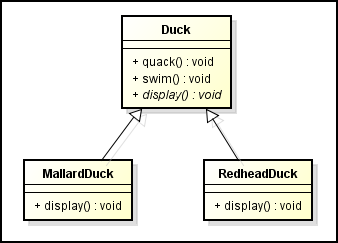
# 策略模式(Strategy)

## 背景：



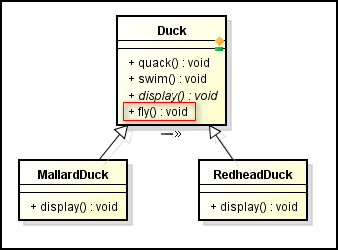
已有的系统如上所示，因为不同的鸭子有不同的颜色，所以display是一个abstract方法，有许多的子类继承自Duck类。

## 问题的提出:

现在想为游戏提供“飞”的功能，同时将来也要增加“叫”的功能

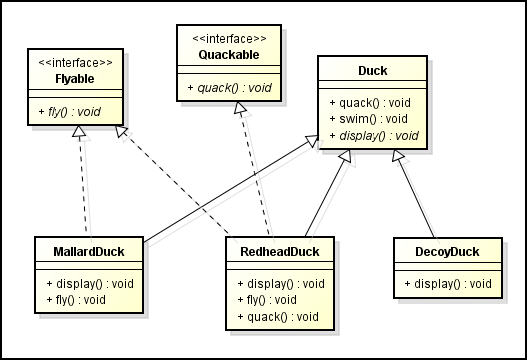
## 错误的解决方案：

### 1.



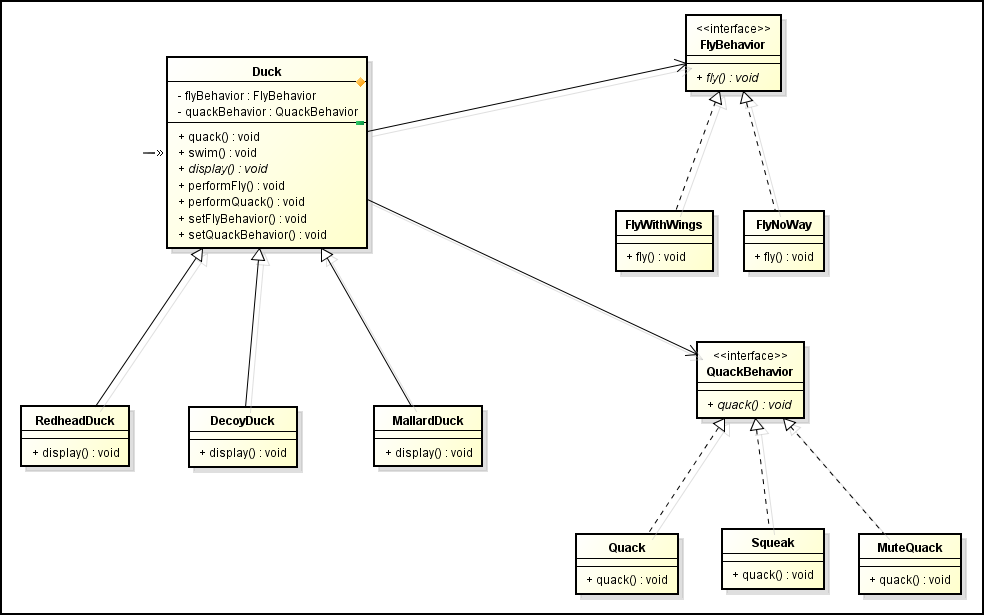
这种方案会污染所有的子类，即不应该能飞的鸭子也有这样的行为

### 2.增加两个接口，Flyable和Quackable，需要“飞”的鸭子实现这个接口，并且实现自己的“飞”逻辑，对于“叫”接口也类似，对于不需要“飞”或“叫"鸭子就不实现任何接口。



缺点：每个能”飞“的鸭子的业务都封装在各处的类内，这样无法实现代码重用，如果将来有一个很小的针对”飞“的改动，将导致要每一个实现了”飞“接口的子类都必须作修改。

## 正确的解决方法：



## 设计原则：

1. 找出应用中可能需要变化之处，把它们独立出来，不要和那些不需要变化的代码混在一起（把会变化的部分取出并”封装“起来，好让其它部分不会受到影响）
2. 针对接口编程，而不是针对实现编程
   * 父类不要负责Fly和Quack的接口，反而是由我们制造的一组其它类专门实现FlyBehavior和QuackBehavior，这就称为”行为“类。由行为类而不是Duck类来实现行为接口。
   * 在我们的新设计中，鸭子的子类将使用接口，所表示的行为，所以实现的”实现“不会被绑死在鸭子的子类中。
3. 多用组合，少用继承。

