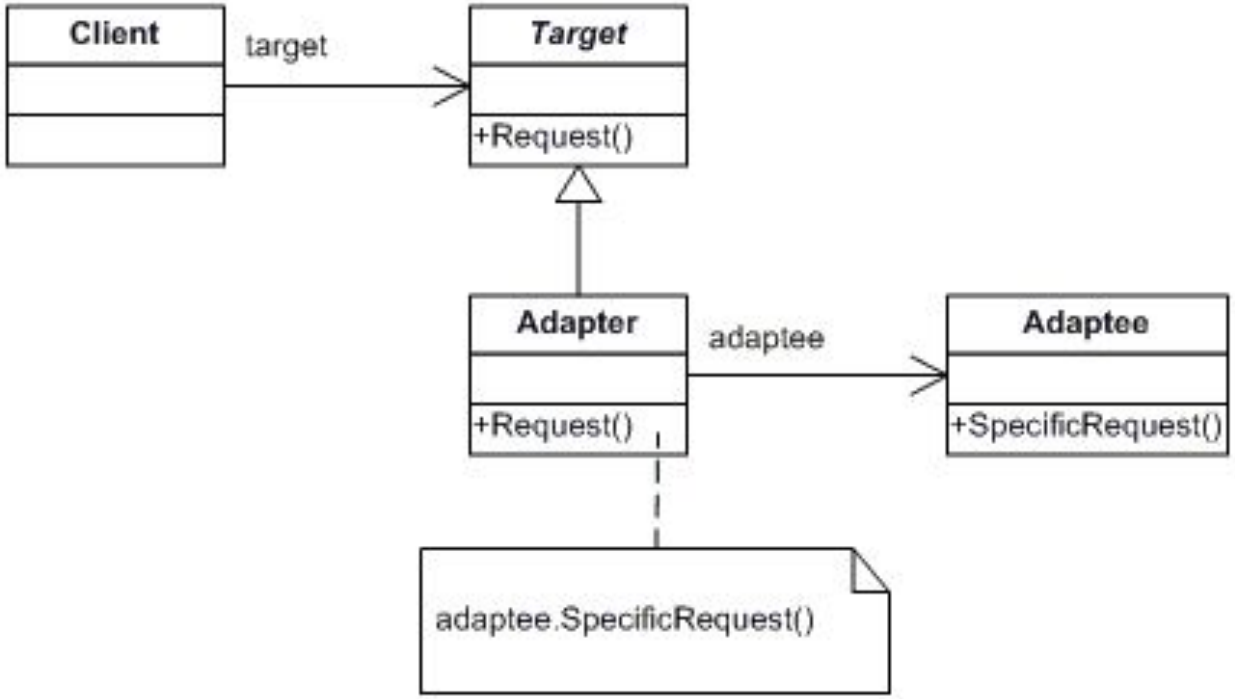
Structural Patterns

Motivation: Adapter

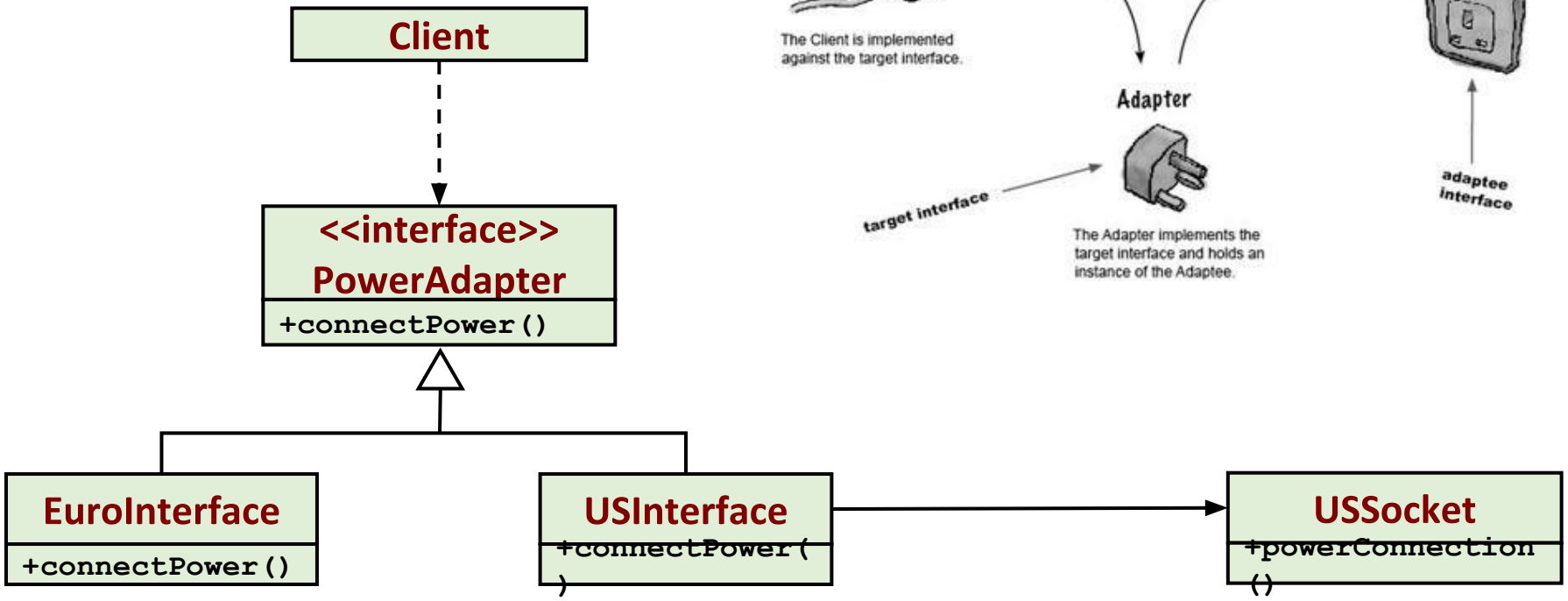
Adapter

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

