Robert C. Martin氏为我们总结了在面向对象的设计（OOD）中应该遵循的原则，这些原则被称为“Principles of OOD”，关于“Principles of OOD”的相关文章可以从Object Menter得到。

# 面向对象设计基础

## 类图关系

UML的类图关系分为：关联、聚合/组合、依赖、泛化（继承）。而其中关联又分为双向关联、单向关联、自身关联。

### 关联

关联（Association）关系：表示类与类之间的联接，它使一个类知道另一个类的属性和方法。

**双向关联**

E:\70 TMP\QQ图片20150410135347.png

C1-C2：指双方都知道对方的存在，都可以调用对方的公共属性和方法。

使用ROSE 生成的代码是这样的：

class C1 { public: C2\* theC2; };

class C2 { public: C1\* theC1; };

双向关联在代码的表现为双方都拥有对方的一个指针，当然也可以是引用或者是值。

**单向关联**

E:\70 TMP\QQ图片20150410135347.png

C3->C4：表示相识关系，指C3知道C4，C3可以调用C4的公共属性和方法。没有生命期的依赖。一般是表示为一种引用。

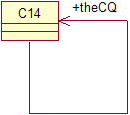
生成代码如下：

class C3 { public: C4\* theC4; };

class C4 { … };

单向关联的代码就表现为C3有C4的指针，而C4对C3一无所知。

**自身关联（反身关联）**



自己引用自己，带着一个自己的引用。

代码如下：

class C14 { public: C14\* theC14; };

就是在自己的内部有着一个自身的引用。

### 聚合/组合

当类之间有整体-部分关系的时候，我们就可以使用组合或者聚合。

**聚合**

E:\70 TMP\QQ图片20150410135347.png

是关联的一种，是强的关联关系，是整体与个体的关系。

表示C9聚合C10，但是C10可以离开C9而独立存在（这句话怎么解，请看下面组合里的解释）。

代码如下：

class C9 { public: C10 theC10; };

class C10 { … };

**组合**

E:\70 TMP\QQ图片20150410135347.png

是关联关系的一种，是比聚合关系强的关联。它要求普通的聚合关系中代表整体的对象负责代表个体的对象的生命周期。合成关系不能共享。

一般是实心菱形加实线箭头表示，如上图所示，表示的是C8被C7包容，而且C8不能离开C7而独立存在。A组合B，则A需要知道B的生存周期，即可能A负责生成或者释放B，或者A通过某种途径知道B的生成和释放。

他们的代码如下：

class C7 { public: C8 theC8; };

class C8 { … };

可以看到，代码和聚合是一样的。具体如何区别，只能用语义来区分了。

### 依赖

E:\70 TMP\QQ图片20150410135347.png

是类与类之间的连接，依赖总是单向的。依赖表示一个类依赖于另一个类的定义。通常被依赖的对象以方法的参数或返回表示。

指C5可能要用到C6的一些方法，也可以这样说，要完成C5里的所有功能，一定要有C6的方法协助才行。C5依赖于C6的定义，一般是在C5类的头文件中包含了C6的头文件。ROSE对依赖关系不产生属性。

注意，要避免双向依赖。一般来说，不应该存在双向依赖。

ROSE生成的代码如下：

// C5.h

#include "C6.h"

class C5 { };

// C6.h

#include "C5.h"

class C6 { };

虽然ROSE不生成属性，但在形式上一般是A中的某个方法把B的对象作为参数使用（假设A依赖于B）。如下：

#include "B.h"

class A { void Func(B &b); }

### 泛化（继承）

E:\70 TMP\QQ图片20150410135347.png

泛化关系：如果两个类存在泛化的关系时就使用，例如父和子，动物和老虎，植物和花等。

表示类与类之间的继承关系，接口与接口之间的继承关系，类对接口的实现关系。

# 度量

## 度量工具和命令

一个依赖提取器/分析器包：

adep：创建别名，将文件聚合进内聚组件。

cdep：从一个文件集合中提取编译时依赖。

ldep：分析一个组件集合中的连接时依赖。

在大型软件系统的整个开发过程中很难过分强调主动使用这样的物理设计工具的重要性。

**留心度量的副作用**：度量会对动机产生影响。人们会对那些被度量的事物更加用心，他们认为度量的目标是在评价他们。要谨慎地选择哪些环节需要被度量。人们会倾向于集中做那些被度量的工作，而忽视未被度量的工作。

## 度量指标

### 响应集

定义：类的各个方法直接调用的函数的数目。

### CCD

组件依赖的度量。可以计算一下重构前、后的。

## 软件开发的度量环节

### 规模

编写的代码总行数

注释总行数

类或者子程序的总数

数据声明的总数

空行总数

### 可维护性

每个类的public子程序数量

传给每个子程序的参数数量

每个类的private子程序和/或private变量的数量

每个子程序使用的局部变量的数量

每个类或子程序调用的子程序的数量

每个子程序中决策点的数量

每个子程序中控制流的复杂度

每个类或子程序中包含的代码行数

每个类或子程序中的注释行数

每个类或子程序中数据声明的数量

每个类或子程序中的空行数量

每个类或子程序中使用goto的数量

每个类或子程序中输入语句或输出语句的数量

### 生产率

项目花费的工作小时数

每个类或子程序花费的工作小时数

每个类或子程序的修改次数

项目花费的钱数

每行代码花费的钱数

每个缺陷花费的钱数

### 整体质量

缺陷总数

每个类或子程序的缺陷数

每千行代码的平均缺陷量

两次系统“失灵”之间的平均时间

编译器检测到的错误数量

### 缺陷跟踪

每个缺陷的严重级别

每个缺陷的位置（类或子程序）

每个缺陷的根源（需求、设计、构造、 测试）

更正每个缺陷的方法

每个缺陷的责任人

每更正一个缺陷所涉及的代码行数

更正每一个缺陷所花费的工作小时数

找出每一个缺陷所需的平均时间

更正每一个缺陷的平均用时

更正每个缺陷的尝试次数

因更正缺陷而引发的新错误数量

## 设计的度量标准

### WMC类方法数

对WMC标准有两种解释。第一种被称为方法复杂度，本书将不予探讨；第二种解释是指在一个类中实现的方法的数量。

面向对象特点：类/方法。

用于度量：复杂性；可用性；可复用性。

度量方法：类中实现的方法的数目；各方法的复杂性之和。

解释：较大=>通过继承对子类的潜在影响较大；具有应用程序依赖性。

经验证明这样的结论：一个类中的方法的数量越大，类的影响就越大。含有大量方法的类的影响范围，从所需的测试工作量到可能只为特定的应用程序所用。方法众多的类会限制复用的可能性。

