### JAVA第五阶段—DAY08-JAVA案例

## 工厂模式

### 步骤 1

创建一个接口。

*Shape.java*

public interface Shape {  
  void draw();  
}

### 步骤 2

创建实现接口的实体类。

*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.");  
  }  
}

### 步骤 3

创建一个工厂，生成基于给定信息的实体类的对象。

*ShapeFactory.java*

public class ShapeFactory {  
  
  //使用 getShape 方法获取形状类型的对象  
  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;  
  }  
}

### 步骤 4

使用该工厂，通过传递类型信息来获取实体类的对象。

*FactoryPatternDemo.java*

public class FactoryPatternDemo {  
​  
  public static void main(String[] args) {  
    ShapeFactory shapeFactory = new ShapeFactory();  
​  
    //获取 Circle 的对象，并调用它的 draw 方法  
    Shape shape1 = shapeFactory.getShape("CIRCLE");  
​  
    //调用 Circle 的 draw 方法  
    shape1.draw();  
​  
    //获取 Rectangle 的对象，并调用它的 draw 方法  
    Shape shape2 = shapeFactory.getShape("RECTANGLE");  
​  
    //调用 Rectangle 的 draw 方法  
    shape2.draw();  
​  
    //获取 Square 的对象，并调用它的 draw 方法  
    Shape shape3 = shapeFactory.getShape("SQUARE");  
​  
    //调用 Square 的 draw 方法  
    shape3.draw();  
  }  
}

### 步骤 5

验证输出。

Inside Circle::draw() method.  
Inside Rectangle::draw() method.  
Inside Square::draw() method.

## 抽象工厂模式

### 步骤 1

为形状创建一个接口。

*Shape.java*

public interface Shape {  
  void draw();  
}

### 步骤 2

创建实现接口的实体类。

*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.");  
  }  
}

### 步骤 3

为颜色创建一个接口。

*Color.java*

public interface Color {  
  void fill();  
}

### 步骤4

创建实现接口的实体类。

*Red.java*

public class Red implements Color {  
​  
  @Override  
  public void fill() {  
    System.out.println("Inside Red::fill() method.");  
  }  
}

*Green.java*

public class Green implements Color {  
​  
  @Override  
  public void fill() {  
    System.out.println("Inside Green::fill() method.");  
  }  
}

*Blue.java*

public class Blue implements Color {  
​  
  @Override  
  public void fill() {  
    System.out.println("Inside Blue::fill() method.");  
  }  
}

### 步骤 5

为 Color 和 Shape 对象创建抽象类来获取工厂。

*AbstractFactory.java*

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

### 步骤 6

创建扩展了 AbstractFactory 的工厂类，基于给定的信息生成实体类的对象。

*ShapeFactory.java*

public class ShapeFactory extends AbstractFactory {  
     
  @Override  
  public Shape getShape(String shapeType){  
    if(shapeType == null){  
        return null;  
    }         
    if(shapeType.equalsIgnoreCase("CIRCLE")){  
        return new Circle();  
    } else if(shapeType.equalsIgnoreCase("RECTANGLE")){  
        return new Rectangle();  
    } else if(shapeType.equalsIgnoreCase("SQUARE")){  
        return new Square();  
    }  
    return null;  
  }  
    
  @Override  
  Color getColor(String color) {  
    return null;  
  }  
}

*ColorFactory.java*

public class ColorFactory extends AbstractFactory {  
  
  @Override  
  public Shape getShape(String shapeType){  
    return null;  
  }  
    
  @Override  
  Color getColor(String color) {  
    if(color == null){  
        return null;  
    }         
    if(color.equalsIgnoreCase("RED")){  
        return new Red();  
    } else if(color.equalsIgnoreCase("GREEN")){  
        return new Green();  
    } else if(color.equalsIgnoreCase("BLUE")){  
        return new Blue();  
    }  
    return null;  
  }  
}

### 步骤 7

创建一个工厂创造器/生成器类，通过传递形状或颜色信息来获取工厂。

*FactoryProducer.java*

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

### 步骤 8

使用 FactoryProducer 来获取 AbstractFactory，通过传递类型信息来获取实体类的对象。

*AbstractFactoryPatternDemo.java*

public class AbstractFactoryPatternDemo {  
  public static void main(String[] args) {  
​  
    //获取形状工厂  
    AbstractFactory shapeFactory = FactoryProducer.getFactory("SHAPE");  
​  
    //获取形状为 Circle 的对象  
    Shape shape1 = shapeFactory.getShape("CIRCLE");  
​  
    //调用 Circle 的 draw 方法  
    shape1.draw();  
​  
    //获取形状为 Rectangle 的对象  
    Shape shape2 = shapeFactory.getShape("RECTANGLE");  
​  
    //调用 Rectangle 的 draw 方法  
    shape2.draw();  
       
    //获取形状为 Square 的对象  
    Shape shape3 = shapeFactory.getShape("SQUARE");  
​  
    //调用 Square 的 draw 方法  
    shape3.draw();  
​  
    //获取颜色工厂  
    AbstractFactory colorFactory = FactoryProducer.getFactory("COLOR");  
​  
    //获取颜色为 Red 的对象  
    Color color1 = colorFactory.getColor("RED");  
​  
    //调用 Red 的 fill 方法  
    color1.fill();  
​  
    //获取颜色为 Green 的对象  
    Color color2 = colorFactory.getColor("Green");  
​  
    //调用 Green 的 fill 方法  
    color2.fill();  
​  
    //获取颜色为 Blue 的对象  
    Color color3 = colorFactory.getColor("BLUE");  
​  
    //调用 Blue 的 fill 方法  
    color3.fill();  
  }  
}

### 步骤 9

验证输出。

Inside Circle::draw() method.  
Inside Rectangle::draw() method.  
Inside Square::draw() method.  
Inside Red::fill() method.  
Inside Green::fill() method.  
Inside Blue::fill() method.

