实验十三

实验目的：

1. 学习设计模式，能在项目设计中运用设计模式进行面向对象设计
2. 编程指导原则：讨论程序编写方式与规则

实验内容：

阅读下面设计模式资料（或查阅其它相关资料），结合项目的进程和开发历程，分析项目采用了那些设计模式

Design Patterns-Elements of Reusable Object-Oriented Software.pdf

The GoF Design Patterns Reference.pdf

Design Patterns - Wikipedia

1. MVC（Model-View-Controller）模式：

前端使用Vue框架，后端使用Spring Boot框架，这两个框架本身就是基于MVC模式设计的。MVC模式将应用程序划分为三个核心组件：模型（Model）、视图（View）和控制器（Controller），以实现数据的分离和业务逻辑的解耦。模型表示数据和业务逻辑，视图负责展示用户界面，控制器处理用户交互并将数据传递给模型或视图。

- 模型（Model）：

模型表示数据和业务逻辑。在本项目中，模型负责处理用户数据、博客内容、评论和其他持久化数据的读取、存储和管理。例如，用户注册和登录模块将处理用户信息的验证和存储，博客模块将负责管理博客的创建、编辑和分类。

- 视图（View）：

视图负责展示用户界面。在前台博客网站中，使用Vue框架构建视图层，通过模板和组件来渲染用户界面。视图将显示博客内容、用户个人信息、评论和其他相关信息。此外，视图还可以与用户进行交互，例如用户注册、登录和评论的表单。

- 控制器（Controller）：

控制器处理用户交互并将数据传递给模型或视图。在前台博客网站中，控制器负责处理用户的行为，例如用户的注册和登录请求、搜索请求、评论请求等。控制器将根据用户的输入和操作调用相应的模型方法，并更新视图以反映最新的数据。

通过MVC模式，项目实现了业务逻辑的分离和解耦。模型处理数据存储和业务逻辑，视图负责展示用户界面，而控制器协调用户交互和数据处理。这种分层结构使得您的代码更加模块化、可维护性更高，同时也提高了开发效率，团队成员可以独立开发和测试各个组件。

2. 观察者模式（Observer Pattern）：

在前台博客网站的博客模块中，用户可以在博客下方进行评论和回复。这涉及到博客模块和评论模块之间的信息传递和更新。观察者模式可以用来实现这种场景，其中博客模块是被观察者（Subject），评论模块是观察者（Observer），当有新评论时，博客模块通知所有注册的评论模块进行更新。

- 被观察者（Subject）：

被观察者是博客模块，在这种情况下，它负责管理博客内容和相关评论。被观察者维护一个观察者列表，并提供注册、注销和通知观察者的方法。当有新的评论添加到博客下方时，被观察者将通知观察者。

- 观察者（Observer）：

观察者是评论模块，对博客的评论状态感兴趣并希望及时获得更新。评论模块将注册为被观察者的观察者，并实现一个接收更新的方法。当被观察者通知观察者有新评论时，观察者将执行相应的操作，例如更新评论列表或显示新评论。通过观察者模式，博客模块和评论模块之间实现了解耦和灵活的通信机制。博客模块不需要直接知道有哪些评论模块存在，它只需要通知注册的观察者即可。同时，观察者模式还支持动态注册和注销观察者，因此其他模块也可以在需要时成为评论模块的观察者。

3. 工厂模式（Factory Pattern）：

在后台管理网站的文章管理模块中，可能存在对文章的创建和编辑操作。工厂模式可以用来封装文章的创建逻辑，通过一个工厂类创建具体的文章对象，而不暴露创建细节给调用方。这样可以降低耦合性，提高代码的可维护性和可扩展性。

以下是工厂模式的组成部分：

- 抽象产品（Abstract Product）：

抽象产品可以是一个接口或基类，定义了文章对象的通用行为和属性。它规定了创建具体产品的方法。

- 具体产品（Concrete Product）：

具体产品是实现了抽象产品定义的具体文章对象。每个具体产品都有自己的实现，可能包括不同类型的文章（如新闻、教程、博客）或其他属性（如标题、内容、作者）。

- 工厂（Factory）：

工厂是负责创建具体产品的类。它封装了创建产品的细节，并提供一个公共的创建方法供外部调用。在您的文章管理模块中，可以有一个文章工厂类，根据传入的参数或条件，创建具体的文章对象并返回。