### RFC类应答数

Response For a Class。

面向对象特点：类/方法。

用于度量：设计可用性；可测试性。

度量方法：因回应消息而被激活的度量数。

解释：较大=>复杂性高，降低可理解度；测试和调试更复杂。

类应答数RFC度量标准就是类中存在的、可被发送到类的对象的消息激活的方法的数量，它包括类等级结构中所有的可访问的方法。

类中可被消息激活的方法的数量越大，类的复杂性就越大。如果大量的方法可因应答消息被激活，类的测试和调试就变得复杂起来，因为对测试人员来说，需要更深层次的理解。

RFC标准用于评价系统设计、可用性和可测试性。

### LCOM方法关联性缺乏度

Lack of Cohesion Of Mefhods。

面向对象特点：类/关联。

用于度量：设计；可复用性。

度量方法：类中由属性决定的方法间的相似性。

解释：低=>类中的关联性好高=>类中的关联性差——将类拆分。

方法关联性缺乏度LCOM度量标准通过实例变量或属性衡量方法的相似程度。任何方法分离性的度量都有助于识别类设计中的瑕疵。

关联性是类中的方法的相互关联，共同协作，提供完整周密的行为的程度，所以关联性指设计各部分的内在一致性。关联性以被封装在对象中的数据为中心，关注方法是如何与数据交互，完成整个行为的。方法的相似程度是类关联性的一个重要方面，我们的目标是实现最大的关联度。适应性好和可复用的程序具有耦合性低，关联性高的特点。

有效的面向对象设计可最大限度地增加关联性，因为它能提高封装性能。方法关联性缺乏度标准就是针对关联性的。

度量关联性至少有两种不同的方法：

1、对类中的每一个数据项，计算使用该数据项的方法在方法中所占的百分比，然后对这些百分比平均，再从100%中减去这个平均百分比。得到的百分数越低，意味着类中的数据和方法的关联性越高。

2、如果方法对相同的属性操作，那么，它们的相似性较大。在由方法使用的属性集合的交集中，对产生的不连通集合的数量进行计数，这是另一种计算关联性的方法。

关联度高表明类的划分合理，缺乏关联性或关联性低会增加系统的复杂性，因而会增加开发过程中的错误概率。低关联性的类可被划分为两个或多个关联性高的子类。这个标准用于评价设计实现和可复用性。

### CBO对象类之间的耦合度

Coupling Between Object Classes。

面向对象特点：耦合。

用于度量：设计；可复用性。

度量方法：明确的非继承相关的类的数量。

解释：高=>设计低劣；难以理解；降低复用性；增加维护成本。

对象类之间的耦合度CBO度量是与一个类耦合的其他类的数量。它是通过计算与一个类所依赖的继承等级结构明确不相关的类的数量而获得的。

耦合是衡量一个实体与另一个实体之间所建立的联系的强度的尺度。类（或对象）的耦合表现在三个方面：

1、当消息在对象之间传递时，对象是耦合的。

2、当类之间相互交互时，它们是耦合的，即在一个类中声明的方法使用另一个类中的方法。

3、继承性在父类和子类之间引入了重要而紧密的耦合关系。

在这里考虑继承等级结构中的耦合关系没有意义。CBO度量标准是根据类之间的交互活动而不是根据继承关系计算出来的。

过分的耦合对模块设计有害，会妨碍代码复用。类越独立，就越容易被其他应用程序复用。耦合度的数值越大，对其他设计部分变化的敏感性越高，因而维护也就越困难。强耦合关系会使系统复杂化，因为如果其中的模块与其他模块关联，就难于理解、改变和纠错。系统设计过程中，若能保证模块间的耦合尽可能达到最弱，就会减低系统的复杂性，这样就可以提高设计的模块化程度和封装性。

### DIT继承树深度

Depth of Inheritance Tree。

面向对象特点：继承。

用于度量：可复用性；可理解性；可测试性。

度量方法：从类节点到根的最大长度。

解释：较高=>更复杂；可重用性好。

继承树深度DU定义为，继承等级结构图中的一个类的深度是从该类的节点到继承树的根的最大长度，通过祖先类的数量来度量。

面向对象系统中另一个设计抽象方式是使用继承。继承是类之间的一种关系，能让编程者复用以前定义的对象，包括变量和操作符。通过降低操作和操作符的数量，继承降低了复杂性，这种对象抽象方式可能造成维护和设计的困难。用于度量继承关系的两个标准是继承等级结构的深度和宽度。

一个类在等级结构中的位置越深，它继承的方法的数量就可能越多，从而就越难于预测它的行为。较深的继承树会增大设计复杂度，因为会涉及较多的方法和类，但继承的方法被复用的潜力也较大。

一个支持DIT的度量标准是继承方法数（Number of Methods Inherited, NMI）。

### NOC子类数量

Number Of Children。

面向对象特点：继承。

用于度量：设计。

度量方法：直接子类的数量。

解释：较高=>可复用；设计不好，增加测试量。过度的关注一个类。

子类数量NOC指继承等级结构中直接从属于一个类的子类的数量，它是一个类在设计和系统中的潜在影响力的指示器。

子类的数量越大，对父类抽象不当的可能性也就越大，因此很可能是一个子类错用的案例。但是，子类的数量越大，复用程度也越髙，因为继承就是一种复用。如果一个类有大量的子类，可能需要对类的方法做更多的测试，这样就拉长了测试时间。因此，NOC主要度量可测试性和设计。

## Dependometer研究

# 方法论

## 敏捷方法

我们称那些轻型的、有效的方法为敏捷方法。在重型方法中，我们在一些不必要、重复的中间环节上浪费了太多的精力，而敏捷则避免了这种浪费。

我对敏捷的理解包括了几个方面：

较低的管理成本和高质量的产出。软件开发存在两个极端：一个是没有任何的管理成本，所有的工作都是为了软件的产出，但是这种方式却往往导致软件开发过程的混沌，产品的低质量，团队士气的低落。另一个是大量管理活动的加入，评审、变更管理，缺陷跟踪，虽然管理活动的加入能够在一定程度上提高开发过程的有序性，但是成本却因此提高，更糟糕的是，很容易导致团队的低效率，降低创新能力。因此，敏捷方法试图寻找一个平衡点，用低成本的管理活动带来最大的产出，即软件的高质量。

尊重人性。敏捷方法尊重人性，强调效率。软件开发可以说是一种脑力的投入，如果不能保证开发人员的自愿投入，产品就肯定要打折扣。事实多次的证明，一个愿意投入的开发人员和一个不愿意投入的开发人员效率相差在三倍以上，对组织的贡献更是在十倍以上。

沟通和反馈是一切的基础。我们已经讨论过沟通的重要程度，而即时的反馈是拥抱变化的前提条件。

客户是上帝。没有客户就没有一切，客户的重要性可以用一句话来形容，就是以合理的成本建造合适的软件。

敏捷其实也有轻重之分，关键在于是否能够做到有效和灵活。因此，敏捷方法论提倡的一个思想是“刚好够”，不过这个“刚好够”可不是那么容易判断的。一支8个人的团队采用XP方法，随着方法的熟练使用，团队的能力在不断的增强，能够处理的问题越越来越复杂，也许他们能够处理采用重型方法的20个人团队能够处理的问题。可是人数增加的时候，原先的方法肯定要适当的调整，比如说，在原先的敏捷方法上增加一些重型方法的技巧。我们不能够要求一支6个人的团队和一支20个人的团队用同样的方法，前者可能采用轻一些的敏捷方法，后者可能采用重一些的敏捷方法，关键的问题在于，两支团队都把重点放在沟通、反馈、频繁交付软件这些关键的因素上，也就是做到有效和灵活。

