实验十二

实验目的：

1. 培养设计原则实践的能力
2. 学习依赖注入（dependency injection）
3. 面向对象设计原则

实验内容：

1.参考教材6.2，结合项目的进程和开发历程，从设计原则的几个方面，组员对负责设计的模块进行评估，思考存在的问题和解决方案

六个设计原则：模块化，接口，信息隐藏，增量式开发，抽象，通用性

一、用户管理模块

1、 模块化：问题出现在没有明确划分出独立的功能模块，内容耦合紧密，导致一个模块的修改可能会影响其他功能。

解决方案：将用户管理模块分为不同的功能模块，例如登录、注册、身份验证、权限控制等，通过定义清晰的模块接口和单一职责原则来保证模块之间的独立性。

2、 接口：问题出现在接口设计不够清晰，难以满足扩展性和复用性的需求。

解决方案：定义良好的接口规范，包括输入输出参数、返回值类型等，封装接口实现细节，遵循依赖倒置原则来降低模块之间耦合度。

3、 信息隐藏：问题出现在模块内部实现的细节暴露给了外部访问接口，引起无意中的调用或者非法调用。

解决方案：采用封装和隐藏的方法，把模块内部实现保持私有和隐蔽，只向外暴露必要的接口，提高系统安全性和保密性。

4、 增量式开发：问题出现在没有充分考虑到组件的更新和维护，导致升级时需要重构已有的代码。

解决方案：采用增量式开发模式，把系统划分为多个部分，每个小部分相对独立，可以快速迭代和发布。同时，实施测试驱动开发或者持续集成来保证代码质量和可靠性。

5、 抽象：问题出现在没有足够的抽象和泛化，导致某些代码无法重用或者难以扩展。

解决方案：在设计过程中，采用通用的程序结构、数据类型和算法等进行抽象，如MVC框架、ORM库等，提高代码的灵活性和可重用度。

6、 通用性：问题出现在没有考虑到用户需求和不同平台之间的差异性，导致软件缺乏适应性。

解决方案：在设计和开发的早期阶段，要充分了解用户的需求和市场的变化趋势，尽量采用通用的软件设计原则和技术，为未来的扩展和适应性做好准备。

二、文章管理模块

1、 模块化：问题出现在没有明确划分出独立的功能模块，内容耦合紧密，导致一个模块的修改可能会影响其他功能。

解决方案：将文章管理模块分为不同的功能模块，例如发布、编辑、删除、分类、搜索等，通过定义清晰的模块接口和单一职责原则来保证模块之间的独立性。

2、 接口：问题出现在接口设计不够清晰，难以满足扩展性和复用性的需求。

解决方案：定义良好的接口规范，包括输入输出参数、返回值类型等，封装接口实现细节，遵循依赖倒置原则来降低模块之间耦合度。例如，可以定义一个文章管理接口来表示发布、编辑和删除等操作。

3、 信息隐藏：问题出现在模块内部实现的细节暴露给了外部访问接口，引起无意中的调用或者非法调用。

解决方案：采用封装和隐藏的方法，把模块内部实现保持私有和隐蔽，只向外暴露必要的接口，提高系统安全性和保密性。例如，可以把文章的具体实现细节封装在文章管理模块中，对外暴露只是一个简单的文章对象。

4、 增量式开发：问题出现在没有充分考虑到组件的更新和维护，导致升级时需要重构已有的代码。

解决方案：采用增量式开发模式，把系统划分为多个部分，每个小部分相对独立，可以快速迭代和发布。同时，实施测试驱动开发或者持续集成来保证代码质量和可靠性。例如，针对文章管理模块，可以设计一个自动化测试框架，在代码修改或升级时对系统进行测试和验证。

5、 抽象：问题出现在没有足够的抽象和泛化，导致某些代码无法重用或者难以扩展。

解决方案：在设计过程中，采用通用的程序结构、数据类型和算法等进行抽象，如MVC框架、ORM库等，提高代码的灵活性和可重用度。例如，可以定义一个抽象的文章对象，封装

