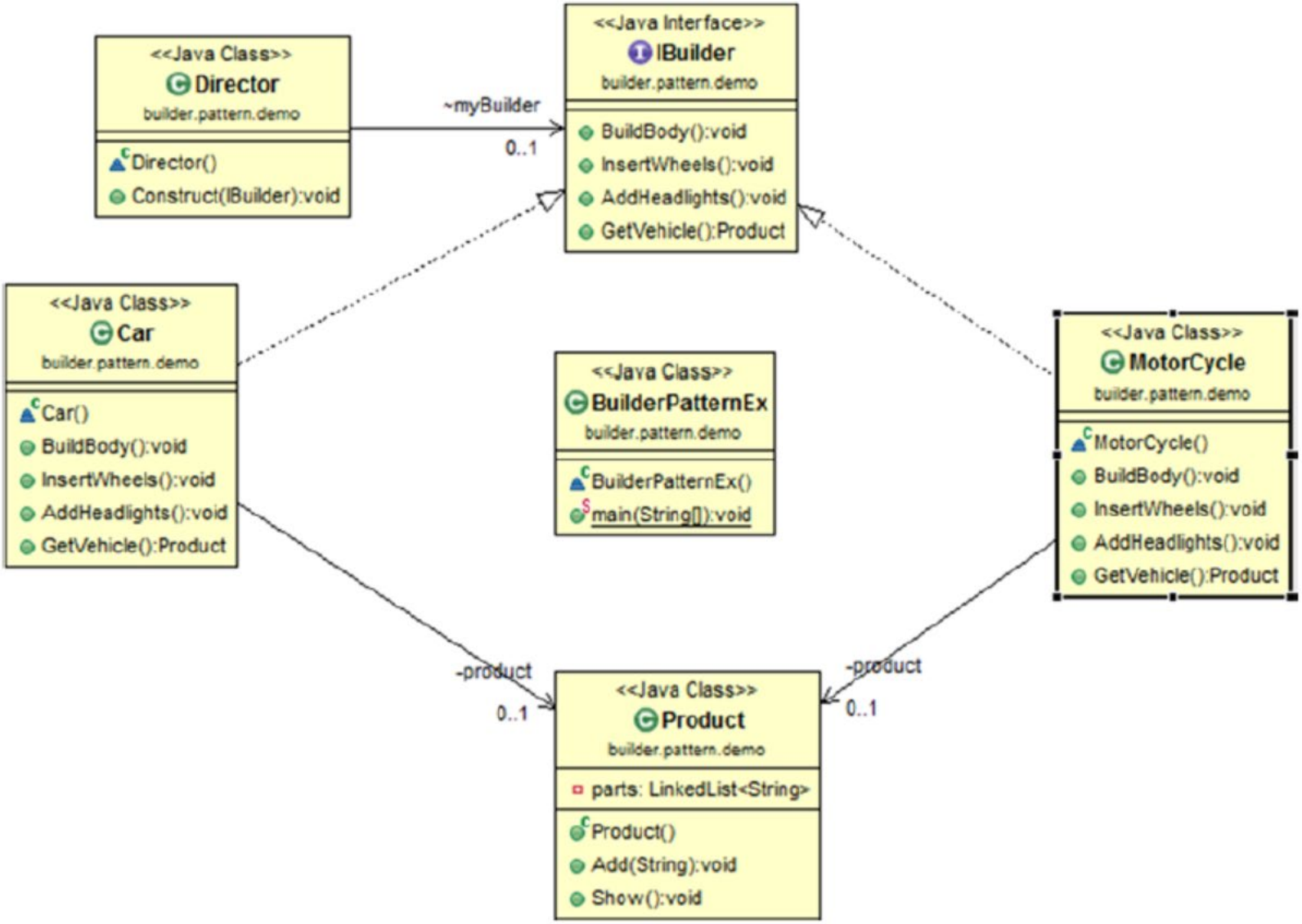
Examples

To create a computer, different parts are assembled depending upon the order received by the customer (e.g., a customer can demand a 500 GB hard disk with an Intel processor; another customer can choose a 250 GB hard disk with an AMD processor).



Consequences

- Lets you vary a product's internal representation
- Isolates construction and representation
- Gives you finer control over the construction process


<<Java Class>>
Director
builder.pattern.demo
Director()
Construct(IBuilder):void
~myBuilder
0..1
<<Java Interface>>
IBuilder
builder.pattern.demo
BuildBody():void
InsertWheels():void
AddHeadlights():void
GetVehicle():Product
<<Java Class>>
Car
builder.pattern.demo
Car()
BuildBody():void
InsertWheels():void
AddHeadlights():void
GetVehicle():Product
<<Java Class>>
BuilderPatternEx
builder.pattern.demo
BuilderPatternEx()
main(String[]):void
<<Java Class>>
MotorCycle
builder.pattern.demo
MotorCycle()
BuildBody():void
InsertWheels():void
AddHeadlights():void
GetVehicle():Product
-product
0..1
-product
0..1
<<Java Class>>
Product
builder.pattern.demo
parts: LinkedList<String>
Product()
Add(String):void
Show():void

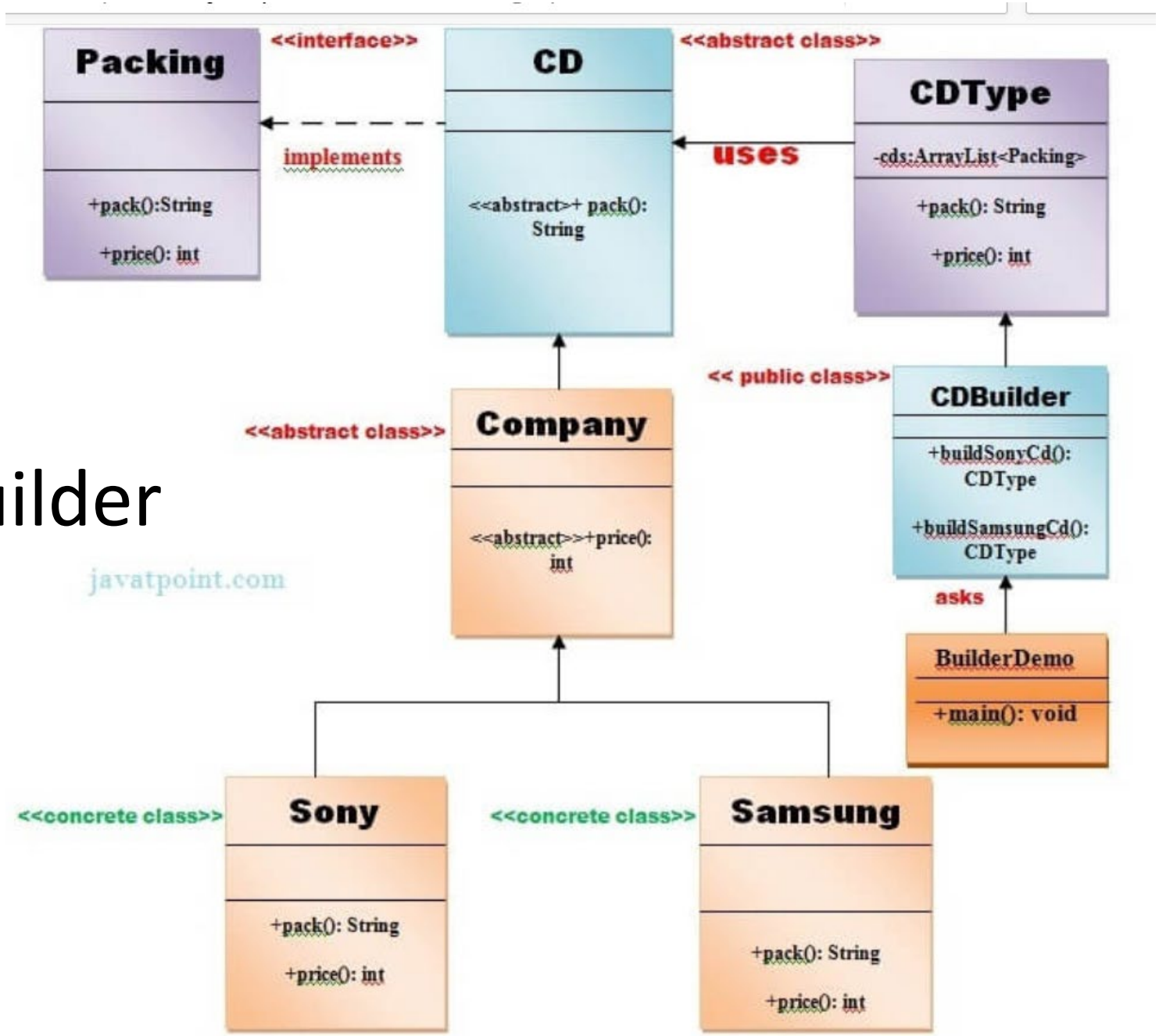
```
// Builders common interface
interface IBuilder
{
        void BuildBody();
        void InsertWheels();
        void AddHeadlights();
        Product GetVehicle();

}

// Car is ConcreteBuilder
class Car implements IBuilder
{
        private Product product = new Product();

        @Override
        public void BuildBody()
        {
                product.Add("This is a body of a Car");
        }
        @Override
        public void InsertWheels()
        {
                product.Add("4 wheels are added");
        }
        @Override
        public void AddHeadlights()
        {
                product.Add("2 Headlights are added");
        }
        public  Product GetVehicle()
        {
                return product;
        }

}
```

```
// "Product"
class Product
{
        // We can use any data structure that you prefer. We have used
        private LinkedList<String> parts;
        public Product()
        {
                parts = new LinkedList<String>();
        }

        public void Add(String part)
        {
                //Adding parts
                parts.addLast(part);
        }

        public void Show()
        {
                System.out.println("\n Product completed as below :");
                for(int i=0;i<parts.size();i++)
                {
                        System.out.println(parts.get(i));
                }
        }
}
```

```
// "Director"
class Director
{
        IBuilder myBuilder;

        // A series of steps-for the production
        public void Construct(IBuilder builder)
        {
                myBuilder=builder;
                myBuilder.BuildBody();
                myBuilder.InsertWheels();
                myBuilder.AddHeadlights();
        }
}
```

```
class BuilderPatternEx
{
        public static void main(String[] args)
        {
                System.out.println("***Builder Pattern Demo***\n");

                Director director = new Director();

                IBuilder carBuilder = new Car();
                IBuilder motorBuilder = new MotorCycle();

                // Making Car
                director.Construct(carBuilder);
                Product p1 = carBuilder.GetVehicle();
                p1.Show();

                //Making MotorCycle
                director.Construct(motorBuilder);
                Product p2 = motorBuilder.GetVehicle();
                p2.Show();
        }
}
```

Builder



File: Packing.java

```
public interface Packing {
        public String pack();
        public int price();
}
```

2) Create 2 abstract classes CD and Company

Create an abstract class CD which will implement Packing interface.

File: CD.java

```
public abstract class CD implements Packing{
public abstract String pack();
}
```

File: Company.java

```
public abstract class Company extends CD{
  public abstract int price();
}
```

File: Sony.java

```
public class Sony extends Company{
    @Override
        public int price(){
                    return 20;
      }
    @Override
    public String pack(){
            return "Sony CD";
        }
}//End of the Sony class.
```

File: Samsung.java

```
public class Samsung extends Company {
    @Override
        public int price(){
                    return 15;
    }
    @Override
    public String pack(){
            return "Samsung CD";
        }
}//End of the Samsung class.
```

File: CDType.java

```
import java.util.ArrayList;
import java.util.List;
public class CDType {
            private List<Packing> items=new ArrayList<Packing>();
            public void addItem(Packing packs) {
                  items.add(packs);
            }
            public void getCost(){
             for (Packing packs : items) {
                      packs.price();
             }
            }
            public void showItems(){
             for (Packing packing : items){
            System.out.print("CD name : "+packing.pack());
            System.out.println(", Price : "+packing.price());
          }
           }
}//End of the CDType class.
```

File: CDBuilder.java

```
public class CDBuilder {
                public CDType buildSonyCD(){
                  CDType cds=new CDType();
                  cds.addItem(new Sony());
                  return cds;
               }
               public CDType buildSamsungCD(){
              CDType cds=new CDType();
              cds.addItem(new Samsung());
              return cds;
               }
}// End of the CDBuilder class.
```

6) Create the BuilderDemo class

File: BuilderDemo.java

```
public class BuilderDemo{
 public static void main(String args[]){
   CDBuilder cdBuilder=new CDBuilder();
   CDType cdType1=cdBuilder.buildSonyCD();
   cdType1.showItems();

   CDType cdType2=cdBuilder.buildSamsungCD();
   cdType2.showItems();
 }
}
```

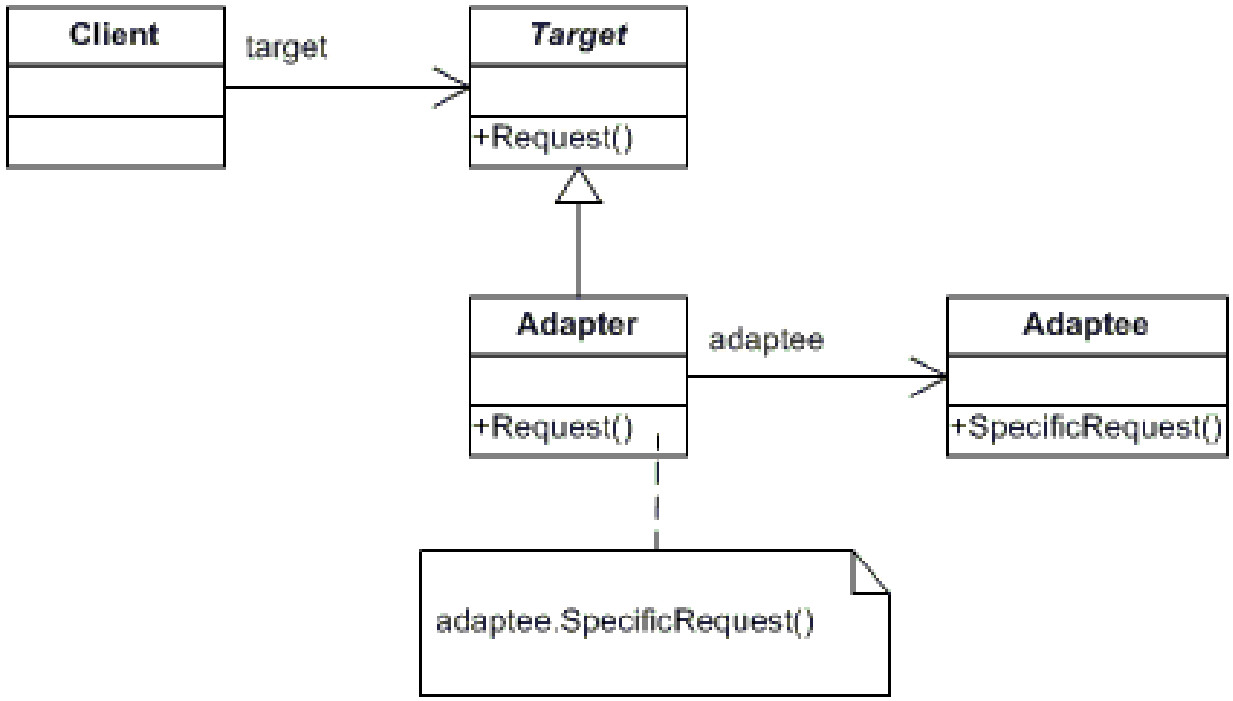
Structural Patterns

Motivation: Adapter

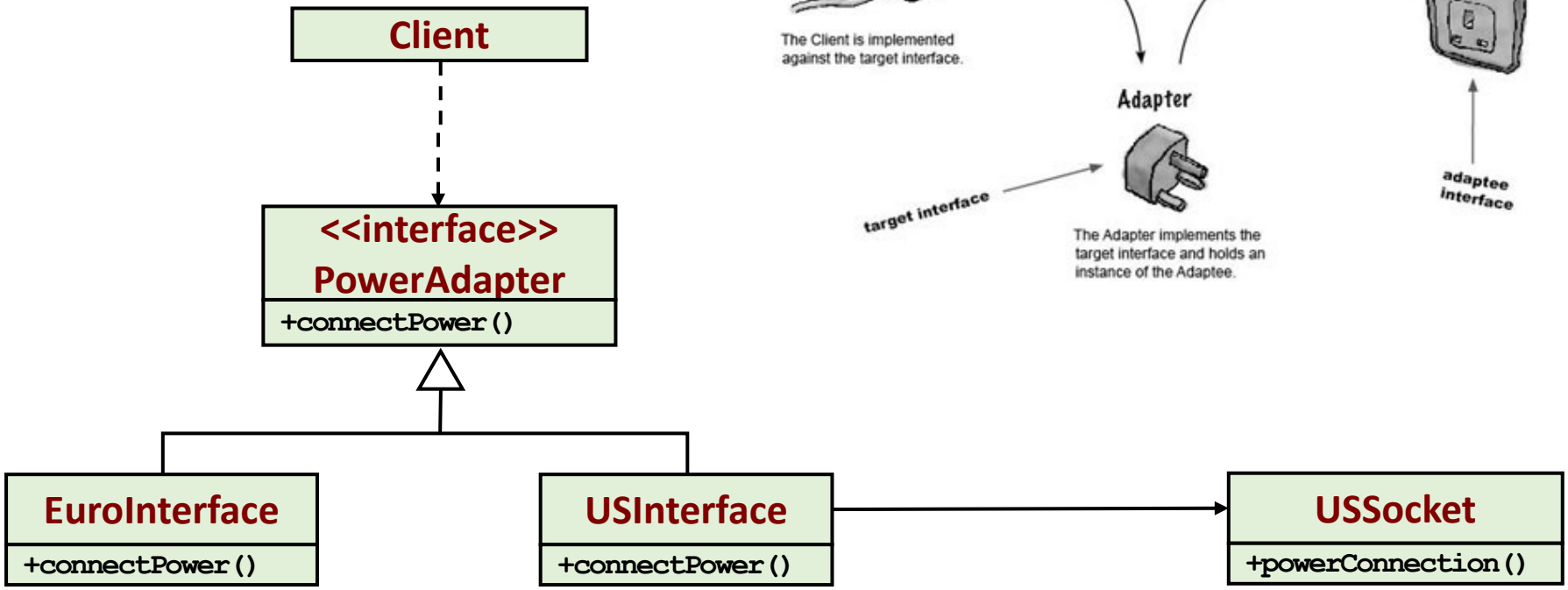
Adapter

Intent	• Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. • Wrap an existing class with a new interface.
Problem	• An "off the shelf" component offers compelling functionality that you would like to reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed.

