DESIGN PATTERN

Hasan Ali Ahmad

Super software

Software engineering

Table of Contents

DESIGN PATTERN

- **What is a Design Pattern?**
  - A (Problem, Solution) pair.
  - A technique to repeat designer success.
  - Borrowed from Civil and Electrical Engineering domains.

ما هو نموذج التصميم؟

- زوج (مشكلة، حل).
- تقنية لتكرار نجاح المصمم.
- اقتبس من مجالات الهندسة المدنية والالكترونية.

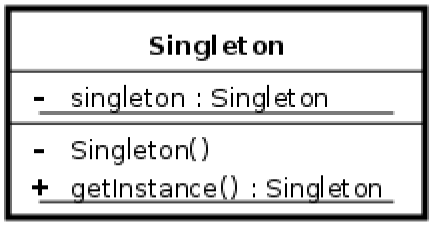
(يكون لدينا مشكلة برمجية مكررة ونحلها باستخدام نموذج تصميم معروف أي هو حل لمشاكل برمجية مكررة).

Pattern Categories:

- **Creational Patterns** concern the process of object creation.
  - نماذج تهتم بعملية خلق الاوبجكت.
- **Structural Patterns** concern with integration and composition of classes and objects.
  - نماذج تهتم بعملية بنية تكامل وارتباط الفئات والكائنات مع بعضها.
- **Behavioral Patterns** concern with class or object communication.
  - نماذج تهتم بعملية سلوك الكائن.

Singleton design pattern

- Creational pattern



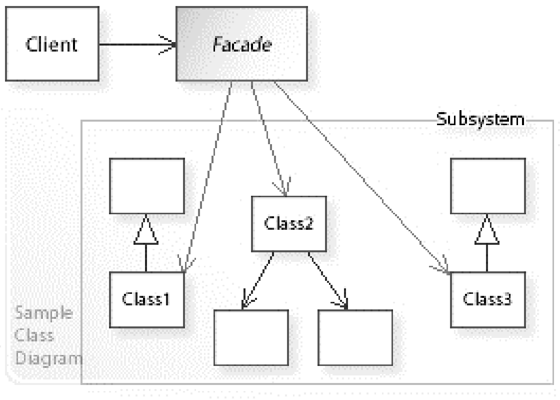
مفيد عندما تكون هناك حاجة إلى كائن واحد بالضبط.

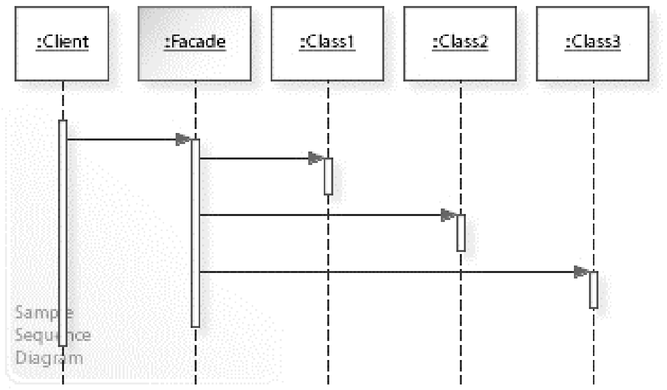
```
class Singleton
{
    //static variable single instance of type Singleton
    private static Singleton single_instance = null;

    // private constructor restricted to this class itself
    private Singleton() {
    }

    // static method to create instance of Singleton class
    public static Singleton getInstance()
    {
        if (single_instance == null)
            single_instance = new Singleton();

        return single_instance;
    }
}
```

Facade design pattern

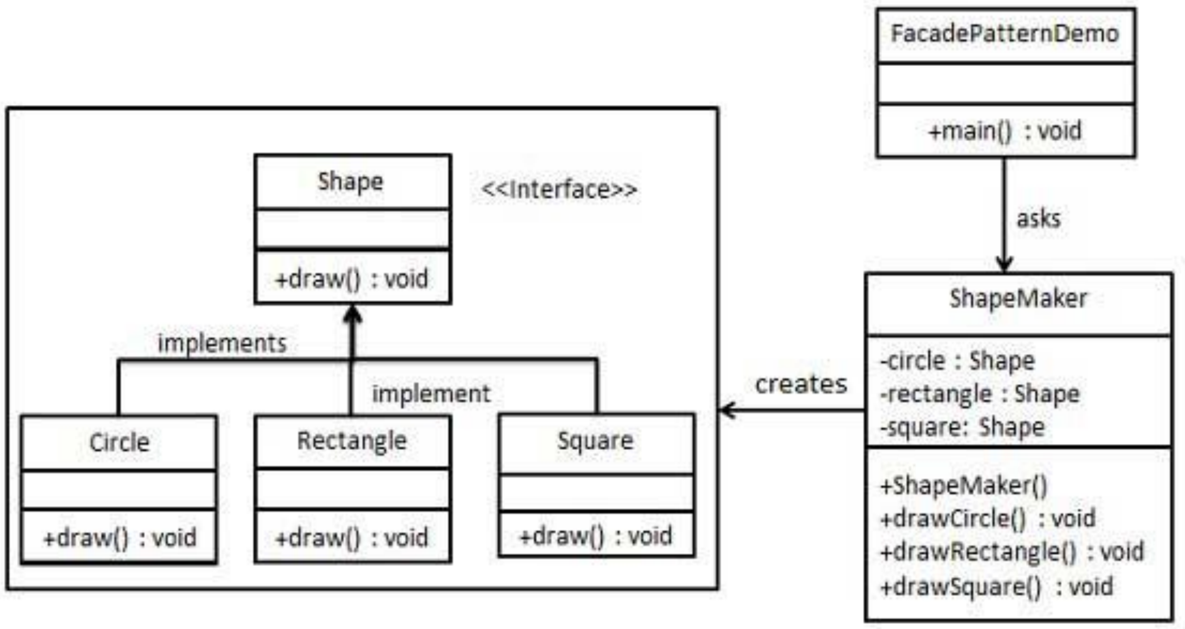
- structural pattern

غالبًا ما يستخدم المطورون نمط تصميم الواجهة عندما يكون النظام معقدًا للغاية أو يصعب فهمه لأن النظام يحتوي على العديد من الأصناف المترابطة أو لأن الكود المصدري غير متوفر. يخفي هذا النمط تعقيدات النظام الأكبر ويوفر واجهة أبسط للعميل.





Example:



Step 1

Create an interface.

Shape.java

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

Step 2

Create concrete classes implementing the same interface.

Rectangle.java

```
public class Rectangle implements Shape {

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

Square.java

```
public class Square implements Shape {

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

Circle.java

```
public class Circle implements Shape {

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

Step 4

Use the facade to draw various types of shapes.

FacadePatternDemo.java

```
public class FacadePatternDemo {
   public static void main(String[] args) {
      ShapeMaker shapeMaker = new ShapeMaker();

      shapeMaker.drawCircle();
      shapeMaker.drawRectangle();
      shapeMaker.drawSquare();
   }
}
```

Step 5

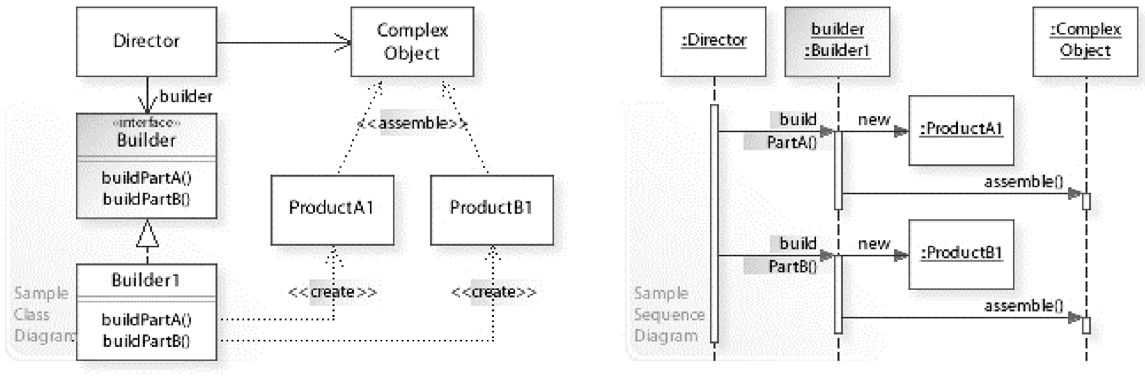
Verify the output.

```
Circle::draw()
Rectangle::draw()
Square::draw()
```

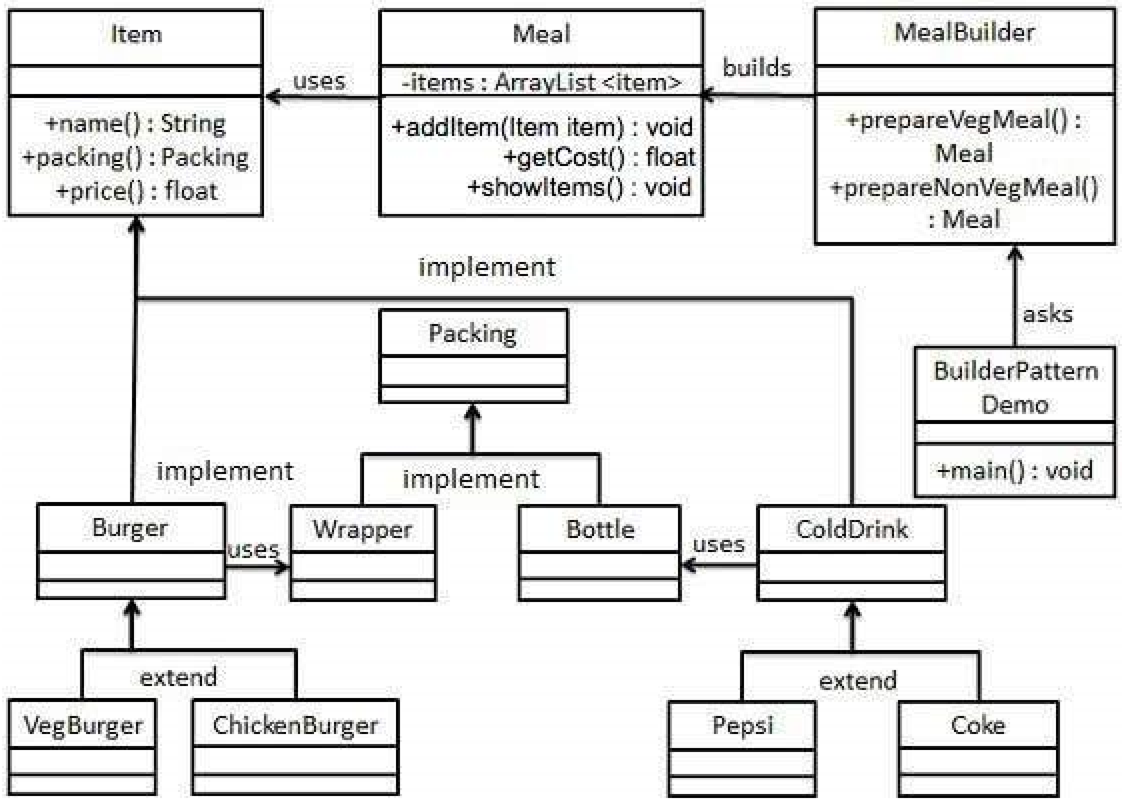
Builder design pattern

- Creational pattern

يفيد في انشاء كائن معقد باستخدام كائنات بسيطة.



Example:



Step 1

Create an interface Item representing food item and packing.

Item.java

```
public interface Item {
   public String name();
   public Packing packing();
   public float price();
}
```

Packing.java

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

Step 2

Create concrete classes implementing the Packing interface.

Wrapper.java

```
public class Wrapper implements Packing {

   @Override
   public String pack() {
      return "Wrapper";
   }
}
```

Bottle.java

```
public class Bottle implements Packing {

   @Override
   public String pack() {
      return "Bottle";
   }
}
```

Step 3

Create abstract classes implementing the item interface providing default functionalities.

Burger.java

```
public abstract class Burger implements Item {

   @Override
   public Packing packing() {
      return new Wrapper();
   }

   @Override
   public abstract float price();
}
```

ColdDrink.java

```
public abstract class ColdDrink implements Item {

       @Override
       public Packing packing() {
        return new Bottle();
       }

       @Override
       public abstract float price();
}
```

Step 4

Create concrete classes extending Burger and ColdDrink classes

VegBurger.java

```
public class VegBurger extends Burger {

   @Override
   public float price() {
      return 25.0f;
   }

   @Override
   public String name() {
      return "Veg Burger";
   }
}
```

ChickenBurger.java

```
public class ChickenBurger extends Burger {

   @Override
   public float price() {
      return 50.5f;
   }

   @Override
   public String name() {
      return "Chicken Burger";
   }
}
```

Coke.java

```
public class Coke extends ColdDrink {

   @Override
   public float price() {
      return 30.0f;
   }

   @Override
   public String name() {
      return "Coke";
   }
}
```

Pepsi.java

```
public class Pepsi extends ColdDrink {

   @Override
   public float price() {
      return 35.0f;
   }

   @Override
   public String name() {
      return "Pepsi";
   }
}
```

Step 5

Create a Meal class having Item objects defined above.

Meal.java

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

public class Meal {
   private List<Item> items = new ArrayList<Item>();

   public void addItem(Item item){
      items.add(item);
   }

   public float getCost(){
      float cost = 0.0f;

      for (Item item : items) {
         cost += item.price();
      }
      return cost;
   }

   public void showItems(){

      for (Item item : items) {
         System.out.print("Item : " + item.name());
         System.out.print(", Packing : " +
item.packing().pack());
         System.out.println(", Price : " + item.price());
      }
   }
}
```

Step 6

Create a MealBuilder class, the actual builder class responsible to create Meal objects.

MealBuilder.java

```
public class MealBuilder {

   public Meal prepareVegMeal (){
      Meal meal = new Meal();
      meal.addItem(new VegBurger());
      meal.addItem(new Coke());
      return meal;
   }

   public Meal prepareNonVegMeal (){
      Meal meal = new Meal();
      meal.addItem(new ChickenBurger());
      meal.addItem(new Pepsi());
      return meal;
   }
}
```

Step 7

BuiderPatternDemo uses MealBuider to demonstrate builder pattern.

BuilderPatternDemo.java

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

      MealBuilder mealBuilder = new MealBuilder();

      Meal vegMeal = mealBuilder.prepareVegMeal();
      System.out.println("Veg Meal");
      vegMeal.showItems();
      System.out.println("Total Cost: " + vegMeal.getCost());

      Meal nonVegMeal = mealBuilder.prepareNonVegMeal();
      System.out.println("\n\nNon-Veg Meal");
      nonVegMeal.showItems();
      System.out.println("Total Cost: " +
nonVegMeal.getCost());
   }
}
```

Step 8

Verify the output.

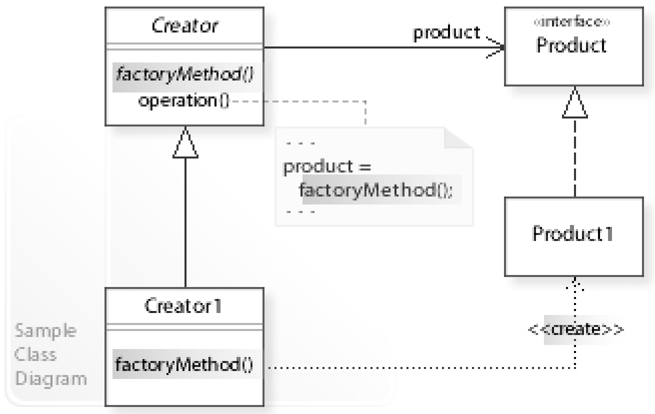
```
Veg Meal
Item : Veg Burger, Packing : Wrapper, Price : 25.0
Item : Coke, Packing : Bottle, Price : 30.0
Total Cost: 55.0


Non-Veg Meal
Item : Chicken Burger, Packing : Wrapper, Price : 50.5
Item : Pepsi, Packing : Bottle, Price : 35.0
Total Cost: 85.5
```

Factoty design pattern

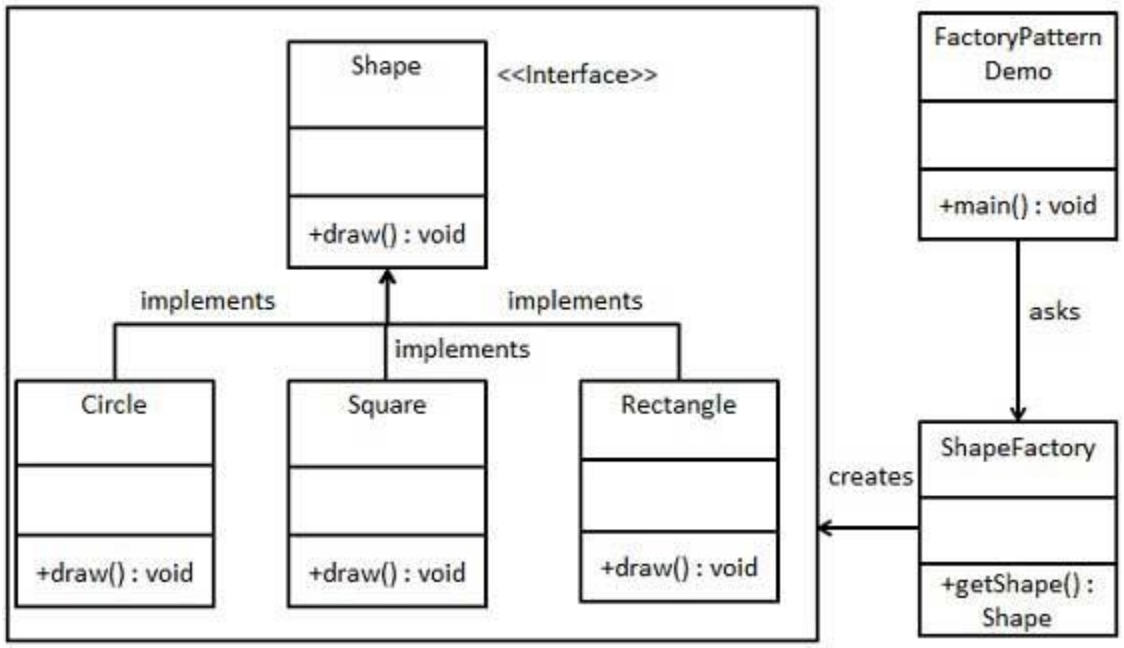
- Creational pattern

يفيد في انشاء كائن عن طريق فئة مشتقة من الفئة الأساسية.

للتعامل مع مشكلة إنشاء الكائنات دون الحاجة إلى تحديد صنف الكائن الذي سيتم إنشاؤه بالضبط.



Example:



Step 1

Create an interface.

Shape.java

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

Step 2

Create concrete classes implementing the same interface.

Rectangle.java

```
public class Rectangle implements Shape {

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

Square.java

```
public class Square implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

Circle.java

```
public class Circle implements Shape {

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

Step 3

Create a Factory to generate object of concrete class based on given information.

ShapeFactory.java

```
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;
   }
}
```

Step 4

Use the Factory to get object of concrete class by passing an information such as type.

FactoryPatternDemo.java

```
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();

      //get an object of Square and call its draw method.
      Shape shape3 = shapeFactory.getShape("SQUARE");

      //call draw method of square
      shape3.draw();
   }
}
```

Step 5

Verify the output.

Inside Circle::draw() method.

