天津大学

设计模式（3）实验报告

学　　院　智算学部

专　　业　软件工程

学　　号　3019213043

姓　　名　刘京宗

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. 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。

2. 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。

3. 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实现。

4. 桥接模式

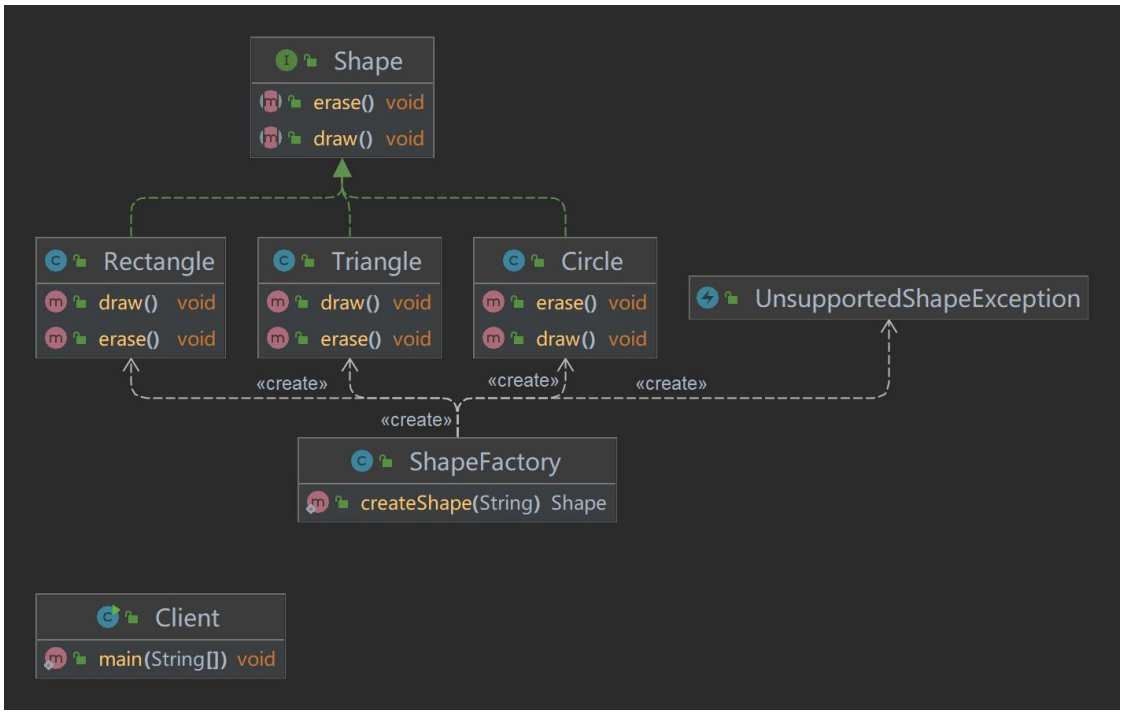
某手机美图 APP 软件支持多种不同的图像格式，例如 JPG、GIF、BMP 等常用图像格式，同时提供了多种不同的滤镜对图像进行处理，例如木刻滤镜（Cutout）、模糊滤镜(Blur）、 锐化滤镜（Sharpen）、纹理滤镜（Texture）等。现采用桥接模式设计该 APP 软件，使得该软件能够为多种图像格式提供一系列图像处理滤镜，同时还能够很方便地增加新的图像格式和滤镜，绘制对应的类图并编程模拟实现。

5. 策略模式

在某云计算模拟平台中提供了多种虚拟机迁移算法，例如动态迁移算法中的 Pre－Copy （预拷贝）算法、 Post － Copy （后拷贝）算法、 CR / RT － Motion 算法等，用户可以灵活地选择所需的虚拟机迁移算法，也可以方便地增加新算法。现采用策略模式进行设计，绘制对应的类图并编程模拟实现。