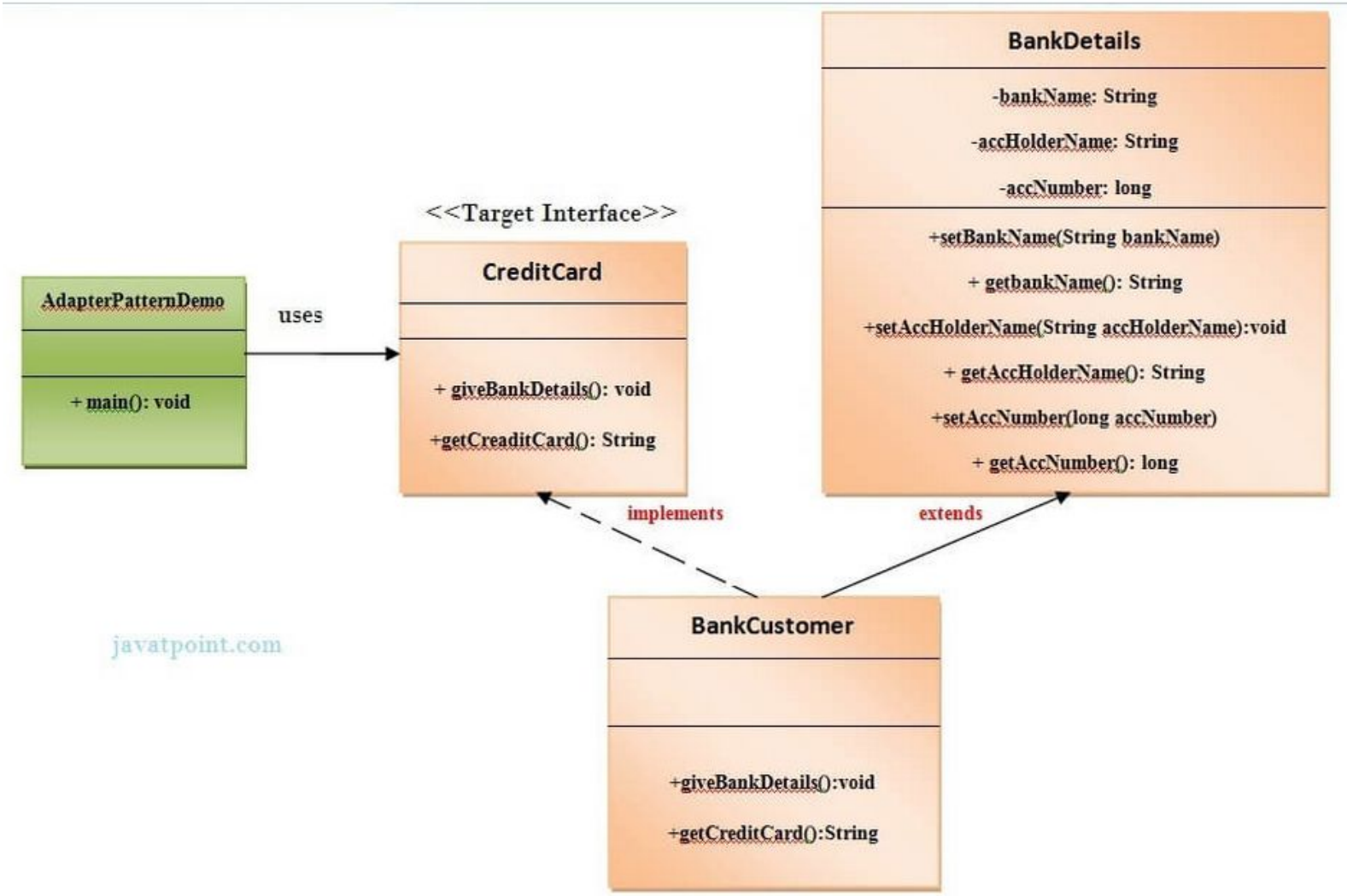
```
public class MediaAdapter implements MediaPlayer {

   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.