## 单例模式

### 步骤 1

创建一个 Singleton 类。

*SingleObject.java*

public class SingleObject {  
​  
  //创建 SingleObject 的一个对象  
  private static SingleObject instance = new SingleObject();  
​  
  //让构造函数为 private，这样该类就不会被实例化  
  private SingleObject(){}  
​  
  //获取唯一可用的对象  
  public static SingleObject getInstance(){  
    return instance;  
  }  
​  
  public void showMessage(){  
    System.out.println("Hello World!");  
  }  
}

### 步骤 2

从 singleton 类获取唯一的对象。

*SingletonPatternDemo.java*

public class SingletonPatternDemo {  
  public static void main(String[] args) {  
​  
    //不合法的构造函数  
    //编译时错误：构造函数 SingleObject() 是不可见的  
    //SingleObject object = new SingleObject();  
​  
    //获取唯一可用的对象  
    SingleObject object = SingleObject.getInstance();  
​  
    //显示消息  
    object.showMessage();  
  }  
}

### 步骤 3

验证输出。

Hello World!

## 单例模式的几种实现方式

### 1、懒汉式，线程不安全

**代码实例：**

public class Singleton {    
  private static Singleton instance;    
  private Singleton (){}    
   
  public static Singleton getInstance() {    
  if (instance == null) {    
      instance = new Singleton();    
  }    
  return instance;    
  }    
}

### 2、懒汉式，线程安全

**代码实例：**

public class Singleton {    
  private static Singleton instance;    
  private Singleton (){}    
  public static synchronized Singleton getInstance() {    
  if (instance == null) {    
      instance = new Singleton();    
  }    
  return instance;    
  }    
}

### 3、饿汉式

**代码实例：**

public class Singleton {    
  private static Singleton instance = new Singleton();    
  private Singleton (){}    
  public static Singleton getInstance() {    
  return instance;    
  }    
}

### 4、双检锁/双重校验锁（DCL，即 double-checked locking）

**代码实例：**

public class Singleton {    
  private volatile static Singleton singleton;    
  private Singleton (){}    
  public static Singleton getSingleton() {    
  if (singleton == null) {    
      synchronized (Singleton.class) {    
      if (singleton == null) {    
          singleton = new Singleton();    
      }    
      }    
  }    
  return singleton;    
  }    
}

### 5、登记式/静态内部类

**代码实例：**

public class Singleton {    
  private static class SingletonHolder {    
  private static final Singleton INSTANCE = new Singleton();    
  }    
  private Singleton (){}    
  public static final Singleton getInstance() {    
  return SingletonHolder.INSTANCE;    
  }    
}

### 6、枚举

**代码实例：**

public enum Singleton {    
  INSTANCE;    
  public void whateverMethod() {    
  }    
}

## 建造者模式

### 步骤 1

创建一个表示食物条目和食物包装的接口。

*Item.java*

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

*Packing.java*

public interface Packing {  
  public String pack();  
}

### 步骤 2

创建实现 Packing 接口的实体类。

*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";  
  }  
}

### 步骤 3

创建实现 Item 接口的抽象类，该类提供了默认的功能。

*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();  
}

### 步骤 4

创建扩展了 Burger 和 ColdDrink 的实体类。

*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";  
  }  
}

### 步骤 5

创建一个 Meal 类，带有上面定义的 Item 对象。

*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());  
    }       
  }    
}

### 步骤 6

创建一个 MealBuilder 类，实际的 builder 类负责创建 Meal 对象。

*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;  
  }  
}

### 步骤 7

BuiderPatternDemo 使用 MealBuider 来演示建造者模式（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());  
  }  
}

### 步骤 8

验证输出。

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

## 原型模式

### 步骤 1

创建一个实现了 *Clonable* 接口的抽象类。

*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;  
  }  
}

### 步骤 2

创建扩展了上面抽象类的实体类。

*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*

public class Circle extends Shape {  
​  
  public Circle(){  
    type = "Circle";  
  }  
​  
  @Override  
  public void draw() {  
    System.out.println("Inside Circle::draw() method.");  
  }  
}

### 步骤 3

创建一个类，从数据库获取实体类，并把它们存储在一个 *Hashtable* 中。

*ShapeCache.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();  
  }  
​  
  // 对每种形状都运行数据库查询，并创建该形状  
  // shapeMap.put(shapeKey, shape);  
  // 例如，我们要添加三种形状  
  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);  
  }  
}

### 步骤 4

*PrototypePatternDemo* 使用 *ShapeCache* 类来获取存储在 *Hashtable* 中的形状的克隆。

*PrototypePatternDemo.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());          
  }  
}

### 步骤 5

验证输出。

Shape : Circle  
Shape : Square  
Shape : Rectangle

## 适配器模式

### 步骤 1

为媒体播放器和更高级的媒体播放器创建接口。

*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);  
}

### 步骤 2

创建实现了 *AdvancedMediaPlayer* 接口的实体类。

*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) {  
    //什么也不做  
  }  
}

*Mp4Player.java*

public class Mp4Player implements AdvancedMediaPlayer{  
​  
  @Override  
  public void playVlc(String fileName) {  
    //什么也不做  
  }  
​  
  @Override  
  public void playMp4(String fileName) {  
    System.out.println("Playing mp4 file. Name: "+ fileName);          
  }  
}

