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）

里氏替换原则（Liskov Substitution Principle,简称LSP）

- **核心思想**：子类必须能够替换它们的基类（IS-A）的关系。这意味着在软件系统中，如果一个软件实体使用的是基类的话，那么一定也适用于其子类，而且程序逻辑不会改变。
- **问题描述**： 子类必须保持父类的接口规范。子类对象能够替换父类对象，而程序逻辑不变。子类可以扩展父类的功能，但不能改变父类原有的功能。
- **好处**：

  约束继承泛滥：确保子类在继承父类时不会随意改变父类的行为，保持了系统的稳定性和一致性。
  开闭原则的一种体现：通过子类扩展功能，而不是修改父类，满足了开闭原则（对扩展开放，对修改关闭）。
  加强程序的健壮性：当需求变更时，通过子类替换父类，可以减少对原有系统的影响，提高系统的可维护性和扩展性。
  降低风险：降低了需求变更时引入的风险，因为子类替换父类通常不会导致程序逻辑的改变。
- **注意事项**：

  方法的前置条件：子类方法的前置条件（即方法的输入、入参）要比父类方法的输入参数更宽松。
  方法的后置条件：子类实现父类的方法时，方法的后置条件（即方法的输出、返回值）要比父类更严格或相等。
  避免破坏原有功能：子类在扩展功能时，要确保不破坏父类的原有功能。

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

- **核心思想**： 单一职责原则的核心思想是：一个类应该仅有一个引起它变化的原因。这意味着一个类应该只有一个职责，对外只提供一种功能。
- **问题描述**： 一个类承担过多的职责会导致类的复杂性增加，不利于维护。当类的某个职责发生变化时，可能会影响到其他职责，导致程序的“脆弱”和“僵硬”。
- **好处**：

  降低类的复杂度：每个类只负责一个职责，逻辑更加清晰、简单。
  提高类的可读性：由于类的职责单一，代码可读性更高。
  提高系统的可维护性：类之间的耦合度较低，更容易进行修改和维护。
  提高类的可复用性：粒度较低的类可复用性更高。
- **注意事项**：

  避免过度拆分：虽然单一职责原则鼓励将类的职责拆分开来，但也要避免过度拆分，导致类的数量过多，反而增加了系统的复杂性。
  合理定义职责：在定义类的职责时，要根据实际需求进行合理划分，确保每个类都承担合适的职责。
  遵循其他设计原则：单一职责原则并不是孤立的，它与其他设计原则（如开闭原则、依赖倒置原则等）是相互补充的，要综合考虑并遵循这些原则来设计系统。

开放封闭原则（Open Close Principle,简称OCP）

- **核心思想：** 开闭原则的核心思想是软件实体（类、模块、函数等）应该对扩展开放，对修改关闭。也就是说，在添加新功能时，不应该修改现有的代码，而是通过扩展来实现。
- **问题描述：** 如果一个系统在每次添加新功能时都需要修改现有代码，这将导致代码频繁变动，容易引入新的错误，增加维护难度，并且破坏了系统的稳定性。
- **好处：**

  提高系统的稳定性：现有代码一旦稳定，尽量不去修改，从而减少引入新错误的风险。

  增强系统的可扩展性：通过扩展来添加新功能，可以更灵活地应对需求变化。

  提高代码复用性：通过抽象和接口设计，增强了代码的可复用性和灵活性。
- **注意事项：**

  设计抽象：合理设计抽象类和接口，使得新功能可以通过扩展实现，而不是修改原有类。

  避免过度设计：在没有明确需求的情况下，不要过度抽象化，否则会增加系统复杂性。

  遵循单一职责原则：确保每个类或模块只负责一个功能，以便于扩展而不影响其他部分。

迪米特法则（Law of Demeter,简称LoD）

- **核心思想：** 德米特法则的核心思想是一个对象应当对其他对象有尽量少的了解，只与直接的朋友（直接关联的对象）通信，避免过多的耦合。
- **问题描述：**当一个对象直接调用多个其他对象的属性或方法（尤其是链式调用时），会导致高度耦合，使得系统的维护变得复杂，修改一个对象时可能会影响到许多其他对象。
- **好处：**