```

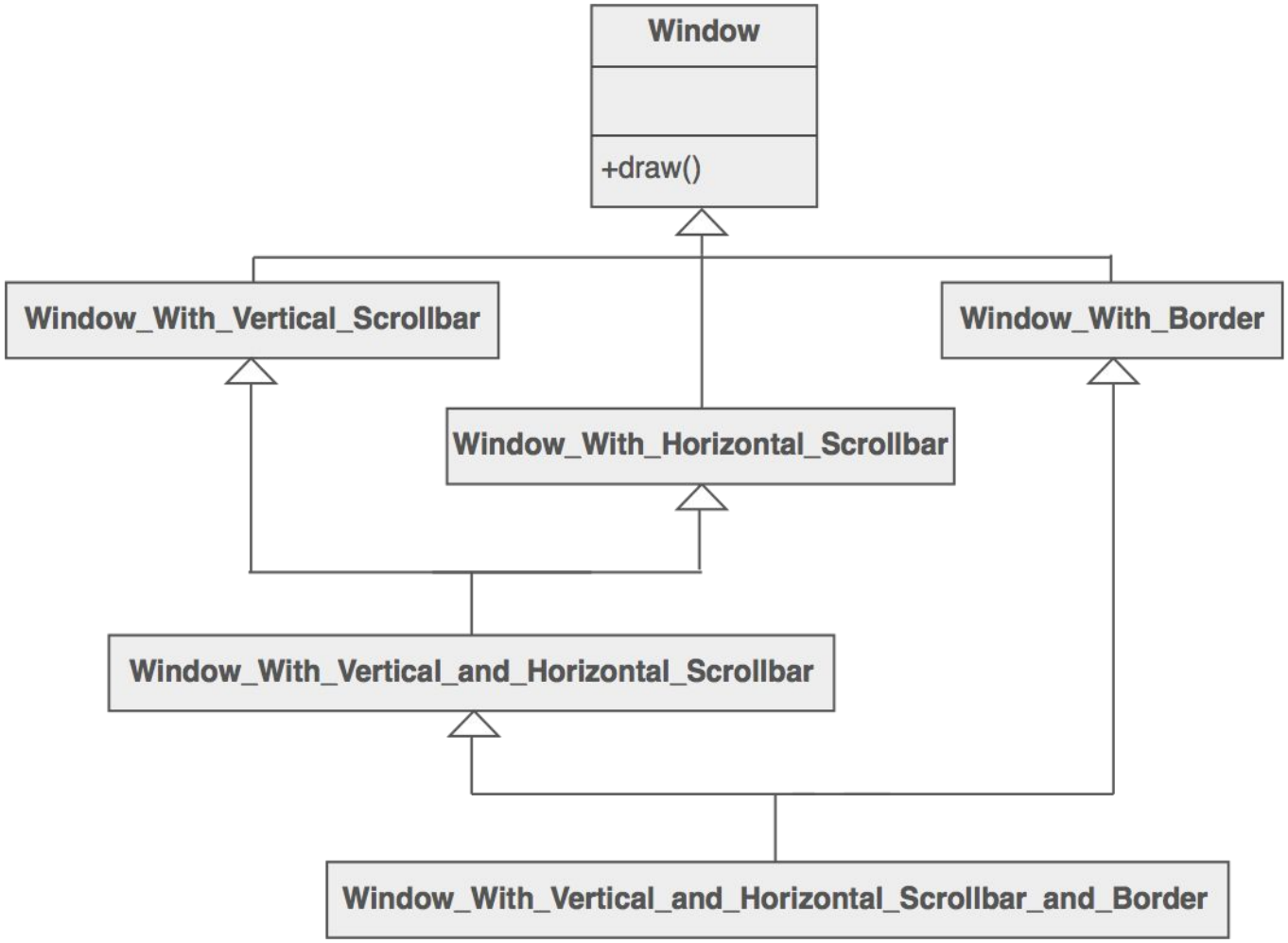
File: AdapterPatternDemo.java

```
//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.