三、评论管理模块

1、 模块化

存在的问题：该模块没有进行有效的模块化划分，导致代码耦合度高、难以维护。

解决方案：进行模块化设计，将该模块划分为更小的、功能相关的子模块，每个子模块具有明确的职责，提高模块的灵活性和可扩展性。

2、 接口

存在的问题：该模块缺乏标准化接口设计，导致不同模块之间无法互相协调开发。

解决方案：明确接口设计，在模块之间使用共享接口的标准，确保代码质量和协调开发。

3、 信息隐藏

存在的问题：未实现信息隐藏原则可能导致潜在的安全漏洞，同时也会影响协同开发的效率。

解决方案：封装变化，限制访问，简化使用流程，降低耦合度，实现信息隐藏，确保系统的稳定性和安全性。

4、 增量式开发

存在的问题：新增功能很难提供增量开发能力，难以使现有功能稳定并保持不变。

解决方案：确保模块具有增量式开发的能力，即新增功能不会影响现有功能的稳定性，从而实现模块的可持续迭代和升级。

5、 抽象

存在的问题：由于类和对象之间分离度不够，可能会导致代码冗余且难以维护。

解决方案：遵守抽象原则，在编写代码时限制具体化，对扩展开放，延迟变化点，确保系统易于维护和扩展。

6、 通用性

存在的问题：评论管理模块的设计缺乏通用性的考虑，导致其灵活性和可定制性较差。

解决方案：引入公共部分，并确保组件在更多情况下高

四、相册管理模块

1、模块化：
问题：相册管理模块的功能过于庞大，难以理解和修改。
解决方案：将相册管理模块拆分为独立的子模块，如相片上传、相片浏览、相片编辑等，每个子模块负责一个明确的任务。确保模块之间的接口清晰，便于理解和维护。

2、接口设计：
问题：相册管理模块的接口定义不清晰，导致其他模块难以正确使用。
解决方案：定义明确的接口，如相片上传接口、相片删除接口等。明确接口的输入和输出，以及预期的行为和返回结果。确保接口的命名清晰、一致，并提供适当的文档说明。

3、信息隐藏：
问题：相册管理模块的内部实现细节暴露给其他模块，导致耦合度高。

解决方案：封装相册管理模块的内部实现细节，只暴露必要的接口给其他模块使用。通过使用私有方法、保护方法和公共方法来控制对内部数据和功能的访问。

4、增量式开发：
问题：相册管理模块一次性开发完成，无法及时验证功能和适应变化的需求。
解决方案：采用增量式开发方法，将相册管理功能划分为小的增量，如基本相册功能、标签功能、分享功能等，并逐步实现、测试和集成。每个增量都能独立运行和演化，使得问题的检测和修复更加容易。

5、抽象：
问题：相册管理模块缺乏合适的抽象层次，导致代码重复和难以扩展。
解决方案：通过抽象将相册管理模块中的共享概念和功能提取出来，形成可复用的组件。例如，可以设计一个通用的相片处理组件，用于上传、编辑和浏览相片。使用设计模式、抽象数据类型等技术来实现高层次的抽象。

6、通用性：
问题：相册管理模块过于专注于当前需求，缺乏通用性。
解决方案：在相册管理模块的设计中考虑到可扩展性和灵活性。可以设计相册管理模块时应考虑支持不同类型的相片和不同的存储方式。提供适当的接口和扩展点，以便未来能够方便地添加新的功能或适应新的需求。

五、说说管理模块

1、模块化：
问题：说说管理模块的功能过于庞大，难以理解和修改。
解决方案：将说说管理模块拆分为独立的子模块，如发布说说、查看说说、评论说说等，每个子模块负责一个明确的任务。确保模块之间的接口清晰，便于理解和维护。