正如例子中所见，使用组合建立系统具有很大的弹性，不仅可将算法封装成类，更可以”在运行时动态地改变行为“，只要组合的行为对象符合正确的接口标准即可。

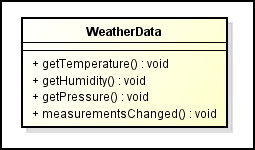
## 定义：

策略模式定义了算法族，分别封装起来，让它们之间可以互相替换，此模式让算法的变化独立于使用算法的客户

# 观察者模式（Observer）

## 背景

需要在不同的布告板（“目前状况”布告板，“气象统计”布告板和“天气预报”布告析）上显示天气的信息。天气站发布的信息类如下：



错误的示范：

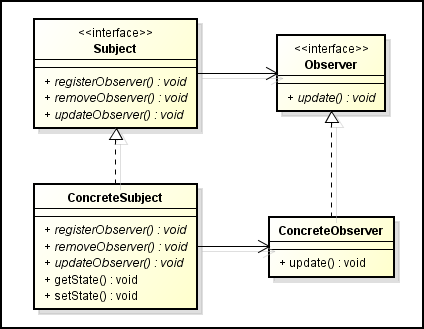
|  |
| --- |
| **public** **class** WeatherData {  **public** **void** measurementsChanged(){  **float** temp = getTemperature();  **float** humidity = getHumidity();  **float** pressure = getPressure();    currentConditionsDisplay.update(temp, humidity, pressure);  statisticsDisplay.update(temp, humidity, pressure);  forecastDisplay.update(temp, humidity, pressure);  }  } |

* 针对具体实现编程，会导致我们以后在增加或删除布告板时必须修改程序
* update(temp, humidity, pressure)这一部分至少看起来就是一个接口，方法名和参数都是一样的
* 红色的部分就是改变的地方，需要封装起来