## 重型方法

我们根据把拥有大量artifact（RUP官方翻译为工件，意思是软件开发过程中的中间产物，如需求规约、设计模型等）和复杂控制的软件开发方法称为重型（Heavy Weight）方法，相对的，我们称artifact较少的方法为轻型（Light Weight）方法。

在传统的观念中，我们认为重型方法要比轻型安全许多。因为我们之所以想出重型方法，就是由于在中大型的项目中，项目经理往往远离代码，他无法有效的了解目前的工程的进度、质量、成本等因素。为了克服未知的恐惧感，项目经理制定了大量的中间管理方法，希望能够控制整个项目，最典型的莫过于要求开发人员频繁的递交各种表示项目目前状态的报告。

在Planning XP一书中有一段讨论轻重型方法论的精辟论述，它把重型方法论归结为一种防御性的姿态（defensive posture），而把轻型方法论归结为一种渴望成功（Plan to win）的心态。如果你采用防御性姿态，那么你的工作就集中在防止和跟踪错误上，大量的工作流程的制定，是为了保证项目不犯错误，而不是项目成功。而这种方法也不可谓不好，但前提是如果整个团队能够满足前面所提到的两个条件的话，项目也肯定会成功，但是重型方法论的一个弊端就在于，大家都在防止错误，都在惧怕错误，因此人和人之间的关系是很微妙的，要达到充分的沟通也是很难的。最终，连对人的评价也变成是以避免错误的多寡作为考评的依据，而不是成就。

管理，被称为科学和艺术的融合体，而管理的艺术性部分很大程度的体现在人的管理上。方法学，一样是科学和艺术的融合体。这是有依据的，其实方法论和管理学是近亲关系，管理学中有一门分支是项目管理，而在软件组织中，项目管理是非常重要的，方法学就是一种针对软件开发的一种特定的项目管理（或是项目管理的一个子集）。

重型方法最大的一个问题就在于他不清楚或忽略了艺术这个层次，忽视了人的因素，把人做为一个计量单位，一种资源，一种线性元素。而人的要素在软件开发中是非常重要的，软件开发实际上是一种知识、智力的转移过程，最终形成的产品是一种知识产品，它的成本取决于开发者的知识价值，因此，人是最重要的因素。而人这个要素是很难衡量的，每个人都有不同的个性、想法、经验、经历，这么多复杂的因素加在一起，就导致了人的不可预见性。因此，我们强调管人的艺术。

最简单的例子是，在重型方法中，我们的基本假设是对人的不信任。项目经理要控制项目，不信任就会产生很多的问题，比如士气不高，计划赶不上变化，创新能力低下，跳槽率升高等等。人都是希望被尊重的，技术人员更看重这一点。敏捷方法的出发点是相互信任，做到这一点是很难的，但是一旦做到了，那这个团队就是非常具有竞争力的。因此，这就产生了一个问题，在没有做到完全的相互信任之前，我们到底相不相信他人呢，这就是我提到的艺术性的问题，什么时候你要相信人？什么时候你不相信人，这些都是需要权衡的问题，也都是表现你艺术性的问题。

## 编程风格信仰

不是所有编程项目的管理者们都会认识到，有一些编程问题与信仰有关。如果你是一名管理者，并且试图要求统一某些编程实践，那么就可能会激怒你的程序员。下面就是事关信仰的一些问题：

1、编程语言

2、缩进风格

3、大括号的摆放位置

4、所用的集成开发环境

5、注释风格

6、效率与可读性的取舍

7、对方法的选择---例如，Scrum、极限编程、渐进交付

8、编程工具

9、命名习惯

10、对goto的使用

11、对全局变量的使用

12、量度，特别是有关生产力的量度，如每天编写的代码行数

以上问题的共同特征是，每一项都是程序员个人风格的反映。如果你认为有必要控制程序员的某些信仰，请考虑这些要点。要清楚地知道你是在处理一个敏感的问题，在全心全意投入之前要先试探程序员们对有关敏感问题的看法。对这些领域要使用“建议”或者“指导原则”，避免制定僵硬的“规则”或者“标准”。

**避免流露明显的意图：**为了规整缩进风格或者大括号的摆放位置，可以要求在源代码宣告完成之前先通过一个格式美化工具来处理。为了规整注释风格，可以要求对所有的代码进行复查，修改那些不清晰的代码，直到代码变得清晰为止。

**让程序员们制定他们自己的标准**：正如在其他地方所提到的，“特定标准的细节”往往没有“存在某个标准”重要。不要给你的程序员们设立标准，但一定要坚持让他们在“那些对你来说非常重要的领域里”标准化。

有哪些信仰的话题值得引发一场激战呢？要求在任何领域里的所有微小细节上都步调一致，其收效恐怕无法弥补由此而对士气造成的影响。如果你发现有人“不分青红皂白地使用goto或全局变量”、程序风格的可读性差、或者有其他影响整个项目的实践行为，那么就得为了提高代码质量而不怕一些摩擦了。如果你的程序员是负责任的，那么这很少会成为问题。最大的斗争往往只在编码风格的一些细微差异上，你完全可以置身事外——只要这对项目没有损失。

每个程序员都有自己喜欢的格式规则，但如果在一个团队中工作，就是团队说了算：一组开发者应当认同一种格式风格，每个成员都应该釆用那种风格。我们想要让软件拥有一以贯之的风格，我们不想让它显得是由一大票意见相左的个人所写成。

## 简单设计

敏捷开发提倡简单设计的实践，“并在实现新需求时抓住机会改进设计”以对同类性质的改动封闭，做到由需求的变化驱动设计的进化（我们不能因为设计的退化而责怪需求的变化），同时经验在此起到十分重要的作用，如有经验的设计人员可以凭经验在初始设计时做出必要的抽象来满足ocp原则等，或是在需求变动时确定系统所需的抽象（所需的封闭），当然应及早的刺激这种变化的出现（如测试驱动的开发方法）。

敏捷建模不建议提前进行各种假想的变化抽象，而是当变化发生第一次的时候抽象这种变化，以后同样的变化就变得很容易。对代码进行重构以保持良好的结构是很重要的，每次抽象都不应该使软件变得越来越僵化。这是非面向对象的语言不具备的优势。

**简化设计**

软件分析和设计的基本思路：

当项目组面临两难选择时，首先要用鱼和熊掌来识别矛盾的两个对立面。项目组用"小鱼"来比喻那些开发和维护代价较小、结构较简单但是缺乏某些灵活性的设计方案，用"熊掌"来比喻那些灵活、易扩展，但结构复杂，开发和维护成本较高的设计方案。在最终进行抉择的时候，项目组必须坚持如下准则：

在满足需求的情况下，尽量选择"小鱼"而舍弃"熊掌"；只有存在无可置疑的理由时，才选择"熊掌"作为设计方案。

人的智力总是有限的。有人说，平均智商的人一次很难理解7 个以上的对象。基于这一原理，程序员应当确保系统设计的最简化。毋庸置疑的是，简单的系统有着更少的错误，系统本身也更容易理解和维护。