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。



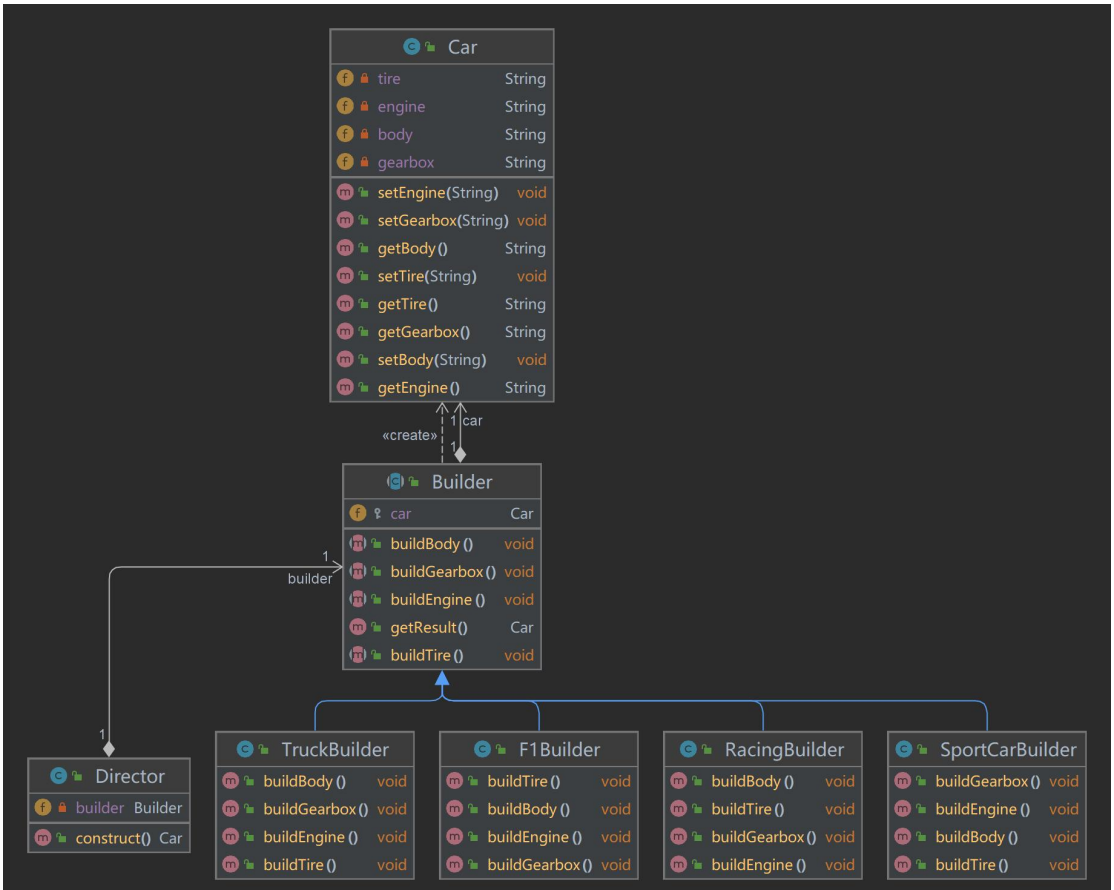
4.1 简单工厂模式

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

    void erase();
}
public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("draw circle");
    }

    @Override
    public void erase() {

        System.out.println("erase circle");
    }
}
public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("draw Rectangle");
    }

    @Override
```

```
    public void erase() {

        System.out.println("erase Rectangle");
    }
}
public class Triangle implements Shape {
    @Override
    public void draw() {
        System.out.println("draw triangle");
    }

    @Override
    public void erase() {

        System.out.println("erase triangle");
    }
}
public class ShapeFactory {
    public static Shape createShape(String name) throws Exception {
        Shape newShape = null;
        if ("circle".equalsIgnoreCase(name)) {
            newShape = new Circle();
        } else if ("rectangle".equalsIgnoreCase(name)) {
            newShape = new Rectangle();
        } else if ("triangle".equalsIgnoreCase(name)) {
            newShape = new Triangle();
        } else {
            throw new UnsupportedShapeException("需要创建的图形不存在");
        }

        return newShape;
    }
}
public class UnsupportedShapeException extends Exception {

    public UnsupportedShapeException() {
    }

    public UnsupportedShapeException(String message) {
        System.out.println(message);
    }
}public class Client {
    public static void main(String[] args) {
        try {
```