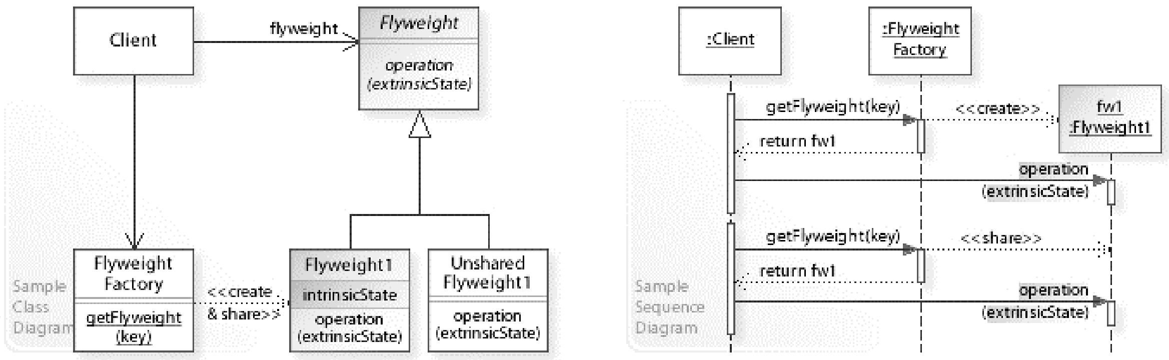
Inside Rectangle::draw() method.

Flyweight design pattern

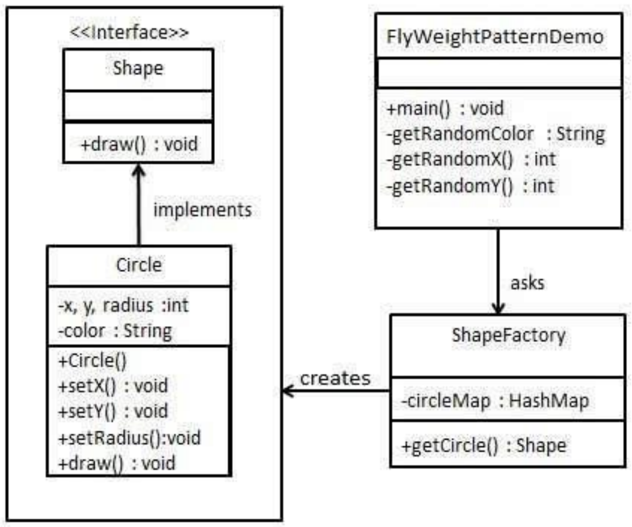
structural pattern

يستخدم هذا النمط بشكل أساسي لتقليل عدد الكائنات التي تم إنشاؤها وتقليل تأثير الذاكرة وزيادة الأداء. يأتي هذا النوع من أنماط التصميم تحت نمط هيكلي حيث يوفر هذا النمط طرقاً لتقليل عدد الكائنات وبالتالي تحسين بنية الكائن للتطبيق.

يحاول هذا النمط إعادة استخدام كائنات مماثلة موجودة بالفعل عن طريق تخزينها وإنشاء كائن جديد عند عدم العثور على كائن مطابق. سوف نوضح هذا النمط من خلال رسم 20 دائرة من مواقع مختلفة ولكننا سننشئ 5 كائنات فقط. تتوفر 5 ألوان فقط لذلك يتم استخدام خاصية اللون للتحقق من كائنات الدائرة الموجودة بالفعل.



Example:



Step 1

Create an interface.

Shape.java

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

Step 2

Create concrete class implementing the same interface.

Circle.java

```
public class Circle implements Shape {
   private String color;
   private int x;
   private int y;
   private int radius;

   public Circle(String color){
      this.color = color;
   }

   public void setX(int x) {
      this.x = x;
   }

   public void setY(int y) {
      this.y = y;
   }

   public void setRadius(int radius) {
      this.radius = radius;
   }

   @Override
   public void draw() {
      System.out.println("Circle: Draw() [Color : " + color +
", x : " + x + ", y :" + y + ", radius :" + radius);
   }
}
```

Step 3

Create a factory to generate object of concrete class based on given information.

ShapeFactory.java

```
import java.util.HashMap;

public class ShapeFactory {

   // Uncomment the compiler directive line and
   // javac *.java will compile properly.
   // @SuppressWarnings("unchecked")
   private static final HashMap circleMap = new HashMap();

   public static Shape getCircle(String color) {
      Circle circle = (Circle)circleMap.get(color);

      if(circle == null) {
         circle = new Circle(color);
         circleMap.put(color, circle);
         System.out.println("Creating circle of color : " +
color);
      }
      return circle;
   }
}
```

Step 4

Use the factory to get object of concrete class by passing an information such as color.

FlyweightPatternDemo.java

```
public class FlyweightPatternDemo {
   private static final String colors[] = { "Red", "Green",
"Blue", "White", "Black" };
   public static void main(String[] args) {

      for(int i=0; i < 20; ++i) {
         Circle circle =
(Circle)ShapeFactory.getCircle(getRandomColor());
         circle.setX(getRandomX());
         circle.setY(getRandomY());
         circle.setRadius(100);
         circle.draw();
      }
   }
   private static String getRandomColor() {
      return colors[(int)(Math.random()*colors.length)];
   }
   private static int getRandomX() {
      return (int)(Math.random()*100 );
   }
   private static int getRandomY() {
      return (int)(Math.random()*100);
   }
}
```

Step 5

Verify the output

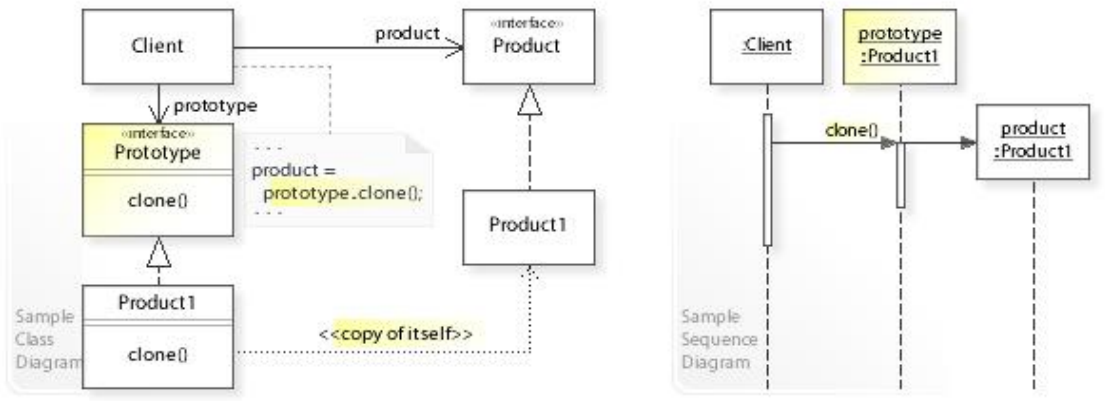
```
Creating circle of color : Black
Circle: Draw() [Color : Black, x : 36, y :71, radius :100
Creating circle of color : Green
Circle: Draw() [Color : Green, x : 27, y :27, radius :100
Creating circle of color : White
Circle: Draw() [Color : White, x : 64, y :10, radius :100
Creating circle of color : Red
Circle: Draw() [Color : Red, x : 15, y :44, radius :100
Circle: Draw() [Color : Green, x : 19, y :10, radius :100
Circle: Draw() [Color : Green, x : 94, y :32, radius :100
Circle: Draw() [Color : White, x : 69, y :98, radius :100
Creating circle of color : Blue
Circle: Draw() [Color : Blue, x : 13, y :4, radius :100
Circle: Draw() [Color : Green, x : 21, y :21, radius :100
Circle: Draw() [Color : Blue, x : 55, y :86, radius :100
Circle: Draw() [Color : White, x : 90, y :70, radius :100
Circle: Draw() [Color : Green, x : 78, y :3, radius :100
Circle: Draw() [Color : Green, x : 64, y :89, radius :100
Circle: Draw() [Color : Blue, x : 3, y :91, radius :100
Circle: Draw() [Color : Blue, x : 62, y :82, radius :100
Circle: Draw() [Color : Green, x : 97, y :61, radius :100
Circle: Draw() [Color : Green, x : 86, y :12, radius :100
Circle: Draw() [Color : Green, x : 38, y :93, radius :100
Circle: Draw() [Color : Red, x : 76, y :82, radius :100
Circle: Draw() [Color : Blue, x : 95, y :82, radius :100
```

Prototype design pattern

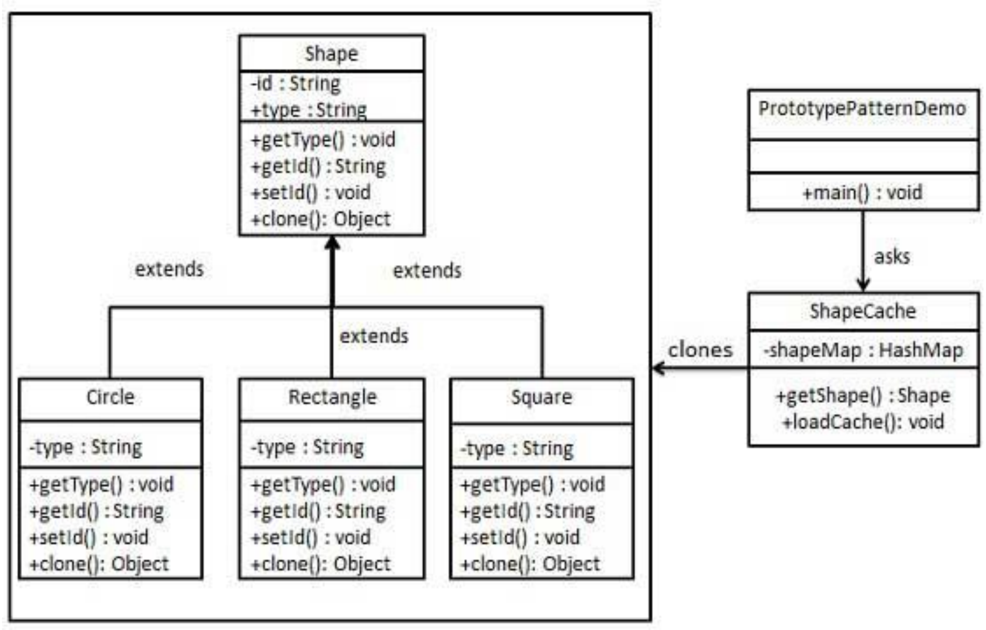
Creational pattern

يفيد في انشاء كائن مكرر مع مراعاة الأداء.

يمكننا تخزين الكائن مؤقتًا وإرجاع نسخته عند الطلب التالي وتحديث قاعدة البيانات عند الحاجة وبالتالي تقليل استدعاءات قاعدة البيانات.



Example:



Step 1

Create an abstract class implementing Clonable interface.

Shape.java

```
public abstract class Shape implements Cloneable {

   private String id;
   protected String type;

   abstract void draw();

   public String getType(){
      return type;
   }

   public String getId() {
      return id;
   }

   public void setId(String id) {
      this.id = id;
   }

   public Object clone() {
      Object clone = null;

      try {
         clone = super.clone();

      } catch (CloneNotSupportedException e) {
         e.printStackTrace();
      }

      return clone;
   }
}
```

Step 2

Create concrete classes extending the above class.

Rectangle.java

```
public class Rectangle extends Shape {

   public Rectangle(){
     type = "Rectangle";
   }

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

Square.java

```
public class Square extends Shape {

   public Square(){
     type = "Square";
   }

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

Circle.java

```java
public class Circle extends Shape {

   public Circle(){
     type = "Circle";
   }

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

Step 3

Create a class to get concrete classes from database and store them in a *Hashtable*.

ShapeCache.java

```java
import java.util.Hashtable;

public class ShapeCache {

   private static Hashtable<String, Shape> shapeMap  = new
Hashtable<String, Shape>();

   public static Shape getShape(String shapeId) {
      Shape cachedShape = shapeMap.get(shapeId);
      return (Shape) cachedShape.clone();
   }

   // for each shape run database query and create shape
   // shapeMap.put(shapeKey, shape);
   // for example, we are adding three shapes

   public static void loadCache() {
      Circle circle = new Circle();
      circle.setId("1");
      shapeMap.put(circle.getId(),circle);

      Square square = new Square();
      square.setId("2");
      shapeMap.put(square.getId(),square);

      Rectangle rectangle = new Rectangle();
      rectangle.setId("3");
      shapeMap.put(rectangle.getId(), rectangle);
   }
}
```

Step 4

PrototypePatternDemo uses ShapeCache class to get clones of shapes stored in a Hashtable.

PrototypePatternDemo.java

```java
public class PrototypePatternDemo {
   public static void main(String[] args) {
      ShapeCache.loadCache();

      Shape clonedShape = (Shape) ShapeCache.getShape("1");
      System.out.println("Shape : " + clonedShape.getType());

      Shape clonedShape2 = (Shape) ShapeCache.getShape("2");
      System.out.println("Shape : " + clonedShape2.getType());

      Shape clonedShape3 = (Shape) ShapeCache.getShape("3");
      System.out.println("Shape : " + clonedShape3.getType());
   }
}
```

Step 5

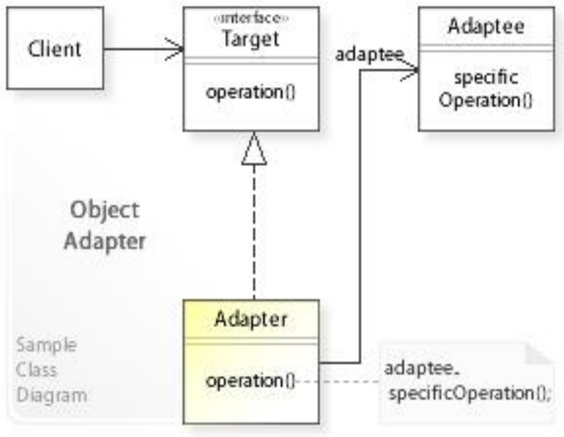
Verify the output.

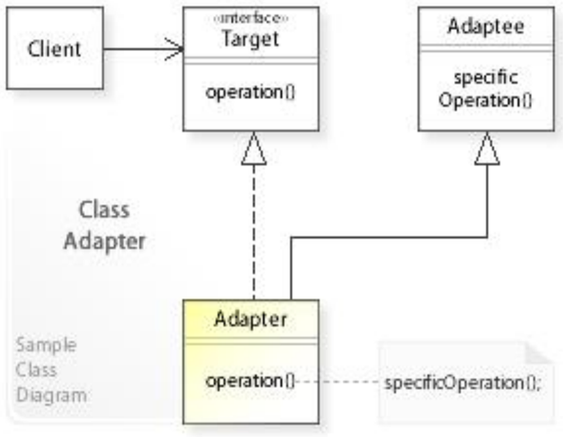
```
Shape : Circle
Shape : Square
Shape : Rectangle
```

Adapter design pattern

يعمل نمط المحول كجسر بين واجهتين غير متوافقين. يأتي هذا النوع من أنماط التصميم تحت نمط هيكلي حيث يجمع هذا النمط بين قدرة واجهتين مستقلتين.

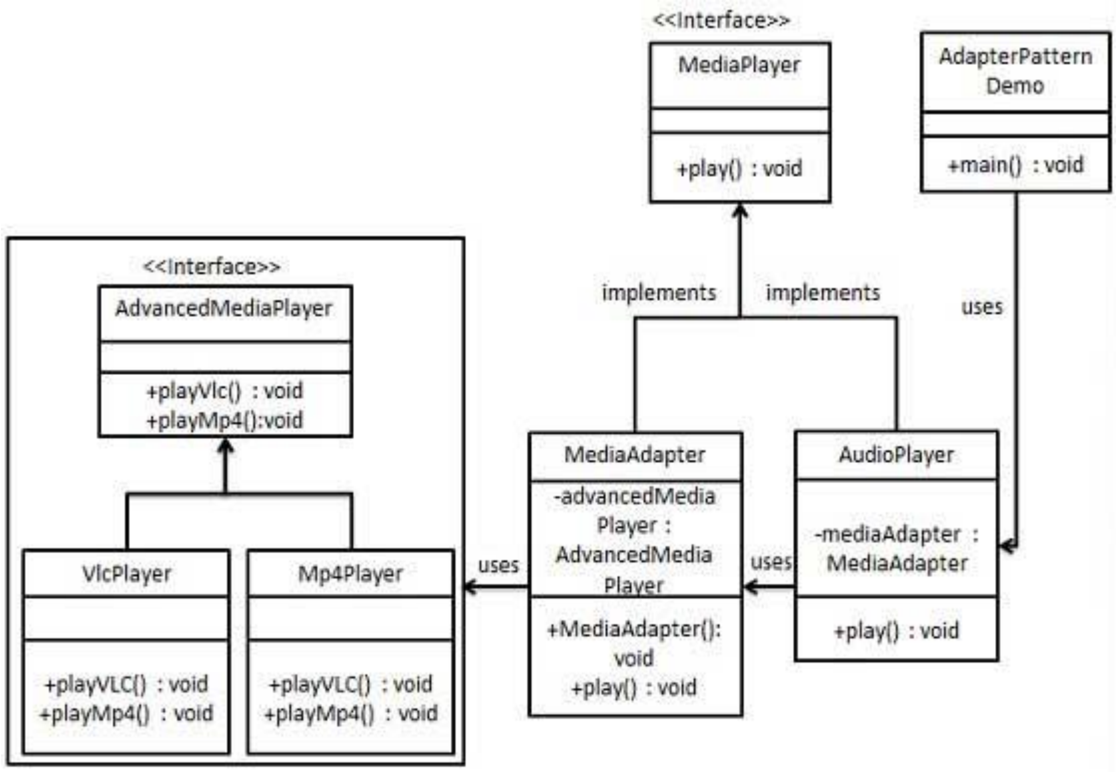
يتضمن هذا النمط فئة واحدة مسؤولة عن ضم وظائف الواجهات المستقلة أو غير المتوافقة. مثال واقعي يمكن أن يكون حالة قارئ بطاقة يعمل كمحول بين بطاقة الذاكرة وجهاز كمبيوتر محمول. تقوم بتوصيل بطاقة الذاكرة بقارئ البطاقات وقارئ البطاقات في الكمبيوتر المحمول بحيث يمكن قراءة بطاقة الذاكرة عبر الكمبيوتر المحمول.





Example:

example in which an audio player device can play mp3 files only and wants to use an advanced audio player capable of playing vlc and mp4 files



Step 1

Create interfaces for Media Player and Advanced Media Player.

MediaPlayer.java

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

AdvancedMediaPlayer.java

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

Step 2

Create concrete classes implementing the *AdvancedMediaPlayer* interface.

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
   }
}
```

Mp4Player.java

```
public class Mp4Player implements AdvancedMediaPlayer{

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

   @Override
   public void playMp4(String fileName) {
      System.out.println("Playing mp4 file. Name: "+ fileName);

   }
}
```

Step 3

Create adapter class implementing the *MediaPlayer* interface.

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);
      }
   }
}
```

Step 4

Create concrete class implementing the *MediaPlayer* interface.

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");
      }
   }
}
```

Step 5

Use the AudioPlayer to play different types of audio formats.

AdapterPatternDemo.java

```java
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");
   }
}
```

Step 6

Verify the output.