### 步骤 3

创建实现了 *MediaPlayer* 接口的适配器类。

*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);  
    }  
  }  
}

### 步骤 4

创建实现了 *MediaPlayer* 接口的实体类。

*AudioPlayer.java*

public class AudioPlayer implements MediaPlayer {  
  MediaAdapter mediaAdapter;   
​  
  @Override  
  public void play(String audioType, String fileName) {          
​  
    //播放 mp3 音乐文件的内置支持  
    if(audioType.equalsIgnoreCase("mp3")){  
        System.out.println("Playing mp3 file. Name: "+ fileName);            
    }   
    //mediaAdapter 提供了播放其他文件格式的支持  
    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");  
    }  
  }     
}

### 步骤 5

使用 AudioPlayer 来播放不同类型的音频格式。

*AdapterPatternDemo.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");  
  }  
}

### 步骤 6

验证输出。

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

## 桥接模式

### 步骤 1

创建桥接实现接口。

*DrawAPI.java*

public interface DrawAPI {  
  public void drawCircle(int radius, int x, int y);  
}

### 步骤 2

创建实现了 *DrawAPI* 接口的实体桥接实现类。

*RedCircle.java*

public class RedCircle implements DrawAPI {  
  @Override  
  public void drawCircle(int radius, int x, int y) {  
    System.out.println("Drawing Circle[ color: red, radius: "  
        + radius +", x: " +x+", "+ y +"]");  
  }  
}

*GreenCircle.java*

public class GreenCircle implements DrawAPI {  
  @Override  
  public void drawCircle(int radius, int x, int y) {  
    System.out.println("Drawing Circle[ color: green, radius: "  
        + radius +", x: " +x+", "+ y +"]");  
  }  
}

### 步骤 3

使用 *DrawAPI* 接口创建抽象类 *Shape*。

*Shape.java*

public abstract class Shape {  
  protected DrawAPI drawAPI;  
  protected Shape(DrawAPI drawAPI){  
    this.drawAPI = drawAPI;  
  }  
  public abstract void draw();     
}

### 步骤 4

创建实现了 *Shape* 接口的实体类。

*Circle.java*

public class Circle extends Shape {  
  private int x, y, radius;  
​  
  public Circle(int x, int y, int radius, DrawAPI drawAPI) {  
    super(drawAPI);  
    this.x = x;    
    this.y = y;    
    this.radius = radius;  
  }  
​  
  public void draw() {  
    drawAPI.drawCircle(radius,x,y);  
  }  
}

### 步骤 5

使用 *Shape* 和 *DrawAPI* 类画出不同颜色的圆。

*BridgePatternDemo.java*

public class BridgePatternDemo {  
  public static void main(String[] args) {  
    Shape redCircle = new Circle(100,100, 10, new RedCircle());  
    Shape greenCircle = new Circle(100,100, 10, new GreenCircle());  
​  
    redCircle.draw();  
    greenCircle.draw();  
  }  
}

### 步骤 6

验证输出。

Drawing Circle[ color: red, radius: 10, x: 100, 100]  
Drawing Circle[ color: green, radius: 10, x: 100, 100]

## 过滤器模式

### 步骤 1

创建一个类，在该类上应用标准。

*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;  
  }    
}

### 步骤 2

为标准（Criteria）创建一个接口。

*Criteria.java*

import java.util.List;  
​  
public interface Criteria {  
  public List<Person> meetCriteria(List<Person> persons);  
}

### 步骤 3

创建实现了 *Criteria* 接口的实体类。

*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*

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;  
  }  
}

### 步骤4

使用不同的标准（Criteria）和它们的结合来过滤 *Person* 对象的列表。

*CriteriaPatternDemo.java*

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()  
          +" ]");  
    }  
  }        
}

### 步骤 5

验证输出。

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 ]

## 组合模式

### 步骤 1

创建 *Employee* 类，该类带有 *Employee* 对象的列表。

*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;  
​  
  //构造函数  
  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+" ]");  
  }     
}

### 步骤 2

使用 *Employee* 类来创建和打印员工的层次结构。

*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);  
​  
    //打印该组织的所有员工  
    System.out.println(CEO);   
    for (Employee headEmployee : CEO.getSubordinates()) {  
        System.out.println(headEmployee);  
        for (Employee employee : headEmployee.getSubordinates()) {  
          System.out.println(employee);  
        }  
    }          
  }  
}

### 步骤 3

验证输出。

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 ]

## 装饰器模式

### 步骤 1

创建一个接口。

*Shape.java*

public interface Shape {  
  void draw();  
}

### 步骤 2

创建实现接口的实体类。

*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");  
  }  
}

### 步骤 3

创建实现了 *Shape* 接口的抽象装饰类。

*ShapeDecorator.java*

public abstract class ShapeDecorator implements Shape {  
  protected Shape decoratedShape;  
​  
  public ShapeDecorator(Shape decoratedShape){  
    this.decoratedShape = decoratedShape;  
  }  
​  
  public void draw(){  
    decoratedShape.draw();  
  }    
}

### 步骤 4

创建扩展了 *ShapeDecorator* 类的实体装饰类。

*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");

}

}

### 步骤 5

使用 *RedShapeDecorator* 来装饰 *Shape* 对象。

*DecoratorPatternDemo.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();  
  }  
}

### 步骤 6

验证输出。

Circle with normal border  
Shape: Circle  
​  
Circle of red border  
Shape: Circle  
Border Color: Red  
​  
Rectangle of red border  
Shape: Rectangle  
Border Color: Red