  降低耦合度：减少对象之间的依赖，提高系统的模块化程度。

  增强可维护性：修改一个对象时，对其他对象的影响较小，便于系统维护。

  提高可读性：代码更容易理解，因为每个对象只处理自身直接相关的部分。
- **注意事项：**

  合理设计接口：确保对象只暴露必要的方法，避免外部对象依赖其内部实现细节。

  避免链式调用：减少对象之间的链式调用，增加中介者或封装调用逻辑。

  关注性能：虽然封装和减少依赖提高了可维护性，但有时会引入额外的调用开销，需在性能和设计之间权衡。

依赖倒转原则（Dependence Inversion Principle,简称DIP）

- **核心思想：** 依赖倒转原则的核心思想是高层模块不应该依赖低层模块，它们都应该依赖其抽象。换句话说，抽象不应该依赖细节，细节应该依赖抽象。这一原则强调的是面向接口编程，通过抽象来减少模块间的依赖，提高系统的灵活性和可维护性。
- **问题描述：**在没有应用依赖倒转原则的情况下，高层模块直接依赖于低层模块的具体实现，这会导致以下问题：
  紧耦合：高层模块和低层模块紧密绑定，一旦低层模块发生变化，高层模块也需要相应地进行修改。

  难以扩展：当需要添加新的功能或模块时，可能会影响到现有的高层模块和低层模块。

  测试困难：由于高层模块直接依赖于低层模块的具体实现，因此在测试高层模块时，需要同时测试低层模块，增加了测试的复杂性和难度。
- **解决方案：** 将类A修改为依赖接口interface，类B和类C各自实现接口interface，类A通过接口interface间接与类B或者类C发生联系，则会大大降低修改类A的几率。

- **好处：** 降低耦合度：通过面向接口编程，减少了高层模块和低层模块之间的直接依赖，降低了模块之间的耦合度。
  提高灵活性：由于高层模块依赖于抽象接口，因此可以更容易地替换低层模块的具体实现，而不需要修改高层模块的代码。
  提高可维护性：由于降低了模块之间的耦合度，使得系统更容易进行维护和扩展。
- **注意事项：**明确抽象接口：在定义抽象接口时，需要确保接口的定义足够清晰和完整，以便能够满足高层模块的需求。
  避免过度抽象：过度抽象可能会导致接口过于复杂和难以理解，因此需要权衡抽象的程度和实际需求。
  确保接口的稳定性：由于高层模块依赖于抽象接口，因此需要确保接口的稳定性和一致性，以避免对高层模块产生不必要的影响。

合成复用原则（Composite/Aggregate Reuse Principle，简称CARP）

- **核心思想：** 合成复用原则的核心思想是尽量使用对象之间的弱关联关系（组合或聚合关系）来达到代码复用的目的，而不是通过继承关系。这一原则强调的是通过组合和聚合来构建复杂的对象，而不是通过继承来扩展对象的功能。
- **问题描述：** 在过度使用继承关系的情况下，可能会导致以下问题：
  高度耦合：子类与父类之间紧密绑定，一旦父类发生变化，子类也需要相应地进行修改。
  继承层次复杂：过深的继承层次会增加系统的复杂性，降低代码的可读性和可维护性。
  功能复用性差：继承关系中的子类只能继承父类的功能，而不能选择性地复用父类的部分功能。
- **好处：**

  降低耦合度：通过组合和聚合关系，可以将功能模块拆分成独立的对象，减少了类之间的直接依赖关系，提高了系统的灵活性。

  提高代码复用性：通过将公共的功能封装成独立的对象，可以在不同的场景中重复利用这些对象，提高了代码的复用性和可读性。

  支持动态组合和配置：由于组合和聚合关系的灵活性，可以在运行时通过动态组合和配置对象来实现不同的功能组合，满足不同的需求。
- **注意事项：**

  合理设计对象关系：在设计对象之间的关系时，需要根据实际需求选择合适的组合或聚合关系，避免过度使用或错误使用这些关系。
  明确对象职责：在构建复杂对象时，需要明确每个对象的职责和边界，避免对象之间出现不必要的依赖和耦合。
  注意性能问题：虽然组合和聚合关系可以提高系统的灵活性和可维护性，但在某些情况下可能会导致性能下降。因此，在设计系统时需要权衡性能和灵活性之间的关系。