## 定义：

观察者模式定义了对象之间的一对多依赖，这样一来，当一个对象改变状态时，它的所有依赖者都会收到通知并自动更新

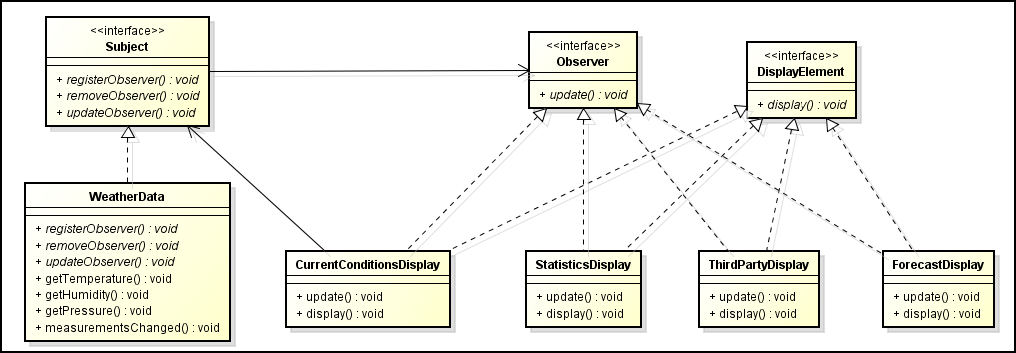
## 类图：



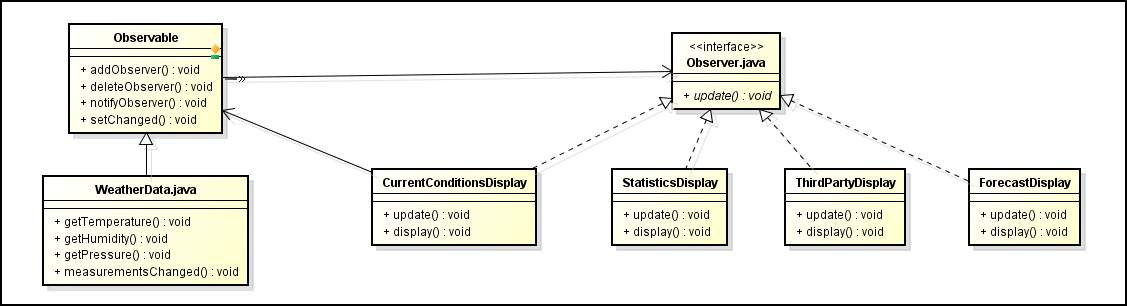
## 设计原则：

* 为了交互对象之间的松耦合设计而努力
* 松耦合设计之所以能让我们建立有弹性的OO系统，能够应付变化，是因为对象之间的互相依赖降到了最低
* 找出程序中会变化的方面，然后将其和固定不变的方面相分离
* 针对接口编程，不针对实现编程
* 多用组合，少用继承

## 解决方案



使用JDK中自带的Observer接口来实现：



需要注意的是Observable是一个具体的类，相当于我们的Subject，且里面的setChanged()方法是protected的

小实现：

|  |
| --- |
| **public** **class** WeatherDataInJDK {  **void** setChanged() {  changed = **true**;  }  **void** notifyObservers(Object arg){  **if**(changed){  **for** every observer on the list{  call update(**this**, arg);  }  changed = **false**;  }  }  **void** measurementsChanged() {  setChanged();  notifyObservers(**null**);  }  } |

Java.util.Observable的黑暗面

是的，你注意到了！如同你发现的一样，Observable是一个类而不是一个接口，更糟糕的是，它甚至没有实现一个接口。不幸的是，java.util.Observable的实现有许多问题，限制了它的使用和复用。这并不是说它没有提供有用的功能，只是想提醒大家注意到这些事实。

这样的实现的问题有：

首先，因为Observable是一个类，你必须要设计一个类继承它。如果某个类想同时具有Observable类和另一个超类的行为，就会陷入两难，毕竟Java不支持多继承。这限制了Observable的复用潜力（而增加复用潜力不正是我们使用模式最原始的功机吗？）

再者，因为没有Observable接口，所以你无法建立自己的实现，和Java内置的Observable API搭配使用，也无法将java.util的实现换成另一套做法的实现（比方说，

Observable将关键的方法保护了起来，如果你看看Observable API，你会发现setChanged()方法被保护起来了（被定义成protected）。那又怎么样呢？这意味着：除非你继承自Observable，否则你无法创建Observable实例并组合到你自己的对象中来。这个设计违反了第二个设计原则：“多用组合，少用继承”

如果你能够扩展java.util. Observable，那么Observable“可能”可以符合你的需求，否则，你可能需要像本章开头的时候一样自己实现这一整套观察者模式。不管用那一种方法，只要你已经熟悉了观察者模式了，应该都能善用它们。