通过工厂模式，可以将创建文章的过程封装在工厂类中，调用方无需关心具体的创建细节。调用方只需要与工厂进行交互，请求创建文章对象。工厂根据传入的参数或条件，实例化适当的具体产品，并返回给调用方。工厂模式的优点之一是降低了系统的耦合性。调用方无需直接依赖于具体产品的创建

过程，而是通过工厂进行创建，从而解耦了调用方和具体产品的实现。这样，如果需要添加新的文章类型或修改创建逻辑，只需修改工厂类而不影响调用方。此外，工厂模式还提高了代码的可维护性和可扩展性。通过将创建逻辑封装在工厂类中，可以更容易地管理和修改创建过程。如果需要添加新的具体产品，只需创建一个新的具体产品类和对应的工厂方法即可，而不需要修改调用方的代码。

4. 代理模式（Proxy Pattern）：

后台管理网站的权限管理模块中，可能需要对用户角色进行管理，并限制对敏感资源的访问。代理模式可以用来实现对用户权限的控制。代理类可以在用户请求访问资源之前进行权限验证，以确保只有具有相应权限的用户才能访问。

在后台管理网站的权限管理模块中，代理模式可以用于实现对用户角色和敏感资源的管理和控制。以下是代理模式在该场景下的组成部分：

- 抽象主题（Subject）：

抽象主题定义了真实主题（敏感资源）和代理主题（代理类）的共同接口。它可以是一个接口或抽象类，定义了代理类和真实主题类共同的行为。

- 真实主题（Real Subject）：

真实主题是具体的敏感资源类，实现了抽象主题定义的接口。真实主题是代理模式中被代理的对象，代理类将通过真实主题来提供服务。

- 代理（Proxy）：

代理类实现了抽象主题接口，并在其内部持有一个真实主题对象的引用。代理类在客户端和真实主题之间充当中介角色，控制对真实主题的访问。代理类可以在访问真实主题之前或之后执行一些额外的操作，如权限验证、日志记录等。

代理类可以在用户请求访问敏感资源之前进行权限验证，确保只有具有相应权限的用户才能访问。代理类可以根据用户的角色信息，决定是否允许用户访问敏感资源，或者在访问之前进行额外的权限检查。这种方式可以实现安全访问控制，确保敏感资源只被授权的用户所访问。代理模式的优点之一是可以实现透明的访问控制。对于客户端来说，代理类和真实主题类具有相同的接口，客户端无需知道代理类的存在，就可以通过代理类访问真实主题。代理模式还可以在保持真实主题类的封装性和独立性的同时，对其进行功能扩展，如添加权限验证、缓存等功能。

给出4种设计模式的例子（语言不限，以组为单位），并总结其特点 （保存到每个小组选定的协作开发平台上）

单例模式（Singleton Pattern）：

例子：

在一个多线程环境下，一个日志记录器类的单例模式可以确保所有线程共享同一个日志实例，避免多个实例写入冲突。以下是一个示例的代码实现（使用Java语言）：

```java
public class Logger {
    private static Logger instance;
    private String logFile;

    private Logger() {
        logFile = "application.log";
    }

    public static synchronized Logger getInstance() {
        if (instance == null) {
            instance = new Logger();
        }
        return instance;
    }

    public void log(String message) {
        // 实现日志写入逻辑
        // ...
    }
}
```

在上述示例中，Logger类被设计为单例模式。它使用了私有的构造函数来限制实例化，通过一个静态方法getInstance()来获取实例。在多线程环境下，通过synchronized关键字修饰getInstance()方法，确保在并发情况下只有一个线程可以创建实例。