2、接口设计：
问题：说说管理模块的接口定义不清晰，导致其他模块难以正确使用。
解决方案：定义明确的接口，如发布说说接口、删除说说接口、评论说说接口等。明确接口的输入和输出，以及预期的行为和返回结果。确保接口的命名清晰、一致，并提供适当的文档说明。

3、信息隐藏：
问题：说说管理模块的内部实现细节暴露给其他模块，导致耦合度高。
解决方案：封装说说管理模块的内部实现细节，只暴露必要的接口给其他模块使用。通过使用私有方法、保护方法和公共方法来控制对内部数据和功能的访问。

4、增量式开发：
问题：说说管理模块一次性开发完成，无法及时验证功能和适应变化的需求。
解决方案：采用增量式开发方法，将说说管理功能划分为小的增量，如基本说说发布功能、说说评论功能、说说点赞功能等，并逐步实现、测试和集成。每个增量都能独立运行和演化，使得问题的检测和修复更加容易。

5、抽象：
问题：说说管理模块缺乏合适的抽象层次，导致代码重复和难以扩展。
解决方案：通过抽象将说说管理模块中的共享概念和功能提取出来，形成可复用的组件。例如，可以设计一个通用的说说数据模型，用于表示说说内容、评论和点赞信息。使用设计模式、抽象数据类型等技术来实现高层次的抽象。

6、通用性：
问题：说说管理模块过于专注于当前需求，缺乏通用性。
解决方案：在说说管理模块的设计中考虑到可扩展性和灵活性。可以设计说说管理模块时应考虑支持不同类型的说说内容（文本、图片、视频等）和灵活的权限控制。提供适当的接口和扩展点，以便未来能

六、友链管理模块

1、模块化：
问题：友链管理模块的功能较为复杂，难以理解和修改。
解决方案：将友链管理模块拆分为独立的子模块，如友链添加、友链编辑、友链删除等，每个子模块负责一个明确的任务。通过模块化的设计，使得每个子模块的职责清晰，易于理解和维护。

2、接口设计：
问题：友链管理模块的接口定义不清晰，导致其他模块难以正确使用。
解决方案：明确定义友链管理模块的接口，如添加友链接口、编辑友链接口、删除友链接口等。确保接口的输入和输出明确，定义好接口的预期行为和返回结果。同时，提供详细的接口文档，以便其他模块能够正确地使用这些接口。

3、信息隐藏：
问题：友链管理模块的内部实现细节暴露给其他模块，导致耦合度高。
解决方案：封装友链管理模块的内部实现细节，只暴露必要的接口给其他模块使用。通过使用私有方法、保护方法和公共方法来控制对内部数据和功能的访问，减少模块之间的耦合性，提高代码的可维护性和可扩展性。

4、增量式开发：
问题：友链管理模块一次性开发完成，无法及时验证功能和适应变化的需求。
解决方案：采用增量式开发的方法，将友链管理功能划分为小的增量，如基本友链添加功能、友链分类功能、友链展示功能等，并逐步实现、测试和集成。每个增量都可以独立运行和演化，能够及时验证功能和适应需求的变化。

5、抽象：
问题：友链管理模块缺乏合适的抽象层次，导致代码重复和难以扩展。
解决方案：通过抽象将友链管理模块中的共享概念和功能提取出来，形成可复用的组件。我们可以设计一个友链数据模型，包含友链的名称、链接、描述等信息，并定义友链管理模块的操作方法。通过抽象，减少代码的重复性，提高代码的可维护性和可扩展性。

6、通用性：
问题：友链管理模块过于专注于当前需求，缺乏通用性。
解决方案：在友链管理模块的设计中考虑到可扩展性

七、日志管理模块

1、模块化：
问题：操作日志管理模块功能较为庞大，难以理解和修改。
解决方案：将操作日志管理模块拆分为独立的子模块，如日志记录、日志查询、日志分析等，每个子模块负责一个明确的任务。通过模块化的设计，降低模块的复杂度，使得每个子模块的职责清晰，易于理解和维护。