JDK中大量使用了观察者模式，如JButton等。

## 要点

* 观察者模式定义了对象之间一对多的关系
* 主题（也就是可观察者）用一个共同的接口来更新观察者
* 观察者和可观察者之间用松耦合的方式结合（loosecoupling），可观察者不知道观察者的细节，只知道观察者实现了观察者接口
* 使用此模式时，你可从被观察者处推或拉数据（然而，推的方式更被认为更“正确”）
* 有多个观察者时，不可以依赖特定的通知次序
* Java有多种观察者模式的实现，包括了通用的java.util.Observable
* 要注意java.util.Observable实现上所带来的一些问题
* 如果有必要的话，可以实现自己的Observable，这并不难，不要害怕
* Swing大量使用观察者模式，许多GUI框架也是如此
* 此模式也被应用在许多地方，如JavaBeans，RMI

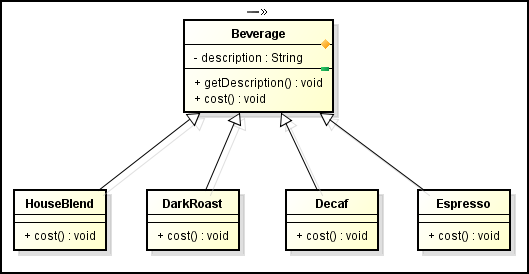
# 装饰模式

你熟悉了装饰的技巧，你将能够在不修改任何底层代码的情况下，给你的（或别人的）对象赋予新的职责。

## 问题提出和背景

在Starbuzz中，在购买咖啡的时候，新引进加入各种调料的业务，如蒸奶(Steamed Milk)，豆浆(Soy)，摩卡(Mocha)或覆盖奶泡。Starbuzz会根据加入的调料收取不同的费用。所以订单系统必须考虑到这些调料部分。

下面是已存在的系统

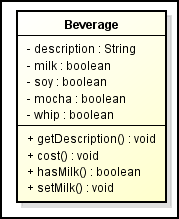


其中cost方法在父类中是一个抽象方法，每一个子类实现它并计算出该饮料要多少钱

## 不好的方案

1. 将每一种调料和咖啡的搭配都导出一个类（会导致类爆炸问题）

2. 在父类Beverage中设计成如下图示：



上图中的余下的getter和setter省略了。

在上面的设计中，cost不再是一个抽象方法，而是在Beverage中提供了实现，让它计算加入各种饮料的调料价钱。子类将覆盖cost方法，但是会调用超类的cost，计算出基本饮料加上调料的价格。

这个方案的问题有：

* 调料的价钱的改变会导致我们更改现有的代码
* 一旦出现新的调料，我们就需要加上新的方法，并改变超类中的cost方法
* 以后可能会开发出新的饮料。对于这些饮料而言（如冰茶），某些调料可能并不合适，但是在这个设计方式中，茶子类仍将继承那些不合适的方法，例如：hasWhip()(加奶泡)
* 万一顾客想要双倍的摩卡咖啡，怎么办？

## 总结：

* 利用继承设计子类的行为，是在编译时静态决定的，而且所有的子类都会继承到相同的行为。然而，如果能够利用组合的做法扩展对象的行为，就可以在运行时动态地进行扩展。
* 我们可以利用此技巧把多个新的职责，甚至是设计超类时还没有想到的职责加在对象上。而且，可以不用修改原来的代码
* 通过动态地组合对象，可以写新的代码添加新的功能，而无须修改现有的代码。既然没有改变现有的代码，那么引进bug或产生意外副作用的机会将大幅度减少
* 代码应该如同晚霞中的莲花一样地关闭（免于改变），如同晨曦中的莲花一样地开放（能够扩展）

## 设计原则

类应该对扩展开放，对修改关闭

## 认识装饰者模式

如果顾客想要摩卡和奶泡深焙咖啡，那么，要做的是：

* 拿一个深焙咖啡(DarkRoast)对象
* 以摩卡（Mocha）对象装饰它
* 以奶泡(Whip)对象装饰它
* 调用cost()方法，并依赖委托将调料的价钱加上去

