**第八次作业**



**IoC（控制反转）**

​**​控制反转​**​是一种软件设计原则，核心是将程序的控制权从内部业务逻辑转移到外部容器或框架。传统编程中，对象主动创建和管理自己的依赖，而在IoC模式下，这些职责交由外部容器统一处理。

例如，框架负责对象的实例化、依赖注入及生命周期管理，应用程序只需关注业务逻辑。IoC通过解耦组件依赖，提升了系统的灵活性和可维护性，常见的实现方式包括依赖注入和服务定位器。

**DIP（依赖倒置原则）**

​**​依赖倒置原则​**​是SOLID设计原则中的“D”，强调​**​高层模块不应直接依赖低层模块，而应共同依赖抽象接口​**​。具体实现细节应通过接口或抽象类向上层解耦。

例如，在电商系统中，订单处理模块（高层）不应直接依赖支付接口的具体实现（如支付宝、微信支付），而应依赖一个抽象的支付接口。低层支付实现类再通过继承或实现该接口完成功能。DIP通过抽象隔离了变化，使系统更易于扩展和修改。

**Dependency Injection（依赖注入）**

​**​依赖注入​**​是实现IoC和DIP的具体技术手段，其核心思想是​**​从外部向对象注入依赖，而非对象内部主动创建​**​。

例如，一个用户服务类需要数据库连接，传统方式可能在类内直接实例化数据库对象，而依赖注入则通过构造函数、Setter方法或接口将已创建的数据库实例传入。这种方式将依赖关系外部化，使得组件更松散耦合，便于单元测试（如替换为Mock对象）和功能扩展（如切换数据库类型）。

**总结如下：**



**IoC举例：**

class UserService {

private Database db; // 依赖抽象

public UserService(Database db) { // 依赖由外部传入

this.db = db;

}

}

**DIP举例：**

// 定义抽象接口

interface Logger {

void log(String message);

}

// 高层模块依赖抽象

class ReportService {

private Logger logger;

public ReportService(Logger logger) { // 依赖接口

this.logger = logger;

}

void generate() {

logger.log("生成报告");

}

}

// 低层模块实现接口

class FileLogger implements Logger {

@Override

void log(String message) { /\* 写入文件 \*/ }

}

class DatabaseLogger implements Logger { // 可扩展新的日志实现

@Override

void log(String message) { /\* 写入数据库 \*/ }

}

**Dependency Injection举例：**

class UserService {

private Database db;

public UserService(Database db) { // 依赖通过构造函数传入

this.db = db;

}

}