2、接口设计：
问题：操作日志管理模块的接口定义不清晰，导致其他模块难以正确使用。
解决方案：明确定义操作日志管理模块的接口，如记录日志接口、查询日志接口、分析日志接口等。确保接口的输入和输出明确，定义好接口的预期行为和返回结果。同时，提供详细的接口文档，以便其他模块能够正确地使用这些接口。

3、信息隐藏：
问题：操作日志管理模块的内部实现细节暴露给其他模块，导致耦合度高。
解决方案：封装操作日志管理模块的内部实现细节，只暴露必要的接口给其他模块使用。通过使用私有方法、保护方法和公共方法来控制对内部数据和功能的访问，减少模块之间的耦合性，提高代码的可维护性和可扩展性。

4、增量式开发：
问题：操作日志管理模块一次性开发完成，无法及时验证功能和适应变化的需求。
解决方案：采用增量式开发的方法，将操作日志管理功能划分为小的增量，如基本日志记录功能、高级日志查询功能、日志统计分析功能等，并逐步实现、测试和集成。每个增量都可以独立运行和演化，能够及时验证功能和适应需求的变化。

5、抽象：
问题：操作日志管理模块缺乏合适的抽象层次，导致代码重复和难以扩展。
解决方案：通过抽象将操作日志管理模块中的共享概念和功能提取出来，形成可复用的组件。我们可以设计一个日志数据模型，包含日志的类型、时间、操作内容等信息，并定义日志管理模块的操作方法。通过抽象，减少代码的重复性，提高代码的可维护性和可扩展性。

2.查阅DI相关资料，学习依赖注入技术

依赖注入(Dependency Injection, DI)是一种设计模式，也是Spring框架的核心概念之一。其作用是去除Java类之间的依赖关系，实现松耦合，以便于开发测试。为了更好地理解DI，先了解DI要解决的问题。

即耦合太紧的问题，如下所示：

如果使用一个类，自然的做法是创建一个类的实例：

```
class Player{
    Weapon weapon;

    Player(){
        // 与 Sword类紧密耦合
        this.weapon = new Sword();
    }

    public void attack() {
        weapon.attack();
    }
}
```

这个方法存在耦合太紧的问题，例如，玩家的武器只能是剑Sword，而不能把Sword替换成枪Gun。要把Sword改为Gun，所有涉及到的代码都要修改，当然在代码规模小的时候这根本就不是什么问题，但代码规模很大时，就会费时费力了。

依赖注入是一种消除类之间依赖关系的设计模式。例如，A类要依赖B类，A类不再直接创建B类，而是把这种依赖关系配置在外部xml文件（或java config文件）中，然后由Spring容器根据配置信息创建、管理bean类。

示例：

```
class Player{
    Weapon weapon;

    // weapon 被注入进来
    Player(Weapon weapon){
        this.weapon = weapon;
    }

    public void attack() {weapon.attack();}public void setWeapon(Weapon weapon){
        this.weapon = weapon;
    }
}
```

如上所示，Weapon类的实例并不在代码中创建，而是外部通过构造函数传入，传入类型是父类 `Weapon` ，所以传入的对象类型可以是任何Weapon子类。

传入哪个子类，可以在外部xml文件（或者java config文件）中配置，Spring容器根据配置信息创建所需子类实例，并注入Player类中，如下所示：

```xml
<bean id="player" class="com.qikegu.demo.Player">
    <construct-arg ref="weapon"/>
</bean>

<bean id="weapon" class="com.qikegu.demo.Gun">
</bean>
```

上面代码中 `<construct-arg ref="weapon"/>` ref指向 `id="weapon"` 的bean，传入的武器类型是 `Gun` ，如果想改为 `Sword` ，可以作如下修改：

```xml
<bean id="weapon" class="com.qikegu.demo.Sword">
</bean>
```