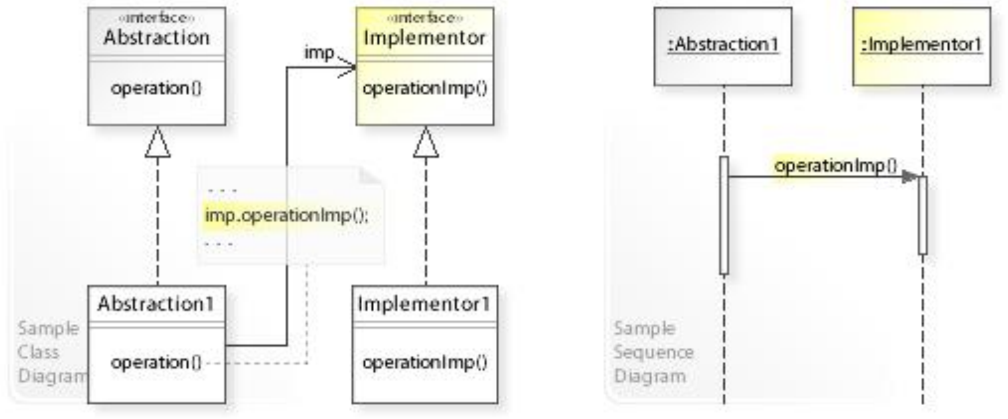
```
Playing mp3 file. Name: beyond the horizon.mp3
Playing mp4 file. Name: alone.mp4
Playing vlc file. Name: far far away.vlc
Invalid media. avi format not supported
```

Bridge design pattern

structural pattern

يقوم على فصل كيفية بداية عمل الكائن (الغرض) و المحتوى أو الناتج النهائي فهو يفصل بين التجريد و التطبيق.

وهذا يفيد في البرامج التي يكون لها تحديثات فجزء منها ثابت وهو الموضوع به التجريد وهو الذي يقوم باستدعاء دالة معينة في الجزء الاخر وهو التطبيق فتظهر النتائج دون تدخل من الجزء الأول.



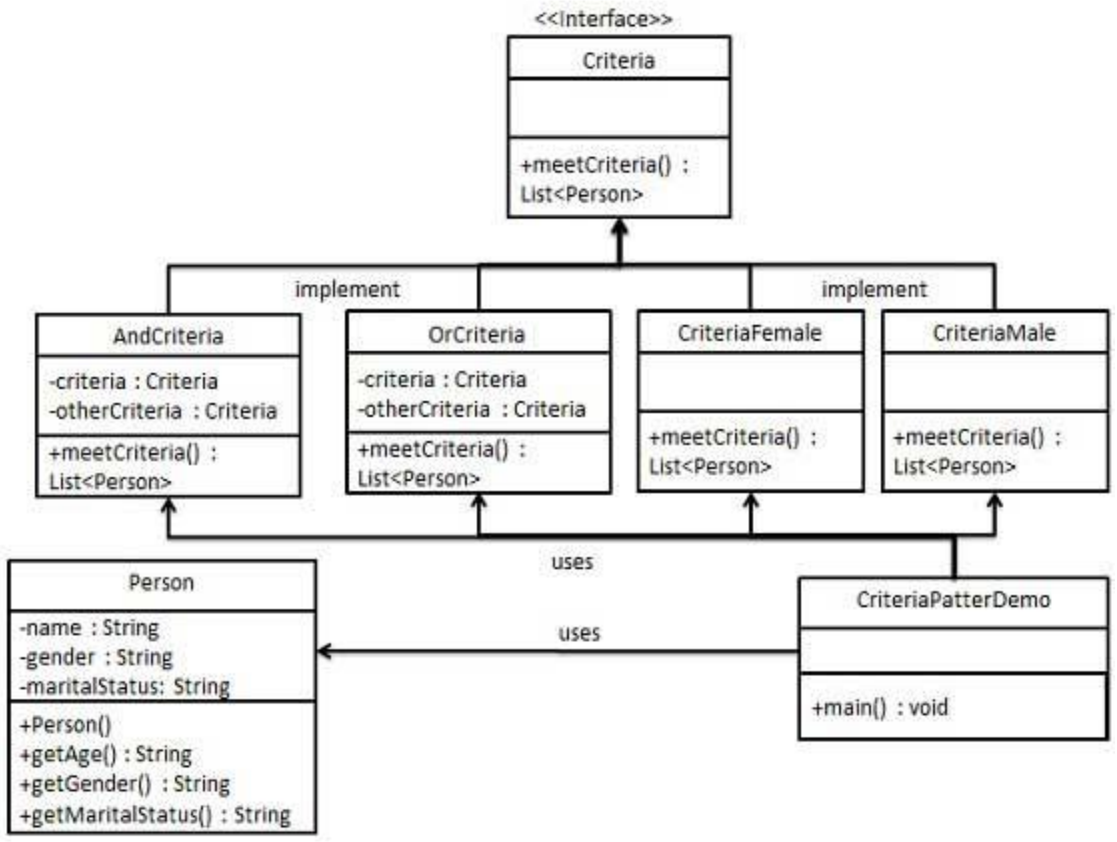
Example:

```
using System;

class BridgePattern
{

    interface Bridge
    {
        string OperationImp();
    }

    // Bridge Pattern Judith Bishop Dec 2006, Aug 2007
    // Shows an abstraction and two implementations proceeding
independently

    class Abstraction
    {
        Bridge bridge;
        public Abstraction(Bridge implementation)
        {
            bridge = implementation;
        }
        public string Operation()
        {
            return "Abstraction" + "<<< BRIDGE >>>> " +
bridge.OperationImp();
        }
    }

    class ImplementationA : Bridge
    {
        public string OperationImp()
        {
            return "ImplementationA";
        }
    }

    class ImplementationB : Bridge
    {
        public string OperationImp()
        {
            return "ImplementationB";
        }
    }
static void Main()
    {
        Console.WriteLine("Bridge Pattern\n");
        Console.WriteLine(new Abstraction(new
ImplementationA()).Operation());
        Console.WriteLine(new Abstraction(new
ImplementationB()).Operation());
    }
}

/* Output
Bridge Pattern
Abstraction <<< BRIDGE >>>>ImplementationA
Abstraction <<< BRIDGE >>>> ImplementationB
*/
```

Filter design pattern

structural pattern

يمكّن المطورين من تصفية مجموعة من الكائنات باستخدام معايير مختلفة وتسلسلها بطريقة منفصلة من خلال العمليات المنطقية. يأتي هذا النوع من أنماط التصميم تحت نمط هيكلي حيث يجمع هذا النمط بين معايير متعددة للحصول على معايير واحدة.



Step 1

Create a class on which criteria is to be applied.

Person.java

```
public class Person {

   private String name;
   private String gender;
   private String maritalStatus;

   public Person(String name, String gender, String maritalStatus){
      this.name = name;
      this.gender = gender;
      this.maritalStatus = maritalStatus;
   }

   public String getName() {
      return name;
   }
   public String getGender() {
      return gender;
   }
   public String getMaritalStatus() {
      return maritalStatus;
   }
}
```

Step 2

Create an interface for Criteria.

Criteria.java

```
import java.util.List;

public interface Criteria {
   public List<Person> meetCriteria(List<Person> persons);
}
```

Step 3

Create concrete classes implementing the *Criteria* interface.

CriteriaMale.java

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

public class CriteriaMale implements Criteria {

   @Override
   public List<Person> meetCriteria(List<Person> persons) {
      List<Person> malePersons = new ArrayList<Person>();

      for (Person person : persons) {
         if(person.getGender().equalsIgnoreCase("MALE")){
            malePersons.add(person);
         }
      }
      return malePersons;
   }
}
```

CriteriaFemale.java

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

public class CriteriaFemale implements Criteria {

   @Override
   public List<Person> meetCriteria(List<Person> persons) {
      List<Person> femalePersons = new ArrayList<Person>();

      for (Person person : persons) {
         if(person.getGender().equalsIgnoreCase("FEMALE")){
            femalePersons.add(person);
         }
      }
      return femalePersons;
   }
}
```

CriteriaSingle.java

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

public class CriteriaSingle implements Criteria {

   @Override
   public List<Person> meetCriteria(List<Person> persons) {
      List<Person> singlePersons = new ArrayList<Person>();

      for (Person person : persons) {
         if(person.getMaritalStatus().equalsIgnoreCase("SINGLE")){
            singlePersons.add(person);
         }
      }
      return singlePersons;
   }
}
```

AndCriteria.java

```
import java.util.List;

public class AndCriteria implements Criteria {

   private Criteria criteria;
   private Criteria otherCriteria;

   public AndCriteria(Criteria criteria, Criteria otherCriteria) {
      this.criteria = criteria;
      this.otherCriteria = otherCriteria;
   }

   @Override
   public List<Person> meetCriteria(List<Person> persons) {

      List<Person> firstCriteriaPersons =
criteria.meetCriteria(persons);
      return otherCriteria.meetCriteria(firstCriteriaPersons);
   }
}
```

OrCriteria.java

```java
import java.util.List;

public class OrCriteria implements Criteria {

   private Criteria criteria;
   private Criteria otherCriteria;

   public OrCriteria(Criteria criteria, Criteria otherCriteria) {
      this.criteria = criteria;
      this.otherCriteria = otherCriteria;
   }

   @Override
   public List<Person> meetCriteria(List<Person> persons) {
      List<Person> firstCriteriaItems =
criteria.meetCriteria(persons);
      List<Person> otherCriteriaItems =
otherCriteria.meetCriteria(persons);

      for (Person person : otherCriteriaItems) {
         if(!firstCriteriaItems.contains(person)){
            firstCriteriaItems.add(person);
         }
      }
      return firstCriteriaItems;
   }
}
```

Step 4

Use different Criteria and their combination to filter out persons.

CriteriaPatternDemo.java

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

public class CriteriaPatternDemo {
   public static void main(String[] args) {
      List<Person> persons = new ArrayList<Person>();

      persons.add(new Person("Robert","Male", "Single"));
      persons.add(new Person("John", "Male", "Married"));
      persons.add(new Person("Laura", "Female", "Married"));
      persons.add(new Person("Diana", "Female", "Single"));
      persons.add(new Person("Mike", "Male", "Single"));
      persons.add(new Person("Bobby", "Male", "Single"));

      Criteria male = new CriteriaMale();
      Criteria female = new CriteriaFemale();
      Criteria single = new CriteriaSingle();
      Criteria singleMale = new AndCriteria(single, male);
      Criteria singleOrFemale = new OrCriteria(single, female);

      System.out.println("Males: ");
      printPersons(male.meetCriteria(persons));

      System.out.println("\nFemales: ");
      printPersons(female.meetCriteria(persons));

      System.out.println("\nSingle Males: ");
      printPersons(singleMale.meetCriteria(persons));

      System.out.println("\nSingle Or Females: ");
      printPersons(singleOrFemale.meetCriteria(persons));
   }

   public static void printPersons(List<Person> persons){

      for (Person person : persons) {
         System.out.println("Person : [ Name : " + person.getName()
+ ", Gender : " + person.getGender() + ", Marital Status : " +
person.getMaritalStatus() + " ]");
      }
   }
}
```

Step 5

Verify the output.

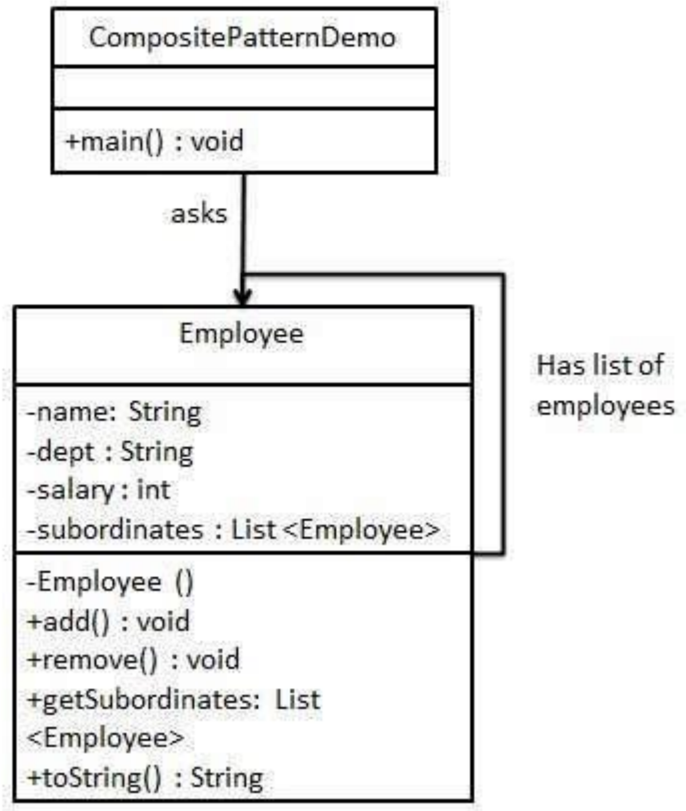
```
Males:
Person : [ Name : Robert, Gender : Male, Marital Status : Single ]
Person : [ Name : John, Gender : Male, Marital Status : Married ]
Person : [ Name : Mike, Gender : Male, Marital Status : Single ]
Person : [ Name : Bobby, Gender : Male, Marital Status : Single ]

Females:
Person : [ Name : Laura, Gender : Female, Marital Status : Married ]
Person : [ Name : Diana, Gender : Female, Marital Status : Single ]

Single Males:
Person : [ Name : Robert, Gender : Male, Marital Status : Single ]
Person : [ Name : Mike, Gender : Male, Marital Status : Single ]
Person : [ Name : Bobby, Gender : Male, Marital Status : Single ]

Single Or Females:
Person : [ Name : Robert, Gender : Male, Marital Status : Single ]
Person : [ Name : Diana, Gender : Female, Marital Status : Single ]
Person : [ Name : Mike, Gender : Male, Marital Status : Single ]
Person : [ Name : Bobby, Gender : Male, Marital Status : Single ]
Person : [ Name : Laura, Gender : Female, Marital Status : Married ]
```

Compsite design pattern

يتم استخدام النمط المركب حيث نحتاج إلى معاملة مجموعة من الكائنات بطريقة مماثلة ككائن واحد. يقوم النمط المركب بتكوين كائنات من حيث هيكل الشجرة لتمثيل التسلسل الهرمي الكامل والجزء الآخر. يأتي هذا النوع من أنماط التصميم تحت نمط هيكلي لأن هذا النمط يخلق بنية شجرية لمجموعة من الكائنات.

ينشئ هذا النمط فئة تحتوي على مجموعة من الكائنات الخاصة بها. توفر هذه الفئة طرقًا لتعديل مجموعتها من نفس الكائنات.

نحن نعرض استخدام النمط المركب من خلال المثال التالي الذي سنعرض فيه التسلسل الهرمي للموظفين في المؤسسة.



Step 1

Create *Employee* class having list of *Employee* objects.

Employee.java

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

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 + ", 
salary :" + salary+" ]");
   }
}
```

Step 2

Use the *Employee* class to create and print employee hierarchy.

CompositePatternDemo.java

```
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);
         }
      }
   }
}
```

Step 3

Verify the output.

```
Employee :[ Name : John, dept : CEO, salary :30000 ]
Employee :[ Name : Robert, dept : Head Sales, salary :20000 ]
Employee :[ Name : Richard, dept : Sales, salary :10000 ]
Employee :[ Name : Rob, dept : Sales, salary :10000 ]
Employee :[ Name : Michel, dept : Head Marketing, salary :20000 ]
Employee :[ Name : Laura, dept : Marketing, salary :10000 ]
Employee :[ Name : Bob, dept : Marketing, salary :10000 ]
```

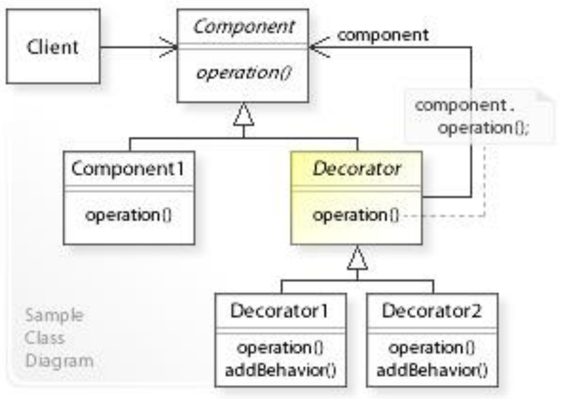
Decarotor design pattern

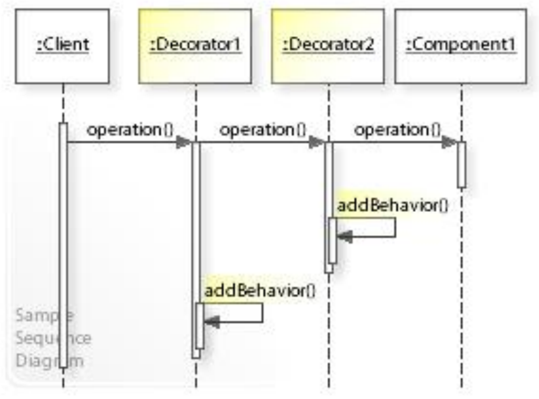
- Structural pattern

يسمح هذا النمط للمستخدم بإضافة وظائف جديدة إلى كائن موجود دون تغيير هيكله. يأتي هذا النوع من أنماط التصميم تحت نمط هيكلي حيث يعمل هذا النمط كغلاف للفئة الحالية.

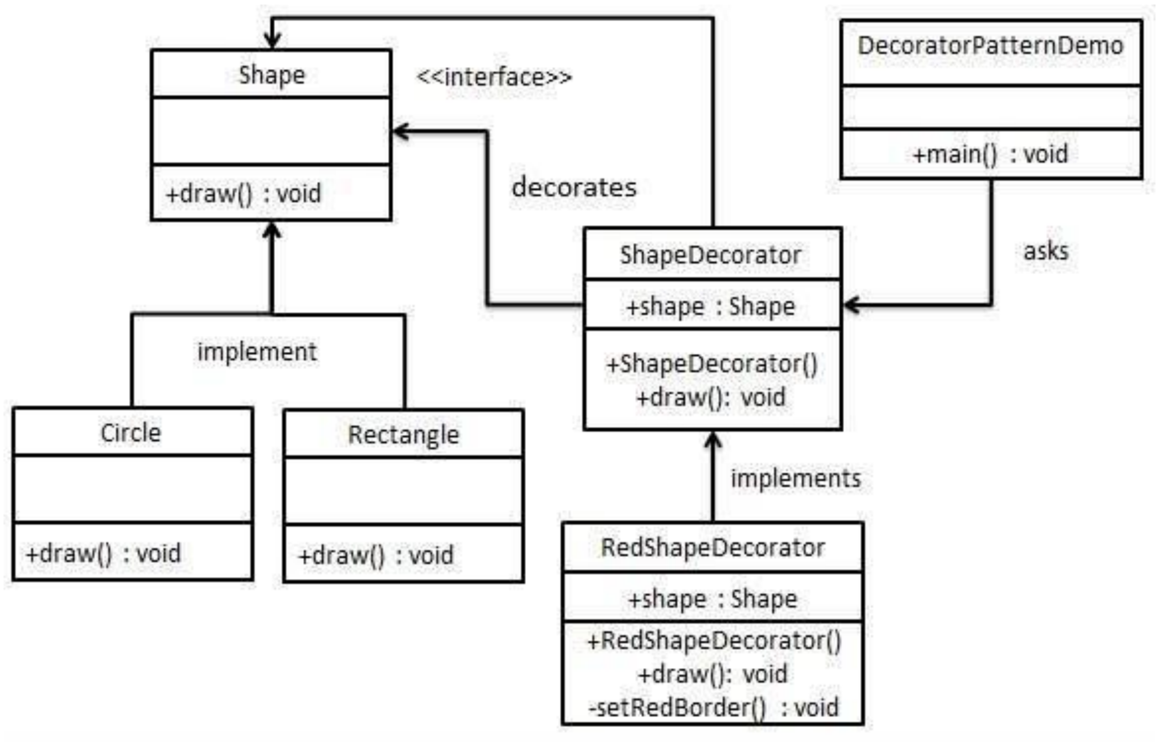
يُنشئ هذا النمط فئة مصممًا تلتف بالفئة الأصلية وتوفر وظائف إضافية للحفاظ على توقيع طرق الفصل كما هو.

نحن نعرض استخدام نمط الديكور من خلال المثال التالي الذي سنقوم فيه بتزيين شكل ببعض الألوان دون تغيير فئة الشكل.





Example:



Step 1

Create an interface.

Shape.java

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

Step 2

Create concrete classes implementing the same interface.

Rectangle.java