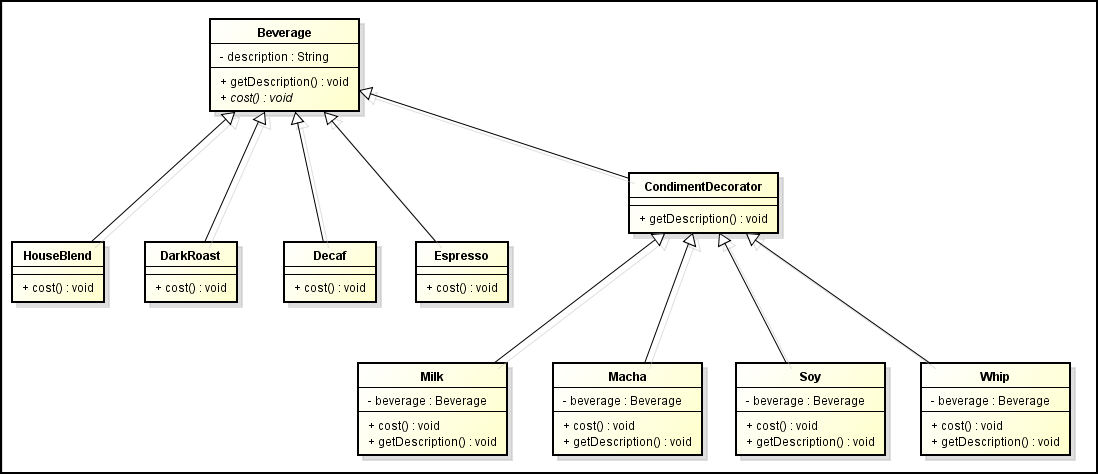
### 特点

* 装饰者和被装饰者对象有相同的超类型
* 你可以用一个或多个装饰者包装一个对象
* 装饰者可以在所委托被装饰者的行为之前与/或之后，加上自己的行为，以达到特定的目的
* 对象可以在任何时候被装饰，所以可以在运行时动态地，不限量地用你喜欢的装饰者来装饰对象
* 装饰者有一个实例变量以保存某个被装饰者的引用