只需修改这一处配置就可以。

松耦合，并不是不要耦合。A类依赖B类，A类和B类之间存在紧密耦合，如果把依赖关系变为A类依赖B的父类B0类，在A类与B0类的依赖关系下，A类可使用B0类的任意子类，A类与B0类的子类之间的依赖关系是松耦合的。

可以看到依赖注入的技术基础是多态机制与反射机制。

3.论述利斯科夫替换原则（里氏代换原则）、单一职责原则、开闭原则、德（迪）米特法则、依赖倒转原则、合成复用原则，结合自己的实践项目举例说明如何应用

里氏代换原则

里氏代换原则：子类可以扩展父类的功能，但不能改变父类原有的功能

如何规范地遵从里氏替换原则：

1.子类必须完全实现父类的抽象方法，但不能覆盖父类的非抽象方法

2.子类可以实现自己特有的方法

3.当子类覆盖或实现父类的方法时，方法的前置条件（即方法的形参）要比父类方法的输入参数更宽松。

4.当子类的方法实现父类的抽象方法时，方法的后置条件（即方法的返回值）要比父类更严格。

5.子类的实例可以替代任何父类的实例，但反之不成立

子类继承父类机制的优点：

1.代码共享，减少创建类的工作量

2.提高代码复用性

3.提高父类扩展性

应用举例：

用户类型继承：管理员、普通用户、test用户都继承自基础的用户类型。各种用户类型可以在系统中切换。

单一职责原则

单一职责原则：一个类改变的原因不应该超过一个，一个类应该专注于单一功能

单一职责原则的意义：

1.提高类的可维护性和可读写性。一个类的职责少了，复杂度降低了，代码就少了，可读性也就好了，可维护性自然就高了。

2.提高系统的可维护性。系统是由类组成的，每个类的可维护性高，相对来讲整个系统的可维护性就高。

3.降低变更的风险。一个类的职责越多，变更的可能性就越大，变更带来的风险也就越大。

应用举例：

修改用户名和密码的场合：将修改用户名和修改密码逻辑分开，而不是将两个操作集中在一个“修改”类里，这样就避免了可能由于传错“operate_type”产生的错误。

开闭原则

开闭原则：一个软件实体应该对扩展开放，对修改关闭

软件如何遵循开闭原则：

1.对于扩展开放：模块的行为可以扩展，当应用的需求改变时，可以对模块进行扩展，以满足新的需求。

2.对于修改封闭：对模块行为扩展时，不必改动模块的源代码或二进制代码。

开闭原则的意义:

1.提高软件的可扩展性和可维护性。

2.事实上，大部分的设计模式和设计原则都是在实现开闭原则。

应用举例：

通过添加插件来扩展功能。

通过数据库访问层封装数据库操作，而不是在业务逻辑代码中进行数据库操作。

德（迪）米特法则

狭义德米特法则：如果两个类不必彼此直接通信，那么这两个类就不应当发生直接的相关作用。如果其中的一个类需要调用另外一个类的某种方法的话，可以通过第三者转发这个调用。

德米特法则通过减少对象之间的直接关联，降低耦合性，提高系统的可维护性和灵活性。遵循这个原则可以减少对象之间的依赖关系，使系统更易于扩展和修改，同时也使得对象的使用更加简洁和清晰。

但是这种法则有缺点：

1、会在系统中造出大量的小方法，散落在系统的各个角落。这些方法仅仅是传递间接的调用，因此与系统的逻辑无关。

2、会造成系统的不同模块之间通信效率的降低，也会使系统的不同模块之间不容易协调。

比如下面表示的关系：一个人想和陌生人交互，必须先和朋友传递信息，再由朋友去和陌生人传递信息，造成信息冗余。

```
public class Person {//人想和陌生人传递
        private String name = "张三";

        public void talking(Friend friend) {
                System.out.println("你好，我是" + name);
                friend.introduce(name);
        }
}
```

```
public class Friend {//朋友接受并传达
        private Stranger stranger;

        public void introduce(String name) {
                stranger.talking(name);
        }

        public Stranger getStranger() {
                return stranger;
        }

        public void setStranger(Stranger stranger) {
                this.stranger = stranger;
        }
}
```

```java
public class Stranger {//陌生人相应
        private String name;

        public Stranger(String name) {
                this.name = name;
        }

        public void talking(String name) {
                System.out.println("你好 ," + name + ",我是" + this.name);
        }
}
```