## 外观模式

### 步骤 1

创建一个接口。

*Shape.java*

public interface Shape {  
  void draw();  
}

### 步骤 2

创建实现接口的实体类。

*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()");  
  }  
}

### 步骤 3

创建一个外观类。

*ShapeMaker.java*

public class ShapeMaker {  
  private Shape circle;  
  private Shape rectangle;  
  private Shape square;  
​  
  public ShapeMaker() {  
    circle = new Circle();  
    rectangle = new Rectangle();  
    square = new Square();  
  }  
​  
  public void drawCircle(){  
    circle.draw();  
  }  
  public void drawRectangle(){  
    rectangle.draw();  
  }  
  public void drawSquare(){  
    square.draw();  
  }  
}

### 步骤 4

使用该外观类画出各种类型的形状。

*FacadePatternDemo.java*

public class FacadePatternDemo {  
  public static void main(String[] args) {  
    ShapeMaker shapeMaker = new ShapeMaker();  
​  
    shapeMaker.drawCircle();  
    shapeMaker.drawRectangle();  
    shapeMaker.drawSquare();        
  }  
}

### 步骤 5

验证输出。

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

## 享元模式

### 步骤 1

创建一个接口。

*Shape.java*

public interface Shape {  
  void draw();  
}

### 步骤 2

创建实现接口的实体类。

*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);  
  }  
}

### 步骤 3

创建一个工厂，生成基于给定信息的实体类的对象。

*ShapeFactory.java*

import java.util.HashMap;  
​  
public class ShapeFactory {  
  private static final HashMap<String, Shape> 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;  
  }  
}

### 步骤 4

使用该工厂，通过传递颜色信息来获取实体类的对象。

*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);   } }

### 步骤 5

验证输出。

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

## 代理模式

### 步骤 1

创建一个接口。

*Image.java*

public interface Image {  
  void display();  
}

### 步骤 2

创建实现接口的实体类。

*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();  
  }  
}

### 步骤 3

当被请求时，使用 *ProxyImage* 来获取 *RealImage* 类的对象。

*ProxyPatternDemo.java*

public class ProxyPatternDemo {  
   
  public static void main(String[] args) {  
    Image image = new ProxyImage("test\_10mb.jpg");  
​  
    //图像将从磁盘加载  
    image.display();   
    System.out.println("");  
    //图像将无法从磁盘加载  
    image.display();       
  }  
}

### 步骤 4

验证输出。

Loading test\_10mb.jpg  
Displaying test\_10mb.jpg  
​  
Displaying test\_10mb.jpg

## 责任链模式

### 步骤 1

创建抽象的记录器类。

*AbstractLogger.java*

public abstract class AbstractLogger {  
  public static int INFO = 1;  
  public static int DEBUG = 2;  
  public static int ERROR = 3;  
​  
  protected int level;  
​  
  //责任链中的下一个元素  
  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);   }

### 步骤 2

创建扩展了该记录器类的实体类。

*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);  
  }  
}

### 步骤 3

创建不同类型的记录器。赋予它们不同的错误级别，并在每个记录器中设置下一个记录器。每个记录器中的下一个记录器代表的是链的一部分。

*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.");  
  }  
}

### 步骤 4

验证输出。

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.

## 命令模式

### 步骤 1

创建一个命令接口。

*Order.java*

public interface Order {

void execute();

}

### 步骤 2

创建一个请求类。

*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");  
  }  
}

### 步骤 3

创建实现了 *Order* 接口的实体类。

*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();  
  }  
}

### 步骤 4

创建命令调用类。

*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();  
  }  
}

### 步骤 5

使用 Broker 类来接受并执行命令。

*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();  
  }  
}

### 步骤 6

验证输出。

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

## 解释器模式

### 步骤 1

创建一个表达式接口。

*Expression.java*

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

### 步骤 2

创建实现了上述接口的实体类。

*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*

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*

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) &amp;&amp; expr2.interpret(context);  
  }  
}

### 步骤 3

*InterpreterPatternDemo* 使用 *Expression* 类来创建规则，并解析它们。

*InterpreterPatternDemo.java*

public class InterpreterPatternDemo {  
​  
  //规则：Robert 和 John 是男性  
  public static Expression getMaleExpression(){  
    Expression robert = new TerminalExpression("Robert");  
    Expression john = new TerminalExpression("John");  
    return new OrExpression(robert, john);       
  }  
​  
  //规则：Julie 是一个已婚的女性  
  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"));  
  }  
}

### 步骤 4

验证输出。

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

## 迭代器模式

### 步骤 1

创建接口。

*Iterator.java*

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

*Container.java*

public interface Container {  
  public Iterator getIterator();  
}

### 步骤 2

创建实现了 *Container* 接口的实体类。该类有实现了 *Iterator* 接口的内部类 *NameIterator*。

*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;       }         } }

### 步骤 3

使用 *NameRepository* 来获取迭代器，并打印名字。

*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);  
    }     
  }  
}

### 步骤 4

验证输出。

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

## 中介者模式

### 步骤 1

创建中介类。

*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);  
  }  
}

### 步骤 2

创建 user 类。

*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;  
  }  
​  
  public void sendMessage(String message){  
    ChatRoom.showMessage(this,message);  
  }  
}

### 步骤 3

使用 *User* 对象来显示他们之间的通信。

*MediatorPatternDemo.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!");  
  }  
}