Solution


Client
target
Target
+Request()
Adapter
adaptee
Adaptee
+Request()
+SpecificRequest()
adaptee.SpecificRequest()

Example: Car Factory



MediaPlayer.java

```
public interface MediaPlayer {
   public void play(String audioType, String fileName);
}
```

AdvancedMediaPlayer.java

```
public interface AdvancedMediaPlayer {
   public void playVlc(String fileName);
   public void playMp4(String fileName);
}
```

VlcPlayer.java

```
public class VlcPlayer implements AdvancedMediaPlayer{
   @Override
   public void playVlc(String fileName) {
      System.out.println("Playing vlc file. Name: "+ fileName);
   }

   @Override
   public void playMp4(String fileName) {
      //do nothing
   }
}
```

MediaAdapter.java

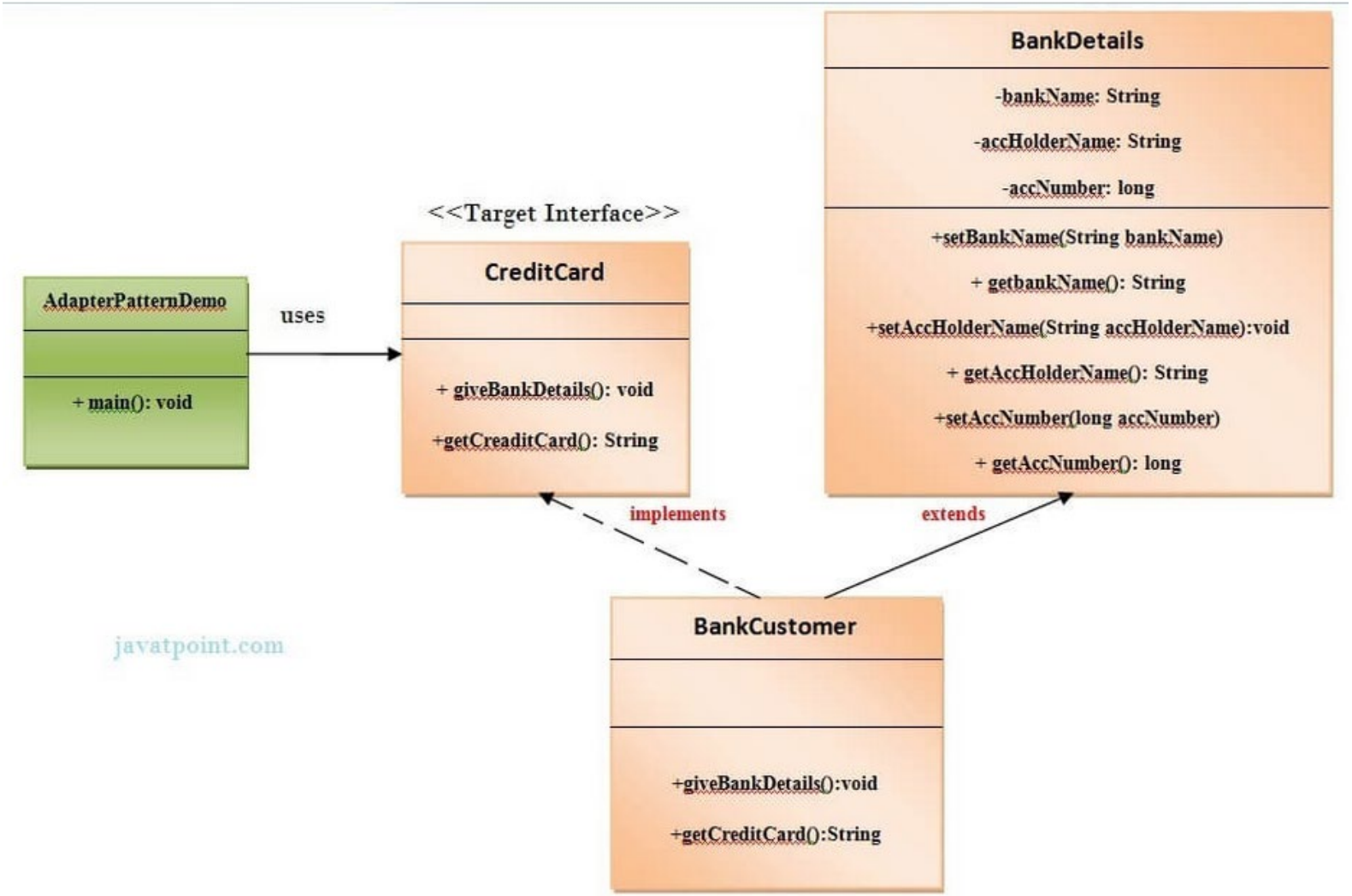
```
public class MediaAdapter implements MediaPlayer {

   AdvancedMediaPlayer advancedMusicPlayer;

   public MediaAdapter(String audioType){

      if(audioType.equalsIgnoreCase("vlc") ){
         advancedMusicPlayer = new VlcPlayer();

      }else if (audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer = new Mp4Player();
      }
   }

   @Override
   public void play(String audioType, String fileName) {

      if(audioType.equalsIgnoreCase("vlc")){
         advancedMusicPlayer.playVlc(fileName);
      }
      else if(audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer.playMp4(fileName);
      }
   }
```

AudioPlayer.java

```
public class AudioPlayer implements MediaPlayer {
   MediaAdapter mediaAdapter;

   @Override
   public void play(String audioType, String fileName) {

      //inbuilt support to play mp3 music files
      if(audioType.equalsIgnoreCase("mp3")){
         System.out.println("Playing mp3 file. Name: " + fileName);
      }

      //mediaAdapter is providing support to play other file formats
      else if(audioType.equalsIgnoreCase("vlc") || audioType.equalsIgnoreCase("mp4")
         mediaAdapter = new MediaAdapter(audioType);
         mediaAdapter.play(audioType, fileName);
      }

      else{
         System.out.println("Invalid media. " + audioType + " format not supported")
      }
   }
}
```

```
public class AdapterPatternDemo {
   public static void main(String[] args) {
      AudioPlayer audioPlayer = new AudioPlayer();

      audioPlayer.play("mp3", "beyond the horizon.mp3");
      audioPlayer.play("mp4", "alone.mp4");
      audioPlayer.play("vlc", "far far away.vlc");
      audioPlayer.play("avi", "mind me.avi");
   }
}
```

Adapter


BankDetails
-bankName: String
-accHolderName: String
-accNumber: long
+setBankName(String bankName)
+ getbankName(): String
+setAccHolderName(String accHolderName):void
+ getAccHolderName(): String
+setAccNumber(long accNumber)
+ getAccNumber(): long
<<Target Interface>>
CreditCard
+ giveBankDetails(): void
+getCreaditCard(): String
AdapterPatternDemo
+ main(): void
uses
implements
extends
BankCustomer
+giveBankDetails():void
+getCreditCard():String
javatpoint.com

```java
public interface CreditCard {
    public void giveBankDetails();
    public String getCreditCard();
}// End of the CreditCard interface.
```

```java
public class BankDetails{
    private String bankName;
    private String accHolderName;
    private long accNumber;

    public String getBankName() {
        return bankName;
    }
    public void setBankName(String bankName) {
        this.bankName = bankName;
    }
    public String getAccHolderName() {
        return accHolderName;
    }
    public void setAccHolderName(String accHolderName) {
        this.accHolderName = accHolderName;
    }
    public long getAccNumber() {
        return accNumber;
    }
    public void setAccNumber(long accNumber) {
        this.accNumber = accNumber;
    }
}// End of the BankDetails class.
```

```java
import java.io.BufferedReader;
import java.io.InputStreamReader;
public class BankCustomer extends BankDetails implements CreditCard {
 public void giveBankDetails(){
  try{
   BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

   System.out.print("Enter the account holder name :");
   String customername=br.readLine();
   System.out.print("\n");

   System.out.print("Enter the account number:");
   long accno=Long.parseLong(br.readLine());
   System.out.print("\n");

   System.out.print("Enter the bank name :");
   String bankname=br.readLine();

   setAccHolderName(customername);
   setAccNumber(accno);
   setBankName(bankname);
   }catch(Exception e){
       e.printStackTrace();
   }
  }
```

```
 public String getCreditCard() {
  long accno=getAccNumber();
  String accholdername=getAccHolderName();
  String bname=getBankName();

  return ("The Account number "+accno+" of "+accholdername+" in "+bname+ "
                  bank is valid and authenticated for issuing the credit card. ");
 }
}//End of the BankCustomer class.
```

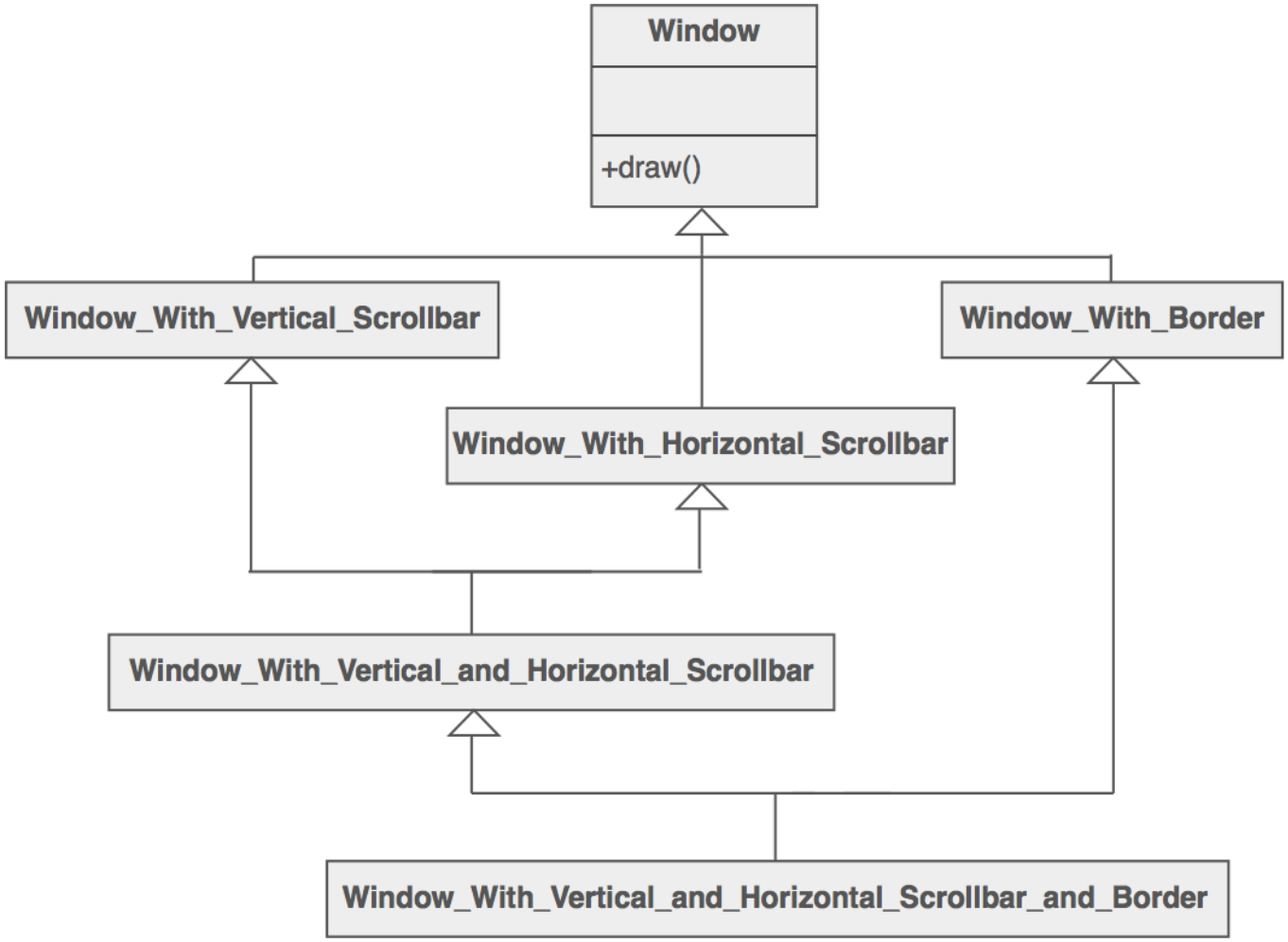
File: AdapterPatternDemo.java

```
//This is the client class.
public class AdapterPatternDemo {
 public static void main(String args[]){
  CreditCard targetInterface=new BankCustomer();
  targetInterface.giveBankDetails();
  System.out.print(targetInterface.getCreditCard());
 }
}//End of the BankCustomer class.
```

Consequences

- Allows pre-existing classes to be used in your code.
- Will not work if existing class is missing some key behavior.