追求最简化这一原则说起来简单，实施起来其实非常难。大多数程序员天生就有追求完美的习惯，许多人在设计软件系统时总想把方案搞得复杂一些，不这样做就不足以显示自己的水平。其实，开发高手的做法恰恰相反，他们追求的总是那些最简洁明了，最容易理解的设计方案。

## 持续改进设计

敏捷团队依靠变化来获取活力。团队几乎不进行预先设计，因此，不需要一个成熟的初始设计。他们更愿意保持系统设计尽可能的干净、简单，并使用许多单元测试和验收测试作为支援。这保持了设计的灵活性、易于理解性。团队利用这种灵活性，持续地改进设计，以便于每次迭代结束所生成的系统都具有最适合于那次迭代中需求的设计。

在要实现新需求时，你抓住这次机会去改进设计，以便设计对于将来的同类变化具有弹性，而不是设法去给设计打补丁。但请注意，**你不是一开始设计该模块时就试图预测程序将如何变化。相反，你是以最简单的方式编写的。直到需求最终确实变化时，你才修改模块的设计，使之对该种变化保持弹性。**

## 过度设计

所谓过度设计（over-engineering），是指代码的灵活性和复杂性超出所需。有些程序员之所以这样做，是因为他们相信自己知晓系统未来的需求。他们推断，最好今天就把方案设计得更灵活、更复杂，以适应明天的需求。这听上去很合理，但是别忘了，这需要你未卜先知。

如果预计错误，浪费的将是宝贵的时间和金钱。花费几天甚至几星期对设计方案进行微调，仅仅为了增加过度的灵活性或者不必要的复杂性，这种情况并不罕见，而且这样只会减少用来添加新功能、排除系统缺陷的时间。

如果预期中的需求根本不会成为现实，那按此编写的代码又将怎样呢？删除是不现实的。删除这些代码并不方便，何况我们还指望着有一天它们能派上用场。无论原因如何，随着过度灵活、过分复杂的代码的堆积，你和团队中的其他程序员，尤其是新成员，就得在毫无必要的更庞大、更复杂的代码基础上工作。

过度设计下的代码会影响生产率，因为当其他人接手一个过度设计的方案时，必须先花上一些时间了解设计中的许多微妙之处，然后才能自如地扩展或者维护它。

过度设计总在不知不觉之中出现，许多架构师和程序员在进行过度设计时甚至自己都不曾意识到，而当公司发现团队的生产率下降时，又很少有人知道是过度设计在作怪。

**不要追求“完美”，“足够好” 就行**

软件设计师，尤其是架构师，在评估针对某个问题的解决方案时，会倾向考虑它是否优雅完美。我们在查看设计或实现时，仿佛自己是选美比赛的评委，会立即找出其中的微小瑕疵，而这些瑕疵只须经由一些调整或迭代重构便可消除。对领域模型中是否存在可被移入基类之中的共同属性或函数过于关注；当在多个实现中发现存在重复的服务时，就大声疾呼一定要把它变为web服务；对“从缓存中得到的”查询和非唯一性索引，则指摘不止。

我的建议是：不要屈服于企图使设计或实现达到完美的诱惑！把目的设定在“足够好”就行，当已经达成目标时，就停下来。

究竟什么是“足够好”？“足够好”指的是，剩余的不完美之处，对系统的功能、可维护性或性能不会产生任何有深远意义的影响。架构和设计协调一致；系统的实现正确可用，并符合性能需求；代码整洁简明，文档化良好。还可以做得更好吗？当然可以，但这样已经足够好，所以就到此为止了吧。可以宣布设计胜利完成，然后转入下一个任务了。

在设计和实现上追求完美，会导致过度设计（overdesigned）和模糊混乱的解决方案，最终使系统难以维护。

要注意避免不必要的抽象或复杂性。为什么保持简单会这么困难呢？因为我们在寻求完美的解决方案！架构师无法忍受设计中可感觉到的不完美，定要除之而后快，但正是这种冲动，向已可工作的简洁的解决方案中引入了额外的复杂性！

应用程序开发不是选美大赛，因此，停止吹毛求疵的做法，不要再浪费时间追求尽善尽美。

**复杂度：尽可能简单**

复杂度就是成本。复杂的软件更难于开发，难于测试，难于调试，难于维护——最重要的，难以学习和使用。而复杂所带来的成本，开发时便汹涌而来，部署后更变本加厉。**复杂是bug滋生的温床**，在整个的软件生存期，世界都将不得安宁。所有种种压力将程序员拖进复杂度的泥沼中。

本质的复杂度：太多复杂问题需要复杂的解决方案。仅仅十行的程序，再优雅也无法控制喷气客机。

偶然复杂度：偶然复杂度的产生是因为没有找到实现规定功能集合的最简方法，偶然复杂度可以由良好的设计或重新设计来去除。

选择复杂度：选择复杂度，同某个期望的功能相关联，只能由改变工程的目标来去除。

## 超前代码

程序中的一些代码似乎是在将来的某个时候才会用到的。在猜测程序将来有哪些功能可能被用到这方面，程序员已经声名狼藉了。超前设计（Designing ahead）常常会遭遇很多可预见的问题。

1、对这些“超前设计”的代码而言，需求不可能定义得很完备，这就意味着程序员对于未来需求的猜测很可能是错误的。因此，程序员所进行的“超前编码”的工作最终将会被丢弃。

2、即使程序员对未来需求的前瞻几近完全准确，他也不可能广泛预见未来需求所有的复杂脉络。这些错综复杂的关系将会埋葬程序员的基本设计构思，也就是说“超前设计”的工作也将不得不抛弃。

3、那些使用“超前设计”代码的未来程序员们并不知道自己手中的代码原本是经过“超前设计”的，或许他们会期望这些代码能比实际情况表现得更好。他们假设这些代码的编码、测试和检查与其他代码进行得同样完美。这些程序员耗费了大量时间来让这些所谓的“超前设计”的代码工作起来，到最后却发现这些代码根本没什么用。

4、“超前设计”的代码是画蛇添足，增加了程序的复杂性，带来了额外的测试、修补缺陷等工作量。其整体效应就是拖了项目的后腿。

专家们认为，对未来需求有所准备的办法并不是去编写空中楼阁式的代码，而是尽可能将满足当前需求的代码清晰直白地表现出来，使未来的程序员理解这些代码到底完成了什么功能，没有完成什么功能，从而根据他们的需要进行修改。

**只实现你真正需要的东西**

极限编程Extreme Programming的理论中有一个基本原则：只实现你真正需要的东西，不要去实现你认为需要的东西。因为软件的特点就是变化：环境在变化，用户在变化，产业在变化，技术也在变化。你基于今天的事实做出的预测，往往无法反映几年甚至几个月后的事态发展情况。与其到时候后悔不迭，还不如把精力花在真正现实的软件需求上。

Note：不仅浪费初期开发成本，也浪费维护成本，增加了系统复杂度，后面不得不理解并维护这一段实际没有用到的功能。

## 设计不足

设计不足比过度设计要常见得多。所谓设计不足（under-engineering），是指所开发的软件设计不良。其产生原因有如下几种：

1、程序员没有时间，没有抽出时间，或者时间不允许进行重构；

2、程序员在何为好的软件设计方面知识不足；

3、程序员被要求在既有系统中快速地添加新功能；

4、程序员被迫同时进行太多项目。

随着时间的推移，设计不足的软件将变成昂贵、难以维护甚至无法维护。Brian Foote和Joseph Yoder曾经创造了一种名为Big Ball of Mud（大泥球）的模式语言，他们是这样描述类似软件的：