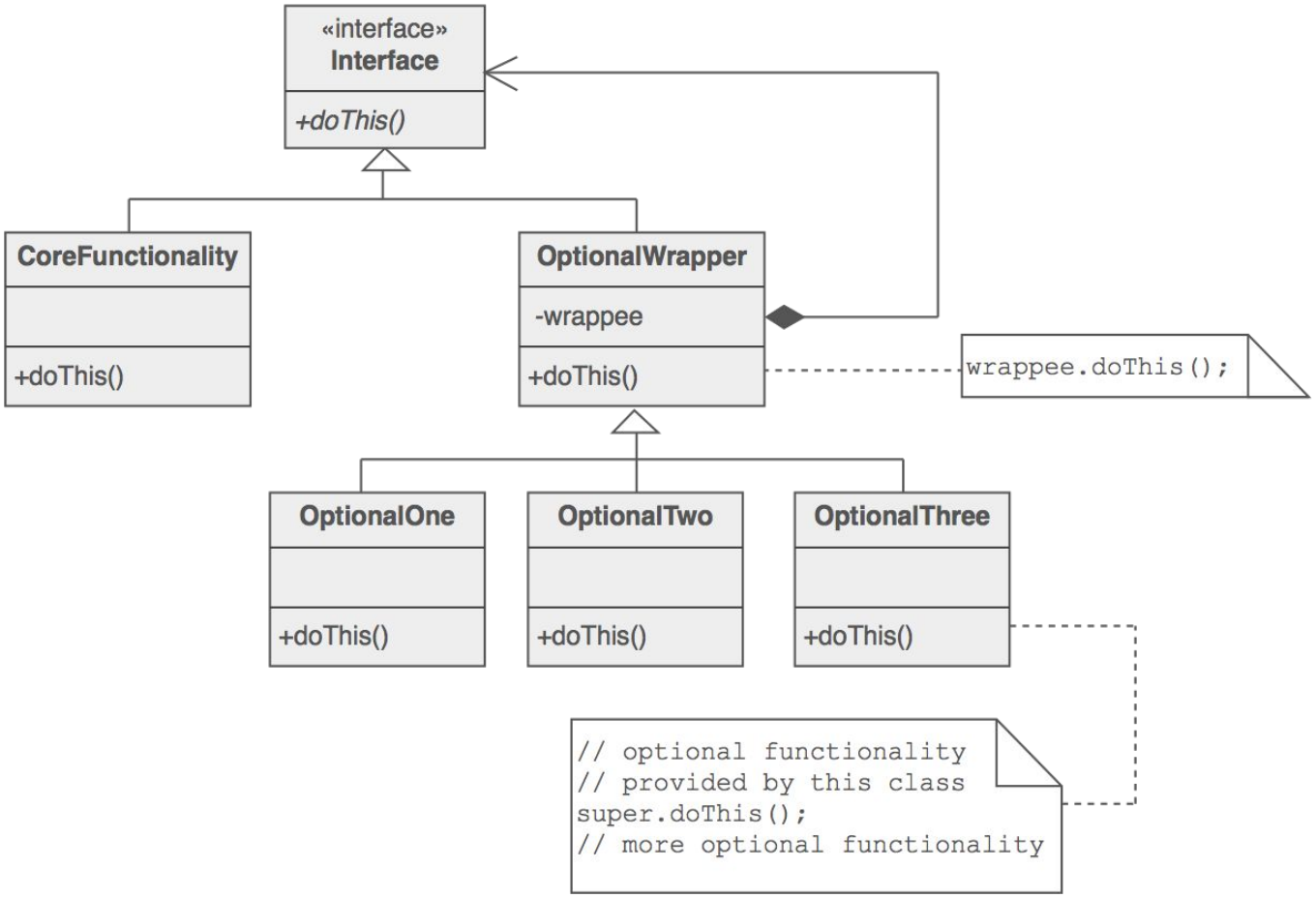
Motivation: Decorator

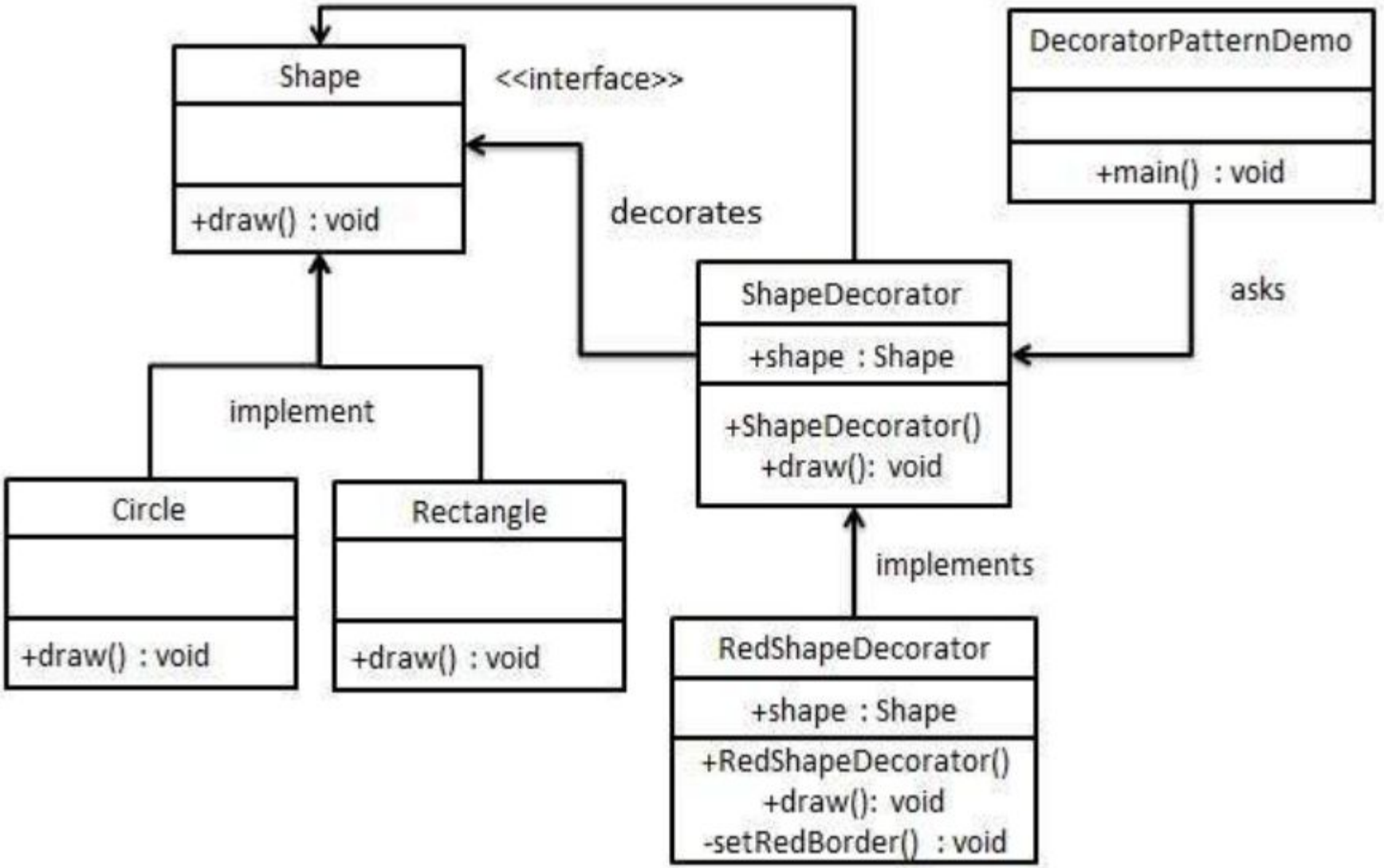


Decorator

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