```
public class Rectangle implements Shape {

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

Circle.java

```
public class Circle implements Shape {

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

Step 3

Create abstract decorator class implementing the *Shape* interface.

ShapeDecorator.java

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

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

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

Step 4

Create concrete decorator class extending the *ShapeDecorator* class.

RedShapeDecorator.java

```
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");
   }
}
```

Step 5

Use the *RedShapeDecorator* to decorate *Shape* objects.

DecoratorPatternDemo.java

```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();
   }
}
```

Step 6

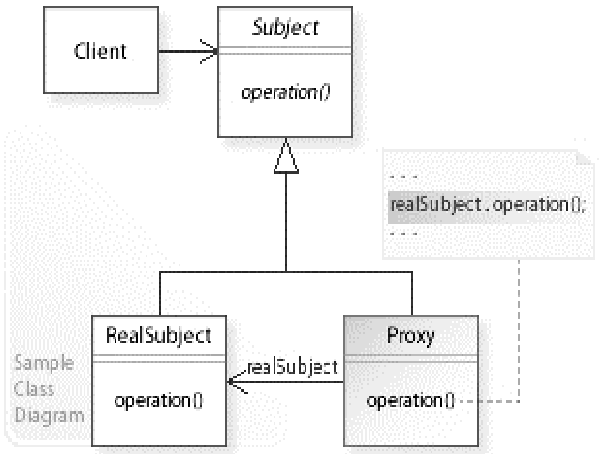
Verify the output.

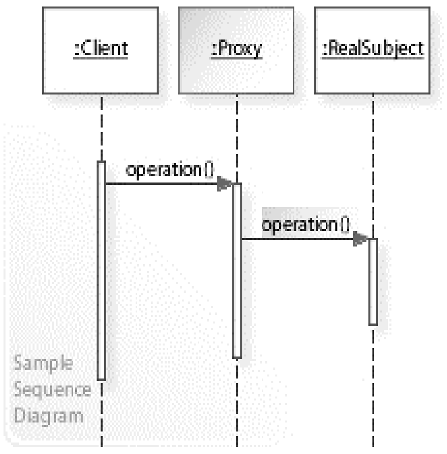
```
Circle with normal border
Shape: Circle

Circle of red border
Shape: Circle
Border Color: Red

Rectangle of red border
Shape: Rectangle
Border Color: Red
```

Proxy design pattern

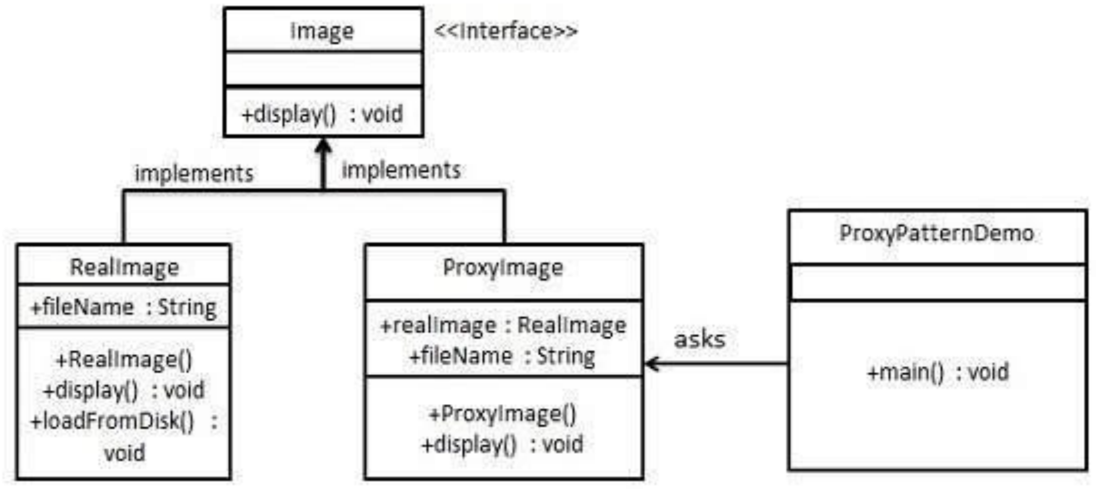
في نمط الوكيل ، تمثل الفئة وظائف فئة أخرى. يأتي هذا النوع من أنماط التصميم تحت نمط هيكلي.

في نمط الوكيل ، نقوم بإنشاء كائن به كائن أصلي لربط وظائفه بالعالم الخارجي.





Example:



Step 1

Create an interface.

Image.java

```
public interface Image {
   void display();
}
```

Step 2

Create concrete classes implementing the same interface.

RealImage.java

```
public class RealImage implements Image {

   private String fileName;

   public RealImage(String fileName){
      this.fileName = fileName;
      loadFromDisk(fileName);
   }

   @Override
   public void display() {
      System.out.println("Displaying " + fileName);
   }

   private void loadFromDisk(String fileName){
      System.out.println("Loading " + fileName);
   }
}
```

ProxyImage.java

```
public class ProxyImage implements Image{

   private RealImage realImage;
   private String fileName;

   public ProxyImage(String fileName){
      this.fileName = fileName;
   }

   @Override
   public void display() {
      if(realImage == null){
         realImage = new RealImage(fileName);
```

```
        }
        realImage.display();
    }
}
```

Step 3

Use the *ProxyImage* to get object of *RealImage* class when required.

ProxyPatternDemo.java

```
public class ProxyPatternDemo {

    public static void main(String[] args) {
        Image image = new ProxyImage("test_10mb.jpg");

        //image will be loaded from disk
        image.display();
        System.out.println("");

        //image will not be loaded from disk
        image.display();
    }
}
```

Step 4

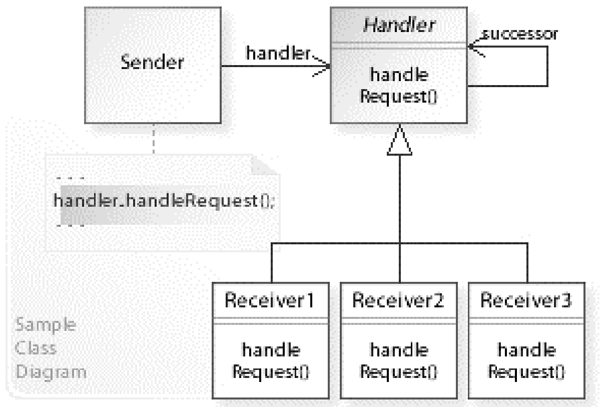
Verify the output.

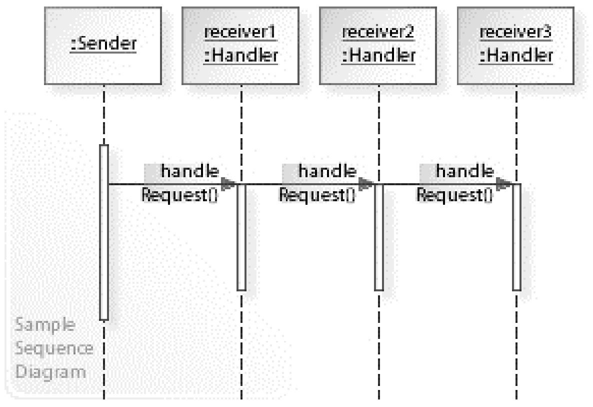
```
Loading test_10mb.jpg
Displaying test_10mb.jpg

Displaying test_10mb.jpg
```

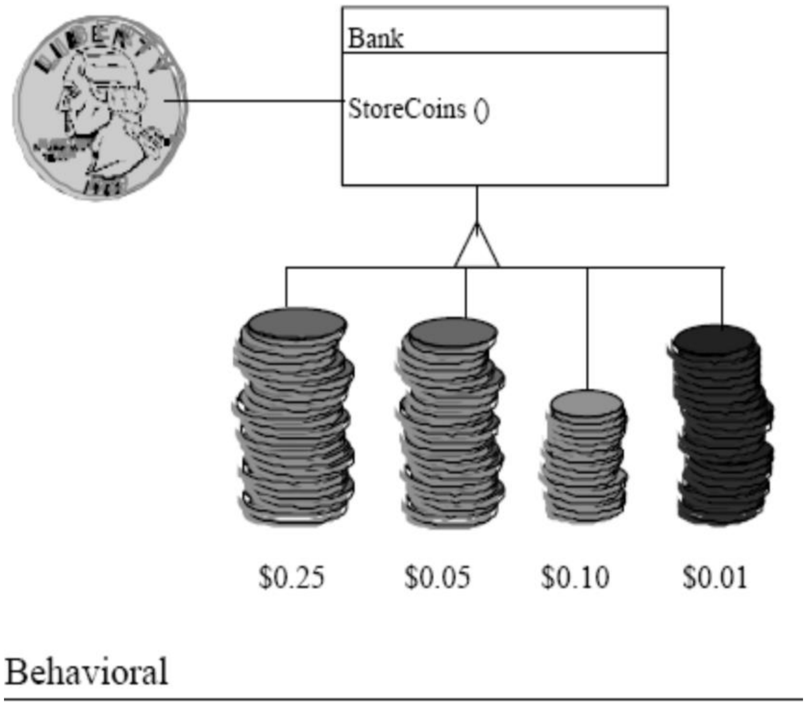
Chain of responsabilty

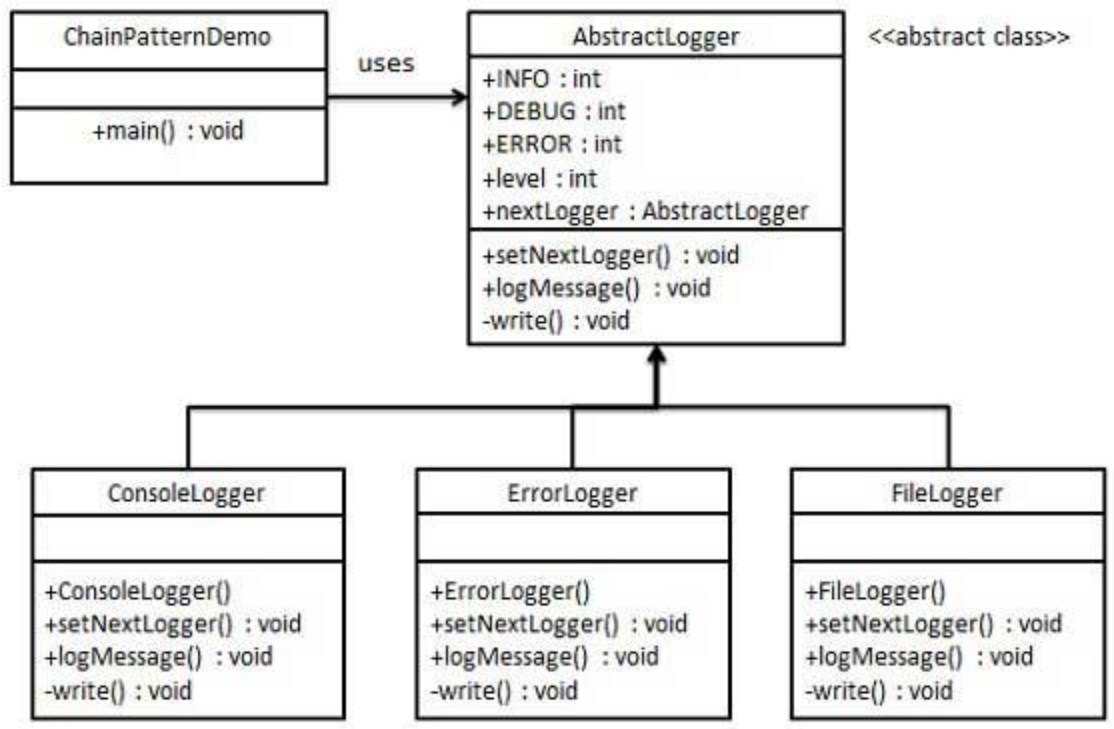
Behavioral pattern

سلسلة من كائنات المعالجة. يحتوي كل كائن معالجة على منطق يحدد أنواع كائنات الأوامر التي يمكنه التعامل معها ؛ يتم تمرير الباقي إلى كائن المعالجة التالي في السلسلة









Step 1

Create an abstract logger class.

AbstractLogger.java

```
public abstract class AbstractLogger {
   public static int INFO = 1;
   public static int DEBUG = 2;
   public static int ERROR = 3;

   protected int level;

   //next element in chain or responsibility
   protected AbstractLogger nextLogger;

   public void setNextLogger(AbstractLogger nextLogger){
      this.nextLogger = nextLogger;
   }

   public void logMessage(int level, String message){
      if(this.level <= level){
         write(message);
      }
      if(nextLogger !=null){
         nextLogger.logMessage(level, message);
      }
   }

   abstract protected void write(String message);

}
```

Step 2

Create concrete classes extending the logger.

ConsoleLogger.java

```
public class ConsoleLogger extends AbstractLogger {

   public ConsoleLogger(int level){
      this.level = level;
   }

   @Override
   protected void write(String message) {
      System.out.println("Standard Console::Logger: " + message);
   }
}
```

ErrorLogger.java

```
public class ErrorLogger extends AbstractLogger {

   public ErrorLogger(int level){
      this.level = level;
   }

   @Override
   protected void write(String message) {
      System.out.println("Error Console::Logger: " + message);
   }
}
```

FileLogger.java

```
public class FileLogger extends AbstractLogger {
   public FileLogger(int level){
      this.level = level;
   }

   @Override
   protected void write(String message) {
      System.out.println("File::Logger: " + message);
   }
}
```

Step 3

Create different types of loggers. Assign them error levels and set next logger in each logger. Next logger in each logger represents the part of the chain.

ChainPatternDemo.java

```
public class ChainPatternDemo {

   private static AbstractLogger getChainOfLoggers(){

      AbstractLogger errorLogger = new
ErrorLogger(AbstractLogger.ERROR);
      AbstractLogger fileLogger = new
FileLogger(AbstractLogger.DEBUG);
      AbstractLogger consoleLogger = new
ConsoleLogger(AbstractLogger.INFO);

      errorLogger.setNextLogger(fileLogger);
      fileLogger.setNextLogger(consoleLogger);

      return errorLogger;
   }

   public static void main(String[] args) {
      AbstractLogger loggerChain = getChainOfLoggers();

      loggerChain.logMessage(AbstractLogger.INFO,
         "This is an information.");

      loggerChain.logMessage(AbstractLogger.DEBUG,
         "This is an debug level information.");

      loggerChain.logMessage(AbstractLogger.ERROR,
         "This is an error information.");
   }
}
```

Step 4

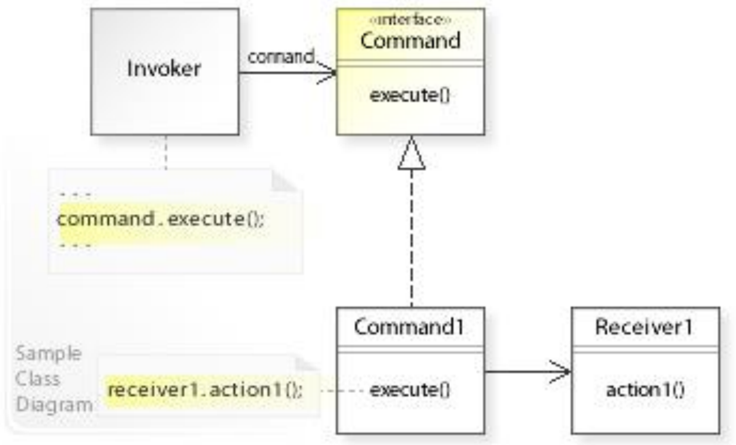
Verify the output.

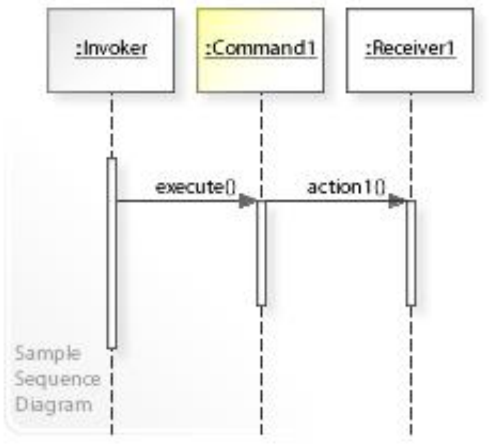
```
Standard Console::Logger: This is an information.
File::Logger: This is an debug level information.
Standard Console::Logger: This is an debug level information.
Error Console::Logger: This is an error information.
File::Logger: This is an error information.
Standard Console::Logger: This is an error information.
```

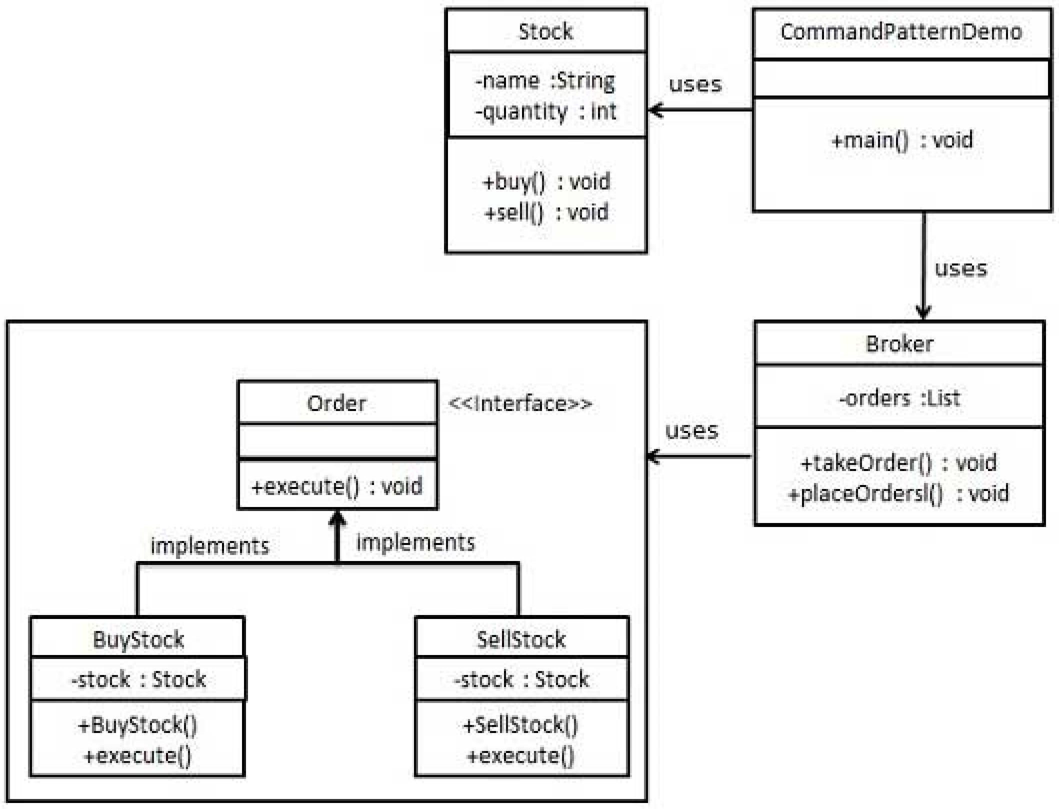
Command pattern

- Behavioral pattern

يستخدم فيه كائن لتغليف جميع المعلومات اللازمة لتنفيذ إجراء أو إطلاق حدث في وقت لاحق. تتضمن هذه المعلومات اسم الأسلوب والكائن الذي يمتلك الأسلوب والقيم لمعلمات الأسلوب. هناك أربعة مصطلحات مرتبطة دائمًا بنمط الأوامر وهي الأوامر ، المتلقي ، المستدعي والعميل.







Step 1

Create a command interface.

Order.java

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

Step 2

Create a request class.

Stock.java

```
public class Stock {

   private String name = "ABC";
   private int quantity = 10;

   public void buy(){
      System.out.println("Stock [ Name: "+name+",
         Quantity: " + quantity +" ] bought");
   }
   public void sell(){
      System.out.println("Stock [ Name: "+name+", 
         Quantity: " + quantity +" ] sold");
   }
}
```

Step 3

Create concrete classes implementing the *Order* interface.

BuyStock.java

```
public class BuyStock implements Order {
   private Stock abcStock;

   public BuyStock(Stock abcStock){
      this.abcStock = abcStock;
   }

   public void execute() {
      abcStock.buy();
   }
}
```

SellStock.java

```
public class SellStock implements Order {
   private Stock abcStock;

   public SellStock(Stock abcStock){
      this.abcStock = abcStock;
   }

   public void execute() {
      abcStock.sell();
   }
}
```

Step 4

Create command invoker class.

Broker.java

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

   public class Broker {
   private List<Order> orderList = new ArrayList<Order>();

   public void takeOrder(Order order){
      orderList.add(order);
   }

   public void placeOrders(){

      for (Order order : orderList) {
         order.execute();
      }
      orderList.clear();
   }
}
```

Step 5

Use the Broker class to take and execute commands.