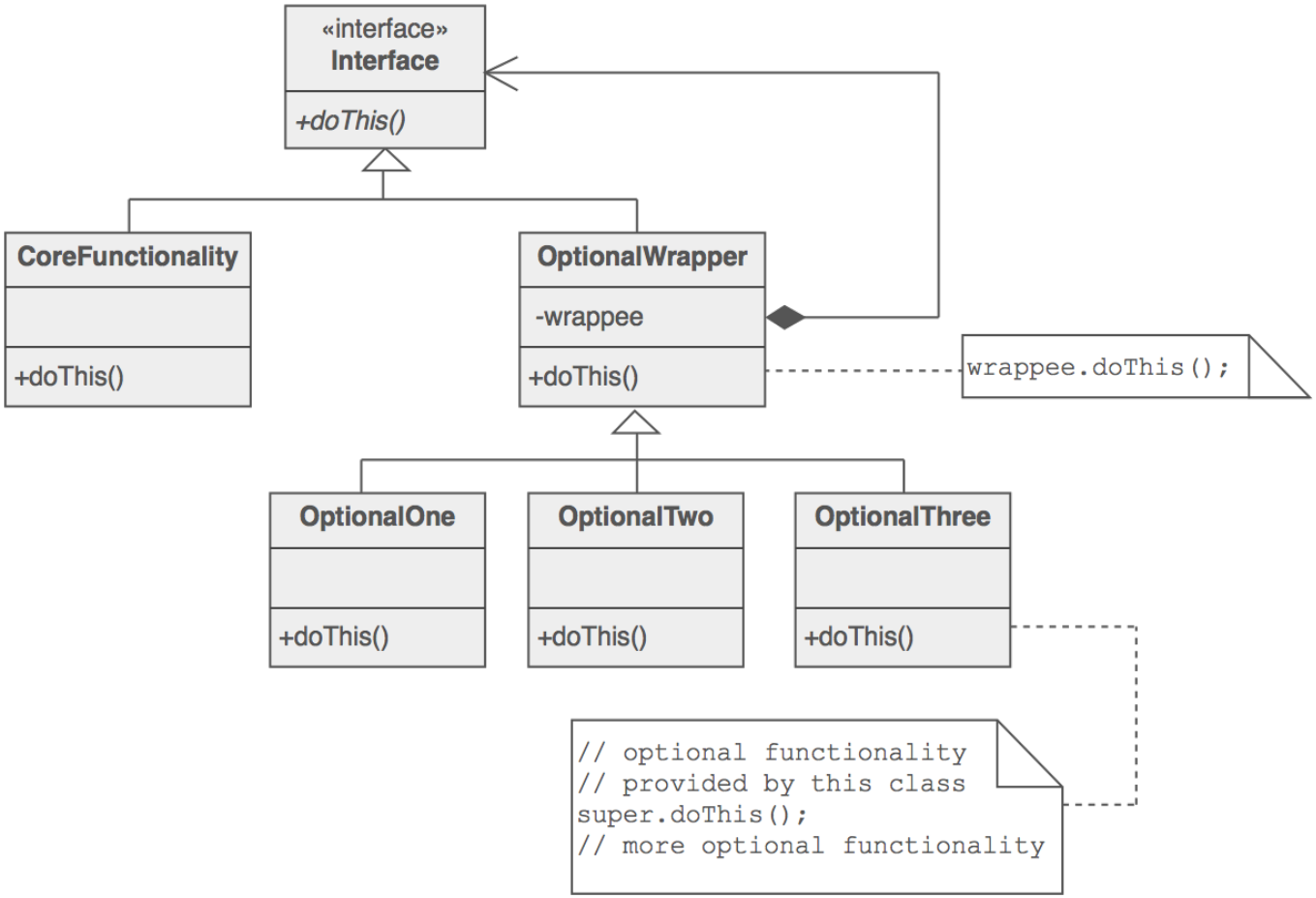
Motivation: Decorator

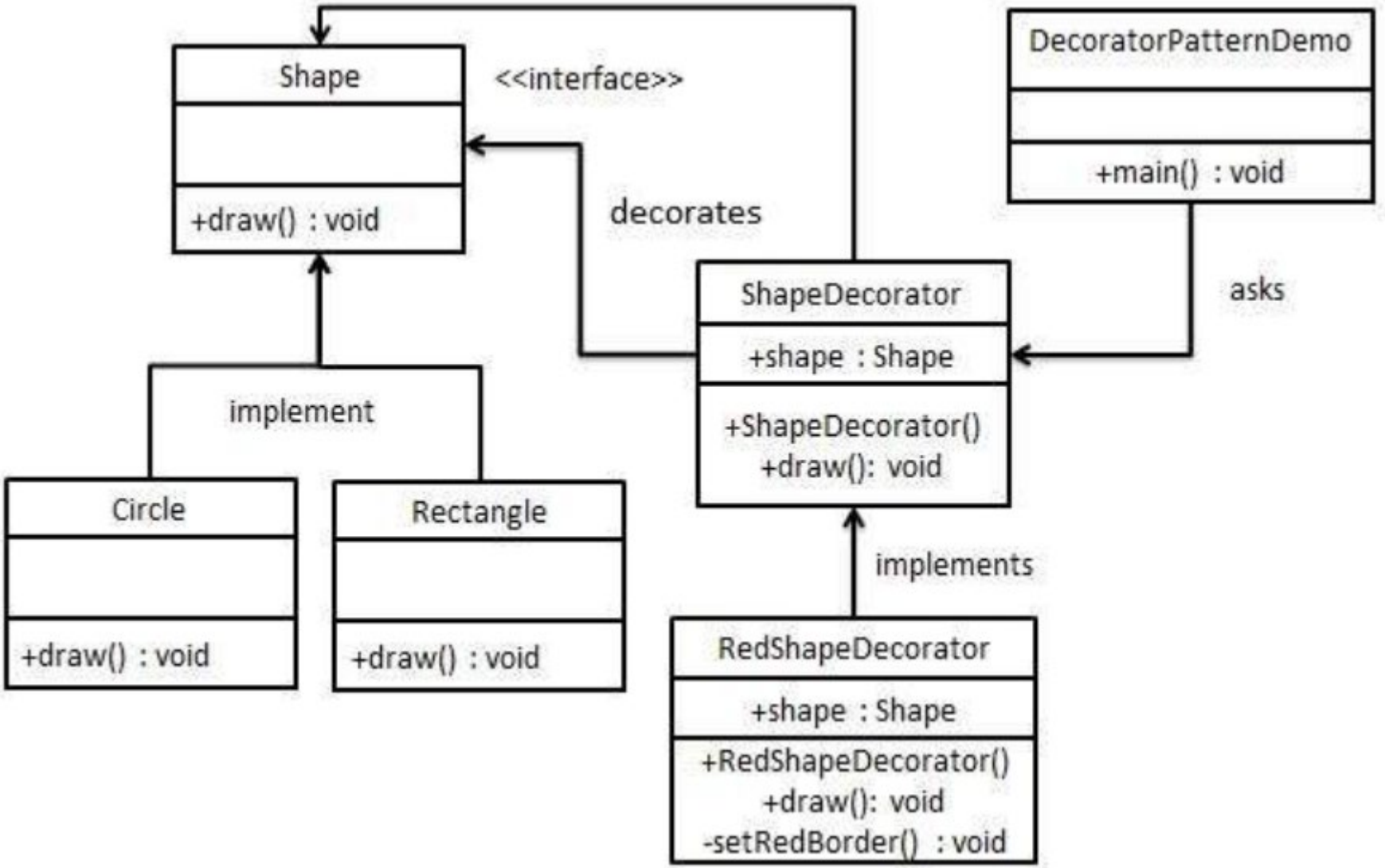


Decorator

Intent	• Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. • Client-specified embellishment of a core object by recursively wrapping it. • Wrapping a gift, putting it in a box, and wrapping the box.
Problem	• You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.

Solution


«interface»
Interface
+doThis()
CoreFunctionality
+doThis()
OptionalWrapper
-wrappee
+doThis()
wrappee.doThis();
OptionalOne
+doThis()
OptionalTwo
+doThis()
OptionalThree
+doThis()
// optional functionality
// provided by this class
super.doThis();
// more optional functionality


Shape
<<interface>>
+draw() : void
decorates
DecoratorPatternDemo
+main() : void
asks
ShapeDecorator
+shape : Shape
+ShapeDecorator()
+draw(): void
implement
Circle
+draw() : void
Rectangle
+draw() : void
implements
RedShapeDecorator
+shape : Shape
+RedShapeDecorator()
+draw(): void
-setRedBorder() : void

Shape.java

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

Circle.java

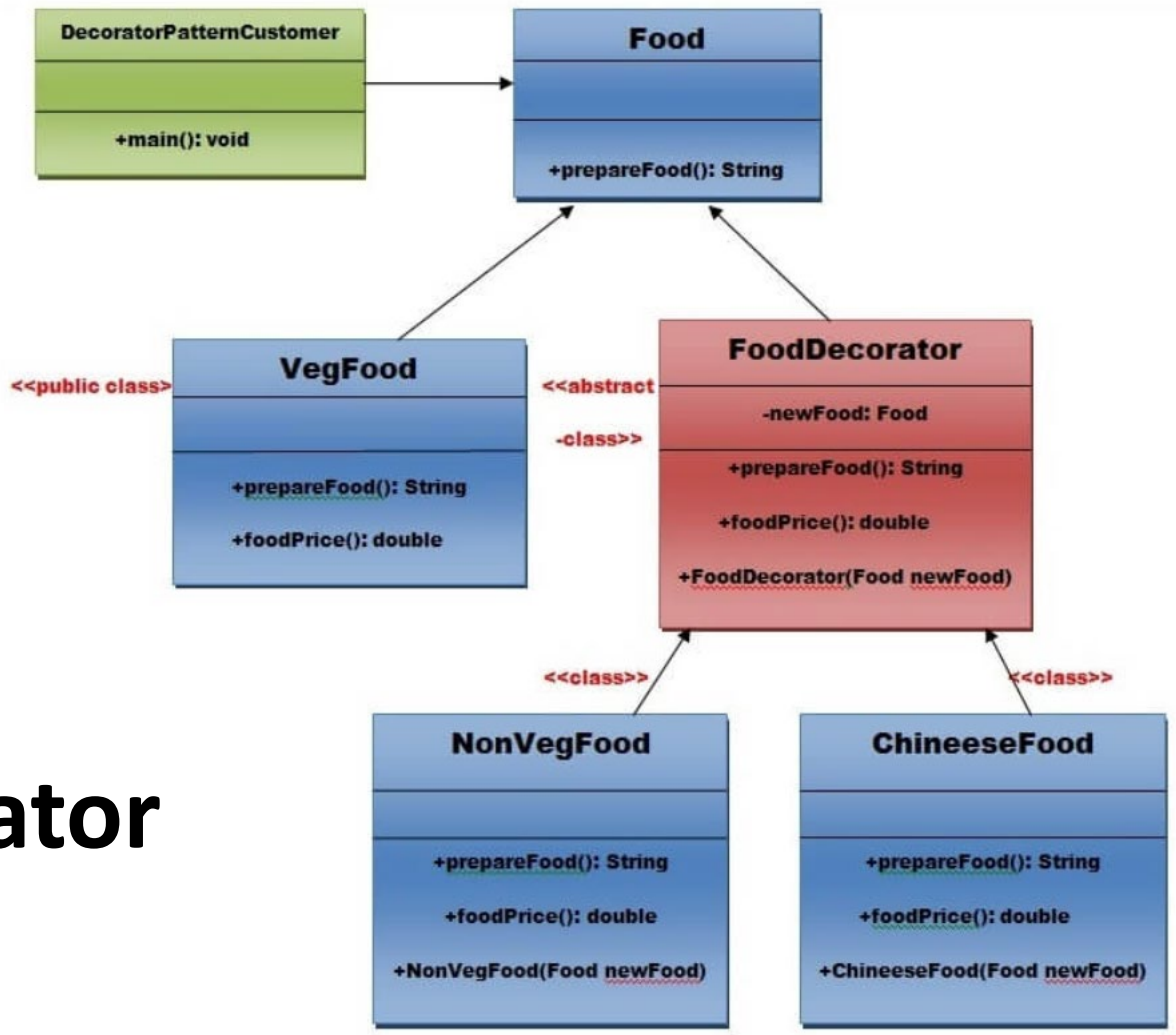
```
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Shape: Circle");
   }
}
```

```
public class RedShapeDecorator extends ShapeDecorator {

   public RedShapeDecorator(Shape decoratedShape) {
      super(decoratedShape);
   }

   @Override
   public void draw() {
      decoratedShape.draw();
      setRedBorder(decoratedShape);
   }

   private void setRedBorder(Shape decoratedShape){
      System.out.println("Border Color: Red");
   }
}
```

```java
public abstract class ShapeDecorator implements Shape {
   protected Shape decoratedShape;

   public ShapeDecorator(Shape decoratedShape){
      this.decoratedShape = decoratedShape;
   }

   public void draw(){
      decoratedShape.draw();
   }
}
```

```java
public class DecoratorPatternDemo {
   public static void main(String[] args) {

      Shape circle = new Circle();

      Shape redCircle = new RedShapeDecorator(new Circle());

      Shape redRectangle = new RedShapeDecorator(new Rectangle());
      System.out.println("Circle with normal border");
      circle.draw();

      System.out.println("\nCircle of red border");
      redCircle.draw();

      System.out.println("\nRectangle of red border");
      redRectangle.draw();
   }
}
```

Decorator


DecoratorPatternCustomer
+main(): void
Food
+prepareFood(): String
VegFood
<<public class>>
+prepareFood(): String
+foodPrice(): double
FoodDecorator
<<abstract
-class>>
-newFood: Food
+prepareFood(): String
+foodPrice(): double
+FoodDecorator(Food newFood)
<<class>>
<<class>>
NonVegFood
+prepareFood(): String
+foodPrice(): double
+NonVegFood(Food newFood)
ChineeseFood
+prepareFood(): String
+foodPrice(): double
+ChineeseFood(Food newFood)

Step 1:**Create a Food interface.**

```
public interface Food {
    public String prepareFood();
    public double foodPrice();
}// End of the Food interface.
```

Step 2: Create a **VegFood** class that will implements the **Food** interface and override its all methods.

File: VegFood.java

```
public class VegFood implements Food {
    public String prepareFood(){
        return "Veg Food";
    }

        public double foodPrice(){
        return 50.0;
    }
}
```

File: FoodDecorator.java

```
public abstract class FoodDecorator implements Food{
   private Food newFood;
   public FoodDecorator(Food newFood)  {
      this.newFood=newFood;
   }
   @Override
   public String prepareFood(){
      return newFood.prepareFood();
   }
   public double foodPrice(){
      return newFood.foodPrice();
   }
}
```

Step 4:Create a **NonVegFood concrete** class that will extend the **FoodDecorator**

File: NonVegFood.java

```
public class NonVegFood extends FoodDecorator{
   public NonVegFood(Food newFood) {
      super(newFood);
   }
   public String prepareFood(){
      return super.prepareFood() +" With Roasted Chiken and Chiken Curry  ";
   }
   public double foodPrice()   {
      return super.foodPrice()+150.0;
   }
}
```

File: ChineeseFood.java

```
public class ChineeseFood extends FoodDecorator{
  public ChineeseFood(Food newFood)    {
        super(newFood);
  }
    public String prepareFood(){
        return super.prepareFood() +" With Fried Rice and Manchurian  ";
    }
    public double foodPrice()   {
        return super.foodPrice()+65.0;
        }
}
```

File: DecoratorPatternCustomer.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class DecoratorPatternCustomer {
   private static int  choice;
   public static void main(String args[]) throws NumberFormatException, IOException    {
     do{
      System.out.print("========== Food Menu ============ \n");
      System.out.print("          1. Vegetarian Food.   \n");
      System.out.print("          2. Non-Vegetarian Food.\n");
      System.out.print("          3. Chineese Food.       \n");
      System.out.print("          4. Exit                 \n");
      System.out.print("Enter your choice: ");
      BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
      choice=Integer.parseInt(br.readLine());
      switch (choice) {
      case 1:{
             VegFood vf=new VegFood();
           System.out.println(vf.prepareFood());
           System.out.println( vf.foodPrice());
         }
         break;

