# 单一职责原则-SRP

就一个类而言，应该仅有一个引起它变化的原因。

为何要把两个职责分离到单独的类？因为每一个职责都是变化的一个轴线。当需求变化时，该变化会反映为类的职责的变化。如果一个类承担了多于一个的职责，那么引起变化的原因就会有多个。如果一个类承担的职责过多，就等于把这些职责耦合在了一起。一个职责的变化可能会削弱或抑制这个类完成其他职责的能力。

# 开放-封闭原则-OCP

软件实体（类、模块、函数等等）应该是可以拓展的，但是不可修改的。OCP建议我们对系统进行重构。

策略模式和模板方法设计模式是满足 OCP最常用的方法。应用它们，可以把一个功能的通用部分和实现细节部分清晰的分离开来。OCP背后的主要机制是抽象和多态。

# 里氏替换原则-LSP

子类型必须能够替换掉它们的基类型。

里氏替换原则至少包含以下两种含义：

1、里氏替换原则是针对继承而言的，如果继承是为了实现代码重用，也就是为了共享方法，那么共享的父类方法就应该保持不变，不能被子类重新定义。子类只能通过新添加方法来扩展功能，父类和子类都可以实例化，而子类继承的方法和父类是一样的，父类调用方法的地方，子类也可以调用同一个继承得来的，逻辑和父类一致的方法，这时用子类对象将父类对象替换掉时，当然逻辑一致，相安无事。

2、如果继承的目的是为了多态，而多态的前提就是子类覆盖并重新定义父类的方法，为了符合LSP，我们应该将父类定义为抽象类，并定义抽象方法，让子类重新定义这些方法，当父类是抽象类时，父类就是不能实例化，所以也不存在可实例化的父类对象在程序里。也就不存在子类替换父类实例（根本不存在父类实例了）时逻辑不一致的可能。

3、不符合LSP的最常见的情况是，父类和子类都是可实例化的非抽象类，且父类的方法被子类重新定义，这一类的实现继承会造成父类和子类间的强耦合，也就是实际上并不相关的属性和方法牵强附会在一起，不利于程序扩展和维护。

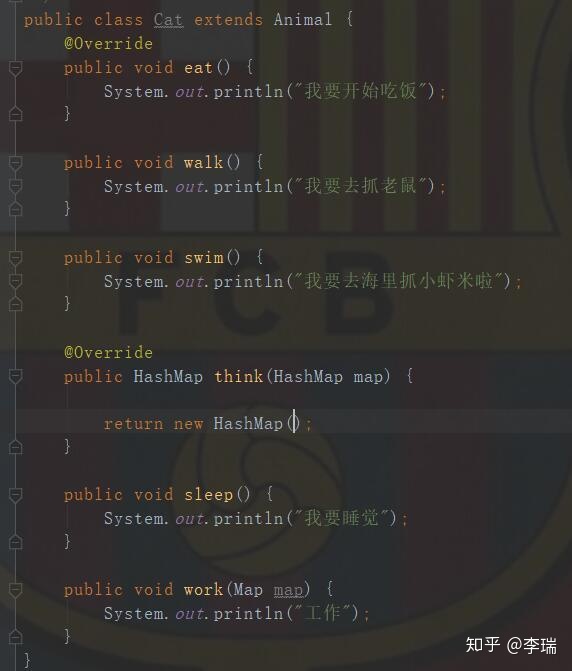
4、当子类的方法重载父类的方法时候，方法的形参要比父类的方法的输入参数更加宽松。

5、当子类的方法实现父类的抽象方法时，方法的返回值要比父类更严格。

使用里氏替换原则原则的目的：

1、增强程序的稳健性

2、使用父类作为参数，传递不同子类完成业务逻辑，具有很高的拓展性。



示例如上：

（1）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。

Cat类的work方法形参要比父类更宽松，所以当参数输入为HashMap类型时，只会执行父类的方法，不会执行Cat中的work方法，这是符合里氏替换原则的。

（2）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。

当Cat重写其父类的think方法时，其返回值为HashMap,而父类返回子类型是Map,子类比父类的返回值范围更小，更严格。如果子类返回值类型比父类还要大，在子类重写该方法时编译器就会报错。

# 依赖倒置原则-DIP

1、高层模块不应该依赖底层模块。二者都应该依赖于抽象。

2、抽象不应该依赖于细节。细节应该依赖于抽象。

高层模块包含了一个应用程序中的重要的策略选择和业务模型。正是这些高层模块才使得其所在的应用程序区别于其他。然而，如果这些高层模块依赖于底层模块，那么底层模块的改动就会直接影响到高层模块，从而迫使他们依次做出改动。

在Java 中抽象指的是接口或者抽象类，两者皆不能实例化。而细节就是实现类，也就是实现了接口或者继承了抽象类的类。他是可以被实例化的。高层模块指的是调用端，底层模块是具体的实现类。在Java中，依赖倒置原则是指模块间的依赖是通过抽象来发生的，实现类之间不发生直接的依赖关系，其依赖关系是通过接口是来实现的。这就是俗称的面向接口编程。