数据结构的构造很随意，甚至近乎不存在；任何东西都要与其他东西通信；所有重要的状态数据都可能是全局的，在状态信息被隔开的地方，需要通过错综复杂的后端通道杂乱地传递，以绕开系统的原有结构。

变量名和函数名信息量不足，甚至会起误导作用；函数可能使用大量全局变量以及定义模糊的冗长的参数列表；函数本身冗长、费解，完成多项毫无关联的任务；代码重复很多；控制流很难看清，难以找到来龙去脉；程序员的意图几乎无法理解；代码完全不可读，近乎难于破译的天书；代码中有许多经过多个维护者之手不断修修补补留下的明显印记，这些维护者几乎都没有理解自己的修补会造成怎样的后果。

虽然你开发的系统也许不会这么恐怖，但是很可能也曾经有过设计不足的时候。迅速使代码运行起来是压倒一切的要求，而这往往伴随着巨大的压力，使我们无法改进既有代码的设计。

长期的设计不足，会使软件开发节奏变成“快，慢，更慢”，可能造成这样的后果：

1、系统的1.0版很快就交付了，但是代码质量很差；

2、系统的2.0版也交付了，但质量低劣的代码使我们慢了下来；

3、在企图交付未来版本时，随着劣质代码的倍增，开发速度也越来越慢，最后人们对系统、程序员乃至使大家陷于这种境地的整个过程都失去了信心；

4、到了4.0版时或者之后，我们意识到这样肯定不行，开始考虑推倒重来。

这种事情在我们的行业里司空见惯。它的代价非常高昂，而且会极大地降低企业本应具备的竞争力。

## 可配置

只要可能，就建立机制把策略决定权交给用户。这种方式产生的程序往往功能强大，专家用户用起来会非常顺手；但它所产生的接口往往选项过多，并且配置文件像杂草一样疯长，从而彻底打击了新手和一般用户。

增加不必要的选项会产生诸多不良后果。其中最不易察觉但最严重的后果是对测试覆盖率的影响。

除非做得非常仔细，否则增加一个开/关配置选项就会使测试量加倍。既然在实践中从来没有人完成双倍测试量，那么实际影响就是减少了特定配置获得的测试量。增加十个选项会产生1024倍测试量，所以不要多久就要讨论可靠性问题了。

## 可读性

维护如此重要而成本又如此高昂：每个程序员都应牢记这一点，开发的程序首先是给人——将来阅读维护源码的人看的，而不是给执行代码的计算机看的。因此，我们应想尽一切办法来提高程序的可读性，只有这样，程序的可扩展性、可维护性才会有本质的提高。

一提到程序的可读性，很多程序员就会想到要在代码中多写注释。但在大多数时候，好的标识符命名方式、规范的代码格式、简洁的算法等都有助于改进代码的可读性。甚至某些职责清晰、内聚性强的代码无需注释就很容易被人理解，这种代码也常被称做自注释的代码。显然，在条件许可的情况下，尽量编写具有自注释功能的代码，这是一件利人利己的大好事。

**清晰原则：清晰胜于技巧**

在选择算法和实现时就应该考虑到将来的可扩展性，而为了取得程序一丁点的性能提升就大幅度增加技术的复杂性和晦涩性，这个买卖做不得，不仅仅是因为复杂的代码容易滋生bug，也因为它会使日后的阅读和维护工作更加艰难。

相反，优雅而淸晰的代码不仅不容易崩溃——而且更易于让后来的修改者立刻理解。

## 模块原则：使用简洁的接口拼合简单的部件

正如Brian Kemighan曾经说过：计算机编程的本质就是控制复杂度，排错占用了大邰分的开发时间。

要编制复杂软件而又不至于一败涂地的唯一方法就是降低其整体复杂度，用淸晰的接口把若干简单的模块组合成一个复杂软件。这样多数问题只会局限于某个局部，那么就有希望对局部进行改进而不至牵动全身。

## 健壮原则：健壮源于透明与简洁

健壮性指软件不仅能在正常情况下运行良好，而且在超出设计者设想的意外条件下也能够运行良好。

大多数软件禁不起磕碰，毛病很多，就是因为过于复杂，很难通盘考虑。如果不能够正确理解一个程序的逻辑，就不能确信其是否正确，也就不能在出错的时候修复它。

这也就带来了让程序健壮的方法，就是让程序的内部逻辑更易于理解。要做到这一点主要有两种方法：透明化和简洁化。

软件的透明性就是指一眼就能够看出来是怎么回事。如果“怎么回事”不算复杂，即人们不需要绞尽脑汁就能够推断出所有可能的情况，那么这个程序就是简洁的。程序越简洁，越透明，也就越健壮。

模块性（代码简朴，接口简洁）是组织程序以达到更简洁目的的一个方法。

## 臭味和原则

设计的腐化是一种症状，是可以主观（如果不能客观的话）进行量度的。腐化常常是由于违法了设计原则中的一个或多个所导致的。例如，僵化性常常是由于对开放－封闭原则OCP不够关注的结果。

**开发团队应该运用相应的设计原则来去除腐化，但当软件还没出现腐化时不应该应用这些原则**。仅仅因为是一个原则就无条件的去遵循它的做法是错误的，这些原则不是可以随意在系统中到处喷洒的香水。**过分遵循这些原则会导致不必要的复杂性**（Needless Complexity）的设计臭味，变成另一种腐化。

## 优化原则：雕琢前先得有原型，跑之前先学会走

原型设计最基本的原则最初来自于Kemighan和Plauger所说的“90%的功能现在能实现，比100%的功能永远实现不了强”。做好原型设计可以帮助你避免为蝇头小利而投入过多的时间，

由于略微不同的一些原因，Donald Knuth（程序设计领域中屈指可数的经典著作之一《计算机程序设计艺术》的作者）广为传播普及了这样的观点：“过早优化是万恶之源”。

还不知道瓶颈所在就匆忙进行优化，这可能是唯一一个比乱加功能更损害设计的错误。从畸形的代码到杂乱无章的数据布局，牺牲透明性和简洁性而片面追求速度、内存或者磁盘使用的后果随处可见。滋生无数bug，耗费以百万计的人时——这点芝麻大的好处，远不能抵消后续排错所付出的代价。

过早的局部优化实际上会妨碍全局优化（从而降低整体性能）。在整体设计中可以带来更多效益的修改常常会受到一个过早局部优化的干扰，结果，出来的产品既性能低劣又代码过于复杂。

在Unix世界里，有一个非常明确的悠久传统：先制作原型，再精雕细琢。优化之前先确保能用。或者：先能走，再学跑。极限编程宗师Kent Beck从另一种不同的文化将这一点有效地扩展为：先求运行，再求正确，最后求快。

