实验十三 设计模块（二）

实验目的：

面向对象设计原则

实验内容：

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 （保存到每个小组选定的协作开发平台上，以组为单位）。

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。

**利斯科夫替换原则（Liskov Substitution Principle, LSP）**

**定义**： 子类对象应该能够替换其基类对象，被程序的行为识别和使用而不引入任何错误。换言之，子类应保持基类的行为。

**特点**：

子类型必须能够替换父类型：这意味着在使用父类型的地方，可以用子类型来替换，而不会产生意料之外的行为。例如，如果有一个函数接受基类对象作为参数，那么它也应该能够接受子类对象作为参数而不产生错误或异常的情况。

行为保持一致性：当子类型替换父类型时，程序的行为应该保持一致。换句话说，对于同样的操作，子类型和父类型应该产生相似的结果，而且不会改变程序的预期行为。

避免破坏原有功能：子类型在替换父类型时不应该破坏原有的功能和约束条件。如果子类型需要修改父类型的行为，那么这可能意味着设计存在问题，需要重新考虑继承关系或接口定义。

提高系统的灵活性和可扩展性：遵循利斯科夫替换原则可以使代码更灵活、更易于扩展和维护。因为子类型可以无缝地替换父类型，所以系统的各个部分可以更容易地被替换或更新，而不会影响整体的稳定性。

**应用**： 假设系统中有各种类型的用户，如Admin、Author和Reader，它们都继承自一个共同的基类User。如果在系统中有一处功能是需要发送消息通知的，那么无论是哪一种类型的用户，都应该通过相同的方式来发送通知，而调用方不必因为用户的具体类型而改变其行为。

**单一职责原则（Single Responsibility Principle, SRP）**

**定义：** 一个类应该只有一个引起改变的原因，即一个类应该只有一个任务或责任。

**特点：**

责任的定义：单一职责原则指出一个类或模块应该有且仅有一个改变的原因。换句话说，一个类或模块应该专注于完成一项特定的功能或任务，而不涉及多种不同的责任。这有助于降低类的复杂度，提高可维护性和可理解性。

类的职责划分：按照单一职责原则，一个类的职责应该清晰明确，不要将多个不相关的功能混合在一起。如果一个类承担了过多的责任，那么它会变得难以理解和修改，同时也增加了对其他类的依赖性。

高内聚性：遵循单一职责原则有助于提高类的内聚性。高内聚的类更加专注于完成特定的功能，代码更加清晰，并且模块之间的耦合性更低，从而提高了代码的灵活性和可维护性。

改变的影响范围：如果一个类具有多个职责，当其中一个职责发生变化时，可能会影响到其他职责，导致代码的不稳定和不可预测性。因此，遵循单一职责原则可以降低代码的脆弱性，使得变更更加可控。

**应用**： 在博客系统中，处理文章（Article）的类应只负责与文章相关的操作，如创建、编辑、删除文章等，而不应负责用户权限的验证。验证用户权限（例如，判断用户是否有编辑文章的权限）应由另一个类（如AuthorizationService）负责处理，以遵守单一职责原则。

**开闭原则（Open/Closed Principle, OCP）**

**定义**： 软件实体（如类、模块、函数等）应该对扩展开放，对修改关闭。意味着已有的代码尽可能少地被修改，同时能轻松添加新功能。

**特点**：

对扩展开放：开闭原则强调当需要添加新功能或修改现有功能时，不应该修改已有的代码，而是通过扩展现有代码来实现。这意味着软件实体应该具有扩展性，能够接受新的功能或行为，而不需要改动其内部实现。

对修改关闭：相对于修改已有的代码，开闭原则鼓励通过添加新的代码来实现新的功能或变化。这样做可以避免破坏已有的稳定性和可靠性，同时也降低了引入错误的风险。

抽象和多态：使用抽象和多态是实现开闭原则的关键手段。通过定义抽象的接口或基类，以及利用多态特性，可以在不修改现有代码的情况下引入新的实现，并且让客户端代码能够透明地使用这些新的实现。