例子：



这个是一个很简单的例子，但是如果我们要新增加一个功能，工人用 螺丝刀来修理东西，在这个类，我们发现是很难做的。因为我们Worker类依赖于一个具体的实现类Hammer。所以我们用到面向接口编程的思想，改成如下的代码：



然后我们的Worker是通过这个接口来于其他细节类进行依赖。代码如下：



Hammer类与Screwdriver类实现这个接口:



这样，通过面向接口编程，我们的代码就有了很高的扩展性，降低了代码之间的耦合度，提高了系统的稳定性。

# 接口隔离原则-ISP

不应该强迫客户依赖于它们不使用的方法。

# 模板方法设计模式

模板方式使用继承来解决问题，模板方法和策略模式都可以分离通用的算法和具体的上下文。

# 策略设计模式

策略模式则使用委托来解决问题，策略模式比模板方法模式涉及更多数量的类和间接层次。ApplicationRunner中委托指针的使用招致了比继承稍微多一点的运行时间和数据空间开销。但是另一方面，如果有许多不同应用程序运行，就可以重用ApplicationRunner，并把许多不同的Application实现传递给它，从而减小通用算法和该算法所控制的具体细节之间的耦合。

# SINGLETON模式和MONOSTATE模式

SINGLETON模式使用私有构造函数，一个静态变量，以及一个静态方法对实例化进行控制和限制。MONOSTATE模式只是简单地把对象的所有变量变成静态的。

区别：SINGLETON模式强调结构上的单一性,它防止创建多个对象实例。

MONOSTATE模式强调行为上的单一性，而没有强加结构方面的限制。

# FACTORY模式

FACTORY模式允许只依赖于抽象接口就能创建出具有具体对象的实例，在开发期间，如果具体类是高度易变得，那么该模式是非常有用的。

# OBSERVER设计模式

观察者（Observer）模式的定义：指多个对象间存在一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。这种模式有时又称作发布-订阅模式、模型-视图模式，它是对象行为型模式。