```
            Shape shape = ShapeFactory.createShape("rectangle");
            shape.draw();
            shape.erase();
            shape = ShapeFactory.createShape("ellipse");
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

4.2 建造者模式




```
public abstract class Builder {
    protected Car car = new Car();

    public abstract void buildBody();

    public abstract void buildEngine();

    public abstract void buildTire();
```

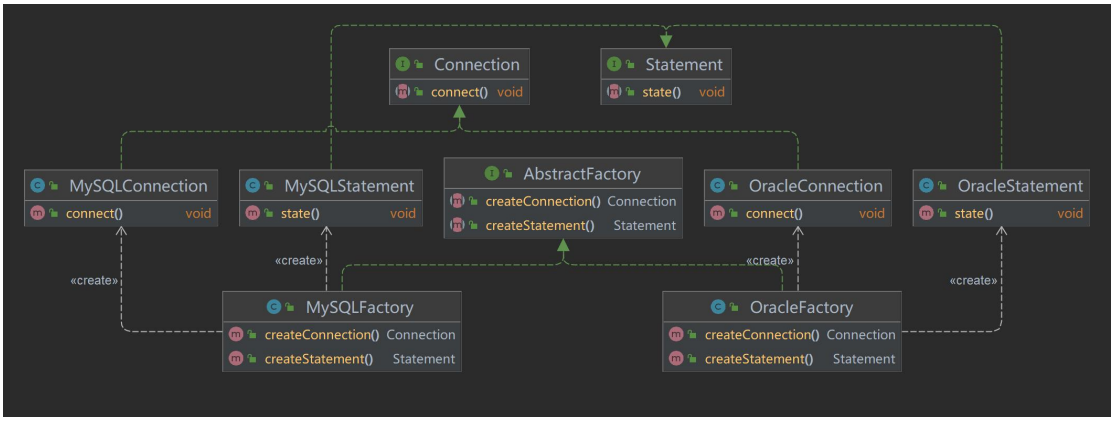
```
    public abstract void buildGearbox();

    public Car getResult() {
        return car;
    }
}
public class Car {
    private String body;
    private String engine;
    private String tire;
    private String gearbox;

    public String getBody() {
        return body;
    }

    public String getEngine() {
        return engine;
    }

    public String getTire() {
        return tire;
    }

    public String getGearbox() {
        return gearbox;
    }

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

    public void setEngine(String engine) {
        this.engine = engine;
    }

    public void setTire(String tire) {
        this.tire = tire;
    }

    public void setGearbox(String gearbox) {
        this.gearbox = gearbox;
    }
}
```

```
public class Director {
	private Builder builder;

	public Director(Builder builder) {
		this.builder = builder;
	}

	public Car construct() {
		builder.buildBody();
		builder.buildEngine();
		builder.buildTire();
		builder.buildGearbox();
		return builder.getResult();
	}
}
public class F1Builder extends Builder {
	@Override
	public void buildBody() {
		car.setBody("F1 body");
	}

	@Override
	public void buildEngine() {
		car.setEngine("F1 engine");
	}

	@Override
	public void buildTire() {
		car.setTire("F1 tire");
	}

	@Override
	public void buildGearbox() {
		car.setGearbox("F1 Gearbox");
	}
}
public class RacingBuilder extends Builder {
	@Override
	public void buildBody() {
		car.setBody("Racing body");
	}

	@Override
	public void buildEngine() {
```

```
            car.setEngine("Racing engine");
        }

        @Override
        public void buildTire() {
            car.setTire("Racing tire");
        }

        @Override
        public void buildGearbox() {
            car.setGearbox("Racing gearbox");
        }
}
public class SportCarBuilder extends Builder {
        @Override
        public void buildBody() {
            car.setBody("SportCar body");
        }

        @Override
        public void buildEngine() {
            car.setEngine("SportCar engine");
        }

        @Override
        public void buildTire() {
            car.setTire("SportCar tire");
        }

        @Override
        public void buildGearbox() {
            car.setGearbox("SportCar gearbox");
        }
}
public class TruckBuilder extends Builder {
        @Override
        public void buildBody() {
            car.setBody("Truck body");
        }

        @Override
        public void buildEngine() {
            car.setEngine("Truck engine");
        }
```

```
    @Override
    public void buildTire() {
        car.setTire("Truck tire");
    }

    @Override
    public void buildGearbox() {
        car.setGearbox("Truck gearbox");
    }
}
```