### 步骤 4

验证输出。

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

## 观察者模式

### 步骤 1

创建 Subject 类。

*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();  
    }  
  }       
}

### 步骤 2

创建 Observer 类。

*Observer.java*

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

### 步骤 3

创建实体观察者类。

*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() );   
  }  
}

### 步骤 4

使用 *Subject* 和实体观察者对象。

*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);  
  }  
}

### 步骤 5

验证输出。

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

## 状态模式

### 步骤 1

创建一个接口。

*State.java*

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

### 步骤 2

创建实现接口的实体类。

*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";  
  }  
}

### 步骤 3

创建 *Context* 类。

*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;  
  }  
}

### 步骤 4

使用 *Context* 来查看当状态 *State* 改变时的行为变化。

*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());  
  }  
}

### 步骤 5

验证输出。

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

## 空对象模式

### 步骤 1

创建一个抽象类。

*AbstractCustomer.java*

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

### 步骤 2

创建扩展了上述类的实体类。

*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;  
  }  
}

### 步骤 3

创建 *CustomerFactory* 类。

*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();   } }

### 步骤 4

使用 *CustomerFactory*，基于客户传递的名字，来获取 *RealCustomer* 或 *NullCustomer* 对象。

*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());  
  }  
}

### 步骤 5

验证输出。

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

## 策略模式

### 步骤 1

创建一个接口。

*Strategy.java*

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

### 步骤 2

创建实现接口的实体类。

*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;  
  }  
}

### 步骤 3

创建 *Context* 类。

*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);  
  }  
}

### 步骤 4

使用 *Context* 来查看当它改变策略 *Strategy* 时的行为变化。

*StatePatternDemo.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));  
  }  
}

### 步骤 5

验证输出。

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

## 模板模式

### 步骤 1

创建一个抽象类，它的模板方法被设置为 final。

*Game.java*

public abstract class Game {  
  abstract void initialize();  
  abstract void startPlay();  
  abstract void endPlay();  
​  
  //模板  
  public final void play(){  
​  
    //初始化游戏  
    initialize();  
​  
    //开始游戏  
    startPlay();  
​  
    //结束游戏  
    endPlay();  
  }  
}

### 步骤 2

创建扩展了上述类的实体类。

*Cricket.java*

public class Cricket extends Game {  
​  
  @Override  
  void endPlay() {  
    System.out.println("Cricket Game Finished!");  
  }  
​  
  @Override  
  void initialize() {  
    System.out.println("Cricket Game Initialized! Start playing.");  
  }  
​  
  @Override  
  void startPlay() {  
    System.out.println("Cricket Game Started. Enjoy the game!");  
  }  
}

*Football.java*

public class Football extends Game {  
​  
  @Override  
  void endPlay() {  
    System.out.println("Football Game Finished!");  
  }  
​  
  @Override  
  void initialize() {  
    System.out.println("Football Game Initialized! Start playing.");  
  }  
​  
  @Override  
  void startPlay() {  
    System.out.println("Football Game Started. Enjoy the game!");  
  }  
}

### 步骤 3

使用 *Game* 的模板方法 play() 来演示游戏的定义方式。

*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();       
  }  
}

### 步骤 4

验证输出。

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!

## 访问者模式

### 步骤 1

定义一个表示元素的接口。

*ComputerPart.java*

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

### 步骤 2

创建扩展了上述类的实体类。

*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);   } }

### 步骤 3

定义一个表示访问者的接口。

*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);  
}

### 步骤 4

创建实现了上述类的实体访问者。

*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.");  
  }  
}

### 步骤 5

使用 *ComputerPartDisplayVisitor* 来显示 *Computer* 的组成部分。

*VisitorPatternDemo.java*

public class VisitorPatternDemo {  
  public static void main(String[] args) {  
​  
    ComputerPart computer = new Computer();  
    computer.accept(new ComputerPartDisplayVisitor());  
  }  
}

### 步骤 6

验证输出。

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

## MVC 模式

### 步骤 1

创建模型。

*Student.java*

public class Student {  
  private String rollNo;  
  private String name;  
  public String getRollNo() {  
    return rollNo;  
  }  
  public void setRollNo(String rollNo) {  
    this.rollNo = rollNo;  
  }  
  public String getName() {  
    return name;  
  }  
  public void setName(String name) {  
    this.name = name;  
  }  
}

### 步骤 2

创建视图。

*StudentView.java*

public class StudentView {  
  public void printStudentDetails(String studentName, String studentRollNo){  
    System.out.println("Student: ");  
    System.out.println("Name: " + studentName);  
    System.out.println("Roll No: " + studentRollNo);  
  }  
}

### 步骤 3

创建控制器。

*StudentController.java*

public class StudentController {  
  private Student model;  
  private StudentView view;  
​  
  public StudentController(Student model, StudentView view){  
    this.model = model;  
    this.view = view;  
  }  
​  
  public void setStudentName(String name){  
    model.setName(name);        
  }  
​  
  public String getStudentName(){  
    return model.getName();       
  }  
​  
  public void setStudentRollNo(String rollNo){  
    model.setRollNo(rollNo);         
  }  
​  
  public String getStudentRollNo(){  
    return model.getRollNo();       
  }  
​  
  public void updateView(){                
    view.printStudentDetails(model.getName(), model.getRollNo());  
  }   
}

### 步骤 4

使用 *StudentController* 方法来演示 MVC 设计模式的用法。

*MVCPatternDemo.java*

public class MVCPatternDemo {  
  public static void main(String[] args) {  
​  
    //从数据可获取学生记录  
    Student model = retriveStudentFromDatabase();  
​  
    //创建一个视图：把学生详细信息输出到控制台  
    StudentView view = new StudentView();  
​  
    StudentController controller = new StudentController(model, view);  
​  
    controller.updateView();  
​  
    //更新模型数据  
    controller.setStudentName("John");  
​  
    controller.updateView();  
  }  
​  
  private static Student retriveStudentFromDatabase(){  
    Student student = new Student();  
    student.setName("Robert");  
    student.setRollNo("10");  
    return student;  
  }  
}