1、可以向各种对象注册观察者，而不用这些对象显示的调用你，这种间接关系是一种有用的管理依赖关系的方式。

2、两种模式：

推模式：如果被观察对象比较复杂，并且观察者需要一个提示，那么推模式是合适的。

拉模式：如果被观察对象比较简单，那么拉模式很适合。

3、模式的结构

观察者模式的主要角色：

抽象主题（Subject）角色：也叫抽象目标类，它提供了一个用于保存观察者对象的聚集类和增加、删除观察者对象的方法，以及通知所有观察者的抽象方法。

具体主题（Concrete Subject）角色：也叫具体目标类，它实现抽象目标中的通知方法，当具体主题的内部状态发生改变时，通知所有注册过的观察者对象。

抽象观察者（Observer）角色：它是一个抽象类或接口，它包含了一个更新自己的抽象方法，当接到具体主题的更改通知时被调用。

具体观察者（Concrete Observer）角色：实现抽象观察者中定义的抽象方法，以便在得到目标的更改通知时更新自身的状态。

# Visitor模式：

解决的问题：需要向类层次结构中增加新的方法，但是增加起来会很费劲或者会破坏原设计。

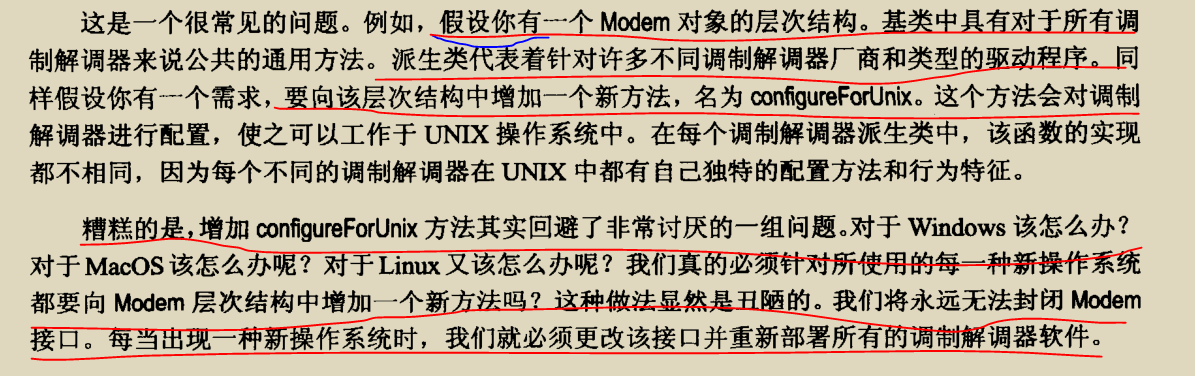
1、常规-核心：双重分发

2、DECORATOR模式

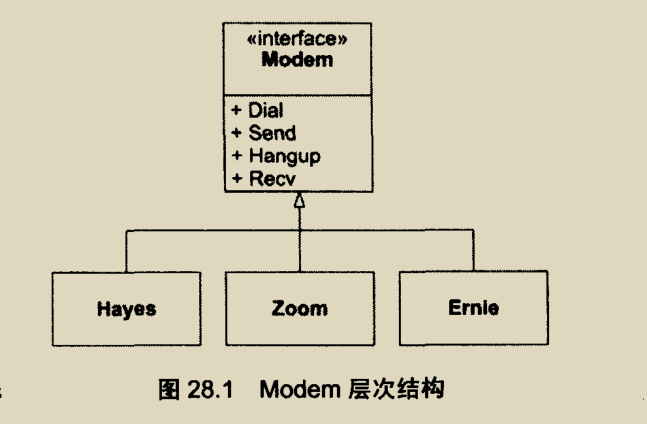
3、VISITOR模式允许在不更改现有类的层次的结构下向其中增加新方法

实例如下：

问题：



Model层次结构如下：

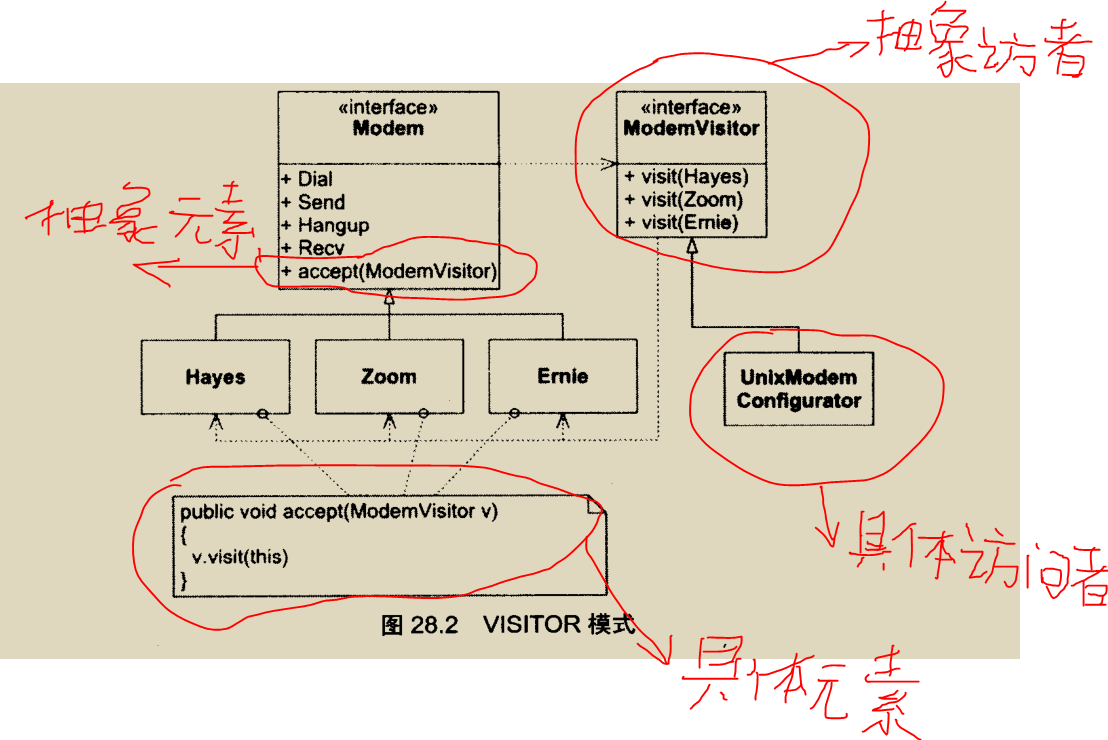


解决方法：

（1）在Modem接口中增加ConfigureForUnix。（这种方法违背了开闭原则，如果是window，MAC那么还有增加方法）

（2）如果不在Modem接口中增加ConfigureForUnix方法，那么我们怎么才能把这些调制解调器配置为可以在UNINX中使用呢？可以使用一项为双重分发的技术，这项技术是VISITOR模式的核心机制。

使用VISITOT模式以后的Model层次结构如下：



# 包的设计原则

1、把类划分到包中的原则：

重用发布等价原则：重用的粒度就是发布的粒度

共同重用原则：一个包中的所有类应该是重用的。如果重用了包中的一个类，那么就要重用包中的所有类。

共同封闭原则：包中的所有类对于同一性质的变化应该是封闭的。一个变化若对一个包产生影响，则将对该包中所有类产生影响，而对于其他包不造成影响。

2、处理包之间相互关系的原则：

无环依赖原则：在包的依赖关系图中不允许存在环。

消除依赖环

当我们去重构代码以解决耦合性、简单性以及表达性的问题时，可能会发现代码已经接近于一个特定的模式了。此时，我们把类和变量的名字改成使用模式的名字，并且把代码的结构更改为正规的形式使用模式。这样代码就回归为模式。