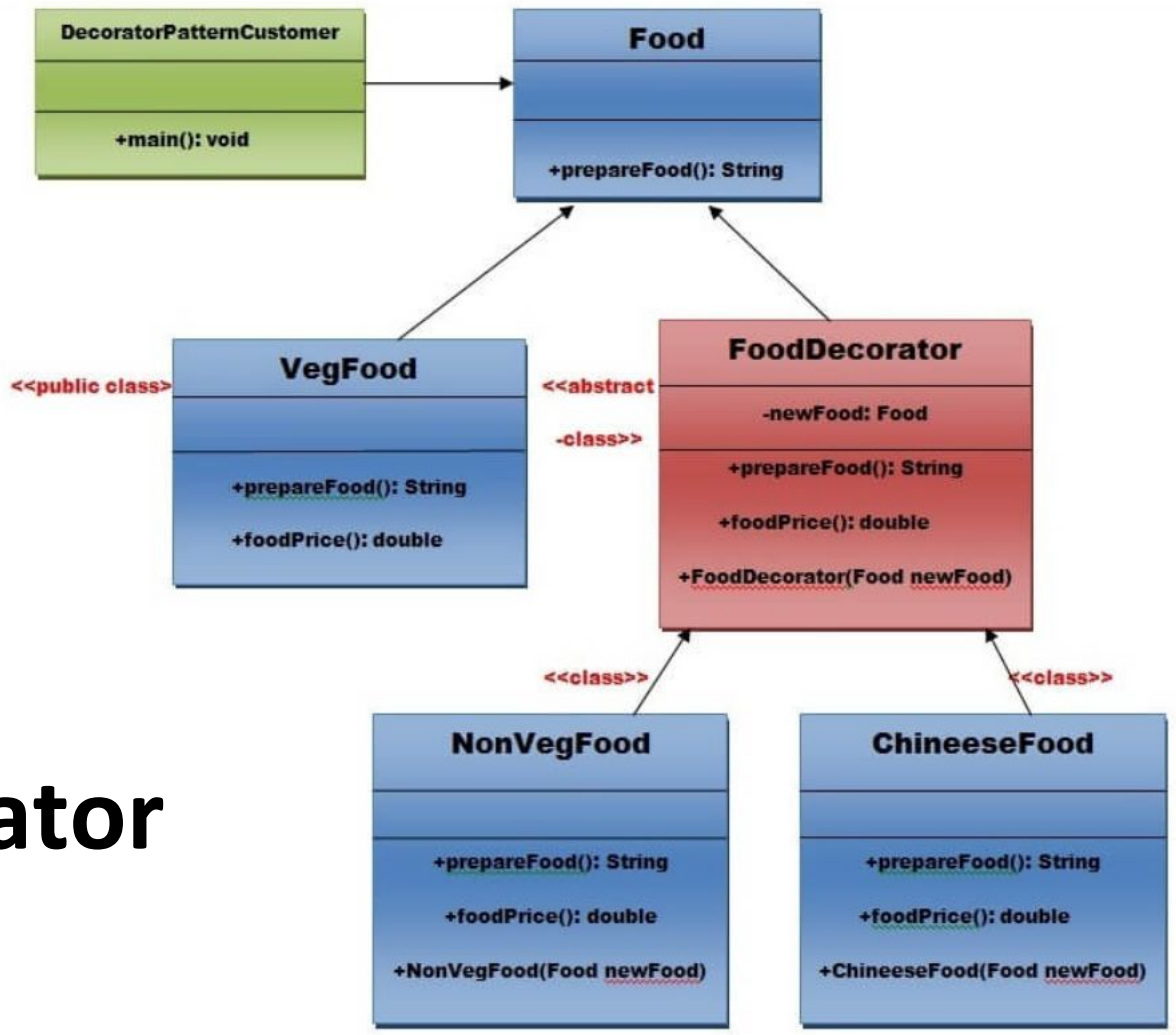
```
public class Circle implements Shape {

   @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