           case 2:{
           Food f1=new NonVegFood((Food) new VegFood());
              System.out.println(f1.prepareFood());
           System.out.println( f1.foodPrice());
      }
```

Consequences

- More flexibility than static inheritance
- Pay-as-you-go approach
- Lots of little objects

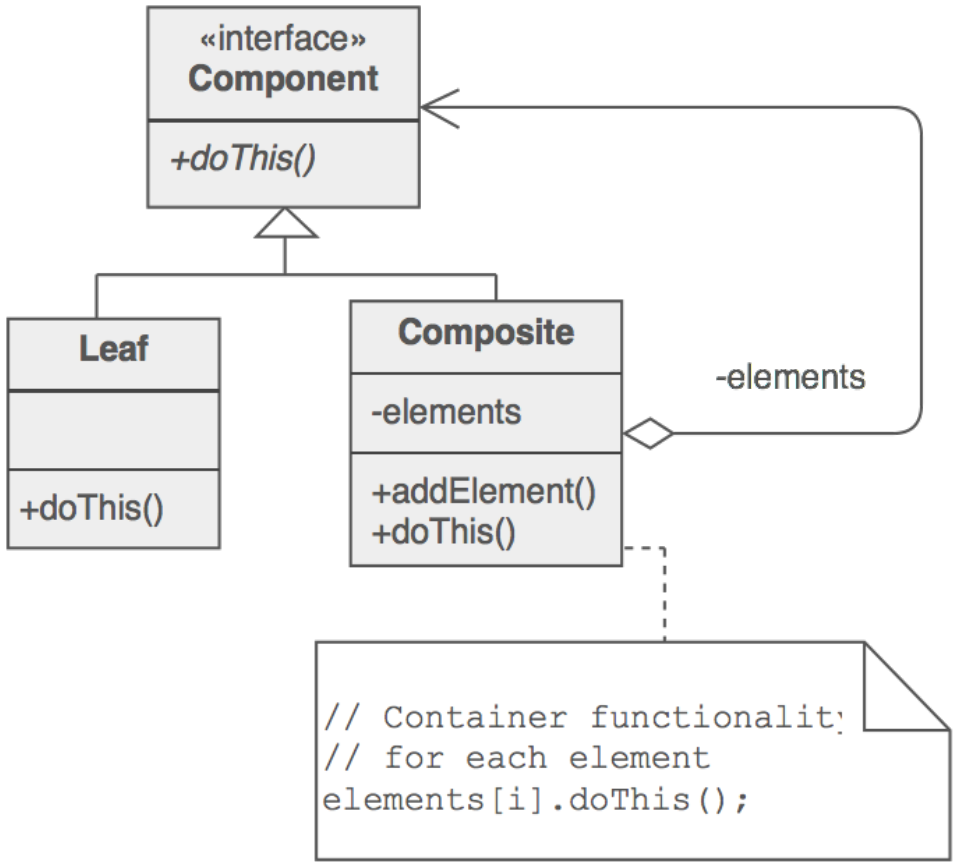
Composite

Intent	• Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. • Recursive composition
Problem	• Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects. Processing of a primitive object is handled one way, and processing of a composite object is handled differently. Having to query the "type" of each object before attempting to process it is not desirable.

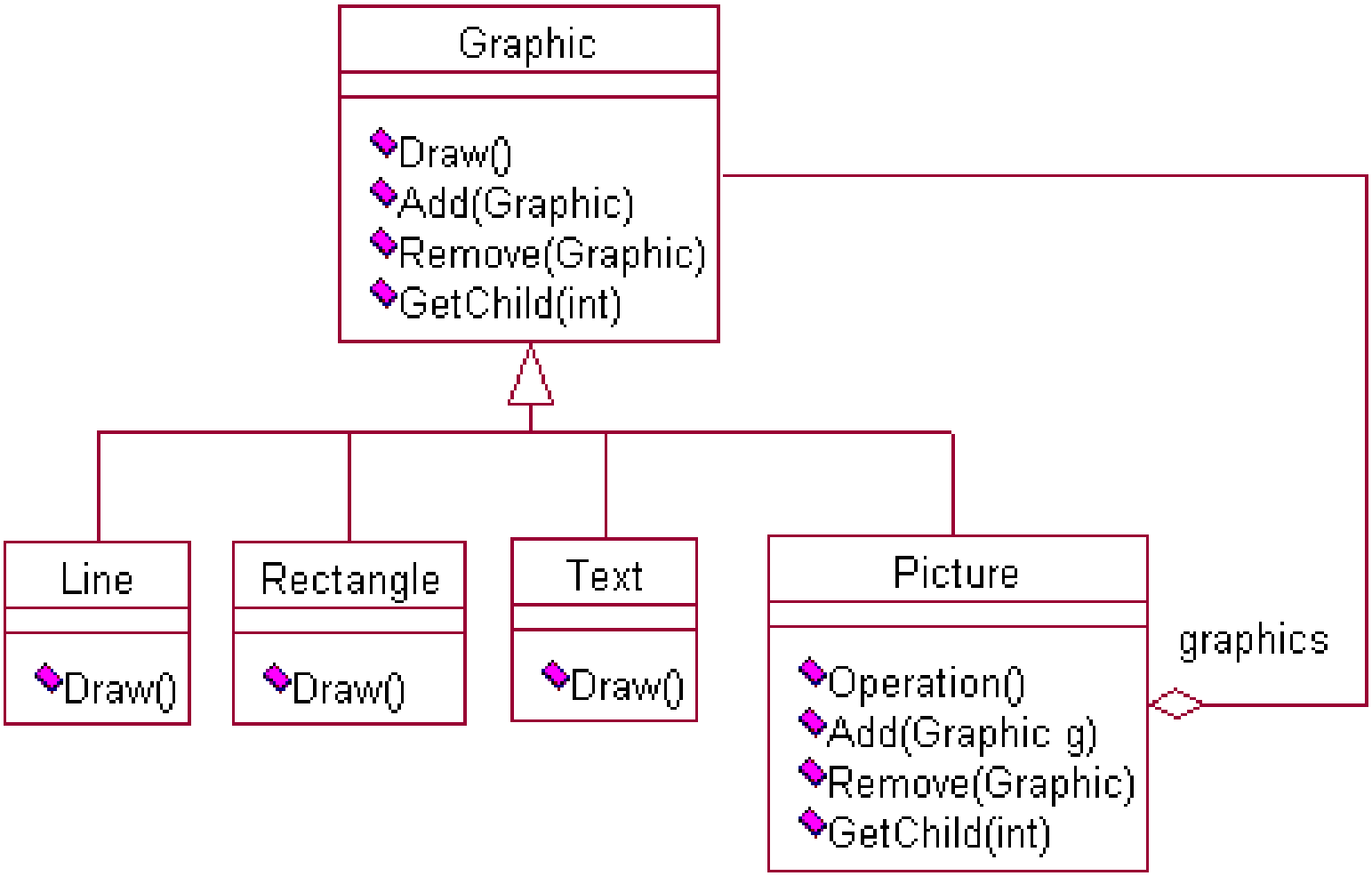
Components

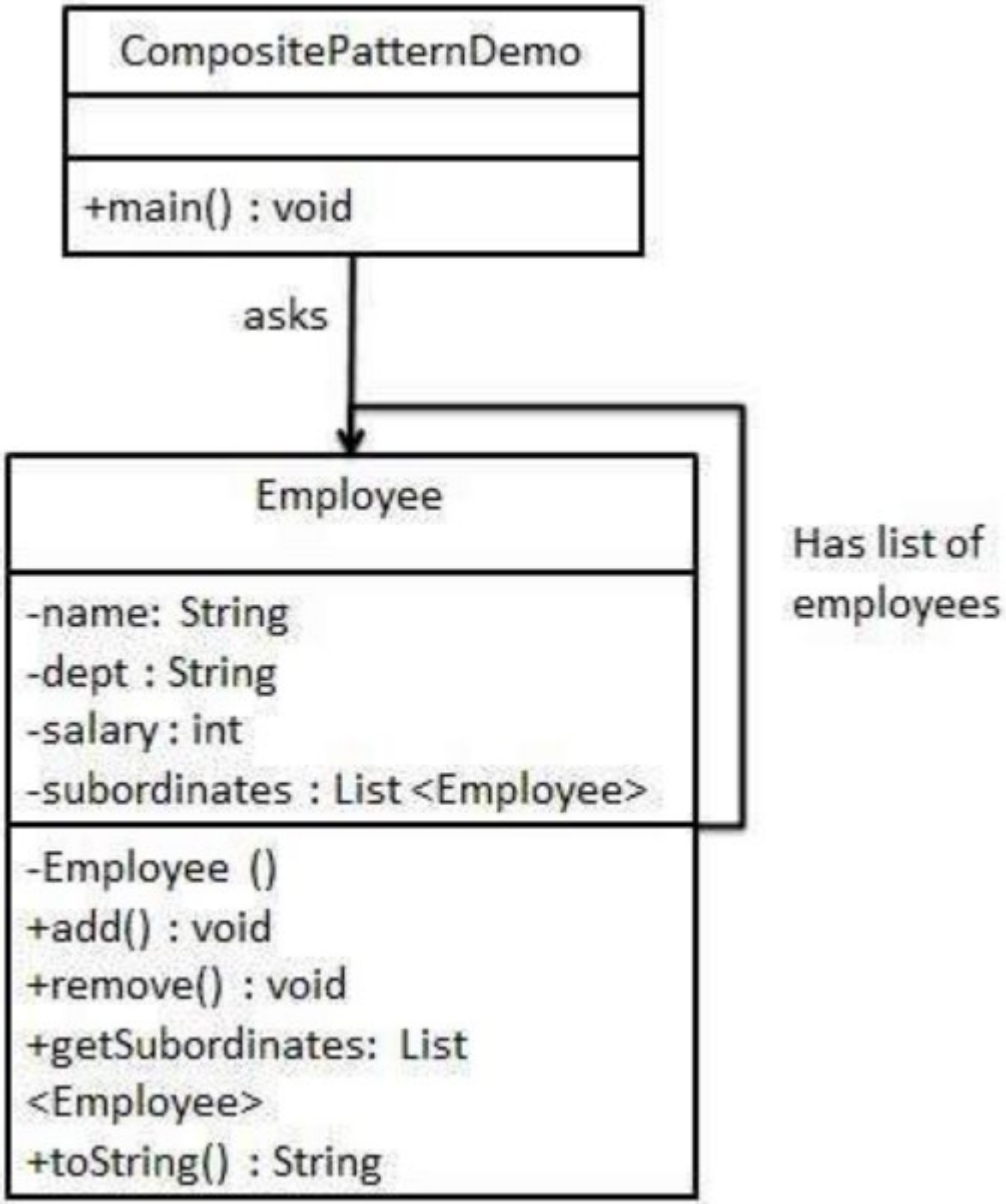
Composite Pattern consists of following objects:

- **Base Component** – Base component is the interface for all objects in the composition, client program uses base component to work with the objects in the composition. It can be an interface or an abstract class with some methods common to all the objects.
- **Leaf** – Defines the behaviour for the elements in the composition. It is the building block for the composition and implements base component. It doesn't have references to other Components.
- **Composite** – It consists of leaf elements and implements the operations in base component.



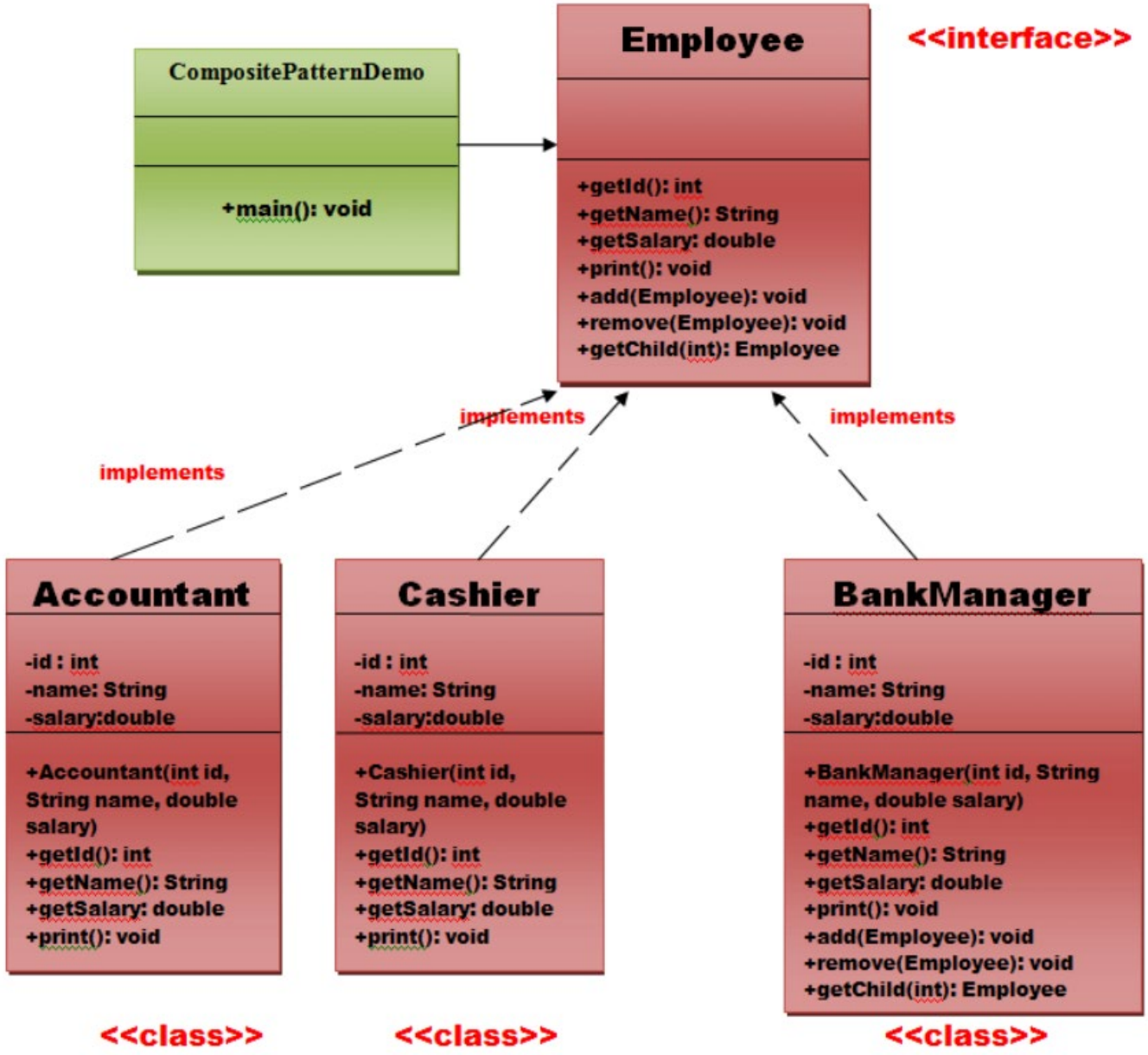
Example


Graphic
Draw()
Add(Graphic)
Remove(Graphic)
GetChild(int)
Line
Draw()
Rectangle
Draw()
Text
Draw()
Picture
Operation()
Add(Graphic g)
Remove(Graphic)
GetChild(int)
graphics


CompositePatternDemo
+main() : void
asks
Employee
-name: String
-dept : String
-salary : int
-subordinates : List <Employee>
-Employee ()
+add() : void
+remove() : void
+getSubordinates: List
<Employee>
+toString() : String
Has list of
employees

```java
public class Employee {
   private String name;
   private String dept;
   private int salary;
   private List<Employee> subordinates;

   // constructor
   public Employee(String name,String dept, int sal) {
      this.name = name;
      this.dept = dept;
      this.salary = sal;
      subordinates = new ArrayList<Employee>();
   }

   public void add(Employee e) {
      subordinates.add(e);
   }

   public void remove(Employee e) {
      subordinates.remove(e);
   }

   public List<Employee> getSubordinates(){
     return subordinates;
   }

   public String toString(){
      return ("Employee :[ Name : " + name + ", dept : " + dept + ",
   }
}
```

```
public class CompositePatternDemo {
   public static void main(String[] args) {

      Employee CEO = new Employee("John","CEO", 30000);

      Employee headSales = new Employee("Robert","Head Sales", 20000);

      Employee headMarketing = new Employee("Michel","Head Marketing", 20000);

      Employee clerk1 = new Employee("Laura","Marketing", 10000);
      Employee clerk2 = new Employee("Bob","Marketing", 10000);

      Employee salesExecutive1 = new Employee("Richard","Sales", 10000);
      Employee salesExecutive2 = new Employee("Rob","Sales", 10000);

      CEO.add(headSales);
      CEO.add(headMarketing);

      headSales.add(salesExecutive1);
      headSales.add(salesExecutive2);

      headMarketing.add(clerk1);
      headMarketing.add(clerk2);

      //print all employees of the organization
      System.out.println(CEO);

      for (Employee headEmployee : CEO.getSubordinates()) {
         System.out.println(headEmployee);

         for (Employee employee : headEmployee.getSubordinates()) {
            System.out.println(employee);
         }
      }
   }
```


CompositePatternDemo
+main(): void
Employee
<<interface>>
+getId(): int
+getName(): String
+getSalary: double
+print(): void
+add(Employee): void
+remove(Employee): void
+getChild(int): Employee
implements
implements
implements
Accountant
-id : int
-name: String
-salary:double
+Accountant(int id, String name, double salary)
+getId(): int
+getName(): String
+getSalary: double
+print(): void
Cashier
-id : int
-name: String
-salary:double
+Cashier(int id, String name, double salary)
+getId(): int
+getName(): String
+getSalary: double
+print(): void
BankManager
-id : int
-name: String
-salary:double
+BankManager(int id, String name, double salary)
+getId(): int
+getName(): String
+getSalary: double
+print(): void
+add(Employee): void
+remove(Employee): void
+getChild(int): Employee
<<class>>
<<class>>
<<class>>

```java
// this is the Employee interface i.e. Component.
public interface Employee {
    public  int getId();
    public String getName();
    public double getSalary();
      public void print();
    public void add(Employee employee);
      public void remove(Employee employee);
      public Employee getChild(int i);
}// End of the Employee interface.
```

```java
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class BankManager implements Employee {
    private int id;
    private String name;
    private double salary;

    public BankManager(int id,String name,double salary) {
     this.id=id;
     this.name = name;
     this.salary = salary;
    }
       List<Employee> employees = new ArrayList<Employee>();
    @Override
    public void add(Employee employee) {
      employees.add(employee);
    }
       @Override
    public Employee getChild(int i) {
     return employees.get(i);
    }
    @Override
    public void remove(Employee employee) {
     employees.remove(employee);
    }
```

```
public int getId()  {
 return id;
}
@Override
public String getName() {
 return name;
}
@Override
public double getSalary() {
 return salary;
}
@Override
public void print() {
 System.out.println("==================
 System.out.println("Id ="+getId());
 System.out.println("Name ="+getName());
 System.out.println("Salary ="+getSalary());
 System.out.println("==================

 Iterator<Employee> it = employees.iterator();

    while(it.hasNext())  {
      Employee employee = it.next();
      employee.print();
    }
}
}// End of the BankManager class.
```

File: Cashier.java

```
public  class Cashier implements Employee{
  /*
     In this class,there are many methods which are not applicable to cashier because
     it is a leaf node.
   */
     private int id;
       private String name;
     private double salary;
     public Cashier(int id,String name,double salary)  {
       this.id=id;
       this.name = name;
       this.salary = salary;
     }
     @Override
     public void add(Employee employee) {
       //this is leaf node so this method is not applicable to this class.
     }
     @Override
     public Employee getChild(int i) {
       //this is leaf node so this method is not applicable to this class.
       return null;
     }
     @Override
     public int getId() {
       // TODO Auto-generated method stub
       return id;
     }
```

```
    public String getName() {
        return name;
    }
    @Override
    public double getSalary() {
        return salary;
    }
    @Override
    public void print() {
        System.out.println("=================
        System.out.println("Id ="+getId());
        System.out.println("Name ="+getName());
        System.out.println("Salary ="+getSalary());
        System.out.println("=================
    }
    @Override
    public void remove(Employee employee) {
        //this is leaf node so this method is not applicab
    }
}
```

File: Accountant.java

```
public class Accountant implements Employee{
/*
   In this class,there are many methods which are not applicable to cashier becaus
   it is a leaf node.
*/
   private int id;
   private String name;
   private double salary;
  public Accountant(int id,String name,double salary)  {
      this.id=id;
      this.name = name;
      this.salary = salary;
  }
  @Override
  public void add(Employee employee) {
      //this is leaf node so this method is not applicable to this class.
  }
  @Override
  public Employee getChild(int i) {
      //this is leaf node so this method is not applicable to this class.
      return null;
  }
  @Override
   public int getId() {
       // TODO Auto-generated method stub
       return id;
   }
```

```
    public String getName() {
        return name;
    }
    @Override
    public double getSalary() {
        return salary;
    }
    @Override
    public void print() {
        System.out.println("============================");
        System.out.println("Id ="+getId());
        System.out.println("Name ="+getName());
        System.out.println("Salary ="+getSalary());
        System.out.println("============================");
    }
   @Override
    public void remove(Employee employee) {
        //this is leaf node so this method is not applicable to this class.
    }
}
```

File: CompositePatternDemo.java

```
public class CompositePatternDemo {
   public static void main(String args[]){
       Employee emp1=new Cashier(101,"Sohan Kumar", 20000.0);
       Employee emp2=new Cashier(102,"Mohan Kumar", 25000.0);
       Employee emp3=new Accountant(103,"Seema Mahiwal", 30000.0);
       Employee manager1=new BankManager(100,"Ashwani Rajput",100000.0);