所有这些话的实质其实是一个意思：先给你的设计做个未优化的、运行缓慢、很耗内存但是正确的实现，然后进行系统地调整，寻找那些可以通过牺牲最小的局部简洁性而获得较大性能提升的地方。

借助原型化找出哪些功能不必实现，有助于对性能进行优化；那些不用写的代码显然无需优化。

## 建立试验性原型

有些时候，除非你更好地了解了一些实现细节，否则很难判断一种设计方法是否奏效。

有一种技术能够低成本地解决这个问题，那就是建立试验性原型。

## 要做多少设计才够

有些时候，编码之前只制订出系统架构的一个最小梗概。而在另一些时候，开发团队会把设计做得非常详细，使编码变成了一种近乎机械式的工作。那么在编码前的时间到底需要做多少设计呢？

对于实施正式编码阶段前的设计工作量和设计文档的正规程度，很难有个准确的定论。有很多因素，如团队的经验、系统的预定寿命、想要得到的可靠度、 项目的规模和团队的大小等等都需要考虑进去。表5-2总结了这些因素是如何影响设计活动的。

**设计文档的正规化以及所需的细节层次**

因素 开始构建之前的设计所需的细化程度 文档正规程度

设计或构建团队在应用程序领域有很丰富的经验 低 低

设计或构建团队有很丰富的经验，但是在这个应用枉序領域缺乏经验 中 中

设计或构建团队缺乏经验 中到高 低到中

设计或构建团队人员变动适中或者较高 中 —

应用栈序是安全攸关的 高 高

应用程序是使命攸关的 中 中到高

項目是小空的 低 低

項目是大型的 中 中

软件預期的生命用期很短（几星期或者几个月） 低 低

软件預期的生命用期很长 (几个月或者几年） 中 中

对于一个特定的项目来说，这其中的两个或者更多因素可能起着关键作用。在某些情况下，从这些因素也可能得出有矛盾的建议。比如，你带一支经验非常丰富的团队开发对安全性要求很高的软件，这时，你可能宁愿釆用更高的设计细化程序以及更正规的设计文档。在这种情况下，你需要去评估每一项因素的重要性，然后根据结果来做出重要性的权衡。

一些项目因太过于专注对设计进行文档化而导致失败。Gresham法则是这么说的，“程序化的活动容易把非程序化的活动驱逐出去”(Simon 1965)。过早地去润色设计方案就是这一法则所描述的例子。我宁愿看到有80%的设计精力用于创建和探索大量的备选设计方案，而20%的精力用于创建并不是很精美的文档，也不愿看到把20%的精力花在创建平庸的设计方案上，而把80%的精力用于对不良的设计进行抛光润色。

我从来没有遇到过哪个人愿意阅读17000页的文档，如果真有这样的人，我想把他杀掉，让他从人类基因库中消失。—Joseph Costello

# 软件质量

## 质量属性之间的关系

外在特性指的是该产品的用户所能够感受到的部分，质量的外在特性是用户关心的唯一软件特性。用户只会关心软件是否容易使用，而不会关心对于程序员来说修改起来是否容易。他们关心软件是否能正确运行，而不关心里面的代码是否可读，或者是否有良好的结构。这些特性中，有一部分是互相重叠的，但它们都有不同的含义，并且在不同的场合下，重要性也有所不同。

程序员除了关心软件质量的外在特性之外，还要关心它的内在特性。同外在质量特性相仿，一些内在质量特性之间也是有所重叠的，同样，它们中的每一个在特定场合有着不同的重要性。

内在和外在特性并不能完全割裂开来，在某些层次上，内在特性会影响某些外在特性。一个无法从系统内部理解或者维护的软件，其缺陷也是很难修正的，而这又会影响正确性和可靠性等外在特性。一个刻板的软件无法根据用户需要进行改进，这就影响了可用性这一外在特性。关键在于，某些特性强调软件要让用户用起来方便，而另一些特性则强调软件要让程序员维护起来方便。

众多质量属性需求之间往往会有冲突，我们必须权衡。在软件实践中，人们对软件质量属性的重视不够，这己成为很多项目遭受失败的常见根源之一。但作为软件架构师，应该对质量属性需求的折衷有系统性的把握。

下图为经典文献中给出的是其中12种质量属性之间的关系，值得架构师研究和借鉴。

由表中可以看出，不同的质量属性之间可能存在3种关系：1、无影响；2、促进；3、阻碍。

开发期质量属性可以认为是软件项目的内部属性，这些指标的良莠深刻影响着开发人员、维护人员、测试人员和开发管理人员的工作。可测试性、可重用性、可维护性、可扩展性和可移植性这些开发期质量属性之间的关系总结如下：

1、除了不存在影响关系之外，绝大多数运行期质量属性之间存在促进和被促进的关系。例如，可重用性越好，可测试性、可维护性、可扩展性和可移植性就越好；

2、唯一的例外是，可移植性可能对可维护性造成负面影响；

3、很多属性之间是“相互”促进的。例如，可测试性和可维护性是相互促进的关系；

4、可读性好的软件维护起来就会容易一些，扩展性好的软件复用时往往也会非常简单。

5、但也有不少属性之间的促进关系是“单向”的。例如，可移植性促进可测试性，但可测试性却不能促进可移植性（也不会降低可移植性）。

相比之下，和运行期质量属性相关的制约关系就显得比较复杂。但经过仔细分析，我们还是能把握住其中较为明显的规律：

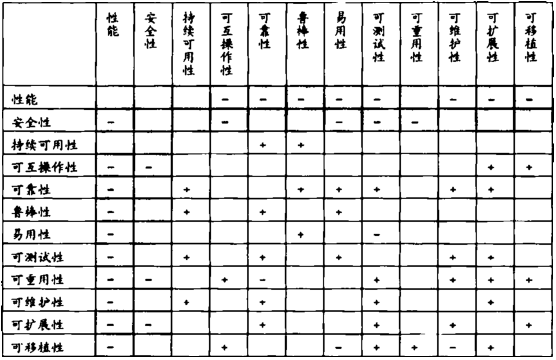
1、非常典型的是，性能和安全性这两个运行期质量属性和其他所有质量属性之间都是抵触关系。

性能和安全性要求高，势必会为其他质量指标带来负面影响。例如，此消彼涨，为了构造更加安全的系统，在系统的易用性、可测试性和可维护性方面就“可能”要做出让步；

2、反过来，其他质量属性指标也会对性能和安全性造成负面影响。例如，某些嵌入式系统对性能要求非常高，但可测试性和可靠性方面也必须达到极高要求，这时就需要架构师非常高超的折衷技能，以达到总体最优的效果。

3、那么，性能和安全性之间呢？很简单，如表中所示，极高的安全性要求将使我们不得不降低性能要求；

4、除了性能和安全性之外，持续可用性、可互操作性、可靠性、鲁棒性和易用性等运行期质量属性相关的制约关系比较复杂，需要更多经验。



## 运行期质量属性

### 性能(Performance)

### 安全性(Security)

软件系统所承担的交易的商业价值极高，系统的安全性非常重要。

### 易用性(Usability)

指软件系统易于学习和使用的程度。

软件产品在正确性方面做得很好，在软件的有效使用方面却做得不够。如果一个软件产品对于具有代表性的预期客户来说，使用起来太复杂、 太困难、太笨拙或者令人痛苦，那么，就不会有人去使用该软件。