CommandPatternDemo.java

```
public class CommandPatternDemo {
   public static void main(String[] args) {
      Stock abcStock = new Stock();

      BuyStock buyStockOrder = new BuyStock(abcStock);
      SellStock sellStockOrder = new SellStock(abcStock);

      Broker broker = new Broker();
      broker.takeOrder(buyStockOrder);
      broker.takeOrder(sellStockOrder);

      broker.placeOrders();
   }
}
```

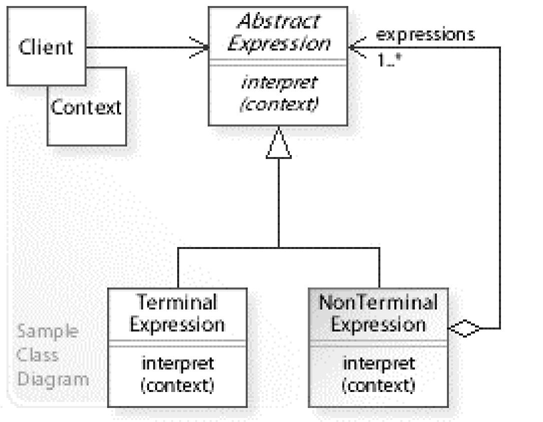
Step 6

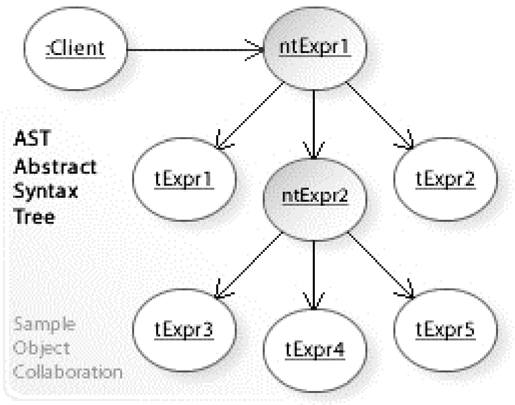
Verify the output.

```
Stock [ Name: ABC, Quantity: 10 ] bought
Stock [ Name: ABC, Quantity: 10 ] sold
```

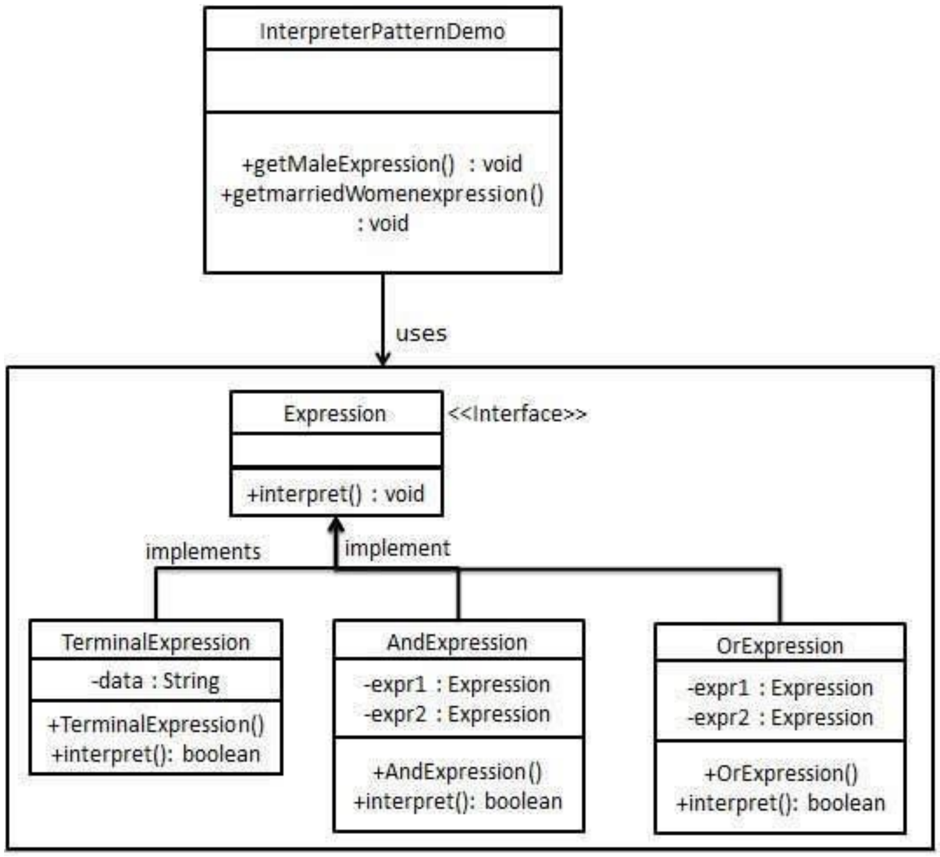
Interpreter design pattern

طريقة لتقييم قواعد اللغة أو التعبير عنها. يأتي هذا النوع من الأنماط تحت نمط سلوكي. يتضمن هذا النمط تنفيذ واجهة تعبير تخبرنا بتفسير سياق معين. يستخدم هذا النمط في معالجة الرموز وما إلى ذلك.





Example:



Step 1

Create an expression interface.

Expression.java

```
public interface Expression {
   public boolean interpret(String context);
}
```

Step 2

Create concrete classes implementing the above interface.

TerminalExpression.java

```
public class TerminalExpression implements Expression {

   private String data;

   public TerminalExpression(String data){
      this.data = data;
   }

   @Override
   public boolean interpret(String context) {

      if(context.contains(data)){
         return true;
      }
      return false;
   }
}
```

OrExpression.java

```java
public class OrExpression implements Expression {

   private Expression expr1 = null;
   private Expression expr2 = null;

   public OrExpression(Expression expr1, Expression expr2) {
      this.expr1 = expr1;
      this.expr2 = expr2;
   }

   @Override
   public boolean interpret(String context) {
      return expr1.interpret(context) || expr2.interpret(context);
   }
}
```

AndExpression.java

```java
public class AndExpression implements Expression {

   private Expression expr1 = null;
   private Expression expr2 = null;

   public AndExpression(Expression expr1, Expression expr2) {
      this.expr1 = expr1;
      this.expr2 = expr2;
   }

   @Override
   public boolean interpret(String context) {
      return expr1.interpret(context) && expr2.interpret(context);
   }
}
```

Step 3

InterpreterPatternDemo uses *Expression* class to create rules and then parse them.

InterpreterPatternDemo.java

```
public class InterpreterPatternDemo {

   //Rule: Robert and John are male
   public static Expression getMaleExpression(){
      Expression robert = new TerminalExpression("Robert");
      Expression john = new TerminalExpression("John");
      return new OrExpression(robert, john);
   }

   //Rule: Julie is a married women
   public static Expression getMarriedWomanExpression(){
      Expression julie = new TerminalExpression("Julie");
      Expression married = new TerminalExpression("Married");
      return new AndExpression(julie, married);
   }

   public static void main(String[] args) {
      Expression isMale = getMaleExpression();
      Expression isMarriedWoman = getMarriedWomanExpression();

      System.out.println("John is male? " +
isMale.interpret("John"));
      System.out.println("Julie is a married women? " +
isMarriedWoman.interpret("Married Julie"));
   }
}
```

Step 4

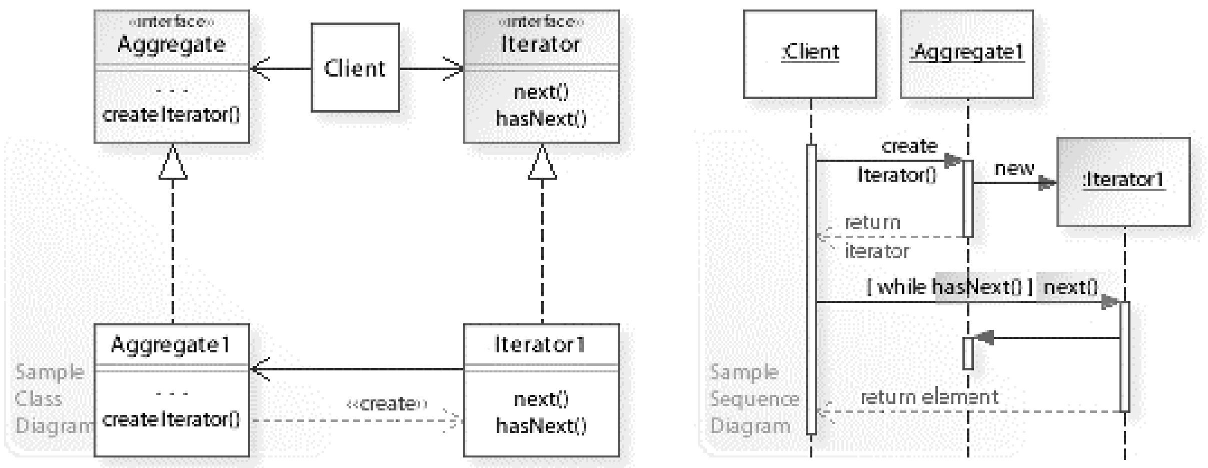
Verify the output.

```
John is male? true
Julie is a married women? true
```

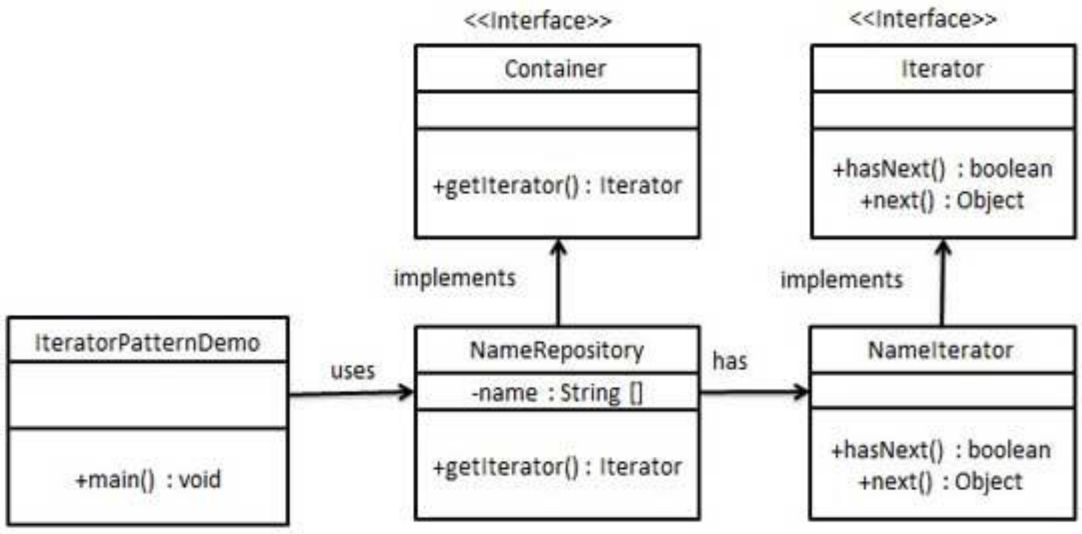
Iterator desin pattern

- Behavioral pattern

نمط للحصول على طريقة للوصول إلى عناصر كائن مجموعة بطريقة متسلسلة دون الحاجة إلى معرفة تمثيله الأساسي.



Example:



Step 1

Create interfaces.

Iterator.java

```
public interface Iterator {
   public boolean hasNext();
   public Object next();
}
```

Container.java

```
public interface Container {
   public Iterator getIterator();
}
```

Step 2

Create concrete class implementing the *Container* interface. This class has inner class *NameIterator* implementing the *Iterator* interface.

NameRepository.java

```
public class NameRepository implements Container {
   public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};

   @Override
   public Iterator getIterator() {
      return new NameIterator();
   }

   private class NameIterator implements Iterator {

      int index;

      @Override
      public boolean hasNext() {

         if(index < names.length){
            return true;
         }
         return false;
      }

      @Override
      public Object next() {

         if(this.hasNext()){
            return names[index++];
         }
         return null;
      }
   }
}
```

Step 3

Use the *NameRepository* to get iterator and print names.

IteratorPatternDemo.java

```
public class IteratorPatternDemo {

   public static void main(String[] args) {
      NameRepository namesRepository = new NameRepository();

      for(Iterator iter = namesRepository.getIterator();
iter.hasNext();){
         String name = (String)iter.next();
         System.out.println("Name : " + name);
      }
   }
}
```

Step 4

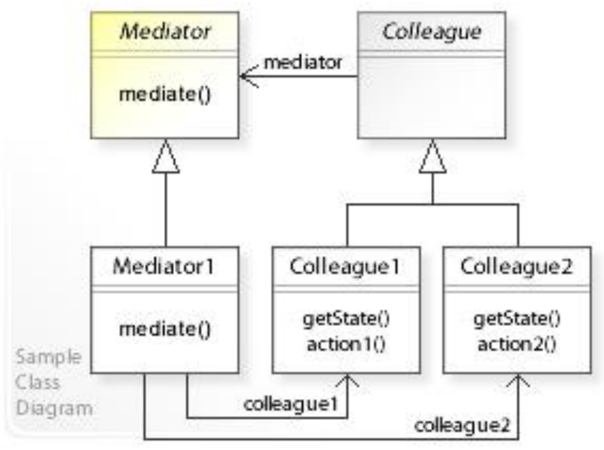
Verify the output.

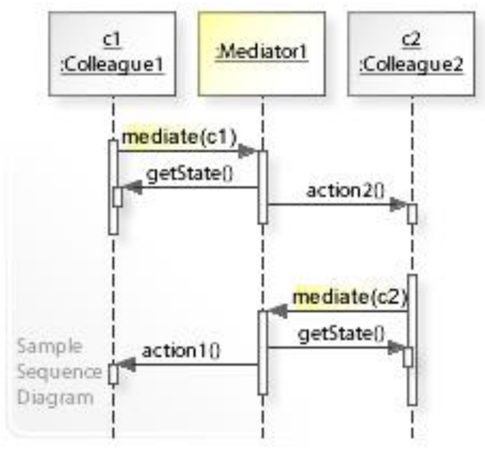
```
Name : Robert
Name : John
Name : Julie
Name : Lora
```

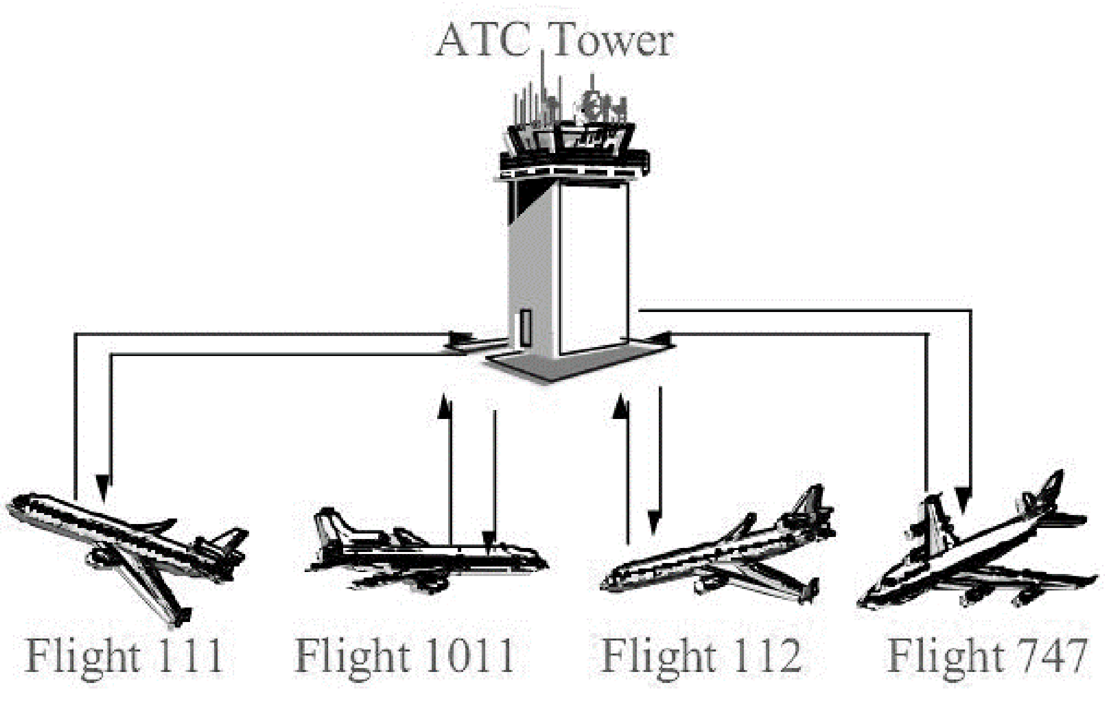
Mediator design pattern

- Behavioral pattern

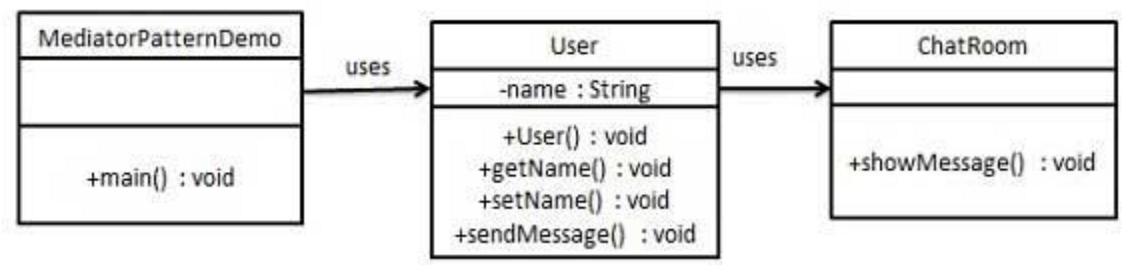
يستخدم نمط الوسيط لتقليل تعقيد الاتصال بين كائنات أو فئات متعددة. يوفر هذا النمط فئة وسيطة تتعامل عادةً مع جميع الاتصالات بين الفئات المختلفة وتدعم الصيانة السهلة للرمز عن طريق الاقتران السائب.







