# 设计模式实验 5-8

## 一、实验目的

1.结合实例，熟练绘制设计模式结构图。
2.结合实例，熟练使用 Java 语言实现设计模式。
3.通过本实验，理解每一种设计模式的模式动机，掌握模式结构，学习如何使用代码实现这些设计模式。

## 二、实验要求

1.结合实例，绘制设计模式的结构图。
2.使用 Java 语言实现设计模式实例，代码运行正确。

## 三、实验内容

**1.单例模式：**某 Web 性能测试软件中包含一个虚拟用户生成器（VirtualUserGenerator）。为了避免生成的虚拟用户数量不一致，该测试软件在工作时只允许启动唯一一个虚拟用户生成器。采用单例模式设计该虚拟用户生成器，绘制类图并分别使用饿汉式单例、双重检测锁和 IoDH 三种方式编程模拟实现。

**2.原型模式：**在某在线招聘网站中，用户可以创建一个简历模板。针对不同的工作岗位，可以复制该简历模板并进行适当修改后，生成一份新的简历。在复制简历时，用户可以选择是否复制简历中的照片：如果选择“是”，则照片将一同被复制，用户对新简历中的照片进行修改不会影响到简历模板中的照片，对模板进行修改也不会影响到新简历；如果选择“否”，则直接引用简历模板中的照片，修改简历模板中的照片将导致新简历中的照片一同修改，反之亦然。现采用原型模式设计该简历复制功能并提供浅克隆和深克隆两套实现方案，绘制对应的类图并编程模拟实现。

**3.简单工厂模式：**简单工厂模式使用简单工厂模式设计一个可以创建不同几何形状（Shape）（例如圆形（Circle）、矩形（Rectangle）和三角形（Triangle）等）的绘图工具类，每个几何图形均具有绘制方法 draw()和擦除方法 erase()，要求在绘制不支持的几何图形时，抛出一个 UnsupportedShapeException 异

常。绘制类图并编程模拟实现。

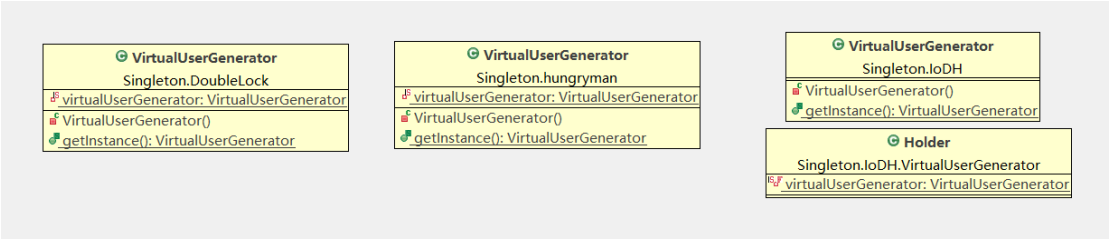
**4.建造者模式：**在某赛车游戏中，赛车包括方程式赛车、场地越野赛车、运动汽车、卡车等类型，不同类型的赛车的车身、发动机、轮胎、变速箱等部件有所区别。玩家可以自行选择赛车类型，系统将根据玩家的选择创建出一辆完整的赛车。现采用建造者模式实现赛车的构建，绘制对应的类图并编程模拟实现。

**5.抽象工厂模式：**某系统为了改进数据库操作的性能，用户可以自定义数据库连接对象 Connection 和语句对象 Statement，针对不同类型的数据库提供不同的连接对象和语句对象，例如提供 Oracle 或 MySQL 专用连接类和语句类，而且用户可以通过配置文件等方式根据实际需要动态更换系统数据库。使用抽象工厂模式设计该系统，绘制对应的类图并编程模拟

## 四、实验结果

需要提供设计模式实例的结构图（类图）和实现代码。

### 1.单例模式




代码实现：

**饿汉式单例**

```
public class VirtualUserGenerator {

    private static VirtualUserGenerator virtualUserGenerator = new
VirtualUserGenerator();

    private VirtualUserGenerator() {
        System.out.println("One VirtualUserGenerator has been
generated");
```

```
    }

    public static VirtualUserGenerator getInstance() {
        return virtualUserGenerator;
    }

}
```