我们所说的用户通常是指系统的最终用户。但是，在一个大型的、设计成层次结构的系统中，组件的客户很可能是别的组件，来自客户（包括别的开发者）的早期反馈对于保证可用性是非常重要的。

### 可用性 (Availability)

持续可用性指系统长时间无故障运行的能力。

### 可靠性(Reliability)

软件系统在一定的时间内无故障运行的能力，应该有很长的平均无故障时间。

### 可伸缩性(Scalability)

软件必须能够在用户的使用率、用户的数目增加很快的情况下，保持合理的性能。只有这样，才能适应用户的市场扩展的可能性。

例如当业务量较小时，软件系统运行在一台服务器上，当业务量增大时，可以通过增加服务器或增加单台服务器上所运行软件系统的个数来提高性能，而无需对软件系统本身进行编程级的修改。

### 互操作性(Interoperability)

本软件系统与其他系统交换数据和相互调用服务的难易程度。

### 鲁棒性/健壮性/容错性(Robustness)

鲁棒性是指软件系统在以下情况下仍能够正常运行的能力：用户进行了非法操作；相连的软硬件系统发生了故障，以及其他非正常情况。

系统在接收无效输入或者处于压力环境时继续正常运行的能力。

## 开发期质量属性

### 易理解性(Understandability)

尤指设计被开发人员理解的难易程度。

指在系统组织和细节语句的层次上理解整个系统的难易程度。与可读性相比，可理解性对系统提出了更高的内在一致性要求。

### 可读性(Readability)

指阅读并理解系统代码的难易程度，尤其是在细节语句的层次上。

可读性对程序的以下方面都有正面影响：

1、可理解性

2、容易复查

3、错误率

4、调试

5、可修改性

6、开发时间——上述因素之综合

7、外在质量---上述因素之综合

如果你能轻松阅读代码，确保该代码能工作也会更容易，这就是应该写可读性好的代码的充分理由。不仅如此，代码在复审过程中也要阅读它；你或别人修改错误时也要读；改动代码时还要读；当别人利用你代码的一部分编写类似的程序时也要读。

使代码可读性好并非是开发过程中的可有可无部分，为了节省编写代码的时间而不顾阅读它的时间，是不经济的。

### 可扩展性(Extensibility)

在新技术出现的时候，一个软件系统应当允许导入新技术，从而对现有系统进行功能和性能的扩展。

为适应新需求或需求的变化为软件增加功能的能力。我们在实际工作中，经常将可扩展性称为灵活性。

### 可重用性(Reusability)

重用软件系统或其一部分的能力的难易程度。

指系统的某些部分可被应用到其他系统中的程度，以及此项工作的难易程度。

### 可测试性(Testability)

对软件测试以证明其满足需求规约的难易程度。在实际工作中主要指进行单元测试、插桩测试等的难易程度。

### 可维护性(Maintainability)

指是否能够很容易对系统进行修改，改变或者增加功能，提高性能，以及修正缺陷。

软件系统的维护包括两方面，一是排除现有的错误，二是将新的软件需求反映到现有系统中去。一个易于维护的系统可以有效地降低技术支持的花费。

便于在系统出现故障时及时方便地找到产生故障的原因和源代码位置，并能方便地进行局部修改、切割；（可维护性与运行可管理性不同）

可维护：可以很方便地修改功能，容易定位bug，功能隔离较好，不会牵一发而动全身。

为了达到下列三种目的之一，而定位修改点并实施修改的难易程度：修改Bug、增加功能、提高质量属性。

如果作者以外的其他人能够顺利地理解和修改软件，则这个软件就是可维护的。可维护性不仅要求代码能够运行；还要求代码能够遵循清晰原则，并且和人以及计算机成功沟通。

### 可移植性(Portability)

指为了在原来设计的特定环境之外运行，对系统所进行修改的难易程度。

不同客户端、应用服务器、数据库管理系统：如果潜在的客户使用的客户端可能使用不同的操作系统或浏览器，其可移植性必须考虑客户端程序的可移植性，或尽量不使业务逻辑放在客户端；数据处理的业务逻辑放在数据库管理系统中会有较好的性能，但如果客户群中不能确定使用的是同一种数据库管理系统，则业务逻辑就不能数据库管理系统中；

达到可移植性一定要注重标准化和开放性：只有广泛采用遵循国际标准，开发出开放性强的产品，才可以保证各种类型的系统的充分互联，从而使产品更具有市场竞争力，也为未来的系统移植和升级扩展提供了基础。

# 设计质量

## 面向对象设计目标

面向对象系统的分析和设计实际上追求的就是两点，一是高内聚 （Cohesion），而是低耦合 （Coupling）。无论是OO 中的封装、继承、多态，还是设计模式的原则和实例都是在为了这两个目标努力。

按照软件工程的理论，面向对象的设计要解决的核心问题就是**可维护性**和**可复用性**。尤其是可维护性，它是影响软件生命周期重要因素，通常情况下，**软件的维护成本远远大于初期开发成本**。

一个可维护性很差的软件设计，形成的原因主要有这么几个：过于僵硬、过于脆弱、复用率低或者黏度过高。相反，一个好的系统设计应该是灵活的、可扩展的、可复用的、可插拔的。在20世纪80到90年代，很多业内专家不断探索面向对象的软件设计方法，陆续提出了一些设计原则。这些设计原则能够显著地提高系统的可维护性和可复用性，成为了我们进行面向对象设计的指导原则：

1、 单一职责原则SRP

2、 开-闭原则OCP

3、 里氏代换原则LSP

4、 依赖倒转原则DIP

5、 接口隔离原则ISP

其中，“开-闭”原则是面向对象的可复用设计的基石，其他设计原则是实现“开-闭”原则的手段和工具。

### 可维护性

一个维护中的软件是不断再生的软件。

修改需求，流程、界面等细节变更，修改积累，往往导致原有软件结构混乱，软件腐烂（可维护性差，可升级性差）。

系统设计师：辩解用户需求变化多；

**可维护性设计目标**

1、可扩展性extensibility；

加入新模块，不影响原有模块

2、灵活性flexibility；

修改一个模块，不影响其他模块

3、可插入性pluggability

可以很容易的去掉一个模块，更换一个模块，加入一个新模块，而不影响其他模块。

### 可复用性

复用的重要性：1、较高的效率；2、较高的软件质量；3、较好的可维护性。

传统的复用：1、复制、剪切代码；2、算法的复用；3、数据结构的复用；

传统复用，往往会破坏软件的可维护性。AB共用C，如果A要修改C，B不允许。

**可复用性（reuse）设计**

一、面向对象设计的复用：

抽象、封装、继承、多态。

复用的焦点集中在含有宏观商业逻辑的抽象层面；而不局限于函数和算法。

二、对可维护性的支持

1、可扩展性extensibility；

封装，继承，多态

由开闭原则、里氏代换原则、依赖倒转原则、组合/聚合复用保证

2、灵活性flexibility；

复用，相对独立，与其他模块松耦合。该模块修改后，不会传递到其他模块。

由开闭原则、迪米特法则、接口隔离原则保证。

3、可插入性pluggability

复用后，新类替代旧类，容易实现。

由开闭原则、里氏代换原则、组合/聚合服用原则和依赖倒转原则保证。

## 软件腐化的原因

什么激发了软件的腐化？答案是需求的变化。由于需求没有按照初始设计预见的方式进行变化，从而导致了设计的退化。通常，改动都很急迫，并且进行改动的开发人员对原始的设计思路并不熟识。因而，虽然对设计的改动可以工作，但是它却以某种方式违反了原始的设计。随着改动的不断进行，这些违反不断地积累，设计开始出现臭味。