### 步骤 5

验证输出。

Student:   
Name: Robert  
Roll No: 10  
Student:   
Name: John  
Roll No: 10

### 步骤 3

创建业务查询服务。

*BusinessLookUp.java*

public class BusinessLookUp {  
  public BusinessService getBusinessService(String serviceType){  
    if(serviceType.equalsIgnoreCase("EJB")){  
        return new EJBService();  
    }else {  
        return new JMSService();  
    }  
  }  
}

### 步骤 4

创建业务代表。

*BusinessDelegate.java*

public class BusinessDelegate {  
  private BusinessLookUp lookupService = new BusinessLookUp();  
  private BusinessService businessService;  
  private String serviceType;  
​  
  public void setServiceType(String serviceType){  
    this.serviceType = serviceType;  
  }  
​  
  public void doTask(){  
    businessService = lookupService.getBusinessService(serviceType);  
    businessService.doProcessing();        
  }  
}

### 步骤 5

创建客户端。

*Client.java*

public class Client {  
    
  BusinessDelegate businessService;  
​  
  public Client(BusinessDelegate businessService){  
    this.businessService = businessService;  
  }  
​  
  public void doTask(){       
    businessService.doTask();  
  }  
}

### 步骤 6

使用 BusinessDelegate 和 Client 类来演示业务代表模式。

*BusinessDelegatePatternDemo.java*

public class BusinessDelegatePatternDemo {  
    
  public static void main(String[] args) {  
​  
    BusinessDelegate businessDelegate = new BusinessDelegate();  
    businessDelegate.setServiceType("EJB");  
​  
    Client client = new Client(businessDelegate);  
    client.doTask();  
​  
    businessDelegate.setServiceType("JMS");  
    client.doTask();  
  }  
}

### 步骤 7

验证输出。

Processing task by invoking EJB Service  
Processing task by invoking JMS Service

## 业务代表模式

### 步骤 1

创建 BusinessService 接口。

*BusinessService.java*

public interface BusinessService {  
  public void doProcessing();  
}

### 步骤 2

创建实体服务类。

*EJBService.java*

public class EJBService implements BusinessService {  
​  
  @Override  
  public void doProcessing() {  
    System.out.println("Processing task by invoking EJB Service");  
  }  
}

*JMSService.java*

public class JMSService implements BusinessService {  
​  
  @Override  
  public void doProcessing() {  
    System.out.println("Processing task by invoking JMS Service");  
  }  
}

### 步骤 3

创建业务查询服务。

*BusinessLookUp.java*

public class BusinessLookUp {  
  public BusinessService getBusinessService(String serviceType){  
    if(serviceType.equalsIgnoreCase("EJB")){  
        return new EJBService();  
    }else {  
        return new JMSService();  
    }  
  }  
}

### 步骤 4

创建业务代表。

*BusinessDelegate.java*

public class BusinessDelegate {  
  private BusinessLookUp lookupService = new BusinessLookUp();  
  private BusinessService businessService;  
  private String serviceType;  
​  
  public void setServiceType(String serviceType){  
    this.serviceType = serviceType;  
  }  
​  
  public void doTask(){  
    businessService = lookupService.getBusinessService(serviceType);  
    businessService.doProcessing();        
  }  
}

### 步骤 5

创建客户端。

*Client.java*

public class Client {  
    
  BusinessDelegate businessService;  
​  
  public Client(BusinessDelegate businessService){  
    this.businessService = businessService;  
  }  
​  
  public void doTask(){       
    businessService.doTask();  
  }  
}

### 步骤 6

使用 BusinessDelegate 和 Client 类来演示业务代表模式。

*BusinessDelegatePatternDemo.java*

public class BusinessDelegatePatternDemo {  
    
  public static void main(String[] args) {  
​  
    BusinessDelegate businessDelegate = new BusinessDelegate();  
    businessDelegate.setServiceType("EJB");  
​  
    Client client = new Client(businessDelegate);  
    client.doTask();  
​  
    businessDelegate.setServiceType("JMS");  
    client.doTask();  
  }  
}

### 步骤 7

验证输出。

Processing task by invoking EJB Service  
Processing task by invoking JMS Service

## 组合实体模式

### 步骤 1

创建依赖对象。

*DependentObject1.java*

public class DependentObject1 {  
    
  private String data;  
​  
  public void setData(String data){  
    this.data = data;   
  }   
​  
  public String getData(){  
    return data;  
  }  
}

*DependentObject2.java*

public class DependentObject2 {  
    
  private String data;  
​  
  public void setData(String data){  
    this.data = data;   
  }   
​  
  public String getData(){  
    return data;  
  }  
}

### 步骤 2

创建粗粒度对象。

*CoarseGrainedObject.java*

public class CoarseGrainedObject {  
  DependentObject1 do1 = new DependentObject1();  
  DependentObject2 do2 = new DependentObject2();  
​  
  public void setData(String data1, String data2){  
    do1.setData(data1);  
    do2.setData(data2);  
  }  
​  
  public String[] getData(){  
    return new String[] {do1.getData(),do2.getData()};  
  }  
}

### 步骤 3

创建组合实体。

*CompositeEntity.java*

public class CompositeEntity {  
  private CoarseGrainedObject cgo = new CoarseGrainedObject();  
​  
  public void setData(String data1, String data2){  
    cgo.setData(data1, data2);  
  }  
​  
  public String[] getData(){  
    return cgo.getData();  
  }  
}

### 步骤 4

创建使用组合实体的客户端类。

*Client.java*

public class Client {  
  private CompositeEntity compositeEntity = new CompositeEntity();  
​  
  public void printData(){  
    for (int i = 0; i < compositeEntity.getData().length; i++) {         System.out.println("Data: " + compositeEntity.getData()[i]);       }   }     public void setData(String data1, String data2){       compositeEntity.setData(data1, data2);   } }