### 双重检测锁

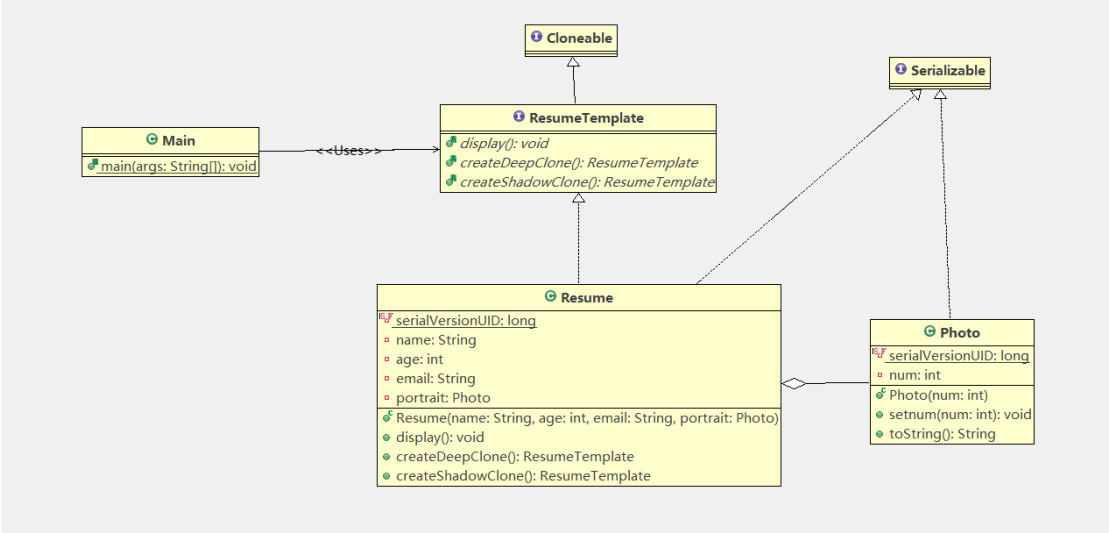
```
public class VirtualUserGenerator {

    // volatile 关键字用于确保多线程下的可见性和禁止指令重排序
    private static volatile VirtualUserGenerator virtualUserGenerator;

    private VirtualUserGenerator() {

    }

    public static VirtualUserGenerator getInstance() {
        // 第一次检查：检查实例是否已经存在
        if (virtualUserGenerator == null) {
            // 进入同步块，确保只有一个线程可以进入
            synchronized (VirtualUserGenerator.class) {
                // 第二次检查：在同步块内再次检查实例是否为null（防止前面的线
程在第一次检查后，进入同步块前停止）
                if (virtualUserGenerator == null) {
                    // 如果仍为null，创建唯一的实例
                    virtualUserGenerator = new VirtualUserGenerator();
                }
            }
        }
        return virtualUserGenerator;
    }

}
```

### IoDH

```
public class VirtualUserGenerator {

    private VirtualUserGenerator() {

    }
```

```
    private static class Holder {
        private static final VirtualUserGenerator virtualUserGenerator =
new VirtualUserGenerator();
    }

    public static VirtualUserGenerator getInstance() {
        return Holder.virtualUserGenerator;
    }

}
```