## 定义

装饰者模式动态地将责任附加到对象上。若要扩展功能，装饰者提供了比继承更有弹性的替代方案

## 解决方案



Beverage相当于抽象的component

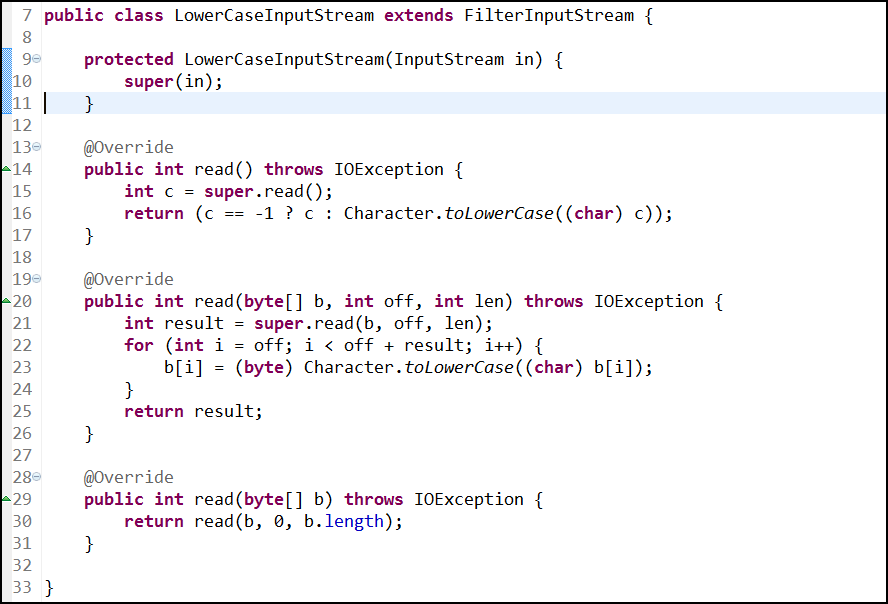
## 为何还是使用继承，并没有使用组合？

* 的确是如此，但是，这么做的重点在于，装饰者和被装饰者必须是一样的类型，也就是有着共同的父类，这是相当关键的地方。在这里，我们利用继承达到“类型匹配”，而不是利用继承获得“行为”
* 由于装饰者必须能够取代被装饰者，所以装饰者需要和被装饰者（亦即被包装的组件）有相同的“接口”
* 当我们将装饰者与组件组合时，就是在加入新的行为。所得到的新的行为，并不是继承自父类，而是由组合对象得到的
* 如果是依赖继承，那么类的行为只能在编译时静态决定。换句话说，行为如果不是来自父类，就是子类覆盖后的版本。反之，利用组合，可以把装饰者混合着用。。。。而且是在运行时
* 我们可以在任何时候实现新的装饰者增加新的行为。如果依赖继承，每当需要新的行为时，还得修改现在的代码。
* 那么为何不把Beverage类设计成一个接口呢？不是不能，而是已有系统中已经是一个抽象类了。

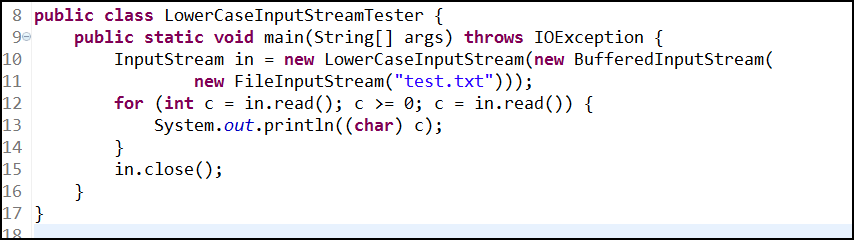
## 装饰者模式的缺点

Java I/O也引出装饰者模式的一个“缺点”：利用装饰者模式，常常造成设计中有大量的小类，数量实在太多，可能会造成使用此API程序员的困扰。但是，现在你已经了解了装饰者模式的工作原理，以后当使用别人的大量的装饰API时，就可以很容易的辨别出他们的装饰者类是如何组织的，以方便用包装方式取得想要的行为

## 实现自己的Java I/O 装饰者



测试代码



## 要点

* 继承属于扩展形式之一，但不见得是达到弹性设计的最佳方案
* 在我们的设计中，应该允许行为可以被扩展，而无须修改现有的代码
* 组合和委托可用于在运行时动态地加上新的行为
* 除了继承，装饰者模式也可以让我们扩展行为
* 装饰者模式意味着一群装饰者类，这些类用来包装具体组件
* 装饰者类反映出被装饰的组件类型（事实上，他们具有相同的类型，都经过接口或继承实现）
* 装饰者可以在被装饰者的行为前面与/或后面加上自己的行为，甚至将被装饰者的行为整个取代掉，而达到特定的目的
* 你可以用无数个装饰者包装一个组件
* 装饰者一般对组件的客户是透明的，除非客户程序依赖于组件的具体实现
* 装饰者会导致设计中出现许多小对象，如果过渡使用，会让程序变得更复杂

# 工厂模式

## 定义

工厂模式定义了一个创建对象的接口，但由子类决定要实例化的类是那一个。工厂方法让类把实例化推迟到了子类

## 特点

尽管只有一个具体创建者，工厂方法模式依然有用，因为它帮助我们将产品的“实现”从“使用”中解耦。如果增加产品或者改变产品的实现，Creator并不会受到影响（因为Creator与任何ConreteProduct之间都不是耦合