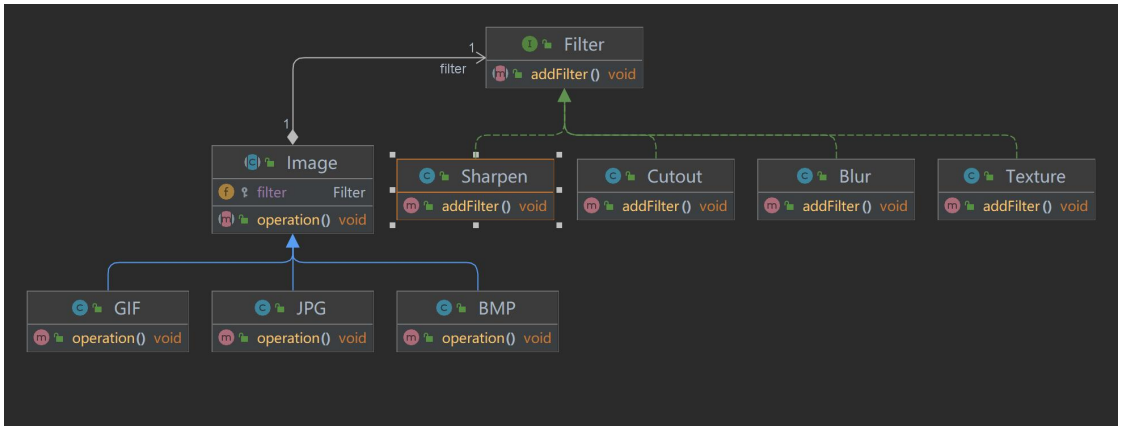
4.3 抽象工厂模式




```
public interface AbstractFactory {
    Connection createConnection();

    Statement createStatement();
}
public interface Connection {
    void connect();
}
public class MySQLConnection implements Connection {
    @Override
    public void connect() {
        System.out.println("MySQL connection");
    }
}
public class MySQLFactory implements AbstractFactory {
    @Override
    public Connection createConnection() {
        return new MySQLConnection();
```

```
    }

    @Override
    public Statement createStatement() {
        return new MySQLStatement();
    }
}
public class MySQLStatement implements Statement {
    @Override
    public void state() {
        System.out.println("MySQL statement");
    }
}
public class OracleConnection implements Connection {
    @Override
    public void connect() {
        System.out.println("Oracle connection");
    }
}
public class OracleFactory implements AbstractFactory {
    @Override
    public Connection createConnection() {
        return new OracleConnection();
    }

    @Override
    public Statement createStatement() {
        return new OracleStatement();
    }
}
public class OracleStatement implements Statement {
    @Override
    public void state() {
        System.out.println("Oracle Statement");
    }
}
public interface Statement {
    void state();
}
```