## 2.原型模式




代码实现：

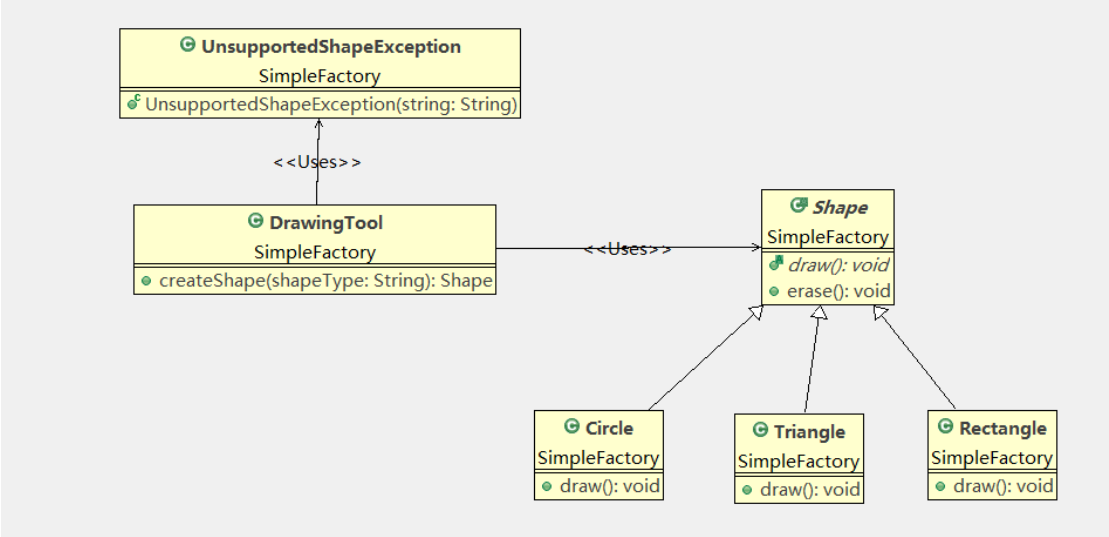
```
public interface ResumeTemplate extends Cloneable{

    public abstract void display();

    public abstract ResumeTemplate createDeepClone();

    public abstract ResumeTemplate createShadowClone();

}

public class Photo implements Serializable{

    private static final long serialVersionUID = 1L;
    private int num;
```

```
    public Photo(int num) {
        this.num = num;
    }

    public void setnum(int num){
        this.num = num;
    }

    @Override
    public String toString() {
        // TODO Auto-generated method stub
        return "MyPhoto's num is " + num;
    }

}

public class Resume implements ResumeTemplate, Serializable{

    private static final long serialVersionUID = -8975396126330142115L;
    private String name;
    private int age;
    private String email;
    private Photo portrait;

    public Resume(String name, int age, String email, Photo portrait) {
        this.name = name;
        this.age = age;
        this.email = email;
        this.portrait = portrait;
    }

    @Override
    public void display() {
        System.out.println("My name is " + name + ".");
        System.out.println("My age is " + age + ".");
        System.out.println("My email is " + email + ".");
        System.out.println("My portrait is " + portrait + ".");

    }

    @Override
    public ResumeTemplate createDeepClone() {

        try {
```

```
        // 将对象写入字节流
        ByteArrayOutputStream outputStream = new
ByteArrayOutputStream();
        ObjectOutputStream objectOutputStream = new
ObjectOutputStream(outputStream);
        objectOutputStream.writeObject(this);

        // 从字节流中读取对象并返回深克隆的副本
        ByteArrayInputStream inputStream = new
ByteArrayInputStream(outputStream.toByteArray());
        ObjectInputStream objectInputStream = new
ObjectInputStream(inputStream);
        return (ResumeTemplate) objectInputStream.readObject();
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }


    }

    @Override
    public ResumeTemplate createShadowClone() {
        ResumeTemplate resumeTemplate = null;
        try {
            resumeTemplate = (ResumeTemplate)clone();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return resumeTemplate;
    }

}

public class Main {

    public static void main(String[] args) {

        Photo p1 = new Photo(1);
        ResumeTemplate r1 = new Resume("A", 1, "1@gmail.com", p1);
        ResumeTemplate r2 = r1.createShadowClone();
        ResumeTemplate r3 = r1.createDeepClone();

        r1.display();
```

```
        System.out.println("");
        r2.display();
        System.out.println("");
        r3.display();

        p1.setnum(2);
        System.out.println("p1.setnum(2)");
        System.out.println("");

        r1.display();
        System.out.println("");
        r2.display();
        System.out.println("");
        r3.display();
    }
}
```