静态工厂的缺点为不能通过继承来改变创建方法的行为

## 设计原则

要依赖抽象，不要依赖具体类（依赖倒置原则）

## 要点

* 所有的工厂都是用来封装对象的创建的
* 简单工厂，虽然并不是真正的设计模式，但仍不失为一个简单的方法，可以将客户程序从具体类中解耦
* 工厂方法使用继承：把对象的创建委托给子类，子类实现工厂方法来创建对象
* 抽象工厂使用对象组合：对象的创建被实现在工厂接口所暴露出来的方法中
* 所有工厂模式都通过减少应用程序和具体类之间的依赖促进松耦合
* 工厂方法允许类将实例化延迟到了子类进行
* 抽象工厂创建相关的对象家族，而不需要依赖它们的具体类
* 依赖倒置原则，指导我们避免依赖具体类型，而要尽量依赖抽象
* 工厂是很有威力的技巧，帮助我们针对抽象编程，而不是针对具体类编程

# 单件模式

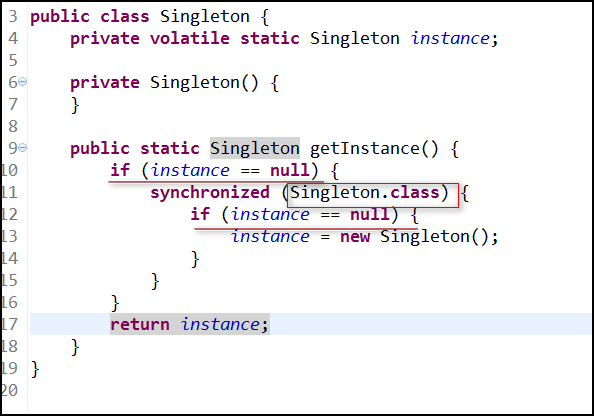
## 定义

单件模式确保一个类只有一个实例，并提供一个全局访问点

同步一个方法可能造成程序执行效率下降100倍

单件类是不能被继承的，因为他们的构造器是私有的

## 所重检查加锁方法实现单件模式



* 所谓的双重检查为执行了两次 if(instance == null)这个语句
* Volatile关键词确保：当instance变量被初始化成Singleton实例时，多个线程正确地处理instance变量
* 双重检查版本不适用于1.4及更早版本的Java。很不幸地，在1.4及更早版本的Java中，许多的JVM对于 volatile关键字的实现会导致双重检查加锁的失效。如果你不能使用Java 5，而必须使用旧版的Java，就请不要利用此技巧实现单件模式
* 两个类加载器可能有机会各处创建自己的单件实例：每个类加载器都定义了一个命名空间，如果有两个以上的类加载器，不同的类加载器可能会加载同一个类，从整个程序来看，同一个类会被加载多次。如果这样的事情发生在单件上，就会产生多个单件并在的怪异现象。所以，如果你的程序有多个类加载器又同时使用了单件模式，请小心。有一个解决办法是：自行指定类加载器，并指定同一个类加载器。
* 在Java1.2之前，垃圾回收器有一个bug，会造成当单件在没有全局的引用时被当作垃圾清除。这造成让人困惑的问题：因为在单件被清除之后，下次调用getInstance会产生一个“全新的”单件。对很多程序来说，这会造成让人困惑的行为，因为对象的实例变量值都不见了，一切加到最原始的位置（例如：网络连接被重新设置）。但在1.2之后这个问题已经被修复了，也不需要一个全局的引用来保护单件。

## 要点

* 单件模式确保程序中一个类最多只有一个实例
* 单件模式也提供访问这个实例的全局点
* 在Java中实现单件模式需要私有的构造器，一个静态方法和一个静态变量
* 确定在性能和资源上的限制，然后小心地选择适当的方案来实现单件，以解决多线程问题（我们必须诊室所有的程序都是多线程的
* 如果不是采用第五版的Java 2，双重检查加锁实现会失效
* 小心，如果你使用多个类加载器，可能导致单件失效而产生多个实例
* 如果使用JVM1.2或之前的版本，你必须建立单件注册表，以免垃圾收集器将单件回收