        manager1.add(emp1);
        manager1.add(emp2);
        manager1.add(emp3);
        manager1.print();
      }
}
```

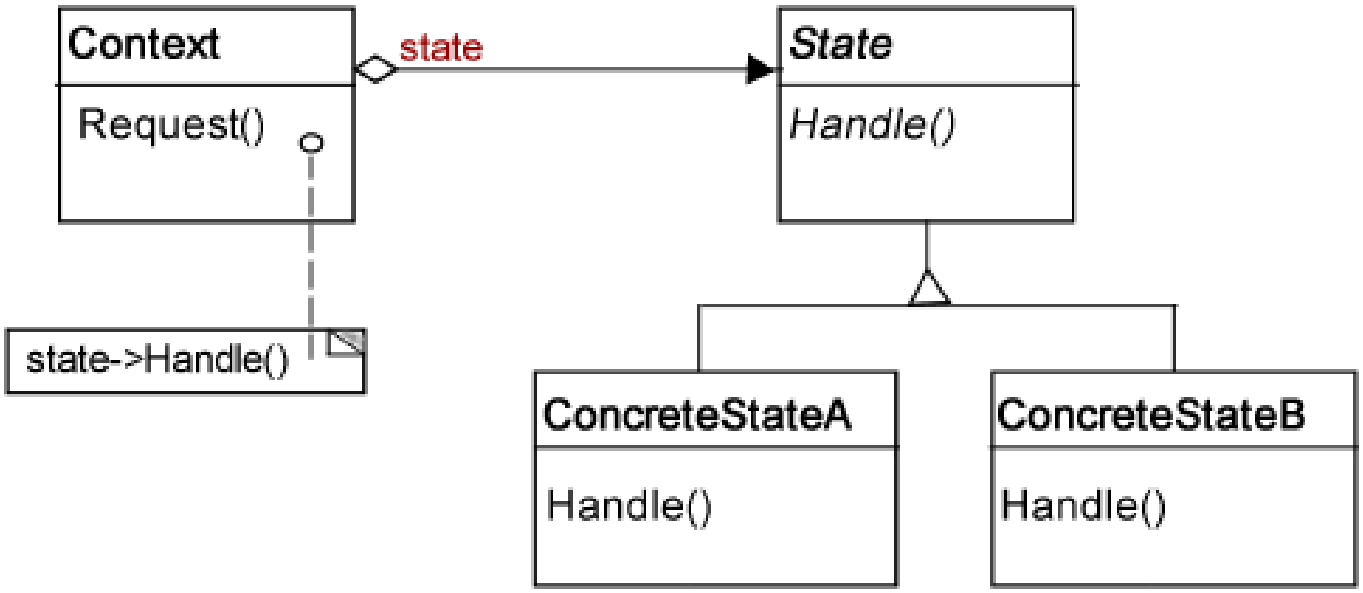
Consequences

- Makes the client simple
- Easier to add new kinds of components

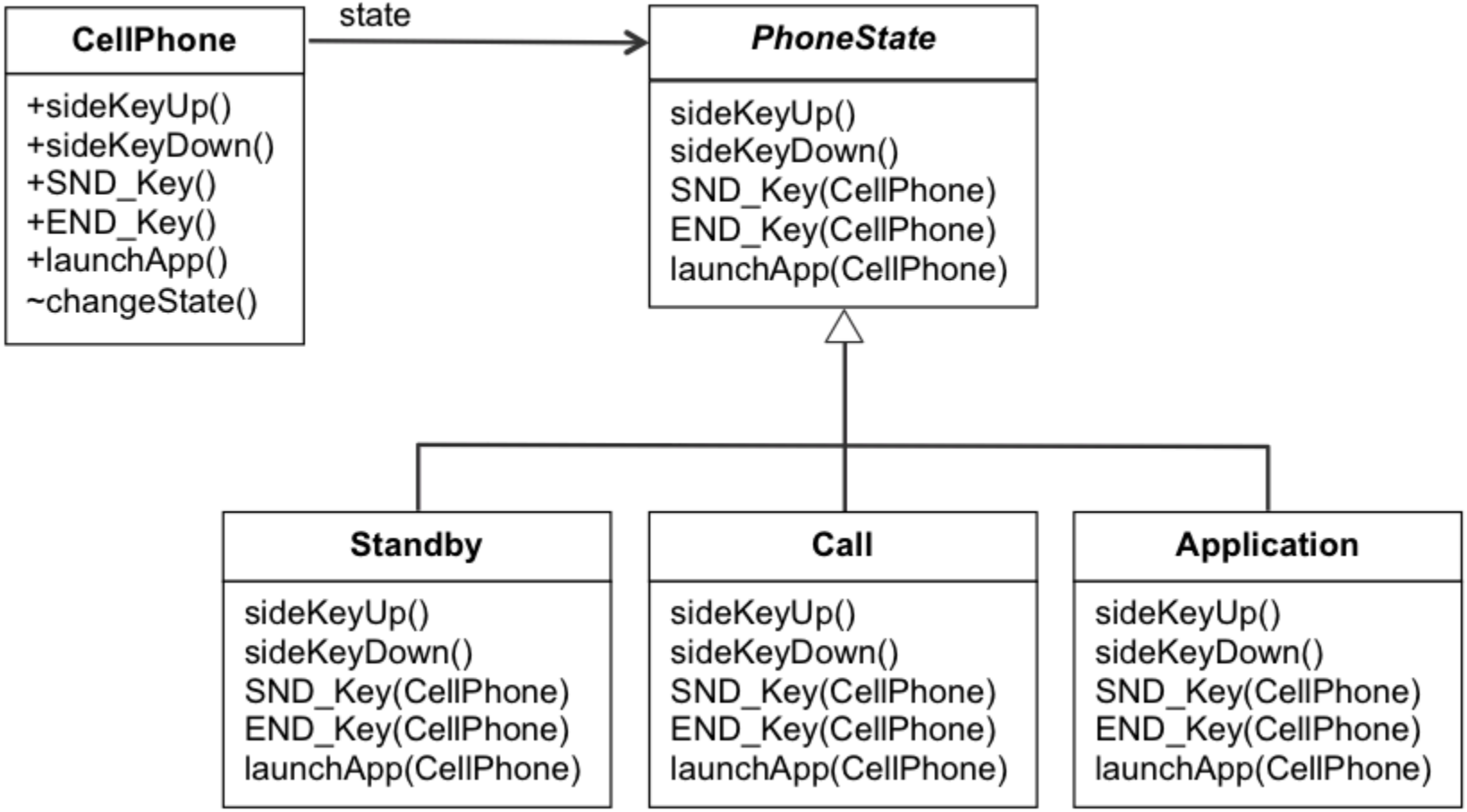
State

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.

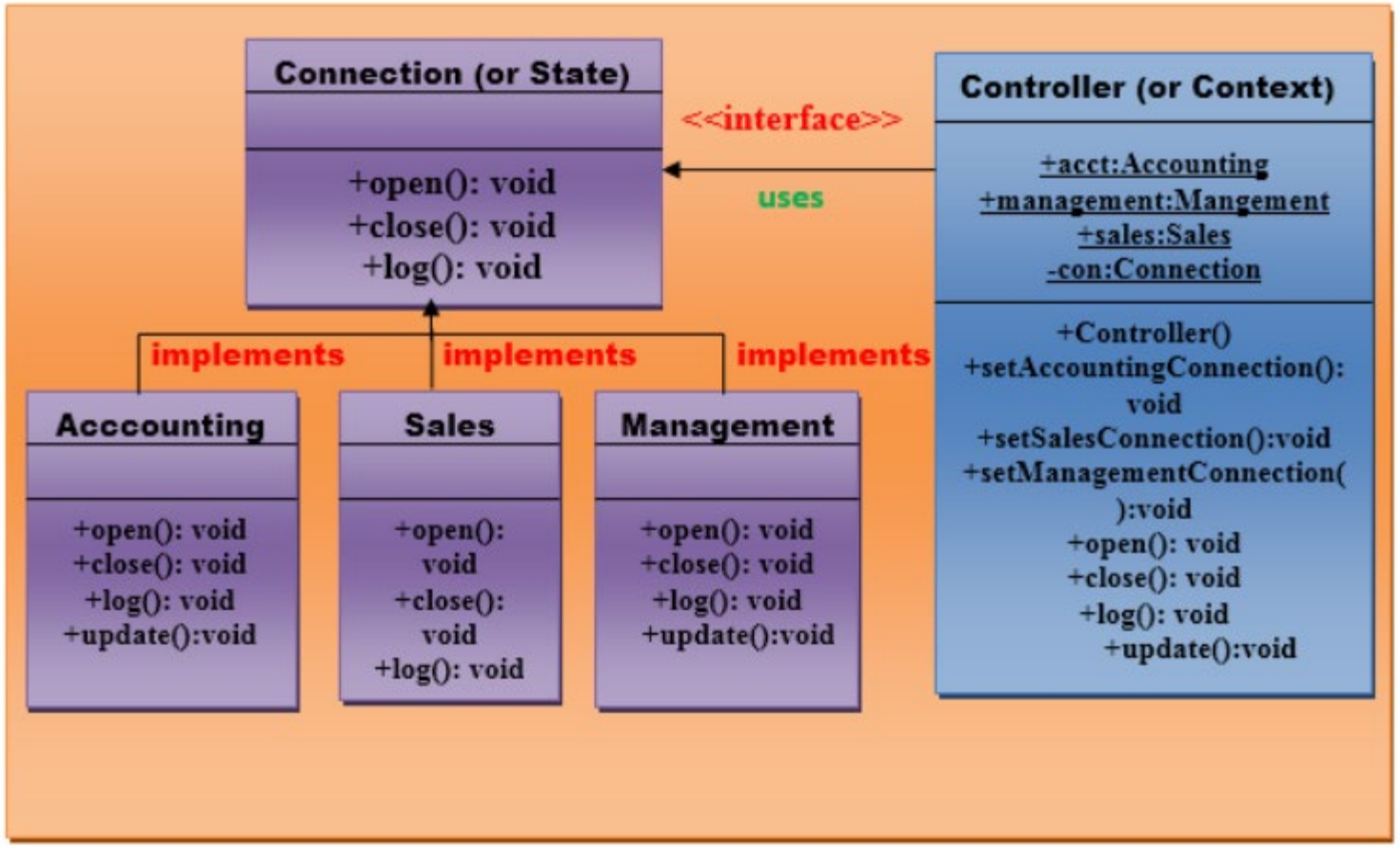
Solution


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)

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

input	sorted result
4PGC938	1ICK750
2IYE230	1ICK750
3CIO720	1OHV845
1ICK750	1OHV845
1OHV845	1OHV845
4JZY524	2IYE230
1ICK750	2RLA629
3CIO720	2RLA629
1OHV845	3ATW723
1OHV845	3CIO720
2RLA629	3CIO720
2RLA629	4JZY524
3ATW723	4PGC938

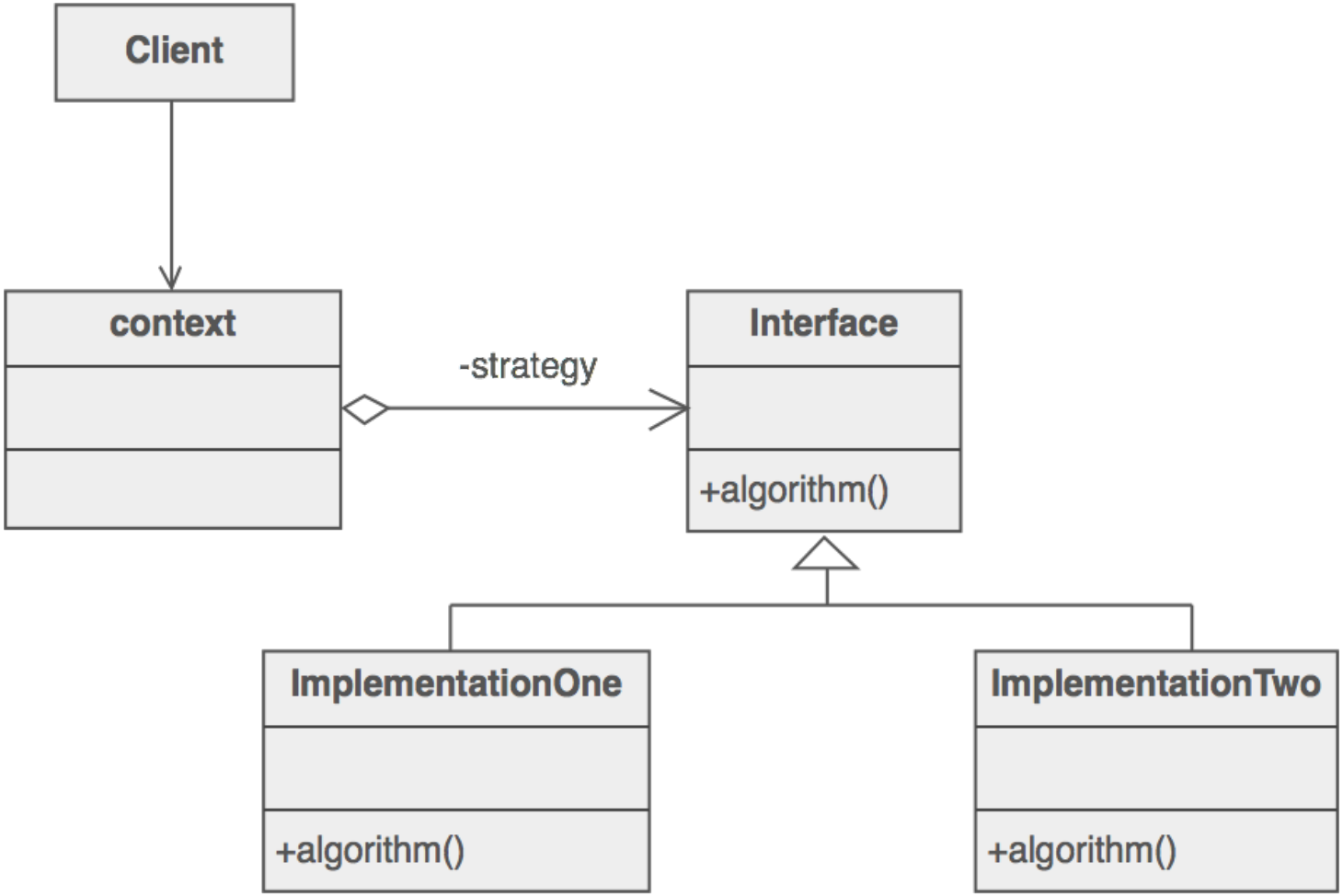
keys are all the same length

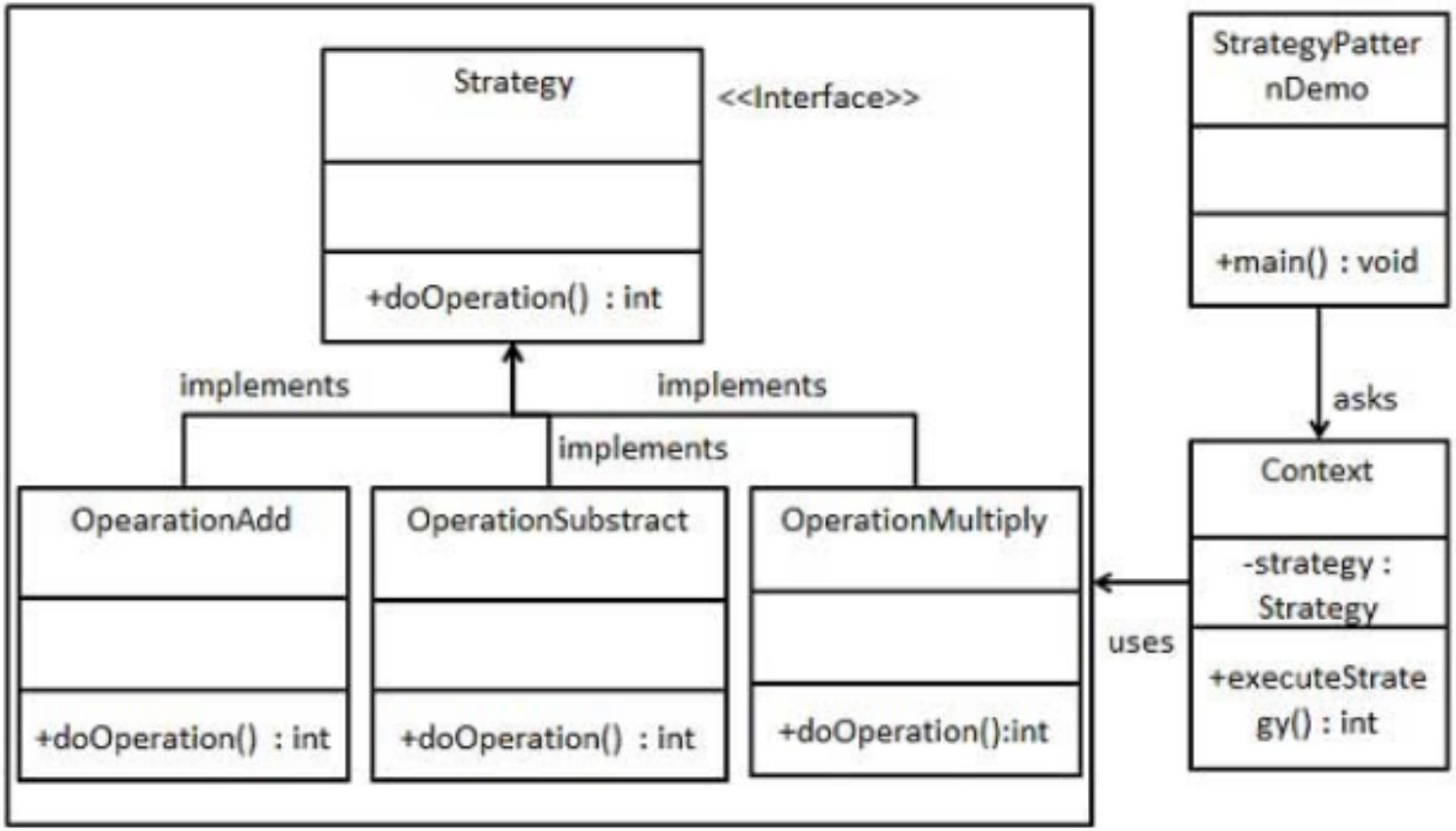
- Quick sort
- Merge sort
- Insertion sort
- Bubble sort
- Radix sort
- Heap sort
- Bucket sort
- ..

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;
   }
}
```