### 3.简单工厂模式



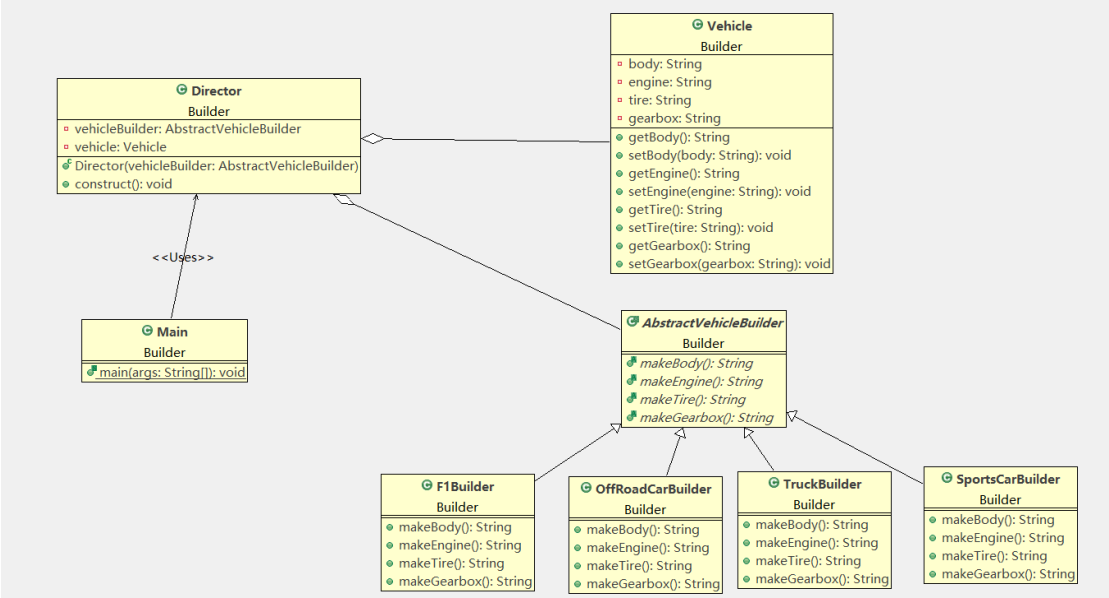

代码实现：

```
public abstract class Shape {

    public abstract void draw();

    public void erase() {
        System.out.println("Erasing the shape");
    }


}
```

```
public class Circle extends Shape{

    @Override
    public void draw() {
        System.out.println("Drawing a circle");

    }

}

public class Rectangle extends Shape{

    @Override
    public void draw() {
        System.out.println("Drawing a rectangle");

    }

}

public class Triangle extends Shape{

    @Override
    public void draw() {
        System.out.println("Drawing a triangle");

    }

}

public class DrawingTool {

    public Shape createShape(String shapeType) throws
UnsupportedShapeException {
        if (shapeType.equalsIgnoreCase("circle")) {
            return new Circle();
        } else if (shapeType.equalsIgnoreCase("rectangle")) {
            return new Rectangle();
        } else if (shapeType.equalsIgnoreCase("triangle")) {
            return new Triangle();
        } else {
            throw new UnsupportedShapeException("Unsupported shape: " +
shapeType);
```

```
        }
    }

}

public class UnsupportedShapeException extends Exception{

    public UnsupportedShapeException(String string) {
        super(string);
    }

}
```

## 4.建造者模式




代码实现：

```
public class Vehicle {

    private String body;
    private String engine;
    private String tire;
    private String gearbox;

    public String getBody() {
        return body;
    }
    public void setBody(String body) {
        this.body = body;
```