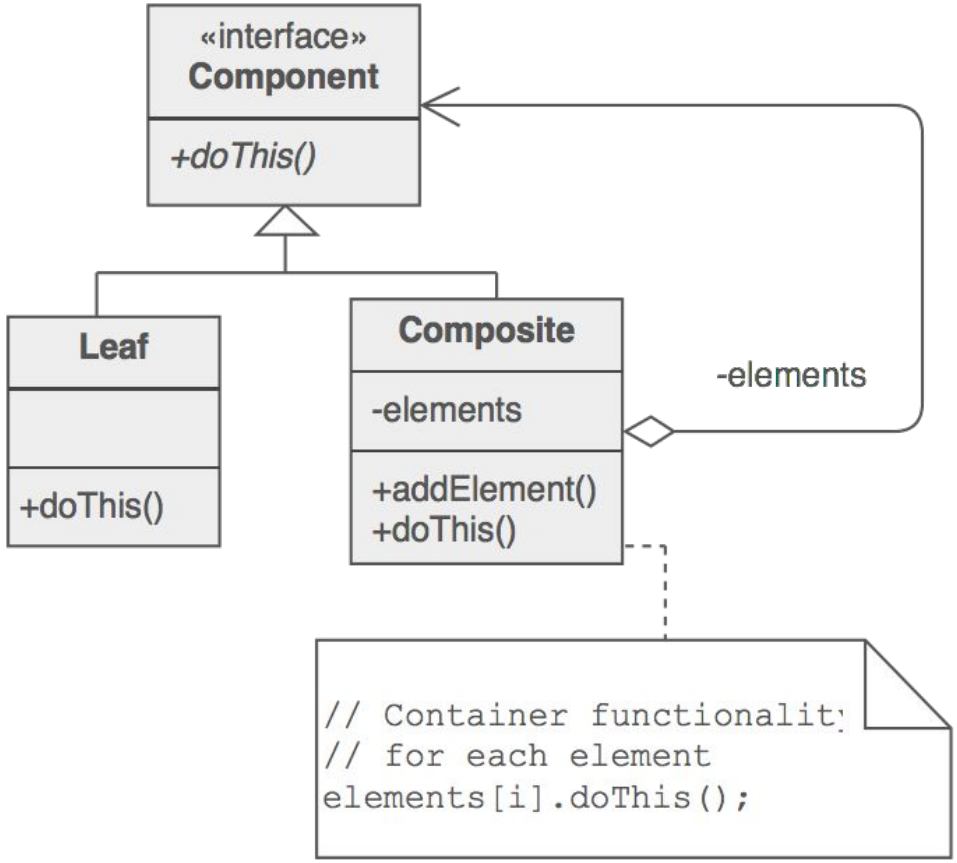
Composite

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.

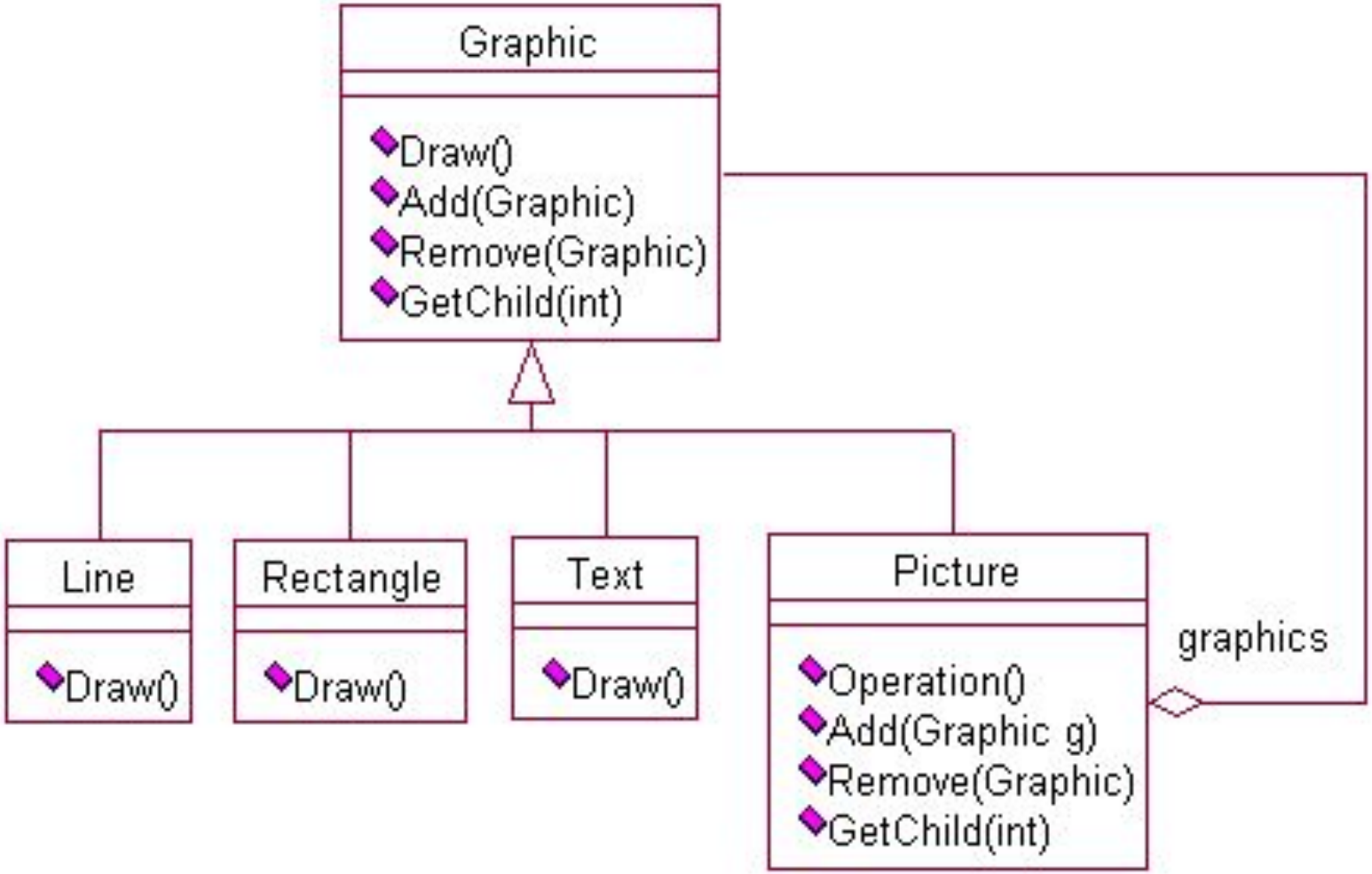
