依赖倒置原则（Dependency-Inversion Principle）是Robert C. Martin在1996年为《C++ Reporter》所写的专栏Engineering Notebook的第三篇，后来加入到他在2002年出版的经典著作《Agile Software  
Development Principles Patterns and Practices》 中提到的，它由两条构成：

***A。High level modules should  
not depend upon low level modules. Both should depend upon abstractions.  
高层模块不应该依赖于低层模块。它  
们都应该依赖于抽象。***

***B。Abstractions should not  
depend upon details. Details should depend upon abstractions.  
抽象不应该依赖于具体。具体应该依赖于抽象。***

*（读者按：简洁而美，像是E=mc2）*

这里的高层模块和低层模块的概念可以参照标准的MVC分层架构。低层模块包含数据持久化等低层次的功能，在整个系统中处于支撑的作用；高层模块包含通常包含重要的业务逻辑，是系统核心功能所在。如果高层模块依赖于低层模块，那么，当我们需要修改低层模块的时候，所有层次在它之上的模块都需要进行修改。而且，由于对低层模块的依赖，高层模块无法在没有低层模块的上下文环境中独立运行。高层模块无法实现复用，而恰恰是这些高层模块才是系统的核心，是最具复用价值的。

在Martin的文章中，举了Copy程序的例子，我使用类重新构造了一下。这个例子很简单，实现的功能就是从键盘读取字符，然后输出到打印机中。这个功能由Copy类  
完成。一个违反DIP原则的设计如下：

“DoCopy”方法的代码可能是这个样子的：

public  
class Copy

{

   public int DoCopy(Keyboard keyboard, Printer printer)

   {

      char c;

      while(c=(keyboard.Read())!=EOF)

      {

         printer.Print(c);

      }

   }

}

Copy依赖于Keyboard和Printer两个类。如果想在没有Keyboard和Printer的上下文环境中复用Copy，比如实现键盘读取字符写入一个文件，则是无法办到的。

一个更合理的的设计是这样的：把Copy对Keyboard和Printer的这种关系中解放出来，Copy类既不依赖于Keyboard也不依赖于Printer，而只依赖于抽象类Reader和Writer。因此，依赖性已经被倒置；Copy类依赖于抽象，而具体的读取器和写出器也依赖相同的抽象。这就是一个符合DIP的设计。以后如果需要复用Copy的功能，比如实现从一个文件读取，写入打印机，完全不需要修改Copy，只需要增加一个继承抽象类Reader的具体实现FileReader就可以了——这又符合了开闭原则的要求！

那么，为什么要称之为依赖倒置呢？实际上，高层模块和低层模块共同依赖的这个抽象层，基本上是由高层模块决定的。也就是说，抽象有那些功能，是由系统的功能和业务逻辑决定的，所以从某种程度上来说，**虽然高层模块的具体功能要藉由低层实现（调用低层），但要实现哪些由高层说了算！**——这有点儿Hollywood的味道了：Don't call us, we'll call you!