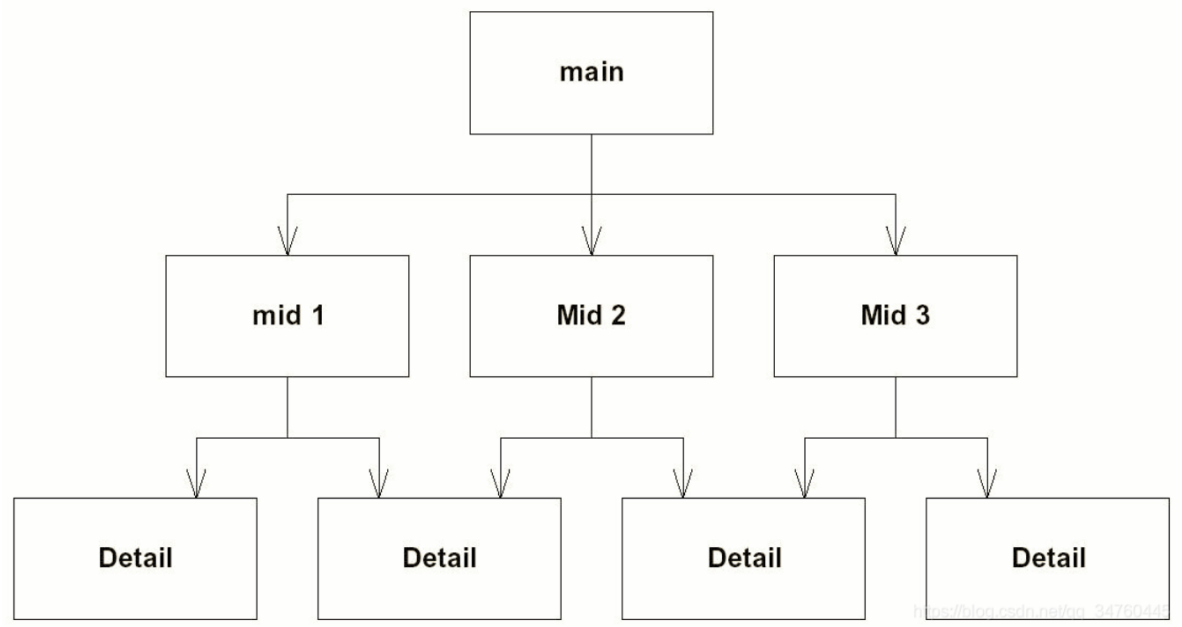
项目上的应用

对于依赖倒置原则，我们做了深入探究，以下是一些概念理解：

依赖： 在程序设计中，如果一个模块a使用/调用了另一个模块b，我们称模块a依赖模块b。

高层模块与低层模块： 往往在一个应用程序中，我们有一些低层次的类，这些类实现了一些基本的或初级的操作，我们称之为低层模块；另外有一些高层次的类，这些类封装了某些复杂的逻辑，并且依赖于低层次的类，这些类我们称之为高层模块。