设计模式和框架：许多设计模式和框架都是遵循开闭原则的，它们通过提供灵活的扩展点和接口，使得系统能够轻松地进行功能扩展而不需要修改现有的代码。

**应用**： 设计插件系统来允许在不修改现有代码的情况下新增功能，如文章的评论系统。可以设计一个PluginInterface接口，所有插件都实现这个接口。当需要添加新功能时，只需添加一个新的插件类实现该接口，而无需修改现有的代码。

**德米特法则（Law of Demeter, LoD）**

**定义：** 一个对象应该对其他对象有最少的了解。简言之，“不要和陌生人说话”，避免强耦合。

**特点：**

降低耦合度：德米特法则旨在降低类之间的耦合度，使得各个类之间的关联尽可能简单和直接。这样做有利于减少类之间的依赖关系，提高系统的灵活性和可维护性。

封装信息：根据德米特法则，一个类应该尽可能少地了解其它类的内部细节，而应该通过接口或者抽象来进行通信。这种封装信息的方式有助于隔离变化，让类的设计更加稳定和可靠。

限制依赖关系：德米特法则也强调了在设计系统时限制类之间的直接依赖关系，尽量减少类之间的交互。这样可以避免类之间的过度耦合，减少系统中的潜在风险和复杂性。

接口约束：遵循德米特法则可以鼓励设计良好的接口和抽象，使得类与类之间的交互更加清晰和简洁。良好定义的接口可以帮助限制类之间的交互，降低系统的复杂度。

**应用**： 设计文章发布的流程时，ArticlePublisher类可能需要使用Article类的一些功能，但不应直接访问其内部的复杂逻辑。例如，若Article类有个updateStatus方法用于更新文章的状态，ArticlePublisher应该调用这个方法，而不是直接修改文章状态的内部字段。

**依赖倒转原则（Dependency Inversion Principle, DIP）**

**定义**： 高层模块不应该依赖低层模块，它们都应该依赖于抽象。抽象不应该依赖于细节，细节应该依赖于抽象。

**特点：**

解耦合：依赖倒置原则旨在降低模块之间的耦合度，通过引入抽象层来解除高层模块对于底层模块的直接依赖。这样可以降低系统的复杂性，提高模块的可复用性和可扩展性。

抽象层：依赖倒置原则鼓励引入抽象层，让高层模块和低层模块都依赖于抽象，而不是具体的实现细节。这种做法有助于隔离变化，让系统更加稳定和灵活。

接口定义：遵循依赖倒置原则需要良好定义的接口或抽象类，这些接口可以作为高层模块和底层模块之间的桥梁，让它们能够进行交互而不需要了解彼此的具体实现。

反转控制：依赖倒置原则也促进了反转控制（Inversion of Control，IoC）的实践，通过将控制权交给框架或容器来管理对象之间的依赖关系，进一步降低了模块之间的耦合度。

**应用**： 对于文章存储，系统可能需要支持多种存储方式（如数据库、文件系统）。在这种情况下，文章存储的接口（ArticleStorage）应该被定义为高层模块，而具体存储实现如DatabaseArticleStorage或FileArticleStorage则实现这个接口。系统中其他部分依赖于ArticleStorage接口，而不是具体的实现类，实现了依赖倒置。

**合成复用原则（Composite Reuse Principle, CRP）**

**定义：** 在一个新的对象中用一些已有的对象来实现新功能时，尽量使用合成/聚合（即对象包含其他对象），而非继承。

**特点**：

优先使用对象组合而不是继承:

组合可以带来更大的灵活性和可重用性,因为我们可以在运行时动态地改变对象的组成。

继承过于僵化,不容易进行扩展和修改。

尽量减少类的继承层次:

过深的继承层次会导致类之间的耦合度增加,继承关系变得难以理解和维护。

应该尽量通过组合来实现功能复用,而不是过度使用继承。

合成/聚合要优于继承:

组合和聚合是更加松耦合的方式,可以在运行时动态地改变对象的组成。

继承则是一种强耦合的关系,子类会被父类的实现细节所束缚。

**应用**： 在处理用户的权限时，系统可能有一组不同的权限检查逻辑。通过合成（composition）而不是继承来重用这些逻辑，可以创建一个PermissionCheck类，其中包含了权限检查的逻辑，然后在需要执行权限检查的类中实例化并使用PermissionCheck类，从而避免了通过继承引入过度的依赖性和复杂性。