4.4. 桥接模式

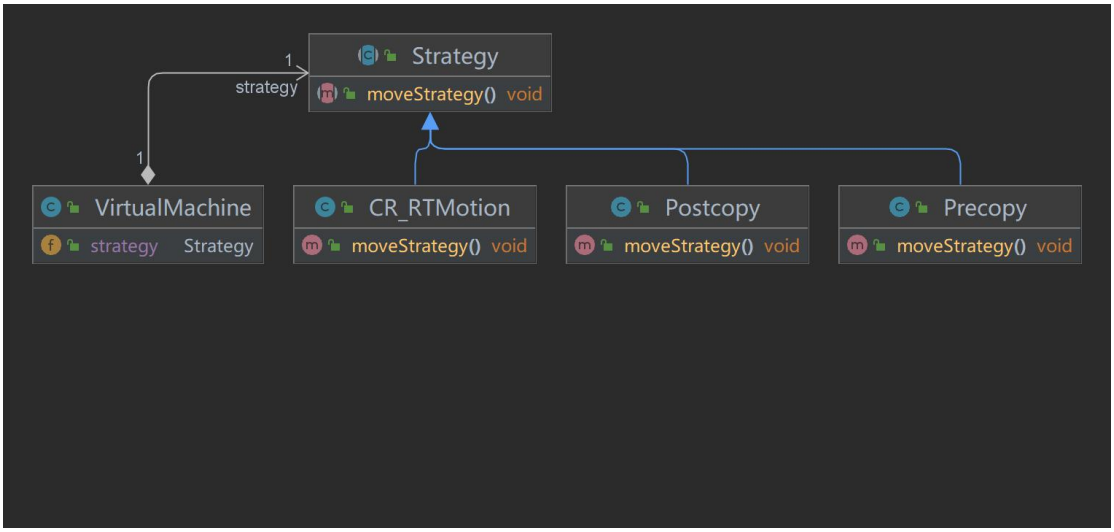



```
public class Blur implements Filter {
    //实现 Filter 接口中声明的方法
    @Override
    public void addFilter() {
        System.out.println("增加模糊滤镜(Blur)!!!");
    }
}
public class BMP extends Image {
    public BMP(Filter filter) {
        super(filter);
    }

    //实现了 Image 中声明的抽象业务方法
    @Override
    public void operation() {
        System.out.print("这是一张 BMP 格式的图片!!!");
        filter.addFilter();//调用 Filter 中定义的业务方法
    }
}
public class Cutout implements Filter {
    //实现 Filter 接口中声明的方法
    @Override
    public void addFilter() {
        System.out.println("增加木刻滤镜(Cutout)!!!");
    }
}
public interface Filter {
    public void addFilter(); //声明方法
}
public class GIF extends Image {
    public GIF(Filter filter) {
        super(filter);
```

```
    }

    //实现了 Image 中声明的抽象业务方法
    @Override
    public void operation() {
        System.out.print("这是一张 GIF 格式的图片!!!");
        filter.addFilter();//调用 Filter 中定义的业务方法
    }
}
public abstract class Image {
    //定义一个 Filter 类型的对象
    protected Filter filter;

    //注入实现类接口对象
    public Image(Filter filter) {
        super();
        this.filter = filter;
    }

    //声明抽象业务方法
    public abstract void operation();
}
public class JPG extends Image {
    public JPG(Filter filter) {
        super(filter);
    }

    //实现了 Image 中声明的抽象业务方法
    @Override
    public void operation() {
        System.out.print("这是一个 JPG 格式的图片!!!");
        filter.addFilter(); //调用 Filter 中定义的业务方法
    }
}
public class Sharpen implements Filter {
    //实现 Filter 接口中声明的方法
    @Override
    public void addFilter() {
        System.out.println("增加模锐化滤镜(Sharpen)!!!");
    }
}
public class Texture implements Filter {
    //实现 Filter 接口中声明的方法
    @Override
```

```
    public void addFilter() {
        System.out.println("增加纹理滤镜（Texture)!!!");
    }
}
```

4.5 策略模式




```
public class VirtualMachine {
    public Strategy strategy;
}
public abstract class Strategy {
    public abstract void moveStrategy();
}
public class Precopy extends Strategy {
    @Override
    public void moveStrategy() {

    }
}
public class Postcopy extends Strategy {
    @Override
    public void moveStrategy() {

    }
}
public class CR_RTMotion extends Strategy {
    @Override
    public void moveStrategy() {

    }
}
```

五、实验小结

通过这次实验，我熟悉了简单工厂模式、建造者模式、抽象工厂模式、桥接模式、策略模式这些设计模式，收益匪浅。