Example:



Step 1

Create mediator class.

ChatRoom.java

```
import java.util.Date;

public class ChatRoom {
   public static void showMessage(User user, String message){
      System.out.println(new Date().toString() + " [" +
user.getName() + "] : " + message);
   }
}
```

Step 2

Create user class

User.java

```
public class User {
   private String name;

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public User(String name){
      this.name  = name;
   }
```

```java
   public void sendMessage(String message){
      ChatRoom.showMessage(this,message);
   }
}
```

Step 3

Use the *User* object to show communications between them.

MediatorPatternDemo.java

```java
public class MediatorPatternDemo {
   public static void main(String[] args) {
      User robert = new User("Robert");
      User john = new User("John");

      robert.sendMessage("Hi! John!");
      john.sendMessage("Hello! Robert!");
   }
}
```

Step 4

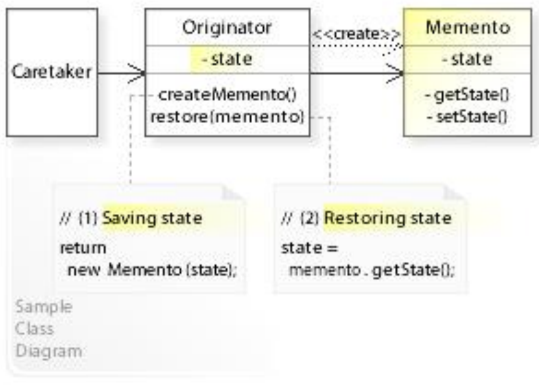
Verify the output.

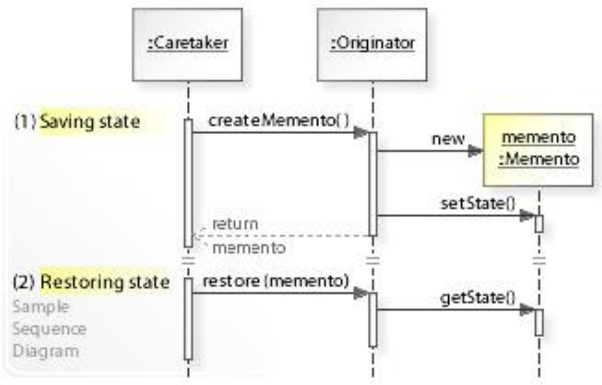
```
Thu Jan 31 16:05:46 IST 2013 [Robert] : Hi! John!
Thu Jan 31 16:05:46 IST 2013 [John] : Hello! Robert!
```

Memento design pattern

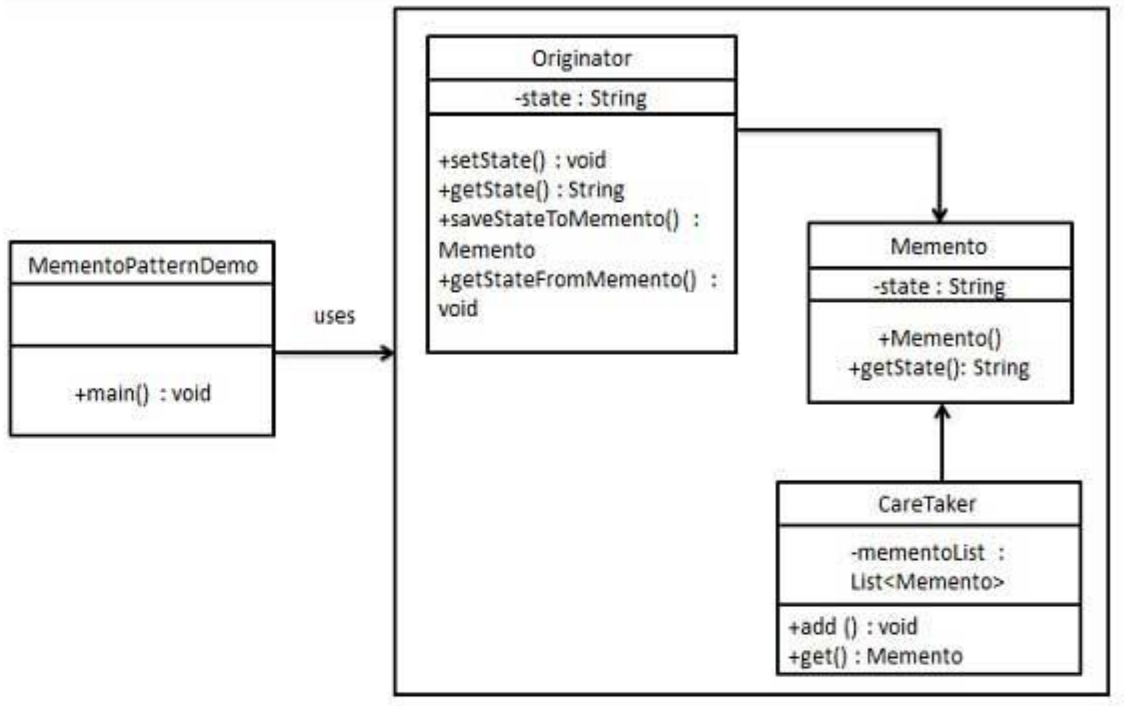
- Behavioral pattern

يستخدم لاستعادة حالة الكائن إلى حالته السابقة.





Example:



Step 1

Create Memento class.

Memento.java

```
public class Memento {
   private String state;

   public Memento(String state){
      this.state = state;
   }

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

Step 2

Create Originator class

Originator.java

```
public class Originator {
   private String state;

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

   public String getState(){
      return state;
   }

   public Memento saveStateToMemento(){
      return new Memento(state);
   }

   public void getStateFromMemento(Memento memento){
      state = memento.getState();
   }
}
```

Step 3

Create CareTaker class

CareTaker.java

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

public class CareTaker {
   private List<Memento> mementoList = new ArrayList<Memento>();

   public void add(Memento state){
      mementoList.add(state);
   }

   public Memento get(int index){
      return mementoList.get(index);
   }
}
```

Step 4

Use *CareTaker* and *Originator* objects.

MementoPatternDemo.java

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

      Originator originator = new Originator();
      CareTaker careTaker = new CareTaker();

      originator.setState("State #1");
      originator.setState("State #2");
      careTaker.add(originator.saveStateToMemento());

      originator.setState("State #3");
      careTaker.add(originator.saveStateToMemento());

      originator.setState("State #4");
      System.out.println("Current State: " + originator.getState());


      originator.getStateFromMemento(careTaker.get(0));
      System.out.println("First saved State: " +
originator.getState());
      originator.getStateFromMemento(careTaker.get(1));
      System.out.println("Second saved State: " +
originator.getState());
```

```
    }
}
```

Step 5

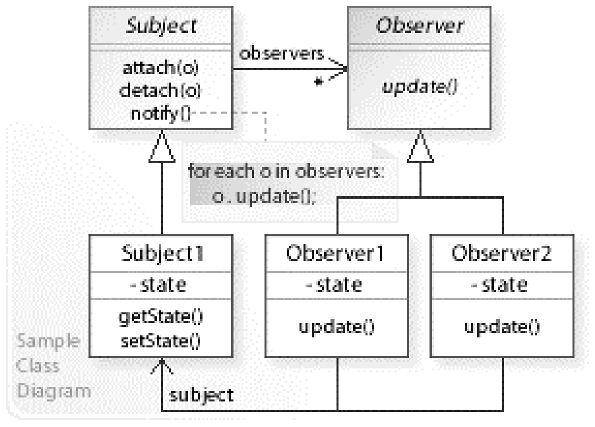
Verify the output.

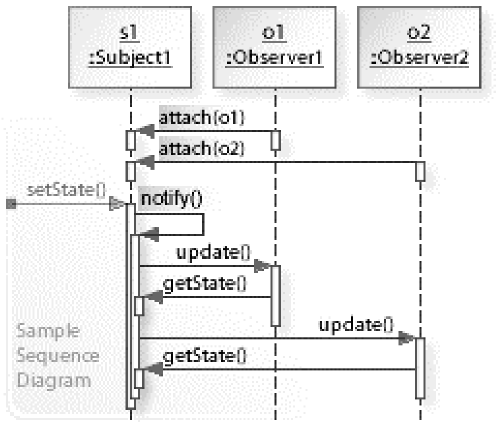
```
Current State: State #4
First saved State: State #2
Second saved State: State #3
```

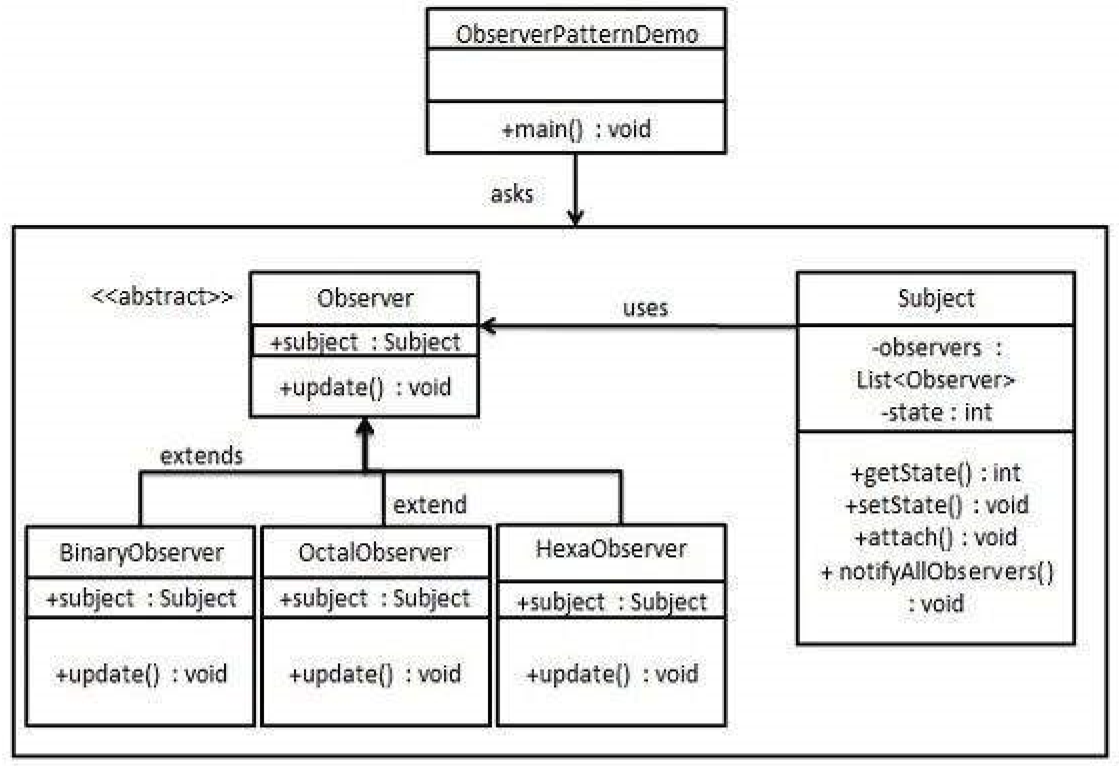
Observer design pattern

- Behavioral pattern

إذا تم تعديل كائن واحد ، فيجب إخطار الكائنات التابعة له تلقائيًا.







Step 1

Create Subject class.

Subject.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();
        }
    }
}
```

Step 2

Create Observer class.

Observer.java

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

Step 3

Create concrete observer classes

BinaryObserver.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.getState() ) );
    }
}
```

OctalObserver.java

```
public class OctalObserver extends Observer{

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

    @Override
    public void update() {
        System.out.println( "Octal String: " + Integer.toOctalString(
subject.getState() ) );
    }
}
```

HexaObserver.java

```
public class HexaObserver extends Observer{

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

    @Override
    public void update() {
        System.out.println( "Hex String: " + Integer.toHexString(
subject.getState() ).toUpperCase() );
    }
}
```

Step 4

Use Subject and concrete observer objects.

ObserverPatternDemo.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);
    }
}
```

Step 5

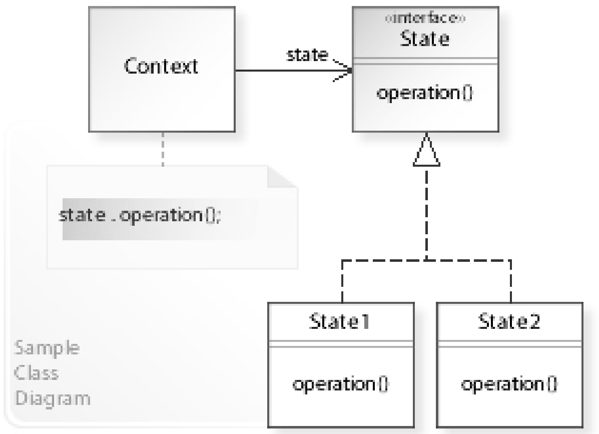
Verify the output.

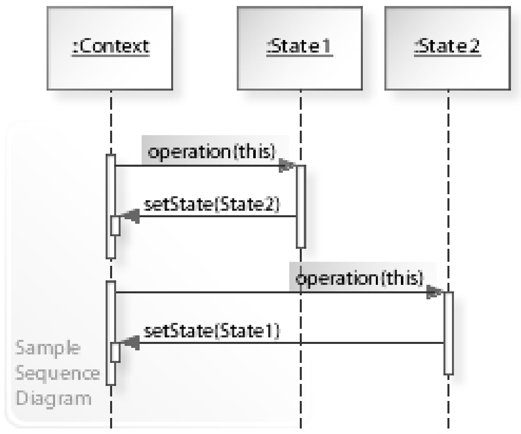
```
First state change: 15
Hex String: F
Octal String: 17
Binary String: 1111
Second state change: 10
Hex String: A
Octal String: 12
Binary String: 1010
```

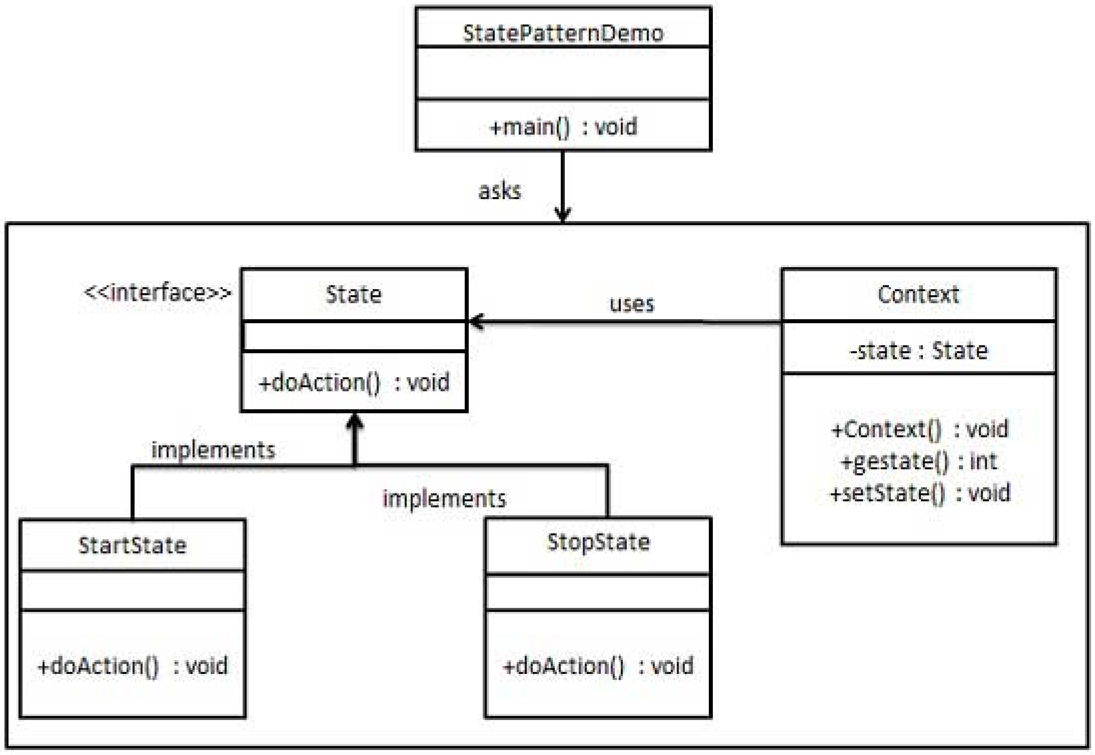
State design pattern

- Behavioral pattern

نمط الحالة هو نمط تصميم برنامج سلوكي يسمح للكائن بتغيير سلوكه عندما تتغير حالته الداخلية. هذا النمط قريب من مفهوم آلات الحالة المحدودة. يمكن تفسير نمط الحالة كنمط إستراتيجي ، وهو قادر على تبديل الإستراتيجية من خلال استدعاءات الأساليب المحددة في واجهة النمط.







Step 1

Create an interface.

State.java

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

Step 2

Create concrete classes implementing the same interface.

StartState.java

```
public class StartState implements State {

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

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

StopState.java

```
public class StopState implements State {

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

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

Step 3

Create *Context* Class.

Context.java

```
public class Context {
   private State state;

   public Context(){
      state = null;
   }

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

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

Step 4

Use the *Context* to see change in behaviour when *State* changes.

StatePatternDemo.java

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

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

      System.out.println(context.getState().toString());
      StopState stopState = new StopState();
      stopState.doAction(context);

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

Step 5

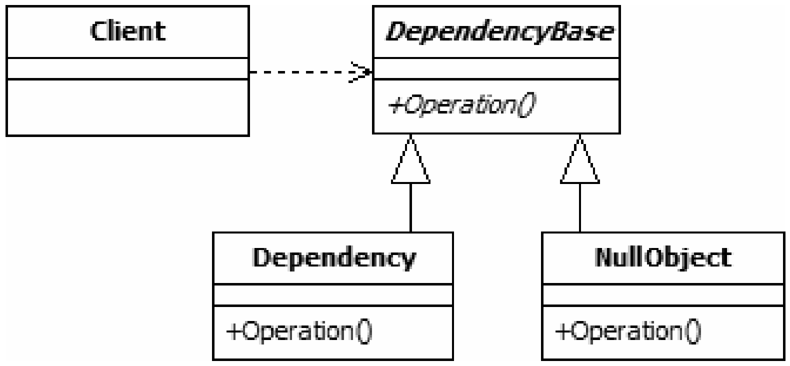
Verify the output.