所以延伸为广义迪米特法则：

迪米特法则讨论的是对象之间的信息流量，流向及信息的影响控制。它的主要用意是控制信息的过载。在运用的过程中，要注意以下几点：

1、在类的划分上，应当创建有弱耦合的类。类之间的耦合越弱，越有利于复用。一个处于弱耦合中德类，一旦被修改，不会对有关系的类造成波及。

2、在类的结构设计中，每一个类都应当尽量降低成员的访问权限。

3、在类的设计中，只要有可能，一个类应当设计成不变类。

4、在对其他类的引用上，一个对象对其对象的引用应当降到最低。

应用：个人博客网站通常包含多个模块或组件，如用户管理、文章管理、评论管理等。在设计这些模块时，应该尽量避免直接依赖其他模块的内部实现细节，而是通过接口或者中间层进行通信。每个模块只需了解其直接相关的模块，而不需要了解其他无关模块的内部结构。而在设计博客网站的各个模块时，应该将模块的内部状态和行为尽可能封装起来，通过接口提供必要的操作方法，而不是直接暴露内部实现细节给其他模块。

依赖倒转原则

1、高层模块不应该依赖低层模块，二者都应该依赖其抽象(抽象类/接口)，不要去依赖一个具体的子类

2、抽象不应该依赖细节，细节应该依赖抽象（这样稳定性会比较好）

3、依赖倒转(倒置)的中心思想是面向接口编程

4、依赖倒转原则是基于这样的设计理念：相对于细节的多变性，抽象的东西要稳定的多。以抽象为基础搭建的架构比以细节为基础的架构要稳定的多。在 java 中，抽象指的是接口或抽象类，细节就是具体的实现类

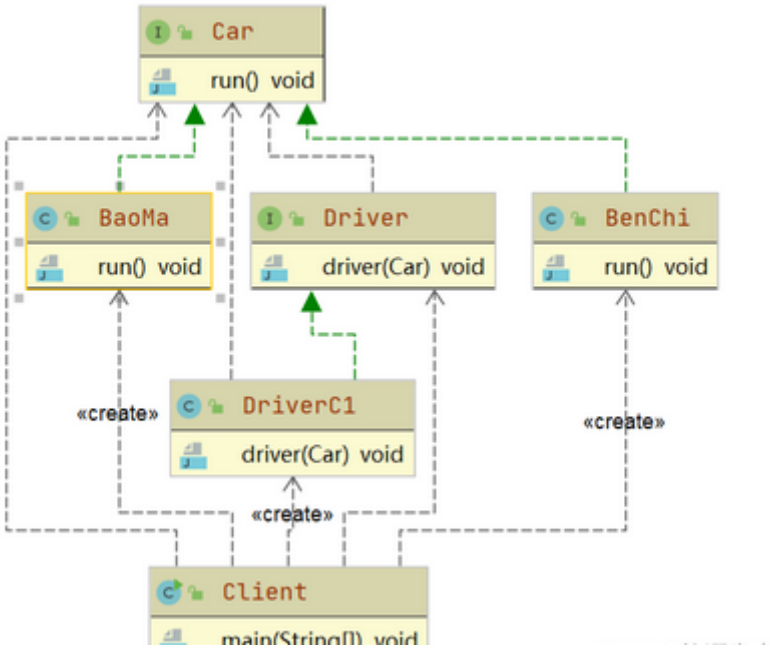
5、使用接口或抽象类的目的是制定好规范，而不涉及任何具体的操作，把展现细节的任务交给他们的实现类去完成（接口和抽象类的价值在于设计）

使用该原则也有需要注意的地方：

1、低层模块尽量都要有抽象类或接口，或者两者都有，程序稳定性更好.