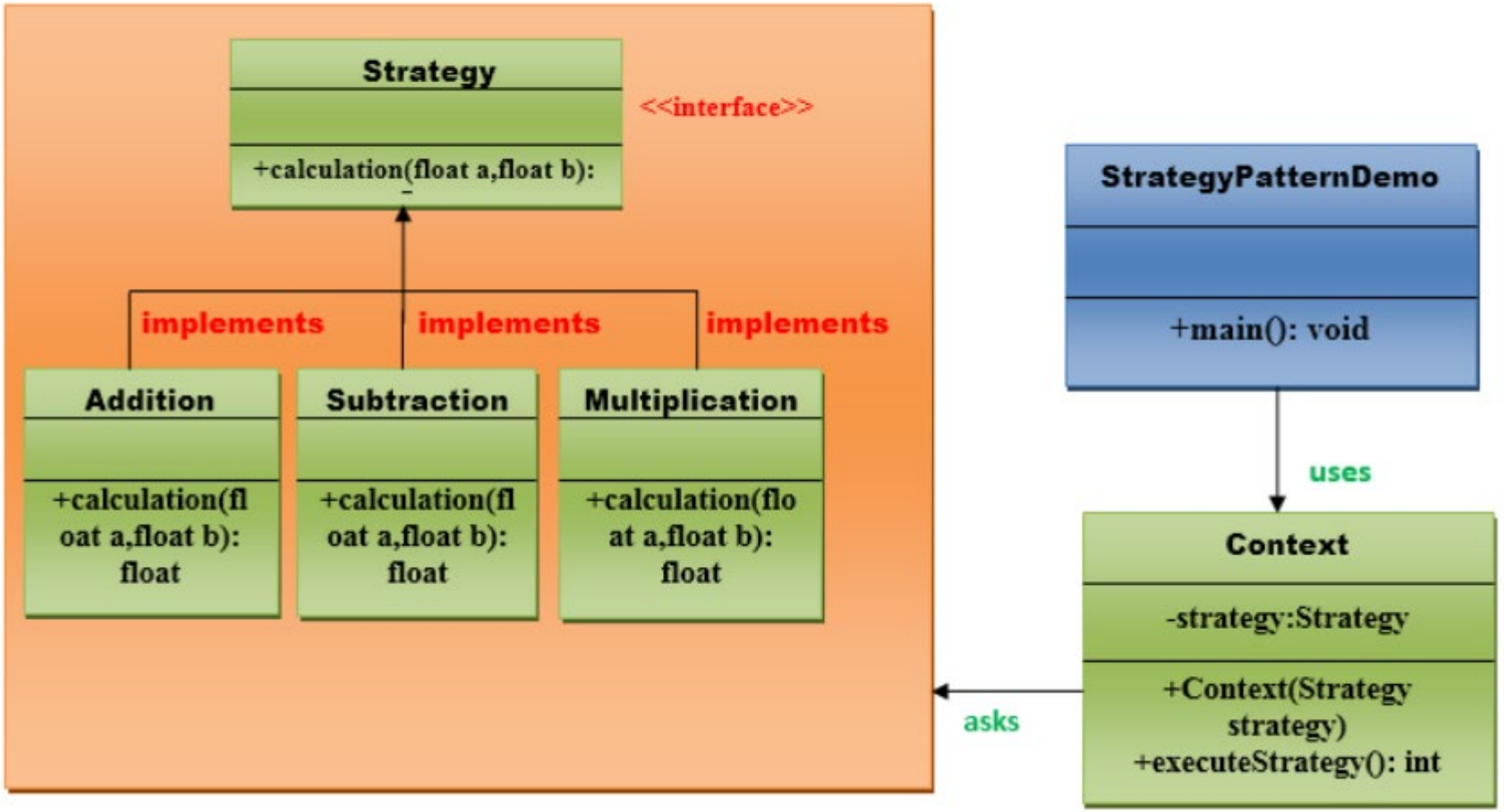
OperationMultiply.java

```
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.
```

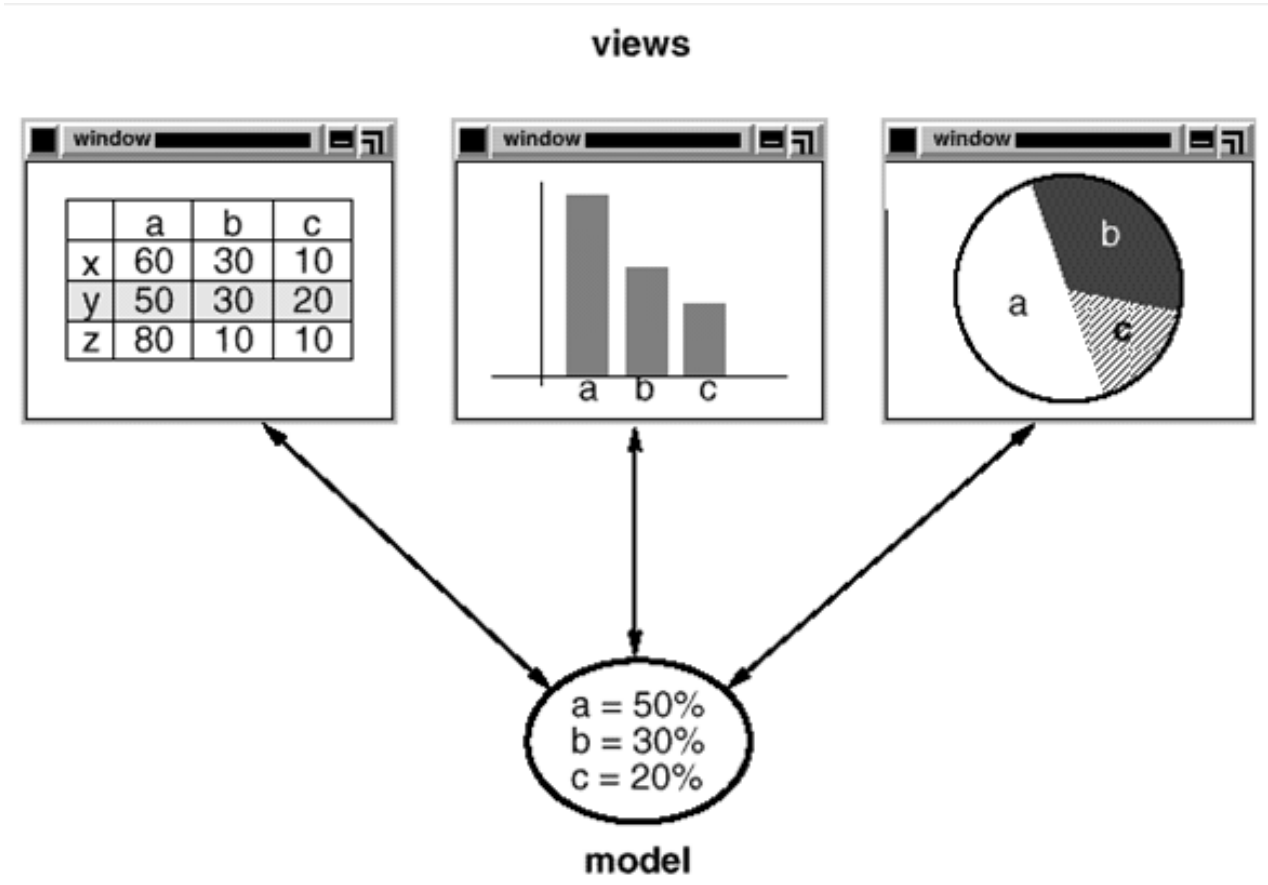
```
public class Context {

    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

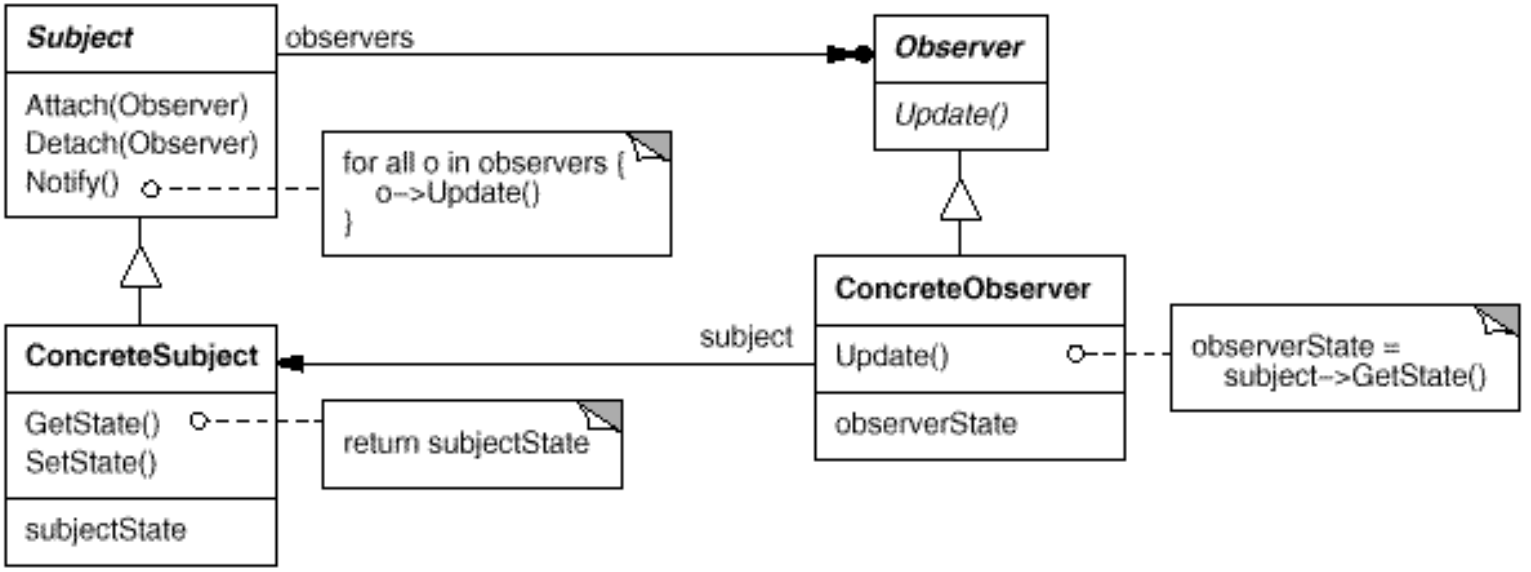
Motivation: Observer

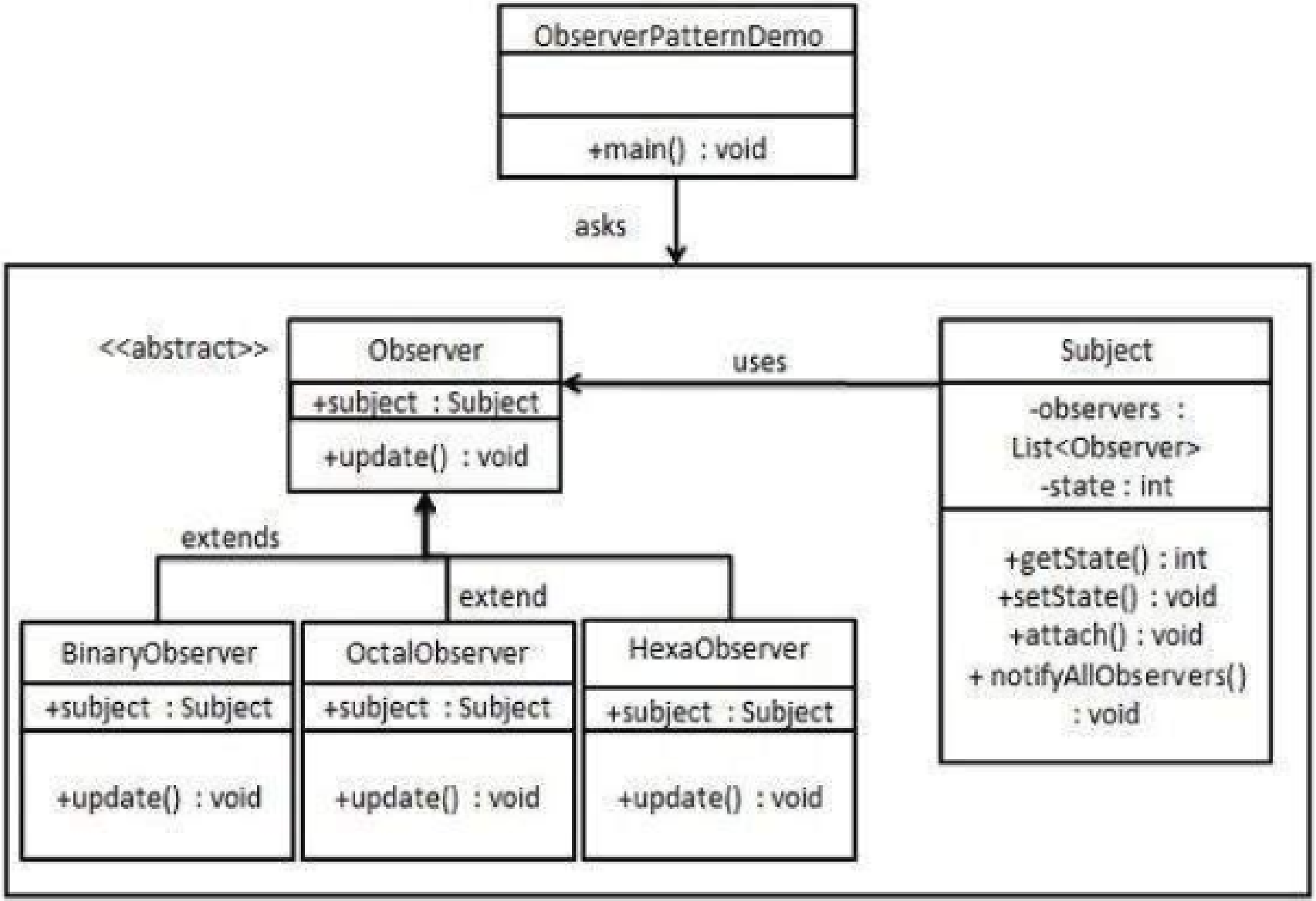


Observer

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

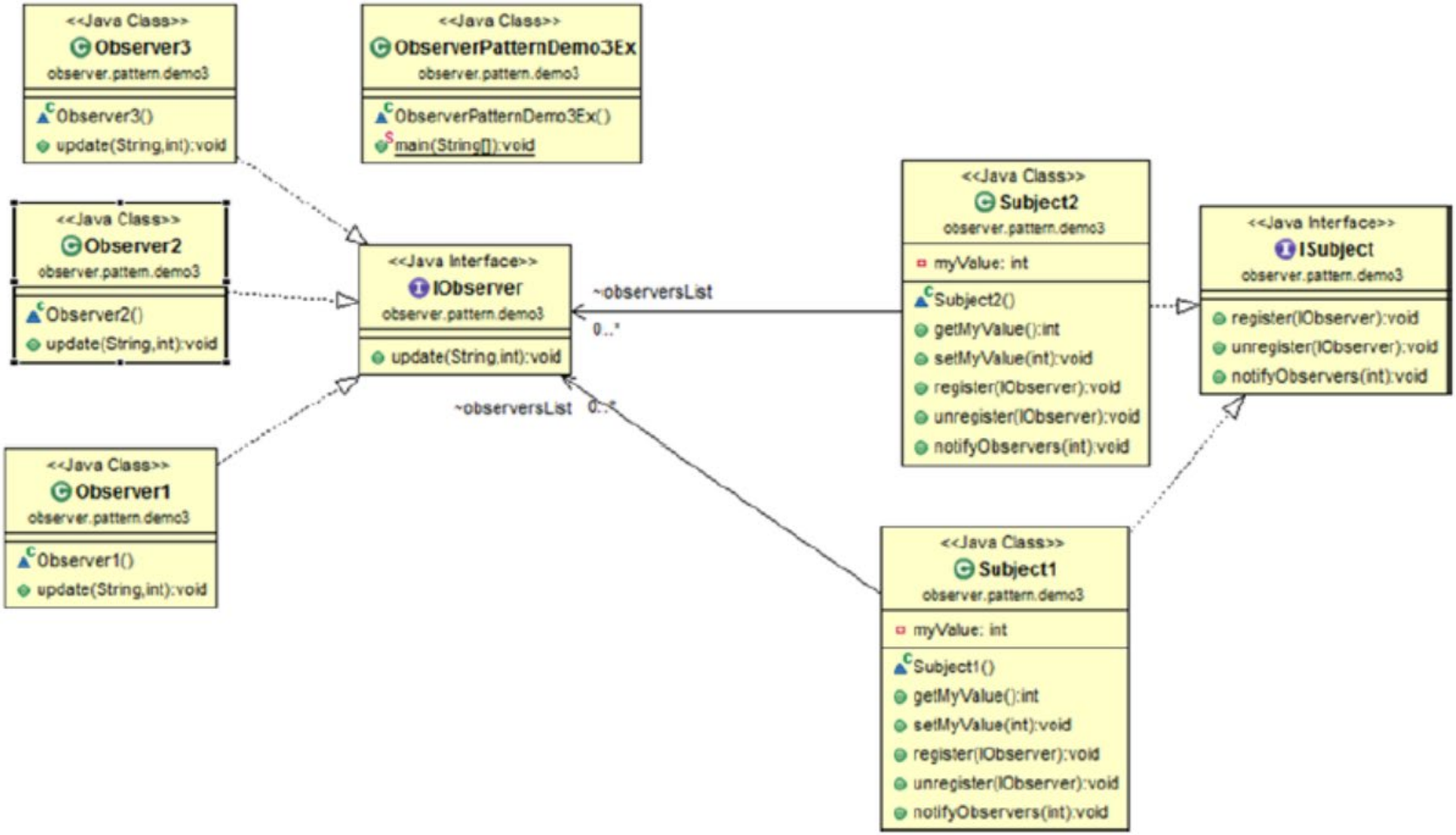
```java
public class ObserverPatternDemo {
   public static void main(String[] args) {
      Subject subject = new Subject();