### 步骤 5

使用 *Client* 来演示组合实体设计模式的用法。

*CompositeEntityPatternDemo.java*

public class CompositeEntityPatternDemo {  
  public static void main(String[] args) {  
      Client client = new Client();  
      client.setData("Test", "Data");  
      client.printData();  
      client.setData("Second Test", "Data1");  
      client.printData();  
  }  
}

### 步骤 6

验证输出。

Data: Test  
Data: Data  
Data: Second Test  
Data: Data1

## 数据访问对象模式

### 步骤 1

创建数值对象。

*Student.java*

public class Student {  
  private String name;  
  private int rollNo;  
​  
  Student(String name, int rollNo){  
    this.name = name;  
    this.rollNo = rollNo;  
  }  
​  
  public String getName() {  
    return name;  
  }  
​  
  public void setName(String name) {  
    this.name = name;  
  }  
​  
  public int getRollNo() {  
    return rollNo;  
  }  
​  
  public void setRollNo(int rollNo) {  
    this.rollNo = rollNo;  
  }  
}

### 步骤 2

创建数据访问对象接口。

*StudentDao.java*

import java.util.List;  
​  
public interface StudentDao {  
  public List<Student> getAllStudents();  
  public Student getStudent(int rollNo);  
  public void updateStudent(Student student);  
  public void deleteStudent(Student student);  
}

### 步骤 3

创建实现了上述接口的实体类。

*StudentDaoImpl.java*

import java.util.ArrayList;  
import java.util.List;  
​  
public class StudentDaoImpl implements StudentDao {  
     
  //列表是当作一个数据库  
  List<Student> students;  
​  
  public StudentDaoImpl(){  
    students = new ArrayList<Student>();  
    Student student1 = new Student("Robert",0);  
    Student student2 = new Student("John",1);  
    students.add(student1);  
    students.add(student2);        
  }  
  @Override  
  public void deleteStudent(Student student) {  
    students.remove(student.getRollNo());  
    System.out.println("Student: Roll No " + student.getRollNo()   
        +", deleted from database");  
  }  
​  
  //从数据库中检索学生名单  
  @Override  
  public List<Student> getAllStudents() {  
    return students;  
  }  
​  
  @Override  
  public Student getStudent(int rollNo) {  
    return students.get(rollNo);  
  }  
​  
  @Override  
  public void updateStudent(Student student) {  
    students.get(student.getRollNo()).setName(student.getName());  
    System.out.println("Student: Roll No " + student.getRollNo()   
        +", updated in the database");  
  }  
}

### 步骤 4

使用 *StudentDao* 来演示数据访问对象模式的用法。

*CompositeEntityPatternDemo.java*

public class DaoPatternDemo {  
  public static void main(String[] args) {  
    StudentDao studentDao = new StudentDaoImpl();  
​  
    //输出所有的学生  
    for (Student student : studentDao.getAllStudents()) {  
        System.out.println("Student: [RollNo : "  
          +student.getRollNo()+", Name : "+student.getName()+" ]");  
    }  
​  
​  
    //更新学生  
    Student student =studentDao.getAllStudents().get(0);  
    student.setName("Michael");  
    studentDao.updateStudent(student);  
​  
    //获取学生  
    studentDao.getStudent(0);  
    System.out.println("Student: [RollNo : "  
        +student.getRollNo()+", Name : "+student.getName()+" ]");       
  }  
}

### 步骤 5

验证输出。

Student: [RollNo : 0, Name : Robert ]  
Student: [RollNo : 1, Name : John ]  
Student: Roll No 0, updated in the database  
Student: [RollNo : 0, Name : Michael ]

## 前端控制器模式

### 步骤 1

创建视图。

*HomeView.java*

public class HomeView {  
  public void show(){  
    System.out.println("Displaying Home Page");  
  }  
}

*StudentView.java*

public class StudentView {  
  public void show(){  
    System.out.println("Displaying Student Page");  
  }  
}

### 步骤 2

创建调度器 Dispatcher。

*Dispatcher.java*

public class Dispatcher {  
  private StudentView studentView;  
  private HomeView homeView;  
  public Dispatcher(){  
    studentView = new StudentView();  
    homeView = new HomeView();  
  }  
​  
  public void dispatch(String request){  
    if(request.equalsIgnoreCase("STUDENT")){  
        studentView.show();  
    }else{  
        homeView.show();  
    }    
  }  
}

### 步骤 3

创建前端控制器 FrontController。

*FrontController.java*

public class FrontController {  
     
  private Dispatcher dispatcher;  
​  
  public FrontController(){  
    dispatcher = new Dispatcher();  
  }  
​  
  private boolean isAuthenticUser(){  
    System.out.println("User is authenticated successfully.");  
    return true;  
  }  
​  
  private void trackRequest(String request){  
    System.out.println("Page requested: " + request);  
  }  
​  
  public void dispatchRequest(String request){  
    //记录每一个请求  
    trackRequest(request);  
    //对用户进行身份验证  
    if(isAuthenticUser()){  
        dispatcher.dispatch(request);  
    }      
  }  
}

### 步骤 4

使用 *FrontController* 来演示前端控制器设计模式。

*FrontControllerPatternDemo.java*

public class FrontControllerPatternDemo {  
  public static void main(String[] args) {  
    FrontController frontController = new FrontController();  
    frontController.dispatchRequest("HOME");  
    frontController.dispatchRequest("STUDENT");  
  }  
}

### 步骤 5

验证输出。

Page requested: HOME  
User is authenticated successfully.  
Displaying Home Page  
Page requested: STUDENT  
User is authenticated successfully.  
Displaying Student Page