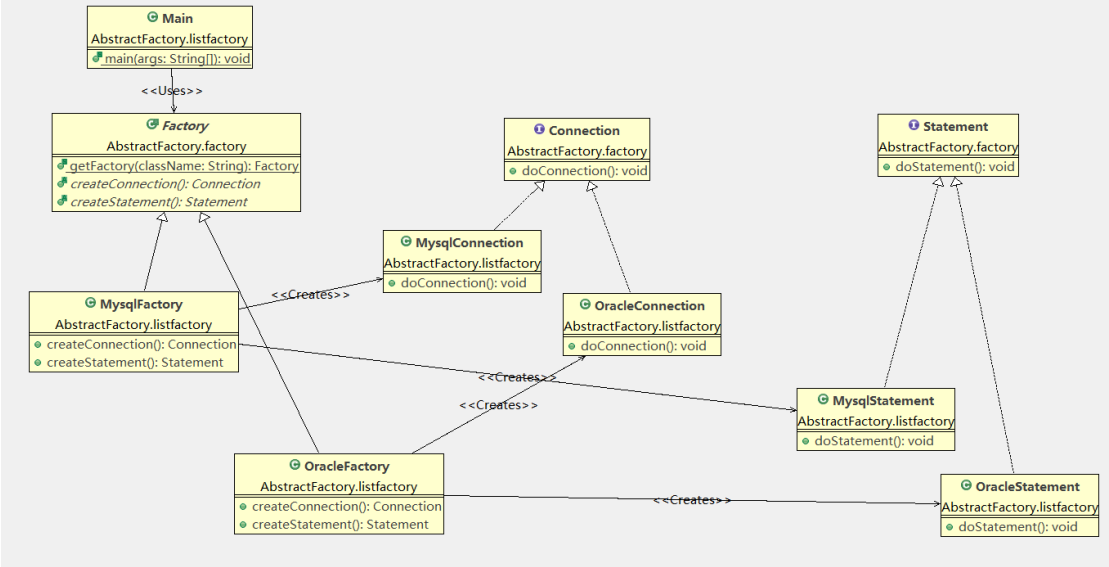
```
    }
    public String getEngine() {
        return engine;
    }
    public void setEngine(String engine) {
        this.engine = engine;
    }
    public String getTire() {
        return tire;
    }
    public void setTire(String tire) {
        this.tire = tire;
    }
    public String getGearbox() {
        return gearbox;
    }
    public void setGearbox(String gearbox) {
        this.gearbox = gearbox;
    }



}

public abstract class AbstractVehicleBuilder {

    public abstract String makeBody();
    public abstract String makeEngine();
    public abstract String makeTire();
    public abstract String makeGearbox();

}

public class F1Builder extends AbstractVehicleBuilder{

    @Override
    public String makeBody() {
        return "F1 Boby";
    }

    @Override
    public String makeEngine() {
        return "F1 Engine";
    }
```

```
    @Override
    public String makeTire() {
        return "F1 Tire";
    }

    @Override
    public String makeGearbox() {
        return "F1 Gearbox";
    }

}

public class OffRoadCarBuilder extends AbstractVehicleBuilder{

    @Override
    public String makeBody() {
        return "OffRoadCar Boby";
    }

    @Override
    public String makeEngine() {
        return "OffRoadCar Engine";
    }

    @Override
    public String makeTire() {
        return "OffRoadCar Tire";
    }

    @Override
    public String makeGearbox() {
        return "OffRoadCar Gearbox";
    }

}

public class SportsCarBuilder extends AbstractVehicleBuilder{

    @Override
    public String makeBody() {
        return "SportsCar Boby";
    }
```

```
    @Override
    public String makeEngine() {
        return "SportsCar Engine";
    }

    @Override
    public String makeTire() {
        return "SportsCar Tire";
    }

    @Override
    public String makeGearbox() {
        return "SportsCar Gearbox";
    }
}

public class TruckBuilder extends AbstractVehicleBuilder{

    @Override
    public String makeBody() {
        return "Truck Boby";
    }

    @Override
    public String makeEngine() {
        return "Truck Engine";
    }

    @Override
    public String makeTire() {
        return "Truck Tire";
    }

    @Override
    public String makeGearbox() {
        return "Truck Gearbox";
    }
}

public class Director {

    private AbstractVehicleBuilder vehicleBuilder;
    private Vehicle vehicle = new Vehicle();
```

```
    public Director(AbstractVehicleBuilder vehicleBuilder) {
        this.vehicleBuilder = vehicleBuilder;
    }

    public void construct() {
        vehicle.setBody(vehicleBuilder.makeBody());
        vehicle.setEngine(vehicleBuilder.makeEngine());
        vehicle.setTire(vehicleBuilder.makeTire());
        vehicle.setGearbox(vehicleBuilder.makeGearbox());

        System.out.println("The construction of this vehicle is " +
vehicle.getBody() + " " + vehicle.getEngine() + " " + vehicle.getTire()
+ " " + vehicle.getGearbox());
    }

}

public class Main {

    public static void main(String[] args) {
        AbstractVehicleBuilder vehicleBuilder1 = new F1Builder();
        AbstractVehicleBuilder vehicleBuilder2 = new TruckBuilder();

        Director director1 = new Director(vehicleBuilder1);
        Director director2 = new Director(vehicleBuilder2);

        director1.construct();
        director2.construct();

    }

}
```