      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

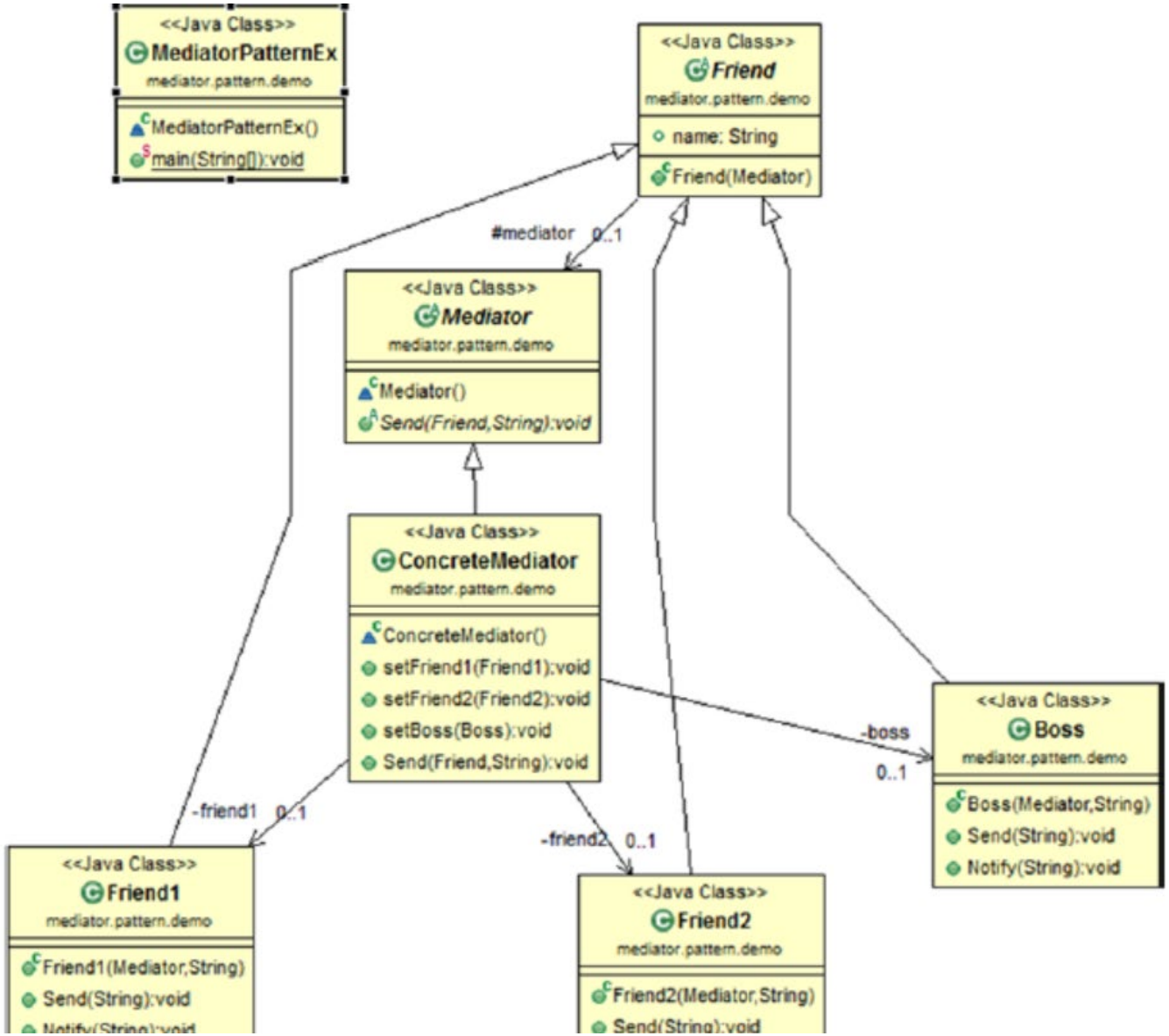


```
class Observer3 implements IObserver
{
    @Override
            public void update(String s,int i)
            {
                    System.out.println("Observer3 is observing:myValue is changed in
                    "+s+" to :"+i);
            }
}

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
```

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);
        }
    }

}
```

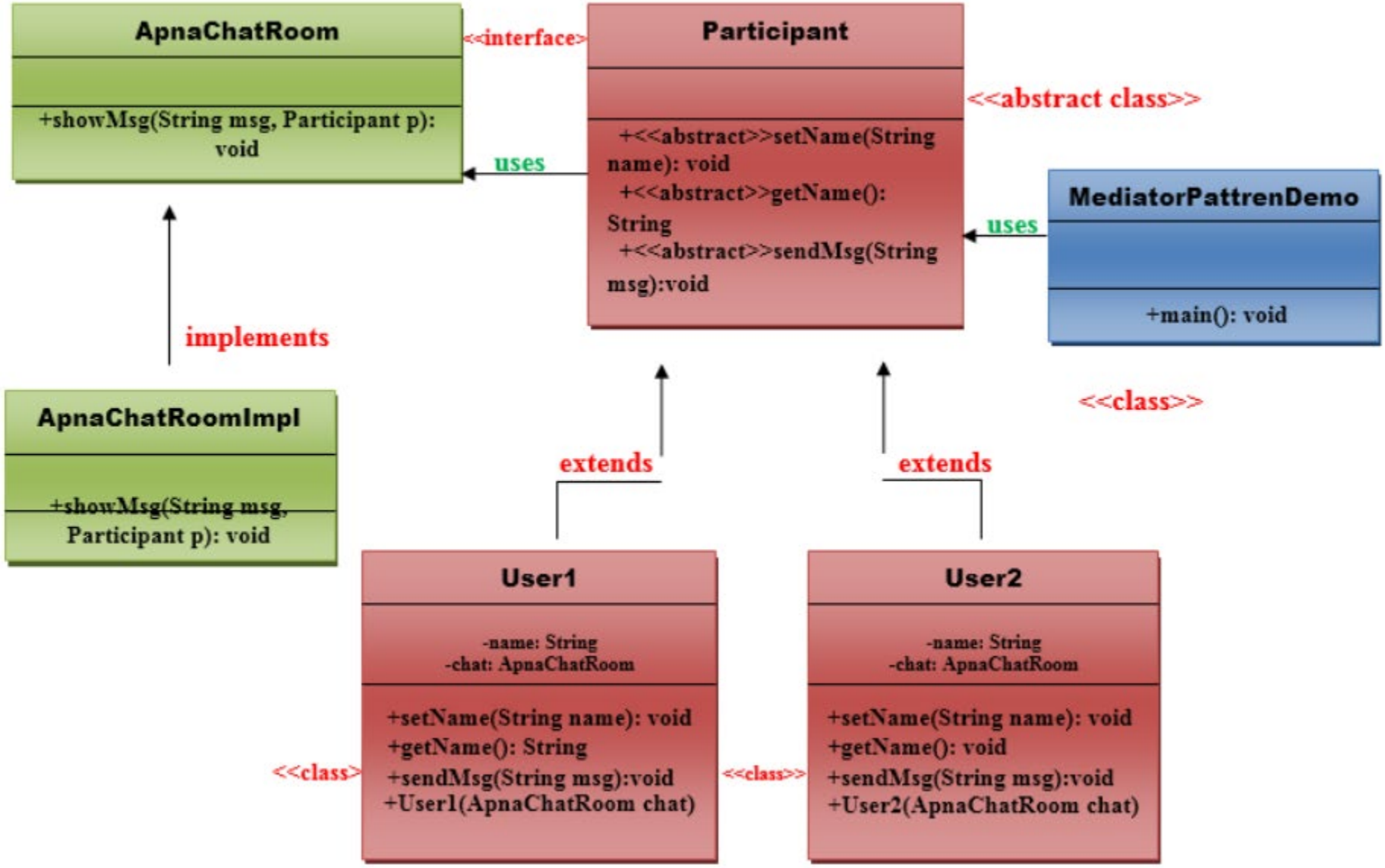
```
abstract class Friend
{
    protected Mediator mediator;
    public String name;

   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.
```

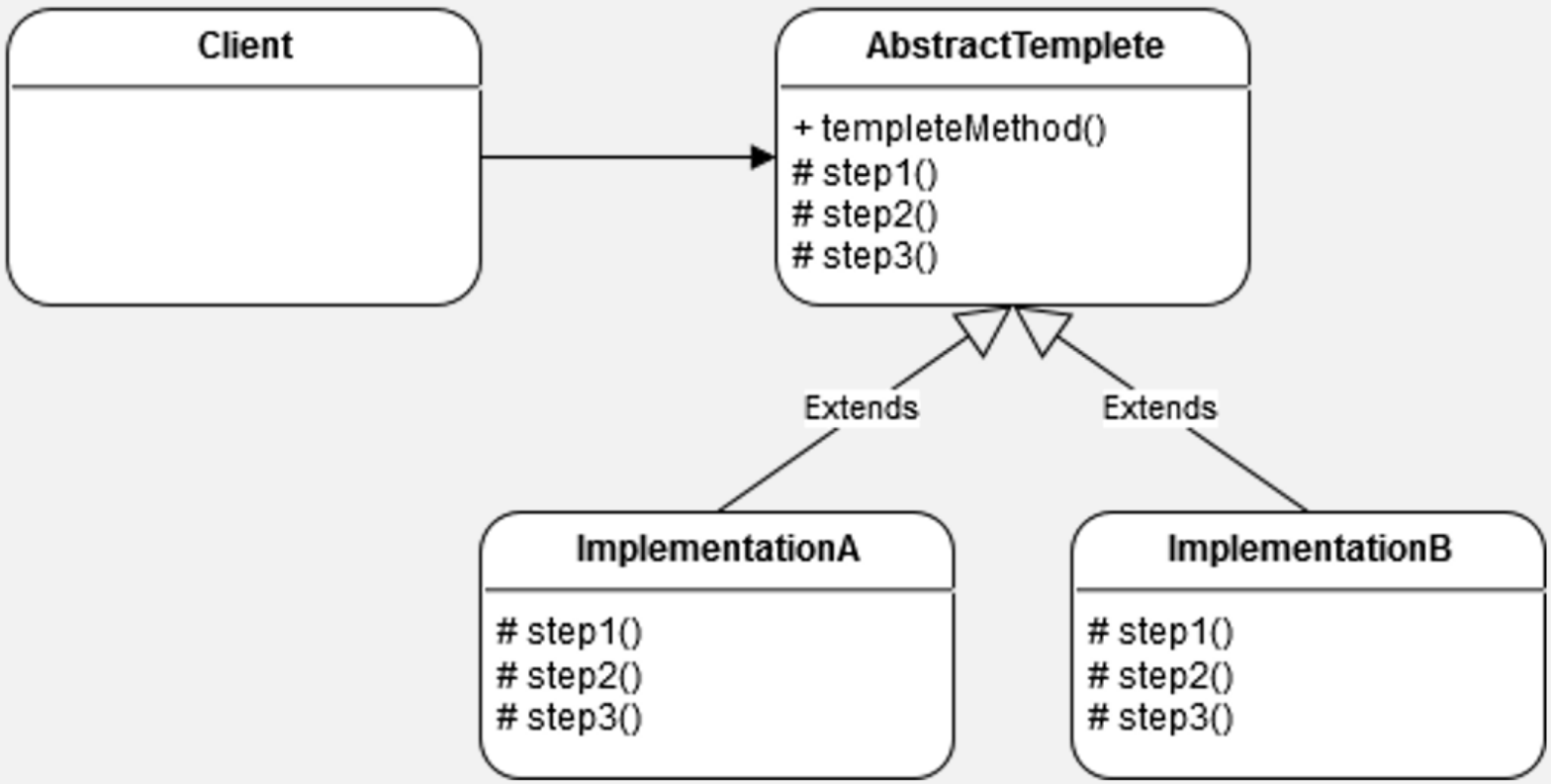
```
public class User2 extends Participant {

    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;
    }
```

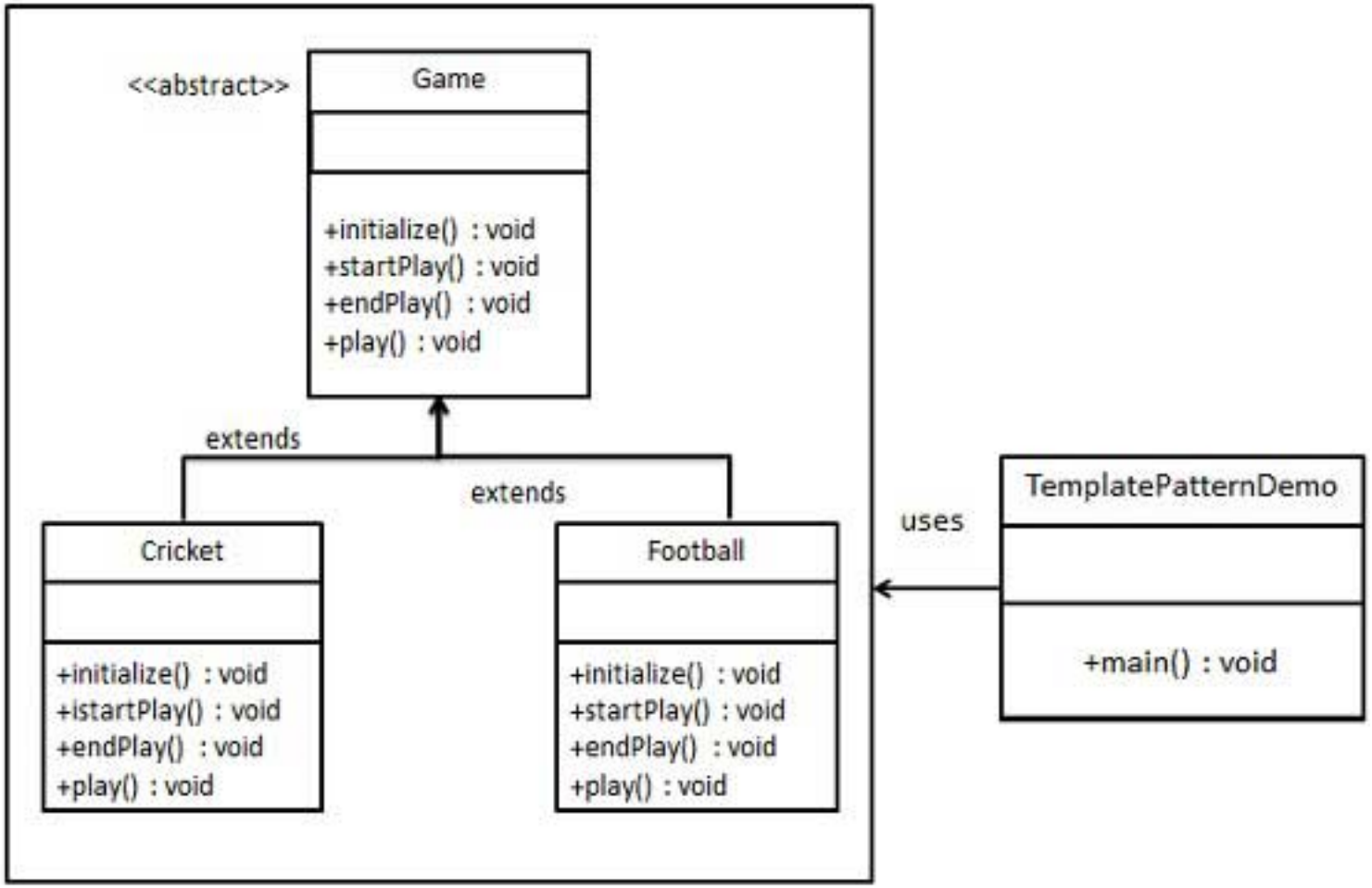
```
public class MediatorPatternDemo {

    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.
```

Template

Intent	• Template Method is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.
Solution	• The Template Method lets you turn a monolithic algorithm into a series of individual steps which can be easily extended by subclasses while keeping intact the structure defined in a superclass.

Templete Method – Class diagram




<<abstract>>
Game
+initialize() : void
+startPlay() : void
+endPlay() : void
+play() : void
extends
extends
Cricket
+initialize() : void
+istartPlay() : void
+endPlay() : void
+play() : void
Football
+initialize() : void
+startPlay() : void
+endPlay() : void
+play() : void
uses
TemplatePatternDemo
+main() : void

Step 1

Create an abstract class with a template method being final.

Game.java

```
public abstract class Game {
   abstract void initialize();
   abstract void startPlay();
   abstract void endPlay();

   //template method
   public final void play(){

      //initialize the game
      initialize();

      //start game
      startPlay();

      //end game
      endPlay();
   }
}
```

Step 2

Create concrete classes extending the above class.

Cricket.java

```
public class Cricket extends Game {

   @Override
   void endPlay() {
      System.out.println("Cricket Game Finished!");
   }

   @Override
   void initialize() {
      System.out.println("Cricket Game Initialized! Start playing.");
   }

   @Override
   void startPlay() {
      System.out.println("Cricket Game Started. Enjoy the game!");
   }
}
```

Football.java

```
public class Football extends Game {

   @Override
   void endPlay() {
      System.out.println("Football Game Finished!");
   }

   @Override
   void initialize() {
      System.out.println("Football Game Initialized! Start playing.");
   }

   @Override
   void startPlay() {
      System.out.println("Football Game Started. Enjoy the game!");
   }
}
```

Step 3

Use the *Game*'s template method play() to demonstrate a defined way of playing game.

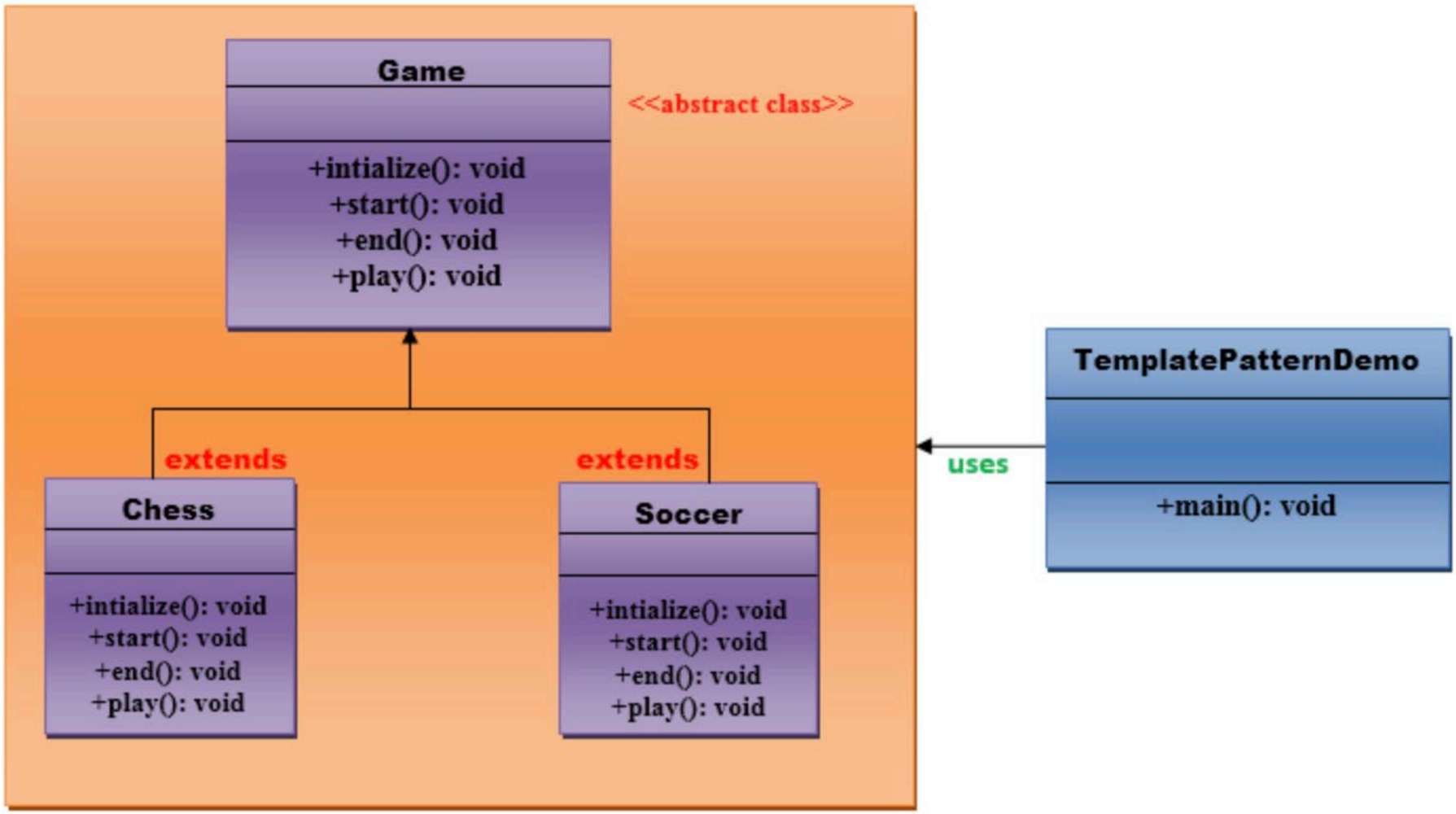
TemplatePatternDemo.java