```
Player is in start state
Start State
Player is in stop state
Stop State
```

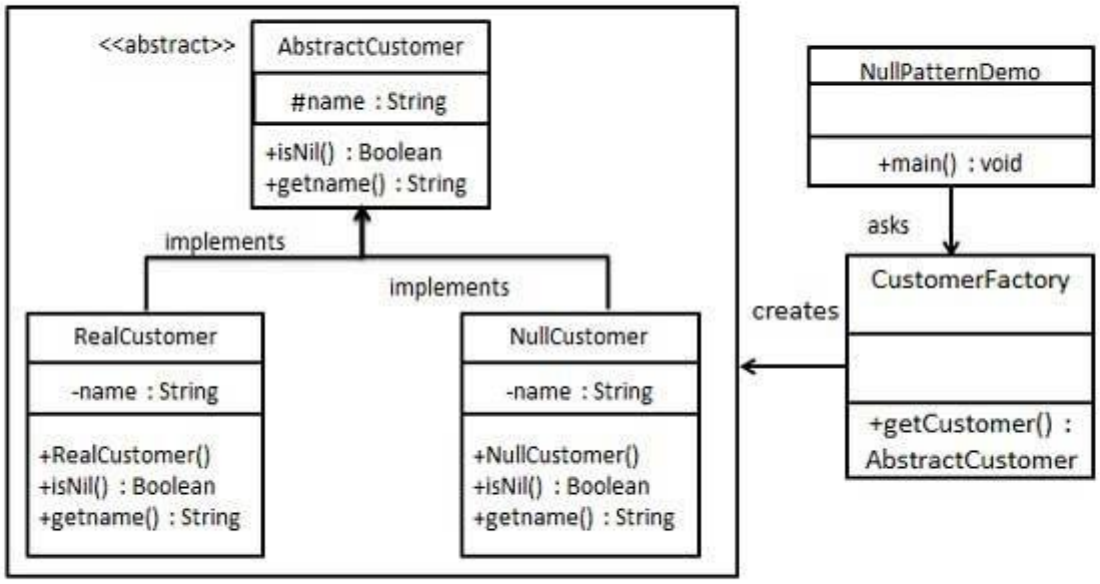
Null object design pattern

- Behavioral pattern

الكائن الفارغ هو كائن بدون قيمة مرجعية أو بسلوك محايد محدد. يصف نمط تصميم الكائن الفارغ استخدامات هذه الكائنات وسلوكها.



Example:



Step 1

Create an abstract class.

AbstractCustomer.java

```
public abstract class AbstractCustomer {
   protected String name;
   public abstract boolean isNil();
   public abstract String getName();
}
```

Step 2

Create concrete classes extending the above class.

RealCustomer.java

```
public class RealCustomer extends AbstractCustomer {

   public RealCustomer(String name) {
      this.name = name;
   }

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

   @Override
   public boolean isNil() {
      return false;
   }
}
```

NullCustomer.java

```
public class NullCustomer extends AbstractCustomer {

   @Override
   public String getName() {
      return "Not Available in Customer Database";
   }

   @Override
   public boolean isNil() {
      return true;
   }
}
```

Step 3

Create *CustomerFactory* Class.

CustomerFactory.java

```
public class CustomerFactory {

   public static final String[] names = {"Rob", "Joe", "Julie"};

   public static AbstractCustomer getCustomer(String name){

      for (int i = 0; i < names.length; i++) {
         if (names[i].equalsIgnoreCase(name)){
            return new RealCustomer(name);
         }
      }
      return new NullCustomer();
   }
}
```

Step 4

Use the *CustomerFactory* to get either *RealCustomer* or *NullCustomer* objects based on the name of customer passed to it.

NullPatternDemo.java

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

      AbstractCustomer customer1 =
CustomerFactory.getCustomer("Rob");
      AbstractCustomer customer2 =
CustomerFactory.getCustomer("Bob");
      AbstractCustomer customer3 =
CustomerFactory.getCustomer("Julie");
      AbstractCustomer customer4 =
CustomerFactory.getCustomer("Laura");

      System.out.println("Customers");
      System.out.println(customer1.getName());
      System.out.println(customer2.getName());
      System.out.println(customer3.getName());
      System.out.println(customer4.getName());
   }
}
```

Step 5

Verify the output.

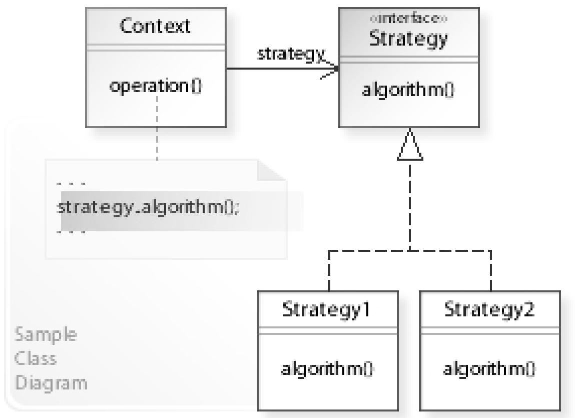
```
Customers
Rob
Not Available in Customer Database
Julie
Not Available in Customer Database
```

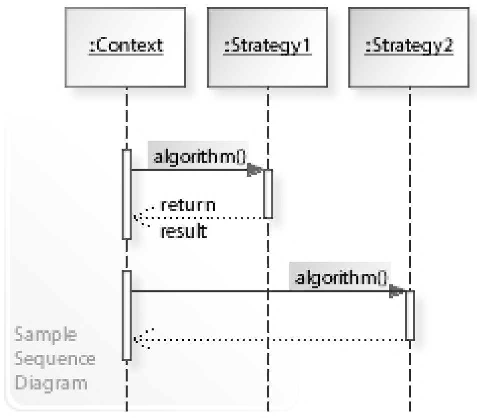
Strategy design pattern

Behavioral pattern

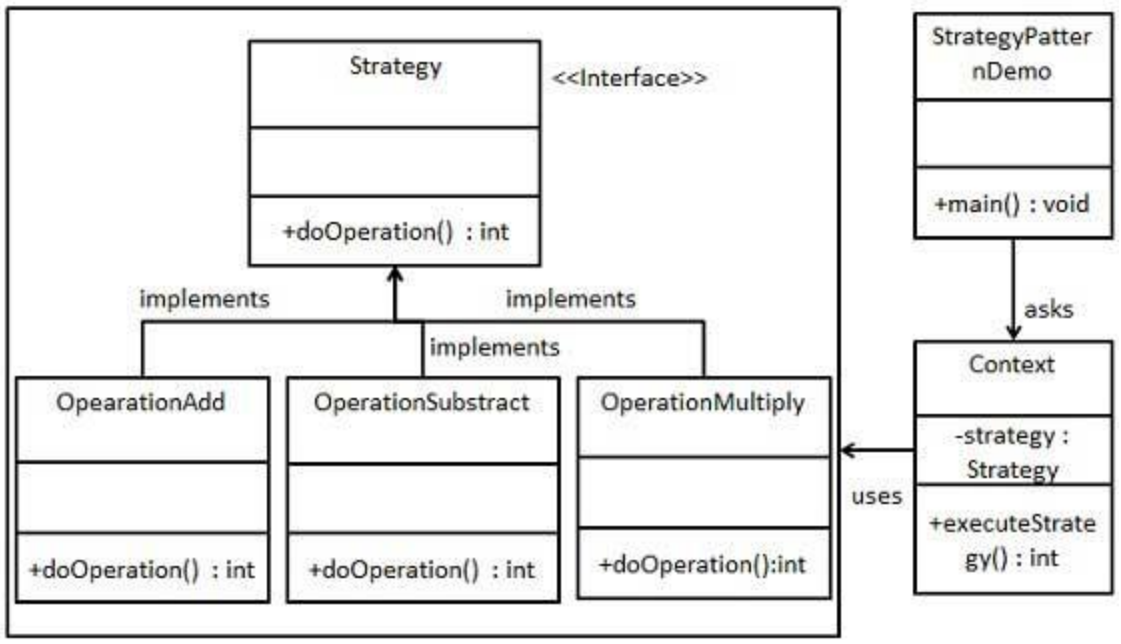
في نمط الإستراتيجية ، نقوم بإنشاء كائنات تمثل استراتيجيات مختلفة وكائن سياق يختلف سلوكه وفقًا لكائن استراتيجيته. يقوم كائن الإستراتيجية بتغيير خوارزمية تنفيذ كائن السياق.

يستعمل هذا النموذج بالتحديد كي يتم اختيار الخوارزمية المناسبة أثناء تشغيل البرنامج.





Example:



Step 1

Create an interface.

Strategy.java

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

Step 2

Create concrete classes implementing the same interface.

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;
   }
}
```

Step 3

Create *Context* Class.

Context.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);
   }
}
```

Step 4

Use the *Context* to see change in behaviour when it changes its *Strategy*.

StrategyPatternDemo.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));
   }
}
```

Step 5

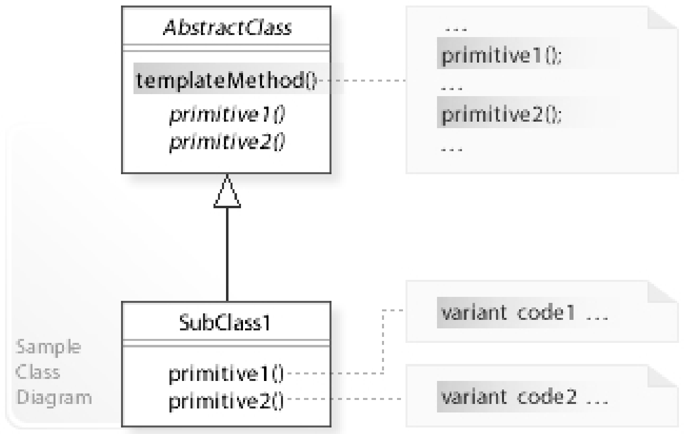
Verify the output.

```
10 + 5 = 15
10 - 5 = 5
10 * 5 = 50
```

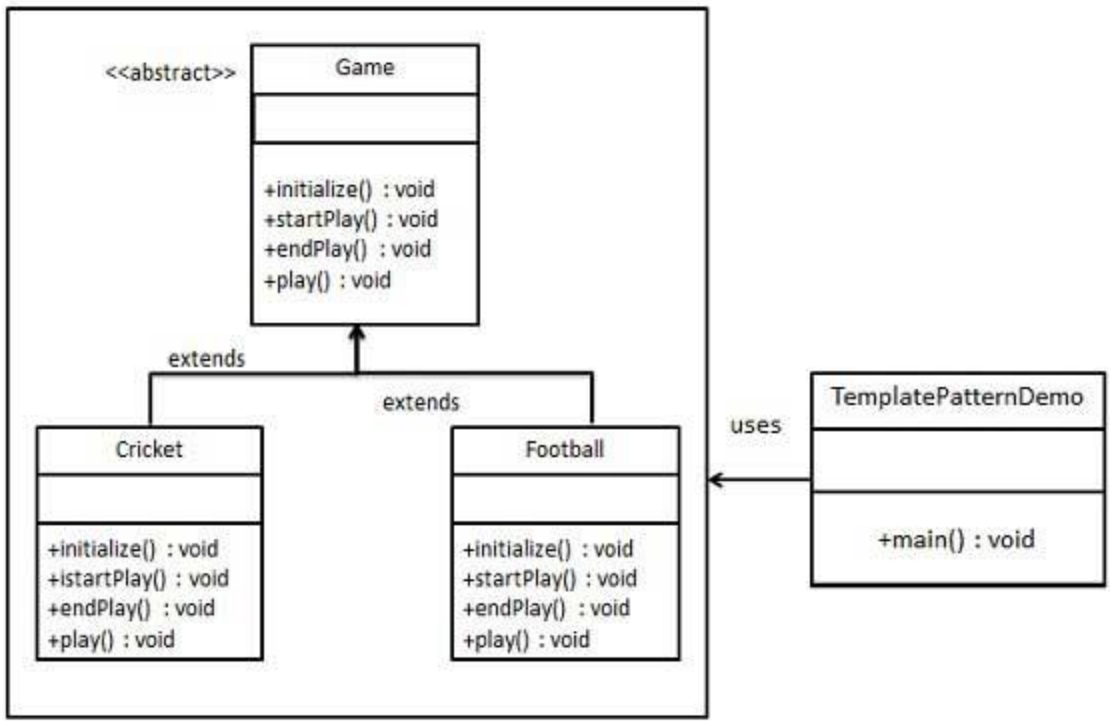
Template design pattern

Behavioral pattern

طريقة القالب هي طريقة في الطبقة الفائقة ، وعادة ما تكون فئة فائقة مجردة ، وتحدد هيكل العملية من حيث عدد من الخطوات عالية المستوى.



Example:



Step 1

Create an abstract class with a template method being final.

Game.java

```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

```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 5

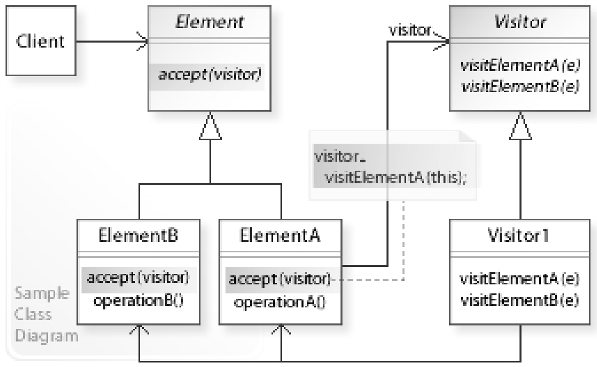
Verify the output.

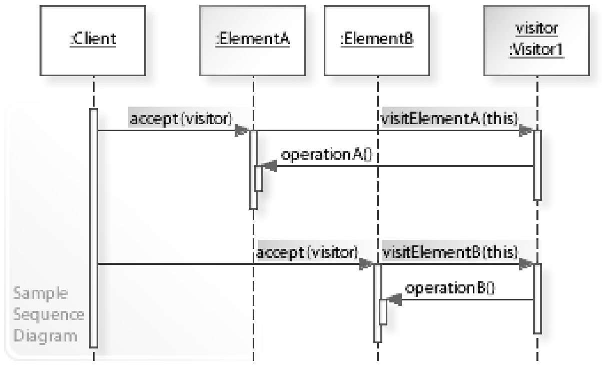
```
Cricket Game Initialized! Start playing.
Cricket Game Started. Enjoy the game!
Cricket Game Finished!

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

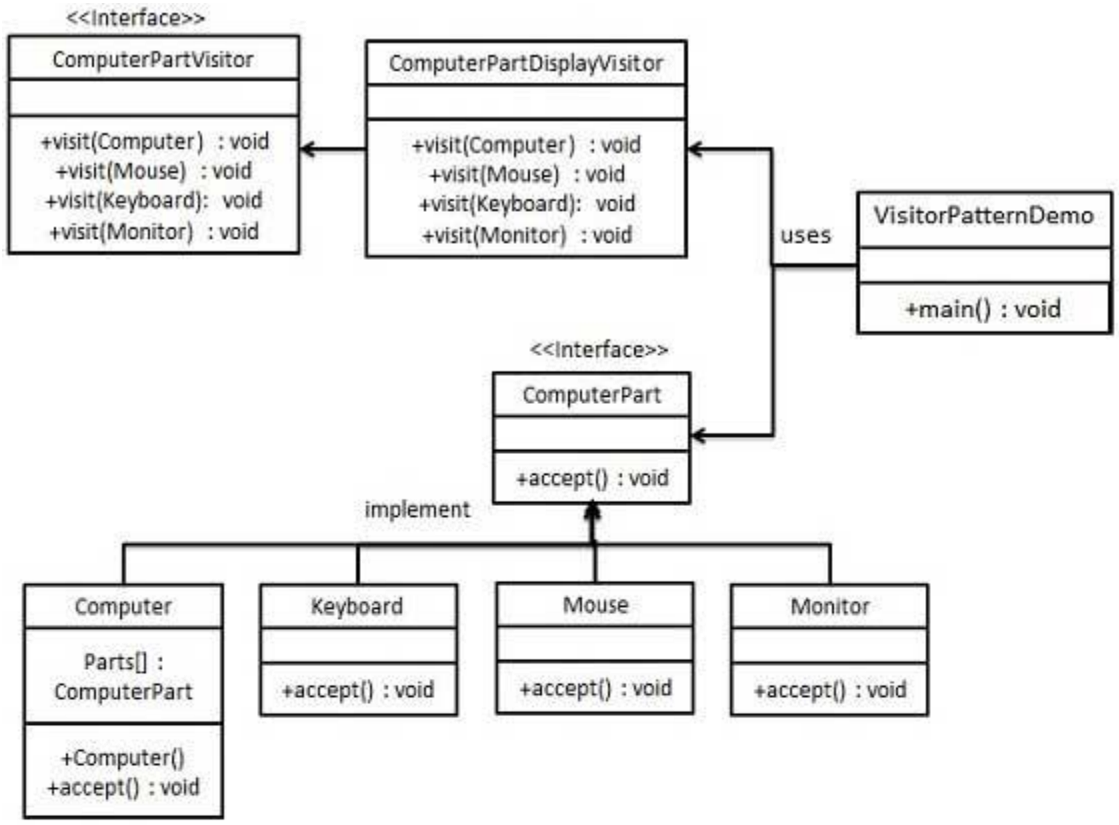
Visitor design pattern

- Behavioral pattern

يعد نمط تصميم الزائر وسيلة لفصل الخوارزمية عن بنية كائن تعمل عليها. النتيجة العملية لهذا الفصل هي القدرة على إضافة عمليات جديدة إلى هياكل الكائنات الموجودة دون تعديل الهياكل. إنها إحدى الطرق لاتباع مبدأ مفتوح / مغلق.







Step 1

Define an interface to represent element.

ComputerPart.java

```
public interface ComputerPart {
   public void accept(ComputerPartVisitor computerPartVisitor);
}
```

Step2

Create concrete classes extending the above class.

Keyboard.java

```
public class Keyboard implements ComputerPart {

   @Override
   public void accept(ComputerPartVisitor computerPartVisitor) {
      computerPartVisitor.visit(this);
   }
}
```

Monitor.java

```
public class Monitor implements ComputerPart {

   @Override
   public void accept(ComputerPartVisitor computerPartVisitor) {
      computerPartVisitor.visit(this);
   }
}
```

Mouse.java

```
public class Mouse implements ComputerPart {

   @Override
   public void accept(ComputerPartVisitor computerPartVisitor) {
      computerPartVisitor.visit(this);
   }
}
```

Computer.java

```
public class Computer implements ComputerPart {

   ComputerPart[] parts;

   public Computer(){
      parts = new ComputerPart[] {new Mouse(), new Keyboard(), new
Monitor()};
   }


   @Override
   public void accept(ComputerPartVisitor computerPartVisitor) {
      for (int i = 0; i < parts.length; i++) {
         parts[i].accept(computerPartVisitor);
      }
      computerPartVisitor.visit(this);
   }
}
```

Step 3

Define an interface to represent visitor.

ComputerPartVisitor.java

```
public interface ComputerPartVisitor {
      public void visit(Computer computer);
      public void visit(Mouse mouse);
      public void visit(Keyboard keyboard);
      public void visit(Monitor monitor);
}
```

Step 4

Create concrete visitor implementing the above class.

ComputerPartDisplayVisitor.java

```
public class ComputerPartDisplayVisitor implements
ComputerPartVisitor {

   @Override
   public void visit(Computer computer) {
      System.out.println("Displaying Computer.");
   }

   @Override
   public void visit(Mouse mouse) {
      System.out.println("Displaying Mouse.");
   }

   @Override
   public void visit(Keyboard keyboard) {
      System.out.println("Displaying Keyboard.");
   }

   @Override
   public void visit(Monitor monitor) {
      System.out.println("Displaying Monitor.");
   }
}
```

Step 5

Use the *ComputerPartDisplayVisitor* to display parts of *Computer*.