传统的结构化程序设计中，高层模块总是依赖于低层模块，如下图所示。

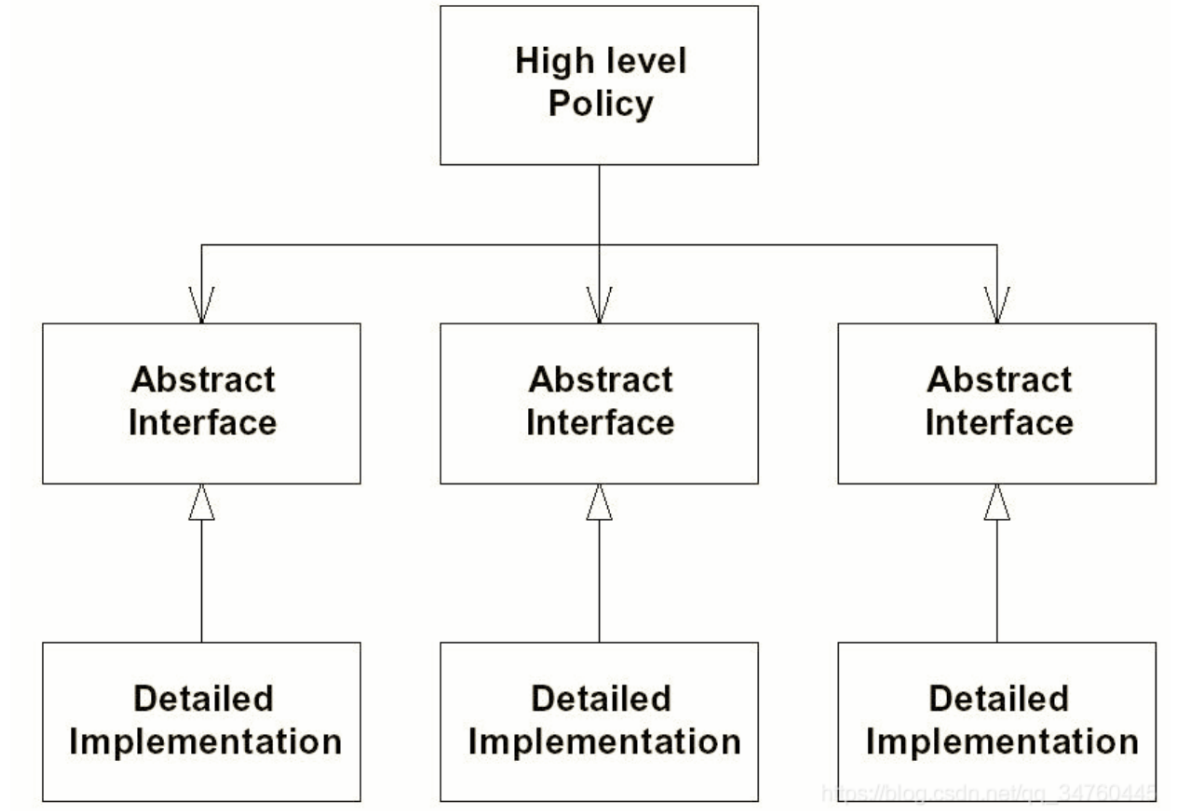


这样的系统“高层模块”过分依赖“低层模块”，具体体现在以下方面：

- 系统很难改变，因为每个改变都会影响其他很多部分。
- 当你对某地方做一修改，系统的看似无关的其他部分都不工作了。
- 系统很难被另外一个应用重用，因为很难将要重用的部分从系统中分离开来。

而一个良好的设计应该是系统的每一部分都是可替换的。如果“高层模块”过分依赖“低层模块”，一方面一旦“低层模块”需要替换或者修改，“高层模块”将受到影响；另一方面，高层模块很难可以重用。

因此，我们应用**依赖倒置原则（Dependency Inversion Principle ， DIP） ，在高层模块与低层模块之间，引入一个抽象接口层 。**



即High Level Classes（高层模块） --> Abstraction Layer（抽象接口层） --> Low Level Classes（低层模块）

抽象接口是对低层模块的抽象，低层模块继承或实现该抽象接口。

这样一来，高层模块不直接依赖低层模块，而是依赖抽象接口层。抽象接口也不依赖低层模块的实现细节，而是低层模块依赖（继承或实现）抽象接口，**类与类之间都通过抽象接口层来建立关系** 。用面向对象技术将两个单元之间的依赖关系颠倒，接触循环。两种解决方法使用后都应形成树形结构，每棵子树都是系统的一部分，可以一次一个软件单元地增量开发系统，依次实现该项目的几个模块。