## 拦截过滤器模式

### 步骤 1

创建过滤器接口 Filter。

*Filter.java*

public interface Filter {  
  public void execute(String request);  
}

### 步骤 2

创建实体过滤器。

*AuthenticationFilter.java*

public class AuthenticationFilter implements Filter {  
  public void execute(String request){  
    System.out.println("Authenticating request: " + request);  
  }  
}

*DebugFilter.java*

public class DebugFilter implements Filter {  
  public void execute(String request){  
    System.out.println("request log: " + request);  
  }  
}

### 步骤 3

创建 Target。

*Target.java*

public class Target {  
  public void execute(String request){  
    System.out.println("Executing request: " + request);  
  }  
}

### 步骤 4

创建过滤器链。

*FilterChain.java*

import java.util.ArrayList;  
import java.util.List;  
​  
public class FilterChain {  
  private List filters = new ArrayList();  
  private Target target;  
​  
  public void addFilter(Filter filter){  
    filters.add(filter);  
  }  
​  
  public void execute(String request){  
    for (Filter filter : filters) {  
        filter.execute(request);  
    }  
    target.execute(request);  
  }  
​  
  public void setTarget(Target target){  
    this.target = target;  
  }  
}

### 步骤 5

创建过滤管理器。

*FilterManager.java*

public class FilterManager {  
  FilterChain filterChain;  
​  
  public FilterManager(Target target){  
    filterChain = new FilterChain();  
    filterChain.setTarget(target);  
  }  
  public void setFilter(Filter filter){  
    filterChain.addFilter(filter);  
  }  
​  
  public void filterRequest(String request){  
    filterChain.execute(request);  
  }  
}

### 步骤 6

创建客户端 Client。

*Client.java*

public class Client {  
  FilterManager filterManager;  
​  
  public void setFilterManager(FilterManager filterManager){  
    this.filterManager = filterManager;  
  }  
​  
  public void sendRequest(String request){  
    filterManager.filterRequest(request);  
  }  
}

### 步骤 7

使用 *Client* 来演示拦截过滤器设计模式。

*InterceptingFilterDemo.java*

public class InterceptingFilterDemo {  
  public static void main(String[] args) {  
    FilterManager filterManager = new FilterManager(new Target());  
    filterManager.setFilter(new AuthenticationFilter());  
    filterManager.setFilter(new DebugFilter());  
​  
    Client client = new Client();  
    client.setFilterManager(filterManager);  
    client.sendRequest("HOME");  
  }  
}

### 步骤 8

验证输出。

Authenticating request: HOME  
request log: HOME  
Executing request: HOME

## 服务定位器模式

### 步骤 1

创建服务接口 Service。

*Service.java*

public interface Service {  
  public String getName();  
  public void execute();  
}

### 步骤 2

创建实体服务。

*Service1.java*

public class Service1 implements Service {  
  public void execute(){  
    System.out.println("Executing Service1");  
  }  
​  
  @Override  
  public String getName() {  
    return "Service1";  
  }  
}

*Service2.java*

public class Service2 implements Service {  
  public void execute(){  
    System.out.println("Executing Service2");  
  }  
​  
  @Override  
  public String getName() {  
    return "Service2";  
  }  
}

### 步骤 3

为 JNDI 查询创建 InitialContext。

*InitialContext.java*

public class InitialContext {  
  public Object lookup(String jndiName){  
    if(jndiName.equalsIgnoreCase("SERVICE1")){  
        System.out.println("Looking up and creating a new Service1 object");  
        return new Service1();  
    }else if (jndiName.equalsIgnoreCase("SERVICE2")){  
        System.out.println("Looking up and creating a new Service2 object");  
        return new Service2();  
    }  
    return null;       
  }  
}

### 步骤 4

创建缓存 Cache。

*Cache.java*

import java.util.ArrayList;  
import java.util.List;  
​  
public class Cache {  
​  
  private List<Service> services;  
​  
  public Cache(){  
    services = new ArrayList<Service>();  
  }  
​  
  public Service getService(String serviceName){  
    for (Service service : services) {  
        if(service.getName().equalsIgnoreCase(serviceName)){  
          System.out.println("Returning cached "+serviceName+" object");  
          return service;  
        }  
    }  
    return null;  
  }  
​  
  public void addService(Service newService){  
    boolean exists = false;  
    for (Service service : services) {  
        if(service.getName().equalsIgnoreCase(newService.getName())){  
          exists = true;  
        }  
    }  
    if(!exists){  
        services.add(newService);  
    }  
  }  
}

### 步骤 5

创建服务定位器。

*ServiceLocator.java*

public class ServiceLocator {  
  private static Cache cache;  
​  
  static {  
    cache = new Cache();         
  }  
​  
  public static Service getService(String jndiName){  
​  
    Service service = cache.getService(jndiName);  
​  
    if(service != null){  
        return service;  
    }  
​  
    InitialContext context = new InitialContext();  
    Service service1 = (Service)context.lookup(jndiName);  
    cache.addService(service1);  
    return service1;  
  }  
}

### 步骤 6

使用 *ServiceLocator* 来演示服务定位器设计模式。

*ServiceLocatorPatternDemo.java*

public class ServiceLocatorPatternDemo {  
  public static void main(String[] args) {  
    Service service = ServiceLocator.getService("Service1");  
    service.execute();  
    service = ServiceLocator.getService("Service2");  
    service.execute();  
    service = ServiceLocator.getService("Service1");  
    service.execute();  
    service = ServiceLocator.getService("Service2");  
    service.execute();          
  }  
}

### 步骤 7

验证输出。

Looking up and creating a new Service1 object

Executing Service1

Looking up and creating a new Service2 object

Executing Service2

Returning cached Service1 object

Executing Service1

Returning cached Service2 object

Executing Service2