2、变量的声明类型尽量是抽象类或接口，这样变量引用和实际对象间，就存在一个缓冲层，利于程序扩展和优化。比如：class A extends B{}，其中B是一个抽象类/接口，在使用时：B obj = new A()，如果A类要进行扩展，只需要在B中增加一个方法即可。

比如下图的模拟开车，把司机和汽车都抽象，使其耦合性降低，便于扩展更多的车。驾驶人员Driver依赖汽车Car类，变成了接口依赖（面向接口编程），方便扩展




```
1  public interface Car {
2      //汽车奔跑
3      public void run();
4  }
5  public interface Driver {
6      //司机开车
7      public void driver(Car car);
8  }
9  public class BenChi implements Car{
10     public void run(){
11         System.out.println("奔驰车跑");
12     }
13 }
14 public class BaoMa implements Car{
15     public void run(){
16         System.out.println("宝马车跑");
17     }
18 }
19 //c1驾驶人员
20 public class DriverC1 implements Driver{
21     @Override
22     public void driver(Car car) {
23         car.run();
```

```
    }
}
public class Client {
    public static void main(String[] args) {
        Driver driver = new DriverC1();
        Car benChi = new BenChi();
        Car baoMa = new BaoMa();
        driver.driver(benChi);
        driver.driver(baoMa);
    }
}
```

依赖倒置原则的优点在小型项目中很难体现出来，在一个大中型项目中，采用依赖倒置原则有非常多的优点，特别是规避一些非技术因素引起的问题。项目越大，需求变化的概率也越大，通过采用依赖倒置原则设计的接口或抽象类对实现类进行约束，可以减少需求变化引起的工作量剧增的情况。人员的变动在大中型项目中也是时常存在的，如果设计优良、代码结构清晰，人员变化对项目的影响基本为零。大中型项目的维护周期一般都很长，采用依赖倒置原则可以让维护人员轻松地扩展和维护。

应用：定义抽象的接口或抽象类来表示高层模块对低层模块的依赖关系。例如，定义一个Logger接口来封装日志记录的功能。高层模块不应直接依赖于低层模块，而是依赖于抽象接口。这意味着，高层模块不关心具体的实现细节，只需要与抽象接口进行交互。低层模块则通过实现抽象接口来提供具体的功能。例如，日志记录模块实现Logger接口。

合成复用原则

尽量使用对象组合，而不是继承来达到复用的目的。

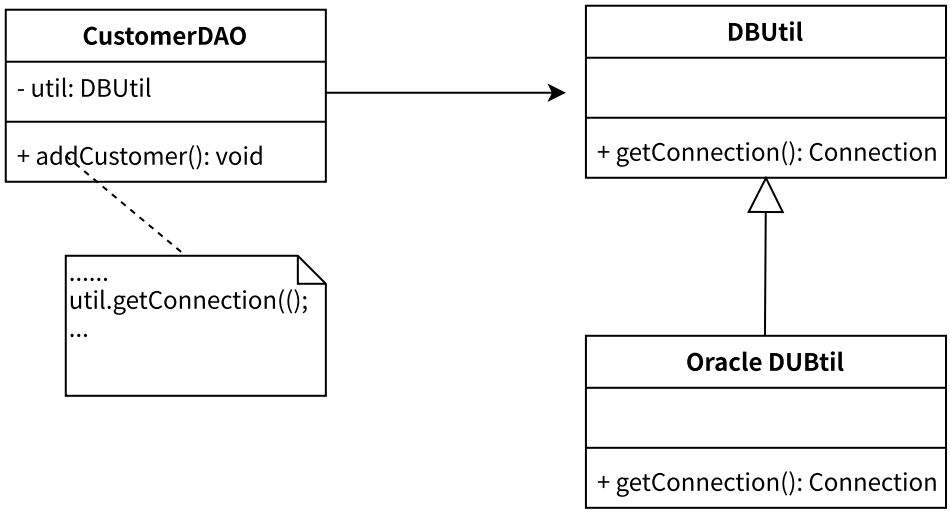
合成复用原则就是在一个新的对象里通过关联关系（包括组合关系和聚合关系）来使用一些已有的对象，使之成为新对象的一部分；新对象通过委派调用已有对象的方法达到复用功能的目的。简言之：复用时要尽量使用组合/聚合关系（关联关系），少用继承。