其他部分的代码实现了日志的写入逻辑，可以通过调用log()方法向日志文件写入日志信息。

使用单例模式的好处是，无论在多少个地方调用Logger.getInstance()方法，都只会返回同一个Logger实例，确保了日志记录的一致性。

特点：

- 单例模式只允许类创建一个实例，所有对该类的实例化操作都返回同一个对象。
- 单例模式提供了一个全局访问点，使得其他对象可以方便地访问该单例对象。
- 单例模式在整个应用程序中只存在一个实例，节省了系统资源。

工厂模式（Factory Pattern）：

例子：

以汽车制造工厂为例，可以通过工厂模式来创建不同品牌的汽车对象。以下是一个示例的代码实现（使用Java语言）：

首先，定义一个抽象的汽车接口：

```java
public interface Car {
```

```
    void drive();
}
```

然后，创建具体的汽车类，例如奥迪（Audi）和宝马（BMW）：

```
public class Audi implements Car {
    @Override
    public void drive() {
        System.out.println("Driving an Audi.");
    }
}

public class BMW implements Car {
    @Override
    public void drive() {
        System.out.println("Driving a BMW.");
    }
}
```

接下来，创建抽象工厂接口，并定义一个用于创建汽车对象的工厂方法：

```
public interface CarFactory {
    Car createCar();
}
```

然后，实现具体的工厂类，分别负责创建奥迪和宝马对象：

```
public class AudiFactory implements CarFactory {
    @Override
    public Car createCar() {
        return new Audi();
    }
}

public class BMWFactory implements CarFactory {
    @Override
    public Car createCar() {
        return new BMW();
    }
}
```

最后，客户端代码可以通过工厂来获取所需的汽车对象，而无需关心具体的实现细节：

```
public class Client {
    public static void main(String[] args) {
        CarFactory factory = new AudiFactory();
        Car car = factory.createCar();
        car.drive();
    }
}
```

在上述示例中，通过定义抽象的汽车接口和具体的汽车类，以及抽象工厂接口和具体工厂类，将对象的创建过程封装在工厂中。客户端代码通过工厂来获取所需的汽车对象，并调用其方法。这样，客户端代码与具体对象的创建过程解耦，使得系统更具灵活性和可扩展性。当需要添加新的汽车品牌时，只需实现一个新的具体工厂类，而不需要修改现有的客户端代码。

特点：

- 工厂模式通过引入一个抽象工厂接口和多个具体工厂类，将对象的创建与客户端代码分离，使得客户端代码只需要与抽象工厂接口交互，而无需关心具体对象的创建细节。
- 工厂模式可以轻松添加新的产品类型，只需实现一个新的具体工厂类即可，而不需要修改现有的客户端代码。
- 工厂模式提供了一种扩展性和灵活性，使得系统更易于维护和扩展。

观察者模式（Observer Pattern）：

例子：

以一个气象站为例，气象站收集天气数据，并将数据通知给不同的显示设备。以下是一个示例的代码实现（使用Java语言）：

首先，定义主题（被观察者）接口：

```
public interface Subject {
    void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}
```

然后，创建具体的主题类，实现主题接口，并维护观察者列表，以及在状态变化时通知观察者：

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

```
3
4  public class WeatherStation implements Subject {
5      private List<Observer> observers;
6      private float temperature;
7
8      public WeatherStation() {
9          observers = new ArrayList<>();
10     }
11
12     @Override
13     public void registerObserver(Observer observer) {
14         observers.add(observer);
15     }
16
17     @Override
18     public void removeObserver(Observer observer) {
19         observers.remove(observer);
20     }
21
22     @Override
23     public void notifyObservers() {
24         for (Observer observer : observers) {
25             observer.update(temperature);
26         }
27     }
28
29     public void setTemperature(float temperature) {
30         this.temperature = temperature;
31         notifyObservers();
32     }
33 }
```

接下来，定义观察者接口：

```
1  public interface Observer {
2      void update(float temperature);
3  }
```

然后，创建具体的观察者类，实现观察者接口，并定义更新逻辑：

```
1  public class DisplayDevice implements Observer {
2      private String name;
3
```

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

    @Override
    public void update(float temperature) {
        System.out.println(name + ": The temperature is " + temperature + " degr
    }
}
```

最后，客户端代码可以创建主题对象和观察者对象，并将观察者注册到主题上，当主题的温度发生变化时，观察者会自动更新显示：