Components

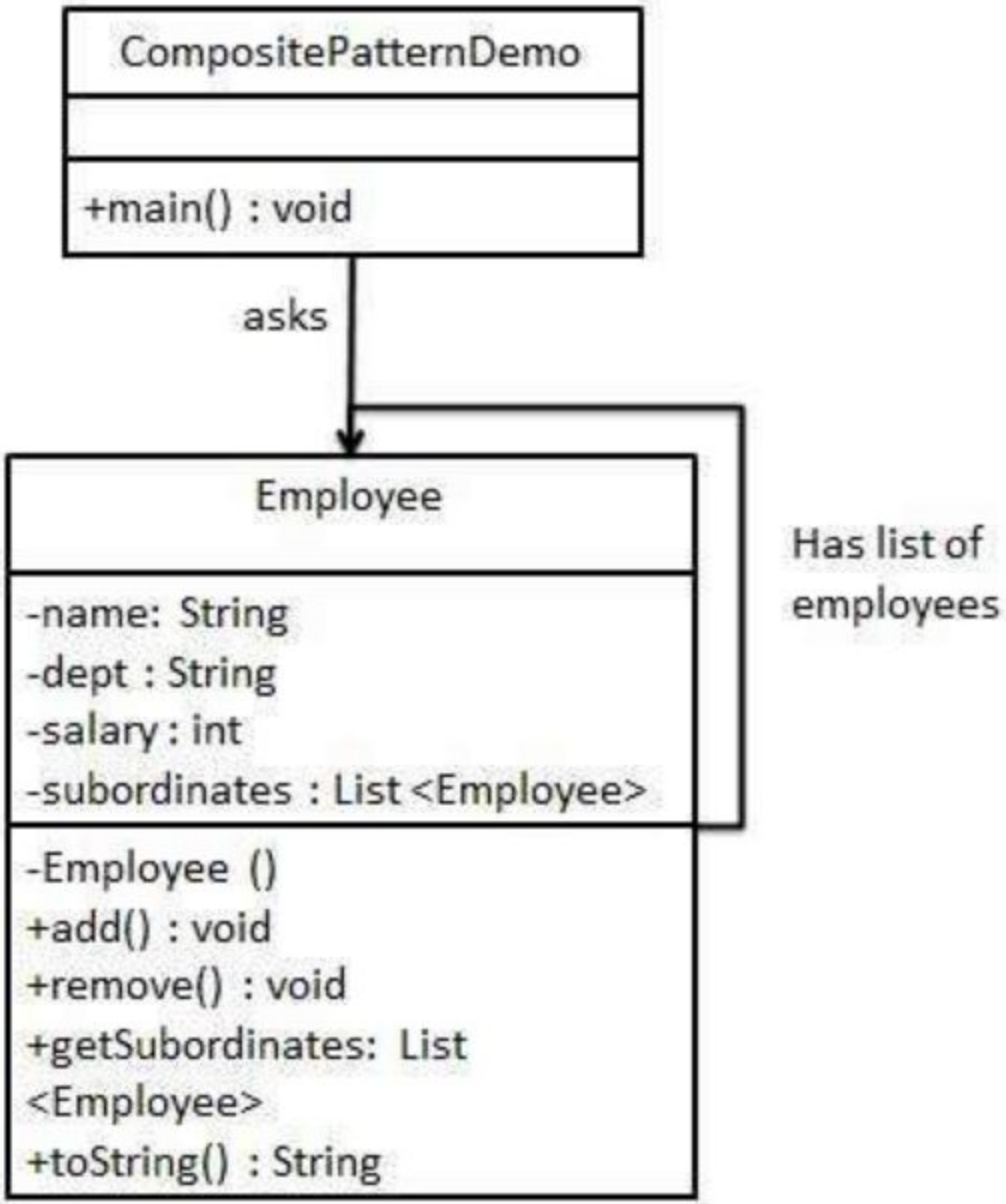
Composite Pattern consists of following objects:

- **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.