```
public class TemplatePatternDemo {
   public static void main(String[] args) {

      Game game = new Cricket();
      game.play();
      System.out.println();
      game = new Football();
      game.play();
   }
}
```

Step 4

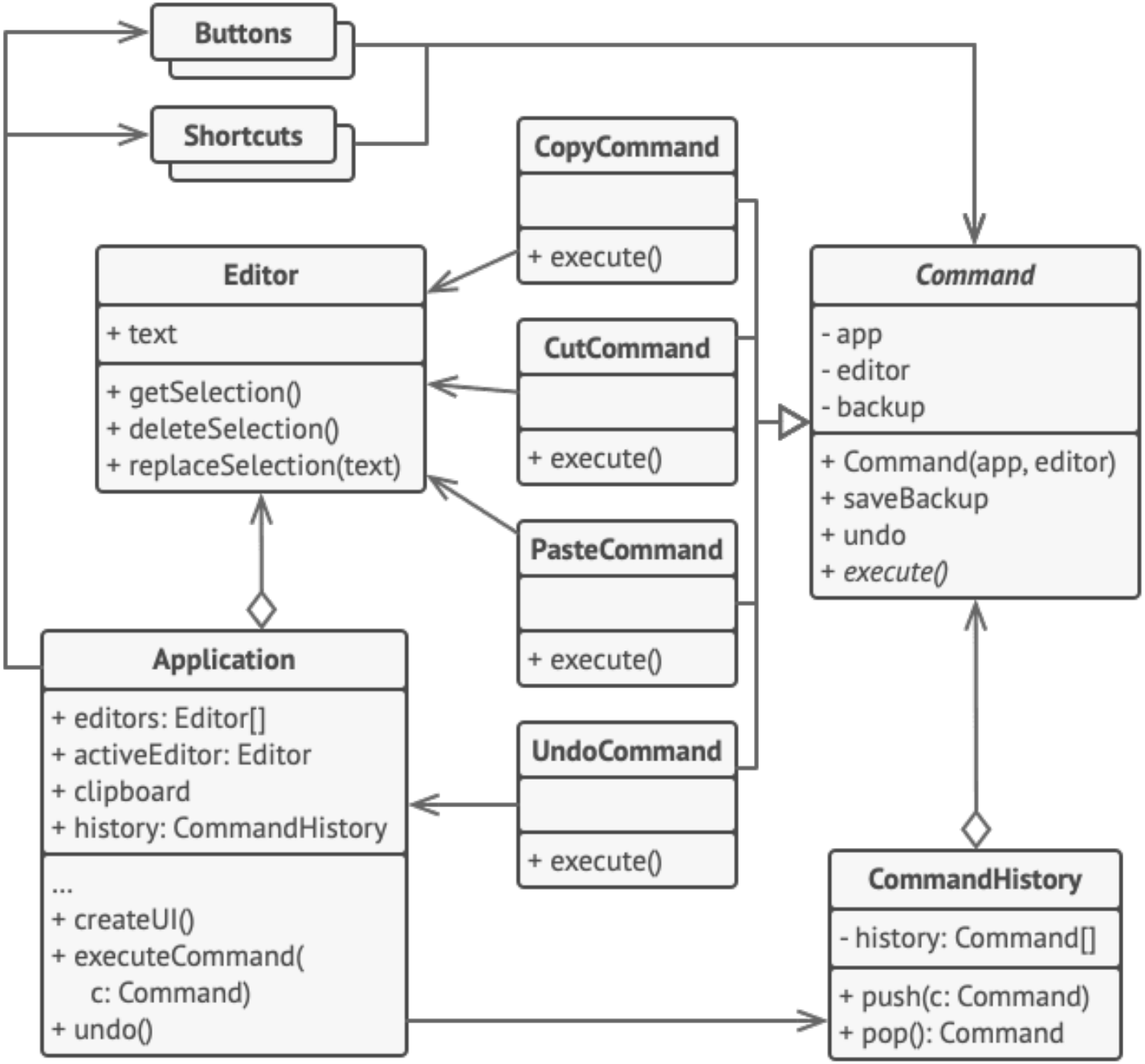
Verify the output.

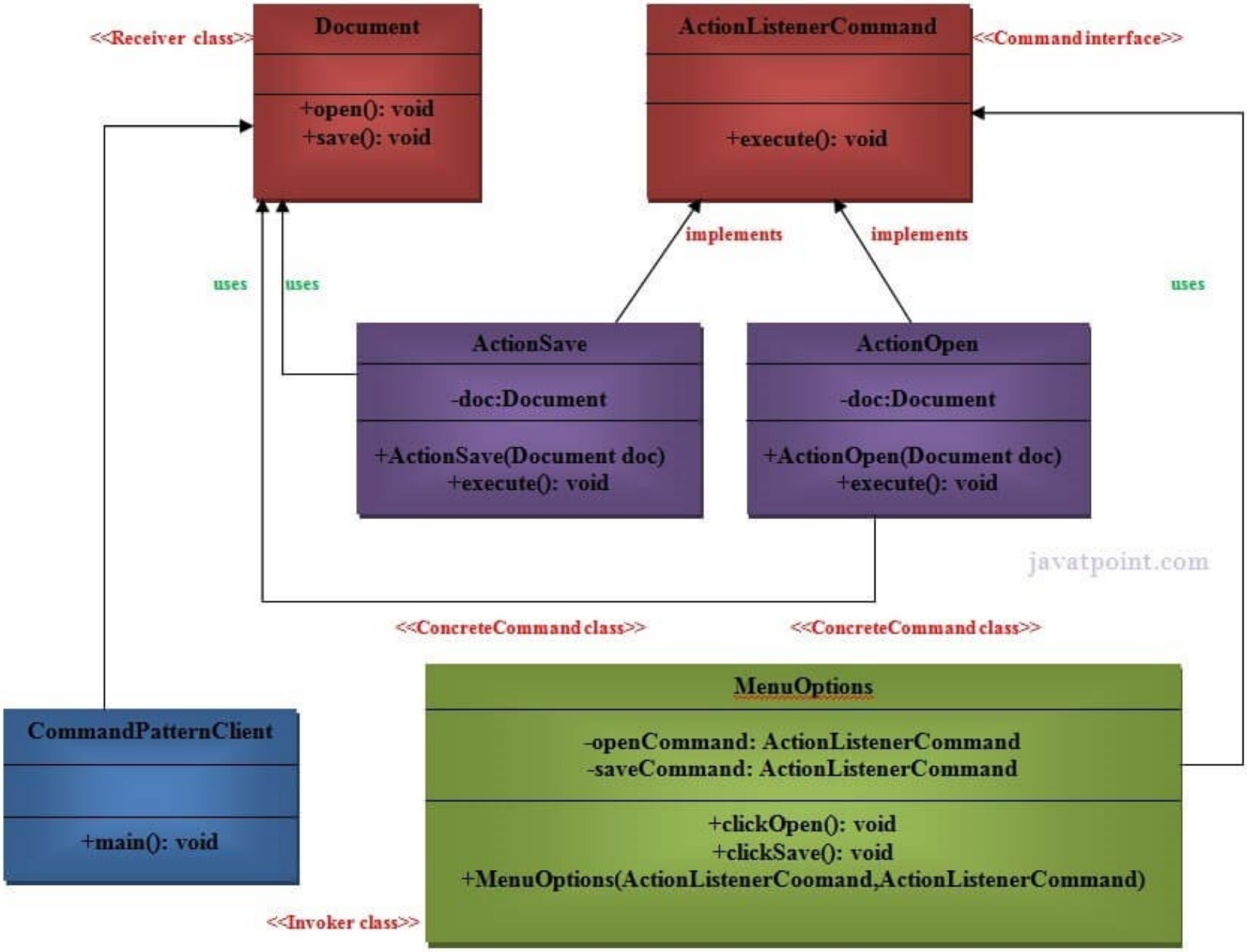
```
Cricket Game Initialized! Start playing.
Cricket Game Started. Enjoy the game!
Cricket Game Finished!

Football Game Initialized! Start playing.
Football Game Started. Enjoy the game!
Football Game Finished!
```


Game
<<abstract class>>
+intialize(): void
+start(): void
+end(): void
+play(): void
extends
extends
Chess
+intialize(): void
+start(): void
+end(): void
+play(): void
Soccer
+intialize(): void
+start(): void
+end(): void
+play(): void
uses
TemplatePatternDemo
+main(): void

Command

Intent	• Command is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you parameterize methods with different requests, delay or queue a request's execution, and support undoable operations.
Solution	• The Command pattern can turn a specific method call into a stand-alone object. This change opens up a lot of interesting uses: you can pass commands as method arguments, store them inside other objects, switch linked commands at runtime, etc


Buttons
Shortcuts
CopyCommand
+ execute()
Editor
+ text
+ getSelection()
+ deleteSelection()
+ replaceSelection(text)
CutCommand
+ execute()
Command
- app
- editor
- backup
+ Command(app, editor)
+ saveBackup
+ undo
+ execute()
PasteCommand
+ execute()
Application
+ editors: Editor[]
+ activeEditor: Editor
+ clipboard
+ history: CommandHistory
...
+ createUI()
+ executeCommand(
c: Command)
+ undo()
UndoCommand
+ execute()
CommandHistory
- history: Command[]
+ push(c: Command)
+ pop(): Command


<<Receiver class>>
Document
+open(): void
+save(): void
ActionListenerCommand
<<Command interface>>
+execute(): void
implements
implements
uses
uses
uses
ActionSave
-doc:Document
+ActionSave(Document doc)
+execute(): void
ActionOpen
-doc:Document
+ActionOpen(Document doc)
+execute(): void
javatpoint.com
<<ConcreteCommand class>>
<<ConcreteCommand class>>
CommandPatternClient
+main(): void
MenuOptions
-openCommand: ActionListenerCommand
-saveCommand: ActionListenerCommand
+clickOpen(): void
+clickSave(): void
+MenuOptions(ActionListenerCoomand,ActionListenerCommand)
<<Invoker class>>

Step 1

Create a **ActionListernerCommand** interface that will act as a Command.

```
public interface ActionListenerCommand {
   public void execute();
}
```

Step 2

Create a **Document** class that will act as a Receiver.

File: Document.java

```
public class Document {
        public void open(){
         System.out.println("Document Opened");
      }
      public void save(){
         System.out.println("Document Saved");
      }
}
```

Step 3

Create a **ActionOpen** class that will act as an ConcreteCommand.

File: ActionOpen.java

```
public class ActionOpen implements ActionListenerCommand{
   private Document doc;
   public ActionOpen(Document doc) {
      this.doc = doc;
   }
   @Override
   public void execute() {
      doc.open();
   }
}
```

Step 4

Create a **ActionSave** class that will act as an ConcreteCommand.

File: AdapterPatternDemo.java

```
public class ActionSave implements ActionListenerCommand{
   private Document doc;
   public ActionSave(Document doc) {
      this.doc = doc;
   }
   @Override
   public void execute() {
      doc.save();
   }
}
```

Step 5

Create a **MenuOptions** class that will act as an Invoker.

File: ActionSave.java

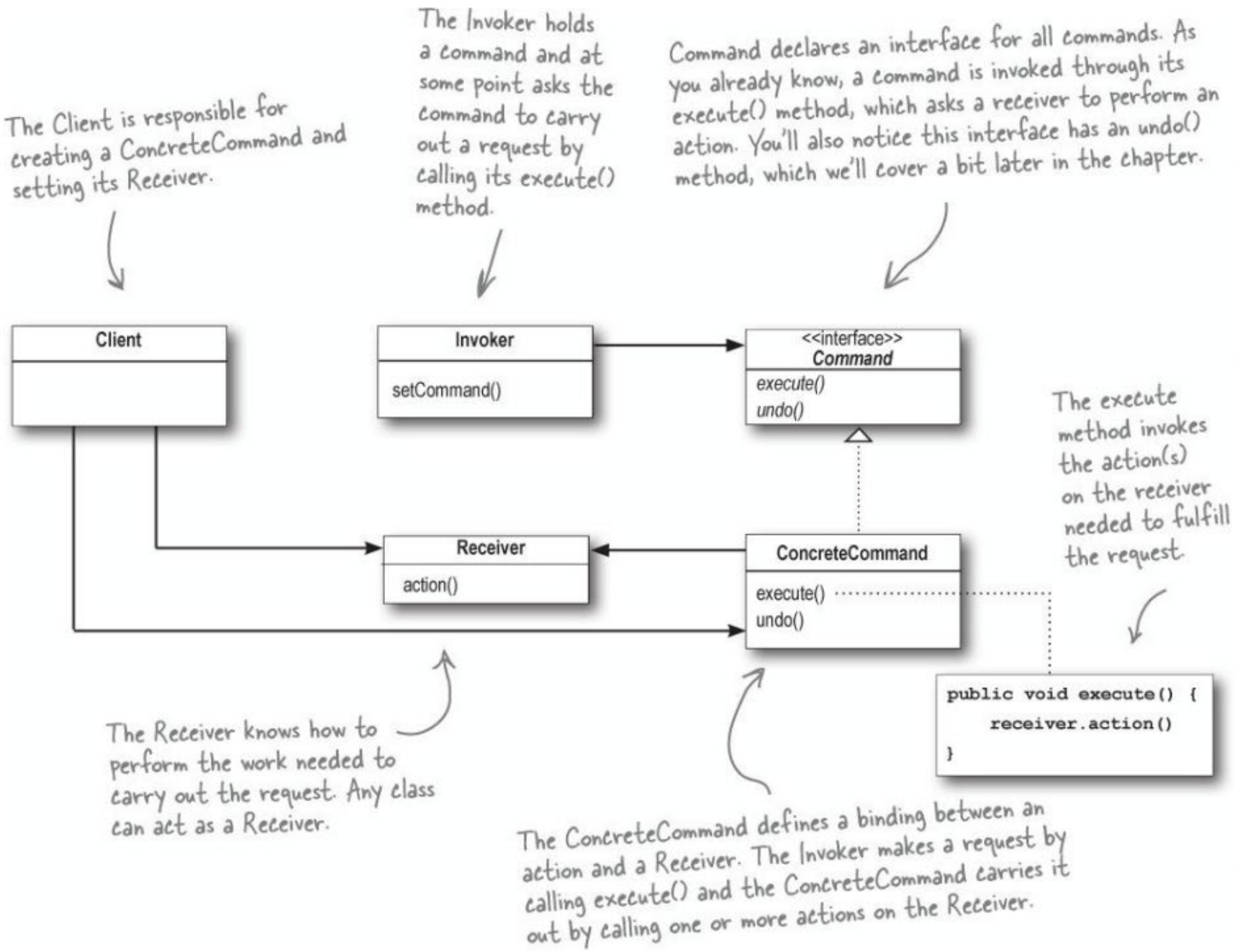
```
public class ActionSave implements ActionListenerCommand{
   private Document doc;
   public ActionSave(Document doc) {
      this.doc = doc;
   }
   @Override
   public void execute() {
      doc.save();
   }
}
```

Step 6

Create a **CommanPatternClient** class that will act as a Client.

File: AdapterPatternDemo.java

```
public class CommandPatternClient {
   public static void main(String[] args) {
      Document doc = new Document();

      ActionListenerCommand clickOpen = new ActionOpen(doc);
      ActionListenerCommand clickSave = new ActionSave(doc);

      MenuOptions menu = new MenuOptions(clickOpen, clickSave);

      menu.clickOpen();
      menu.clickSave();
   }
}
```


The Client is responsible for creating a ConcreteCommand and setting its Receiver.
The Invoker holds a command and at some point asks the command to carry out a request by calling its execute() method.
Command declares an interface for all commands. As you already know, a command is invoked through its execute() method, which asks a receiver to perform an action. You'll also notice this interface has an undo() method, which we'll cover a bit later in the chapter.
Client
Invoker
setCommand()
<<interface>>
Command
execute()
undo()
Receiver
action()
ConcreteCommand
execute()
undo()
The execute method invokes the action(s) on the receiver needed to fulfill the request.
public void execute() {
receiver.action()
}
The Receiver knows how to perform the work needed to carry out the request. Any class can act as a Receiver.
The ConcreteCommand defines a binding between an action and a Receiver. The Invoker makes a request by calling execute() and the ConcreteCommand carries it out by calling one or more actions on the Receiver.