```
public class Client {
    public static void main(String[] args) {
        WeatherStation weatherStation = new WeatherStation();

        DisplayDevice display1 = new DisplayDevice("Display 1");
        DisplayDevice display2 = new DisplayDevice("Display 2");

        weatherStation.registerObserver(display1);
        weatherStation.registerObserver(display2);

        weatherStation.setTemperature(25.5f);
        weatherStation.setTemperature(28.2f);
    }
}
```

在上述示例中，气象站充当主题（被观察者），显示设备充当观察者。当气象站的温度发生变化时，它会通知所有注册的观察者，观察者会自动更新显示。这样，气象站与显示设备之间实现了解耦，观察者模式提供了一种灵活的方式来构建可扩展和可维护的系统。

特点：

- 观察者模式使用了发布-订阅的机制，其中主题（被观察者）维护了一个观察者列表，并在状态发生变化时通知所有观察者。
- 观察者模式实现了松耦合，主题和观察者之间相互独立，可以方便地扩展和修改。
- 观察者模式符合开闭原则，可以在不修改主题和观察者的情况下增加新的观察者。

装饰器模式（Decorator Pattern）：

例子：

在一个图形绘制应用中，可以使用装饰器模式为具体的图形对象附加颜色、边框等装饰器，实现动态添加功能而不改变原有图形对象的结构。以下是一个示例的代码实现（使用Java语言）：

首先，定义一个抽象的图形接口：

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

然后，创建具体的图形类，例如矩形和圆形：

```
public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a rectangle.");
    }
}

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

接下来，定义装饰器接口，并实现具体的装饰器类，例如颜色装饰器和边框装饰器：

```
public interface ShapeDecorator extends Shape {
    // 空接口，继承自Shape接口
}

public class ColorDecorator implements ShapeDecorator {
    private Shape decoratedShape;
    private String color;

    public ColorDecorator(Shape decoratedShape, String color) {
        this.decoratedShape = decoratedShape;
        this.color = color;
    }

    @Override
    public void draw() {
```

```
        decoratedShape.draw();
        System.out.println("Applying color: " + color);
    }
}

public class BorderDecorator implements ShapeDecorator {
    private Shape decoratedShape;
    private int borderWidth;

    public BorderDecorator(Shape decoratedShape, int borderWidth) {
        this.decoratedShape = decoratedShape;
        this.borderWidth = borderWidth;
    }

    @Override
    public void draw() {
        decoratedShape.draw();
        System.out.println("Applying border with width: " + borderWidth);
    }
}
```

最后，客户端代码可以创建原始的图形对象，然后通过装饰器来动态地添加功能：

```
public class Client {
    public static void main(String[] args) {
        // 创建原始的图形对象
        Shape rectangle = new Rectangle();
        Shape circle = new Circle();

        // 通过装饰器为图形对象添加颜色和边框
        Shape decoratedRectangle = new ColorDecorator(rectangle, "Red");
        decoratedRectangle.draw();

        Shape decoratedCircle = new BorderDecorator(circle, 2);
        decoratedCircle.draw();
    }
}
```

在上述示例中，通过定义抽象的图形接口和具体的图形类，以及装饰器接口和具体的装饰器类，实现了图形对象的功能扩展。客户端代码可以创建原始的图形对象，并通过装饰器来动态地添加颜色和边框的功能。装饰器模式提供了一种灵活的方式来扩展对象的功能，同时保持了对象结构的稳定性。

特点：

- 装饰器模式通过组合而不是继承的方式实现功能的扩展，可以在运行时动态地添加、修改或删除对象的功能。
- 装饰器模式遵循开闭原则，允许在不修改现有代码的情况下增加新的装饰器类和功能。
- 装饰器模式通过透明的方式将装饰器对象嵌套在被装饰对象中，对客户端代码而言，被装饰对象和装饰器对象是一致的。

上网查询“阿里编程规范 （如：阿里巴巴JAVA开发手册）；华为 编程军规”等，对照自己的代码看有哪些不符合规范的地方，修改。