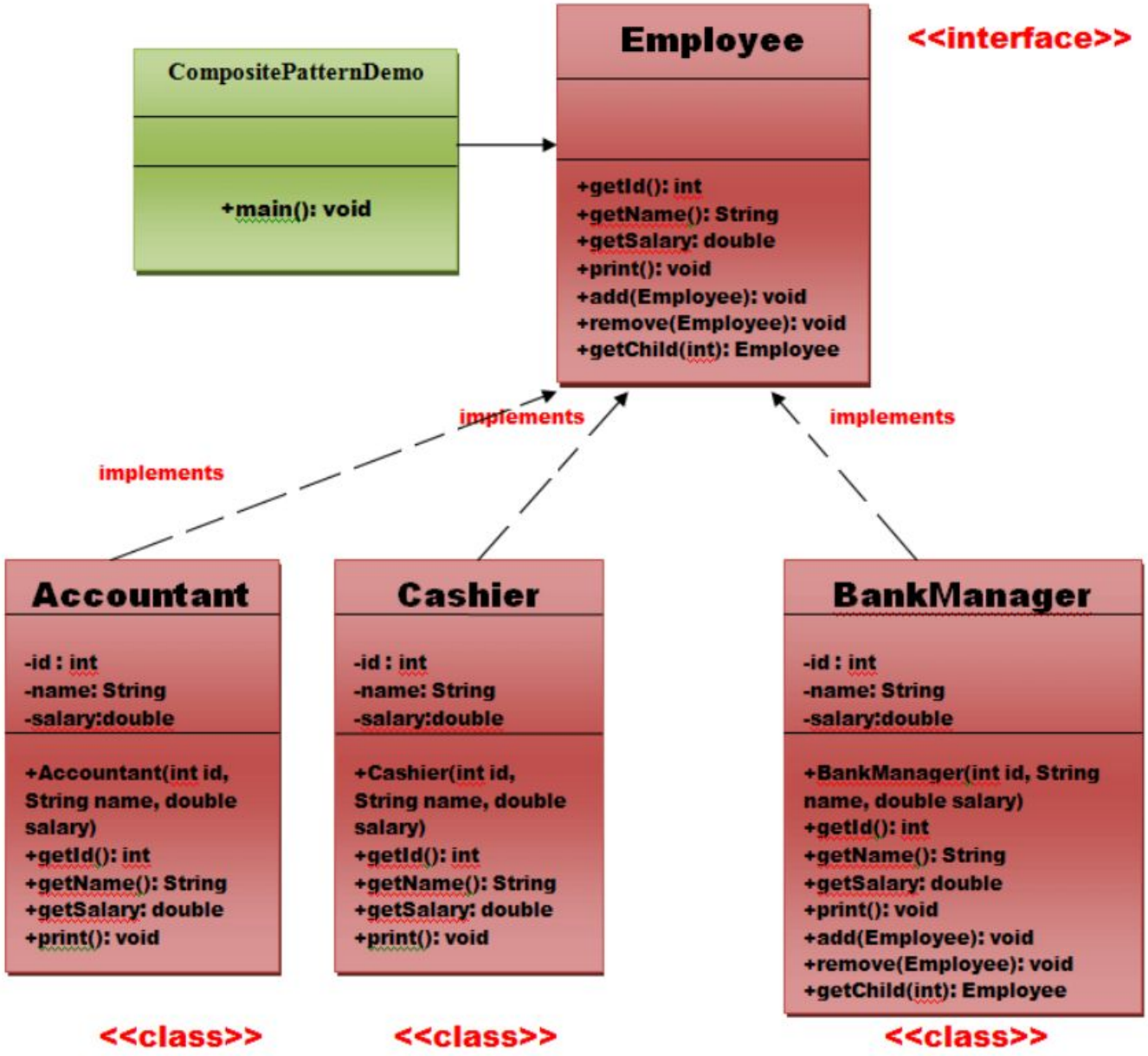
Example


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