在面向对象设计中，可以通过两种方法在不同的环境中复用已有的设计和实现，即通过组合/聚合关系或通过继承，但首先应该考虑使用组合/聚合，组合/聚合可以使系统更加灵活，降低类与类之间的耦合度，一个类的变化对其他类造成的影响相对较少；其次才考虑继承，在使用继承时，需要严格遵循里氏代换原则，有效使用继承会有助于对问题的理解，降低复杂度，而滥用继承反而会增加系统构建和维护的难度以及系统的复杂度，因此需要慎重使用继承复用。

通过继承来进行复用的主要问题在于继承复用会破坏系统的封装性，因为继承会将基类的实现细节暴露给子类，由于基类的内部细节通常对子类来说是可见的，所以这种复用又称“白箱”复用，如果基类发生改变，那么子类的实现也不得不发生改变；从基类继承而来的实现是静态的，不可能在运行时发生改变，没有足够的灵活性；而且继承只能在有限的环境中使用（如类没有声明为不能被继承）。

在下图中，CustomerDAO和DBUtil之间的关系为关联关系，采用依赖注入的方式将DBUtil对象注入到CustomerDAO中，可以使用构造注入，也可以使用Setter注入。如果需要对DBUtil的功能进行扩展，可以通过其子类来实现，如通过子类OracleDBUtil来连接Oracle数据库。由于CustomerDAO针对

DBUtil编程，根据里氏代换原则，DBUtil子类的对象可以覆盖DBUtil对象，只需在CustomerDAO中注入子类对象即可使用子类所扩展的方法。例如在CustomerDAO中注入OracleDBUtil对象，即可实现Oracle数据库连接，原有代码无须进行修改，而且还可以很灵活地增加新的数据库连接方式




```
1
2  #include<iostream>
3  #include<string>
4  using namespace std;
5
6  class AbstaractCat
7  {
8  public:
9          virtual void run() = 0;
10 };
11
12 class Dazhong :public AbstaractCat
13 {
14 public:
15         virtual void run()
16         {
17                 cout << "大众车启动..." << endl;
18         }
19 };
20
21 class Tuolaji :public AbstaractCat
22 {
23 public:
24         virtual void run()
25         {
26                 cout << "拖拉机启动..." << endl;
27         }
28 };
```

```
29 class Person
30 {
31 public:
32         void setCar(AbstaractCat* car)
33         {
34                 this->car = car;
35         }
36
37         void Doufeng()
38         {
39                 this->car->run();
40                 if (this->car != NULL)
41                         delete this->car;
42                 this->car = NULL;
43
44         }
45
46 public:
47         AbstaractCat* car;
48 };
49
50 void test01()
51 {
52         Person *p = new Person;
53         p->setCar(new Dazhong);
54         p->Doufeng();
55         p->setCar(new Tuolaji);
56         p->Doufeng();
57
58         delete p;
59 }
60 int main()
61 {
62         test01();
63         system("pause");
64 }
```

应用：在设计组件时，应明确定义组件的接口和抽象类，以提供一致的操作方式和行为。通过使用接口和抽象类，可以使组件更易于复用和替换，而不依赖于具体实现。将个人博客网站拆分为独立的模块和组件，每个模块或组件负责特定的功能。每个组件应具有单一职责，高内聚性，并且可以独立地进行开发、测试和部署。在设计组件时，尽可能将变化封装起来，以减少对其他组件的影响。