## 5.抽象工厂模式




代码实现：

```
public abstract class Factory {

    public static Factory getFactory(String className) {
        Factory factory = null;
        try {
            factory = (Factory)Class.forName(className).newInstance();
        } catch (ClassNotFoundException e) {
            System.out.println(className + "can not be found.");
        } catch (Exception e) {
            e.printStackTrace();
        }
        return factory;
    }

    public abstract Connection createConnection();
    public abstract Statement createStatement();

}

public interface Connection {

    public void doConnection();

}

public interface Statement {
```

```
    public void doStatement();

}

public class MysqlFactory extends Factory{

    @Override
    public Connection createConnection() {
        return new MysqlConnection();
    }

    @Override
    public Statement createStatement() {
        return new MysqlStatement();
    }

}

public class MysqlConnection implements Connection{

    @Override
    public void doConnection() {
        System.out.println("Doing Mysql Connection");

    }

}

public class MysqlStatement implements Statement{

    @Override
    public void doStatement() {
        System.out.println("Doing Mysql Statement");

    }

}

public class OracleFactory extends Factory{

    @Override
    public Connection createConnection() {
        return new OracleConnection();
    }
```

```
    @Override
    public Statement createStatement() {
        return new OracleStatement();
    }

}

public class OracleConnection implements Connection{

    @Override
    public void doConnection() {
        System.out.println("Doing Oracle Connection");

    }
}

public class OracleStatement implements Statement{

    @Override
    public void doStatement() {
        System.out.println("Doing Oracle Statement");

    }
}

public class Main {

    public static void main(String[] args) {
        Factory factory1 =
Factory.getFactory("AbstractFactory.listfactory.MysqlFactory");
        Factory factory2 =
Factory.getFactory("AbstractFactory.listfactory.OracleFactory");
        Factory factory3 =
Factory.getFactory("AbstractFactory.listfactory.OtherFactory");

        Connection connection = factory1.createConnection();
        Statement statement = factory2.createStatement();

        connection.doConnection();
        statement.doStatement();
    }

}
```

## 五、实验小结

请总结本次实验的体会，包括学会了什么、遇到哪些问题、如何解决这些问题以及存在哪些有待改进的地方。

通过实际编写代码，我加强了对单例模式、原型模式、简单工厂模式，建造者模式和抽象工厂五种设计模式的认识，了解了这五种设计模式在实际运用中的作用和意义。

原型模式中，遇到的问题：需要处理浅克隆和深克隆的情况，以确保正确的对象复制。

解决方案：提供两套实现，一个用于浅克隆，另一个用于深克隆。浅克隆时，直接调用 clone()方法。深克隆时，将对象序列化后再反序列化，得到其副本。