实验十四 设计模块（三 ）

实验目的：

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

实验内容：

1. 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](https://en.wikipedia.org/wiki/Design_Patterns)

（1）文档简介与目的

文档目的：确保项目干系人对软件的高层设计有共同的理解，促进团队沟通，保证开发一致性和高效性。

范围：包括架构描述、数据结构说明、逻辑架构视图等。

（2）架构设计目标

关键功能：涉及用户登录、注册、简历上传、个人中心管理、后台管理等。

关键质量属性：安全性、可用性与响应性、可扩展性、用户界面友好性、兼容性和性能、可维护性和可测试性。

（3）架构设计原则

原则：模块化、接口、信息隐藏、增量式开发、抽象、通用性。

（4）备选架构设计方案及选择

备选方案：微服务架构与单体架构的混合模式、完全基于容器化的微服务架构、基于云原生的Serverless架构。

选择原因：根据项目需求、成本、技术栈熟悉度、系统规模等因素进行选择。

（5）全局数据结构说明

常量与变量：定义了系统中使用的常量和变量。

数据结构：详细列出了系统中使用的数据结构，如报名招聘、在线客服、求职论坛等。

（6）逻辑架构视图

顶层逻辑架构：包括注册中心、服务中心、中间件、存储和持续集成与容器集成技术。

功能逻辑架构：分为个人用户端、企业用户端、管理员端功能模块。

开发视图与运行架构视图：展示了系统的开发和运行视图。

物理架构视图：描述了系统的物理部署，包括硬件选型和软件到硬件的映射。

（7）关键质量属性的设计原理

性能、安全性、可用性、可维护性、可扩展性、兼容性、可测试性、用户体验：针对每个属性，都有相应的设计目标和策略。

**设计模式分析**

文档中没有直接提到具体的设计模式，但根据描述，我们可以推断可能采用了以下设计模式：

MVC（Model-View-Controller）：用于分离视图、模型和控制器，提高代码的可维护性和可扩展性。

微服务架构：通过将应用分解为一组小服务，每个服务实现特定功能，提高系统的可扩展性和可维护性。

API Gateway：作为系统入口，负责路由、认证等，是微服务架构中常用的模式。

Repository模式：在数据访问层使用，用于抽象和封装对数据源的访问。

工厂模式：可能用于创建对象，特别是复杂的对象创建过程中。

单例模式：在需要全局访问点或共享资源时使用，如配置管理。

**开发历程**

开发历程遵循了以下步骤：

需求分析与系统设计。

技术选型与架构决策。

模块化开发与接口设计。

实现关键功能与质量属性。

持续集成与部署（CI/CD）。

测试与质量保证。

系统部署与优化。

维护与迭代开发。

2. 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）

1. 单例模式（Singleton Pattern）

例子：

public class Logger {

private static Logger instance;

private Logger() {}

public static synchronized Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

*// 实际的日志记录逻辑*

}

}

特点：

全局访问点：确保一个类只有一个实例，并提供一个全局访问点。

线程安全：通过同步机制确保多线程环境下的安全性。

延迟初始化：实例在第一次被使用时才创建。

资源开销：由于控制了实例的创建，可以节省资源开销。

测试困难：由于全局状态的存在，可能会导致代码测试变得困难。

2. 工厂方法模式（Factory Method Pattern）

例子：

interface Product {

void use();

}

class ConcreteProduct implements Product {

public void use() {

*// 产品使用逻辑*

}

}

abstract class Creator {

abstract Product factoryMethod();

}

class ConcreteCreator extends Creator {

public Product factoryMethod() {

return new ConcreteProduct();

}

}

特点：

封装性：将对象的创建和使用分离，提高系统的封装性。

扩展性：易于扩展新的产品类，无需修改现有代码。

解耦：客户端不需要知道具体的类是如何实现的，只需知道接口。

一致性：可以保证生成对象的一致性。

3. 观察者模式（Observer Pattern）

例子：

interface Observer {

void update(String message);

}

interface Subject {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

class ConcreteSubject implements Subject {

private List<Observer> observers = new ArrayList<>();

public void registerObserver(Observer o) {

observers.add(o);

}

public void removeObserver(Observer o) {

observers.remove(o);

}

public void notifyObservers() {

for (Observer observer : observers) {

observer.update("Message");

}

}

}

特点：

低耦合：观察者和被观察对象之间的耦合度低。

动态绑定：观察者可以在运行时添加或移除。

广播通信：支持广播通信，一个事件可以通知多个观察者。

内存管理：需要自己管理观察者列表，可能会有内存泄露的风险。

4. 装饰器模式（Decorator Pattern）

例子：

interface Component {

void operate();

}

class ConcreteComponent implements Component {

public void operate() {

*// 组件的操作*

}

}

abstract class Decorator implements Component {

protected Component component;

public Decorator(Component component) {

this.component = component;

}

public void operate() {

component.operate();

}

}

class ConcreteDecoratorA extends Decorator {

public ConcreteDecoratorA(Component component) {

super(component);

}

public void operate() {

super.operate();

*// 添加额外的功能*

}

}

特点：

动态扩展：可以在运行时动态地给对象添加额外的职责。

透明性：装饰者和被装饰者可以是同一个接口，对客户端透明。

灵活性：可以多个装饰者组合使用，提供更复杂的功能扩展。

复杂性：装饰者模式可能会使设计变得更加复杂，特别是当有多个装饰者时。

项目跟踪，建立能反映项目及小组每个人工作的进度、里程碑、工作量的跟踪图或表，将其保存到每个小组选定的协作开发平台上，每周更新。