VisitorPatternDemo.java

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

      ComputerPart computer = new Computer();
      computer.accept(new ComputerPartDisplayVisitor());
   }
}
```

Step 6

Verify the output.

```
Displaying Mouse.
Displaying Keyboard.
Displaying Monitor.
Displaying Computer.
```

Architectural Design Pattern

An architectural pattern is a general, reusable solution to a commonly occurring problem in software architecture within a given context. Architectural patterns are similar to software design pattern but have a broader scope.

النمط المعماري هو حل عام قابل لإعادة الاستخدام لمشكلة تحدث بشكل شائع في هندسة البرمجيات في سياق معين. تشبه الأنماط المعمارية نمط تصميم البرامج ولكن لها نطاق أوسع.

وهي أنماط تحدد معمارية بناء البرمجيات.

1. Layered pattern
2. Client-server pattern
3. Master-slave pattern
4. Pipe-filter pattern
5. Broker pattern
6. Peer-to-peer pattern
7. Event-bus pattern
8. Model-view-controller pattern
9. Blackboard pattern
10. Interpreter pattern

1. LAYERED PATTERN

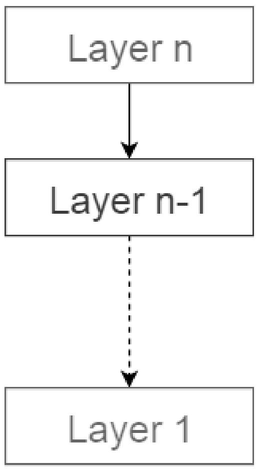
This pattern can be used to structure programs that can be decomposed into groups of subtasks, each of which is at a particular level of abstraction. Each layer provides services to the next higher layer.

The most commonly found 4 layers of a general information system are as follows.

- **Presentation layer** (also known as **UI layer**)
- **Application layer** (also known as **service layer**)
- **Business logic layer** (also known as **domain layer**)
- **Data access layer** (also known as **persistence layer**)

Usage

- General desktop applications.
- E commerce web applications.

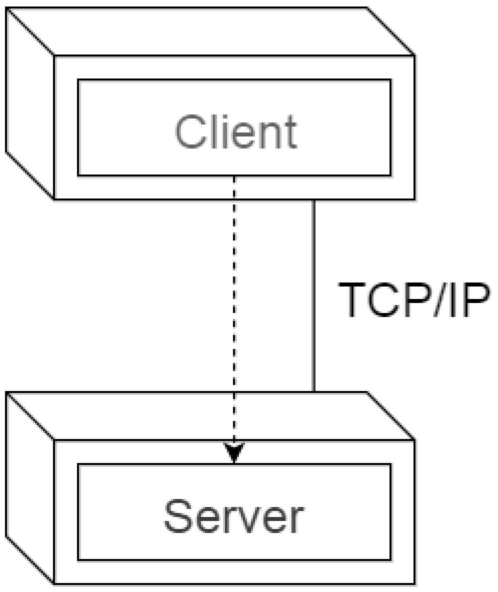


2. CLIENT-SERVER PATTERN

This pattern consists of two parties; a **server** and multiple **clients**. The server component will provide services to multiple client components. Clients request services from the server and the server provides relevant services to those clients. Furthermore, the server continues to listen to client requests.

Usage

- Online applications such as email, document sharing and banking.

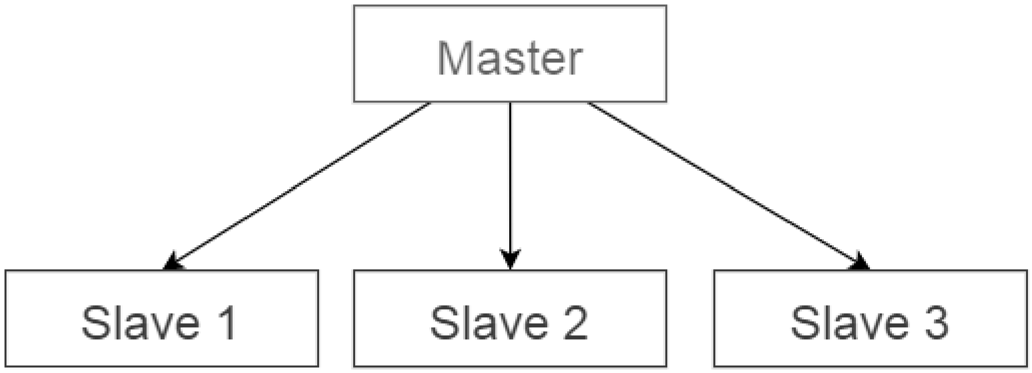


3. MASTER-SLAVE PATTERN

This pattern consists of two parties; **master** and **slaves**. The master component distributes the work among identical slave components, and computes a final result from the results which the slaves return.

Usage

- In database replication, the master database is regarded as the authoritative source, and the slave databases are synchronized to it.
- Peripherals connected to a bus in a computer system (master and slave drives).

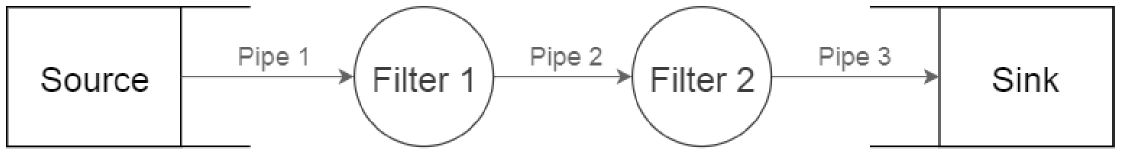


4. PIPE-FILTER PATTERN

This pattern can be used to structure systems which produce and process a stream of data. Each processing step is enclosed within a **filter** component. Data to be processed is passed through **pipes**. These pipes can be used for buffering or for synchronization purposes.

Usage

- Compilers. The consecutive filters perform lexical analysis, parsing, semantic analysis, and code generation.
- Workflows in bioinformatics.



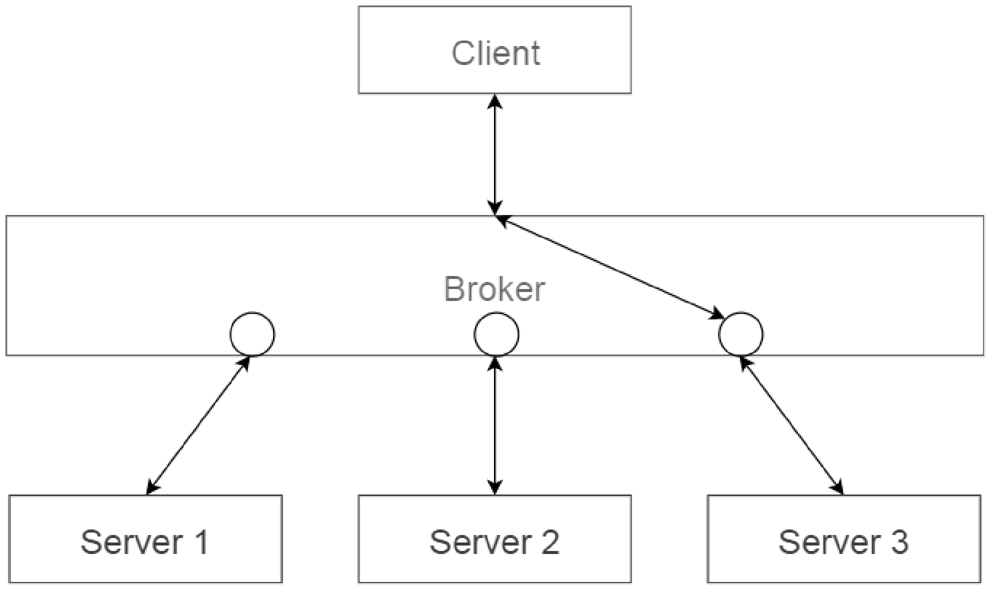
5. BROKER PATTERN

This pattern is used to structure distributed systems with decoupled components. These components can interact with each other by remote service invocations. A **broker** component is responsible for the coordination of communication among **components**.

Servers publish their capabilities (services and characteristics) to a broker. Clients request a service from the broker, and the broker then redirects the client to a suitable service from its registry.

Usage

- Message broker software

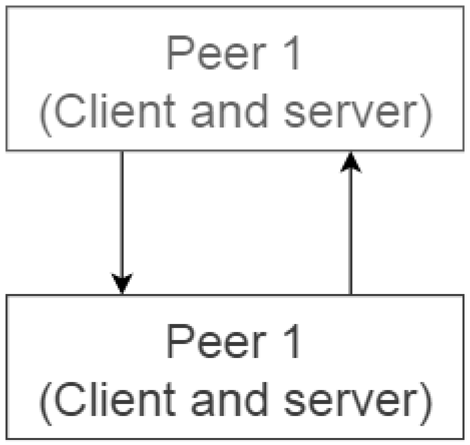


6. PEER-TO-PEER PATTERN

In this pattern, individual components are known as **peers**. Peers may function both as a **client**, requesting services from other peers, and as a **server**, providing services to other peers. A peer may act as a client or as a server or as both, and it can change its role dynamically with time.

Usage

- File-sharing networks such as **Gnutella** and **G2**.
- Multimedia protocols such as **P2PTV** and **PDTP**.
- Cryptocurrency-based products such as **Bitcoin** and **Blockchain**

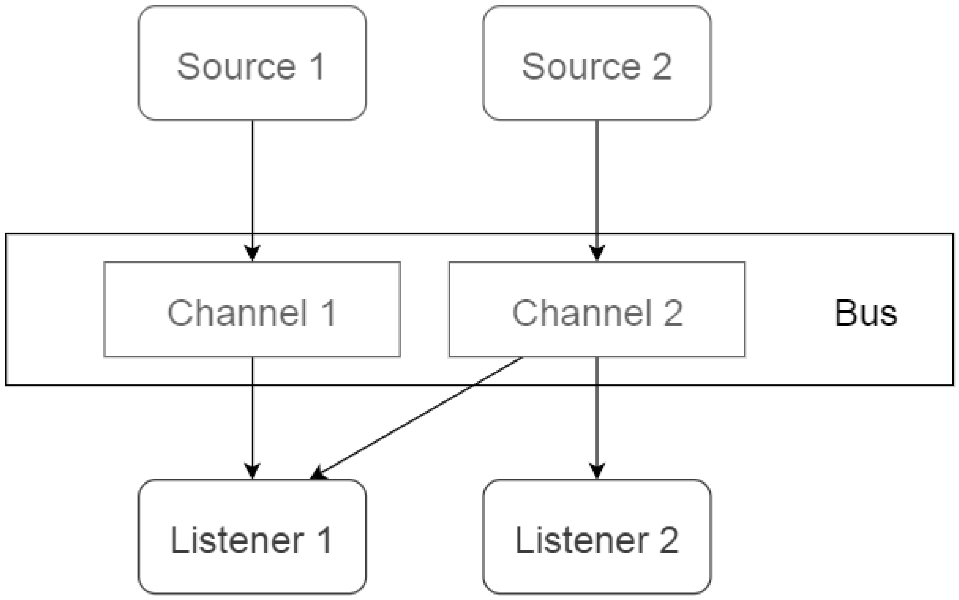


7. EVENT-BUS PATTERN

This pattern primarily deals with events and has 4 major components; **event source**, **event listener**, **channel** and **event bus**. Sources publish messages to particular channels on an event bus. Listeners subscribe to particular channels. Listeners are notified of messages that are published to a channel to which they have subscribed before.

Usage

- Android development
- Notification services



8. MODEL-VIEW-CONTROLLER PATTERN

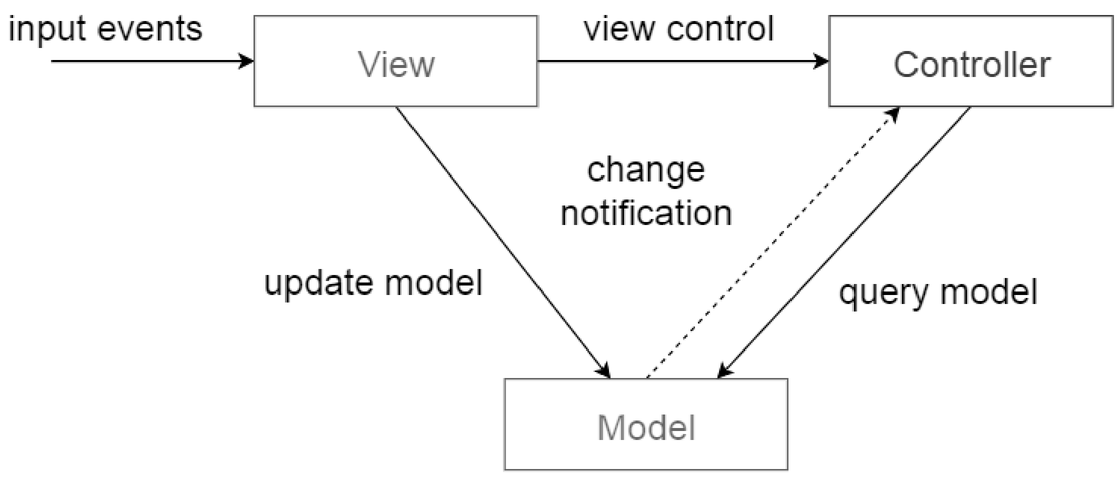
This pattern, also known as MVC pattern, divides an interactive application in to 3 parts as,

1. **model** — contains the core functionality and data
2. **view** — displays the information to the user (more than one view may be defined)
3. **controller** — handles the input from the user

This is done to separate internal representations of information from the ways information is presented to, and accepted from, the user. It decouples components and allows efficient code reuse.

Usage

- Architecture for World Wide Web applications in major programming languages.
- Web frameworks such as **Django** and **Rails**.


input events
View
view control
Controller
change notification
update model
query model
Model

9. BLACKBOARD PATTERN

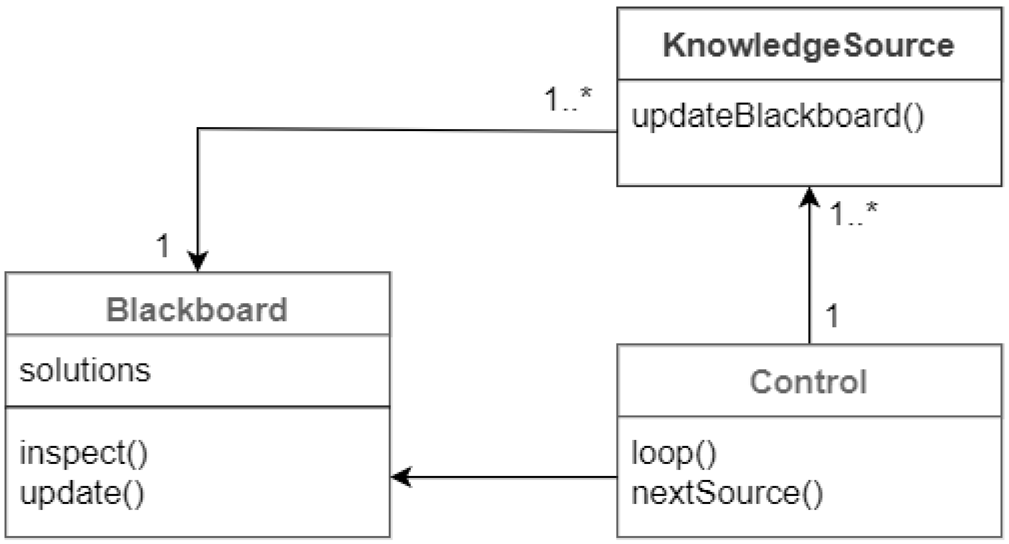
This pattern is useful for problems for which no deterministic solution strategies are known. The blackboard pattern consists of 3 main components.

- **blackboard** — a structured global memory containing objects from the solution space
- **knowledge source** — specialized modules with their own representation
- **control component** — selects, configures and executes modules.

All the components have access to the blackboard. Components may produce new data objects that are added to the blackboard. Components look for particular kinds of data on the blackboard, and may find these by pattern matching with the existing knowledge source.

Usage

- Speech recognition
- Vehicle identification and tracking
- Protein structure identification
- Sonar signals interpretation.

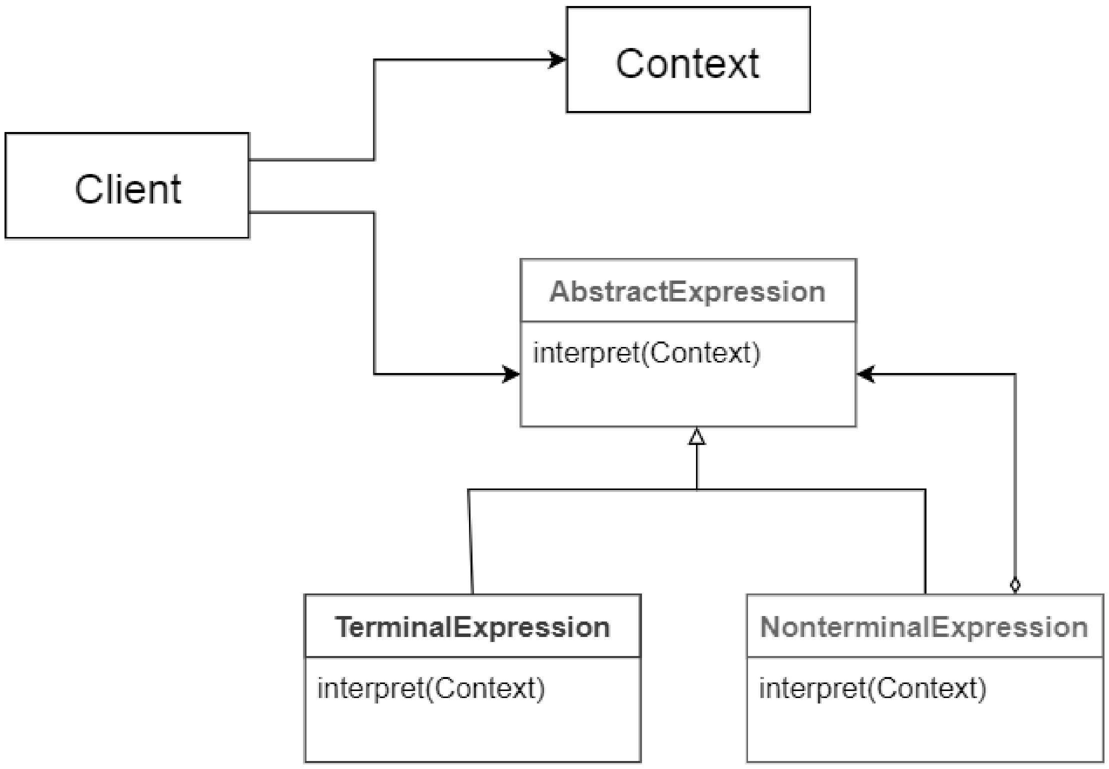

KnowledgeSource
updateBlackboard()
1..*
1
Blackboard
solutions
inspect()
update()
1..*
1
Control
loop()
nextSource()

10. INTERPRETER PATTERN

This pattern is used for designing a component that interprets programs written in a dedicated language. It mainly specifies how to evaluate lines of programs, known as sentences or expressions written in a particular language. The basic idea is to have a class for each symbol of the language.

Usage

- Database query languages such as SQL.
- Languages used to describe communication protocols.


Context
Client
AbstractExpression
interpret(Context)
TerminalExpression
interpret(Context)
NonterminalExpression
interpret(Context)