然而，我们不能因为设计的退化而责怪需求的变化。作为开发人员，我们对需求变化有非常好的了解，大多数人都认识到需求是项目中最不稳定的因素。如果我们的设计由于持续、大量的需求变化而失败，那就表明我们的设计和实践本身是有缺陷的。我们必须要设法找到一种方法，使得设计对于变化具有弹性，并且应用一些实践来防止设计腐化。具体见“敏捷团队不允许软件腐化”。

## 拙劣架构

### 不良影响

不良的架构会产生深远的影响和严重的反弹，缺少预见性和架构设计，会导致下面的问题：

低品质的软件和漫长的版本发布周期；

系统没有弹性，不能够适应变更或添加新的功能；

无处不在的代码问题；

员工问题（压力大、士气低、跳槽等)；

大量混乱的公司内部政治；

公司不能成功；

许多痛苦和面对代码深夜加班。

### 不可理解

导致了一个很难理解的软件系统，实际上几乎不可能修改。新加入项目的团队成员会被复杂性惊呆， 不能够搞清楚状况。

坏的设计确实会招致在它上面叠加坏的设计（**实际上它就是迫使你那样做**），因为没有一种明智的方式可以扩展该设计。在所有能解决手上工作的方法之中，阻力最小的总会被采用，没有明显的办法来修复这些结构问题，所以只要能减少麻烦，就会扔进去新的功能。

注意：重要的是要保持软件设计的品质。坏的架构设计会招致更坏的架构设计。

### 缺乏内聚

系统的组件完全没有内聚性。每个组件本来都应该有一个定义良好的角色，但是它们却包含了一堆杂乱的、不一定相关的功能。这使我们很难确定组件存在的原因，也很难弄明白系统中已经实现了哪项具体的功能。

很自然，这让缺陷修复成为了一场噩梦，严重地影响了软件的品质和可靠性。

功能和数据都放在了系统中错误的地方。许多你认为是“核心服务”的部分却没有在系统的核心部分实现，而是由边远的模块来模拟实现（非常痛苦并且代价很大）。

### 不必要的辑合

“大都市”没有清晰的分层。模块之间的依赖关系不是单向的，耦合常常是双向的。组件A会到达组件B的内部，目的是完成它的一项任务。在其他的地方，组件B又通过硬编码调用了组件A。系统没有最底层，也没有控制中心。它是整体式的一大块软件。这意味着系统的各部分之间耦合非常紧密，你想启动系统骨架就不得不创建所有的组件。单个组件的任何改变都会波及其他组件，需要修改许多侬赖它的组件。孤立地看代码组件没有任何意义。

这使得低层次的测试不能够进行。不仅是代码层次的测试不可能进行，而且组件层次的集成测试也不能够创建，因为每个组件都依赖于几乎所有其他组件。当然，在公司中，测试从来也不具有很高的优先级（我们根本没有时间来做这种测试），所以这“不成为问题”。不必说，这个软件不太可靠。

注意：好的设计考虑到内部组件连接的连接机制和连接数（以及连接性质）。系统的单个部分应该能够独立存在。紧耦合将导致不可测试的代码。

## 软件设计的腐化

以下是一些拙劣设计的症状，当软件出现下面任何一种气味时，就表明软件正在腐化。

### 拙劣设计的症状

脆弱性（Fragility）：设计易于遭到破坏D

牢固性（Iuimobility）：设计难以重用。

粘滞性（Viscosity）：难以做正确的事情。

不必要的复杂性（NeedlessComplexity）：过分设计。

晦涩性（Opacity）：混乱的表达。

这些症状在本质上和代码的“臭味”（smell）相似，但是它们所处的层次稍高一些。它们是遍及整个软件结构的臭味，而不仅仅是一小段代码。

### 僵化性Rigidity

僵化性是指难以对软件进行改动，即使是简单的改动，因为每个改动都会迫使许多对系统其他部分的其他改动。添加一个新功能，很困难，牵扯多，周期长。

出现这样的问题是由于软件模块之间的耦合过于紧密，任何变化都会像涟漪一般引起依赖模块的其他变化。单一的改动会导致有依赖关系的模块中的连锁改动，必须要改动的模块越多，设计就越僵化。

某些功能需求看起来是简单的改动，但是开发人员实际进行改动时，会发现有许多改动带来的影响自己并没有预测到。他们发现要在庞大的代码中搜寻这个变动，并且要更改的模块数目也远远超出预计。

### 脆弱性Fragility

与僵化密切相关的另一个症状是脆弱，软件的每一次变化会涉及到很多不相干的地方。对系统的改动会导致系统中和改动的地方在概念上无关的许多地方出现问题。

与僵化共存，对系统的改动会导致系统中不相干地方出现问题，而且难以预料。**一个地方修改，导致其他地方出现故障，甚至不可预期。（开发一个需求，衍生n个bug，这点不陌生）**

脆弱性是指，在进行一个改动时，程序的许多地方就可能出现问题。常常是，出现新问题的地方与改动的地方并没有概念上的关联。要修正这些问题就又会引出更多的问题，从而使开发团队就像一只不停追逐自己尾巴的狗一样（忙得团团转）。

随着模块脆弱性的增加，改动会引出意想不到的问题的可能性就越来越大。这些模块需要不断地修补——它们从来不会被从错误列表中去掉，开发人员知道需要对它们进行重新设计（但是谁都不愿意去面对重新设计中的难以琢磨性），你越是修正它们，它们就变得越糟。

### 低复用

很难解开系统的纠结，使之成为一些可在其他系统中重用的组件。

**不可移植性：**不可移植性是指应用程序在从其他项目或本项目的其他部分复用类方面的无能。软件工程师找到了一个类，其功能和特点与他们所需要的类似。但是，往往是这样：这个类还有一些不需要的功能，或者类处理的问题稍有区别。进行一番分析后，工程师发现从这个类中分离出所需的功能需要太多的工作量，复用只是一个不可行的建议。结果是，干脆重写这个类，而不再复用了。

### 牢固性Immobility

很难解开系统的纠结，使之成为一些可在其他系统中重用的组件。

牢固性是指，设计中包含了对其他系统有用的部分，但是要把这些部分从系统中分离出来需要的努力和风险是巨大的。

### 粘滞性Viscosity

实施保持设计的更改的难易程度。如果实现保持设计的方法比较困难，就说明该设计的粘性高。

修改，保持原有设计；修改，破坏原有设计。如果总体设计，总是导致后者的实现比前者容易，就是黏度过高。黏度高过会诱导程序员采用错误的方案，并导致软件逐渐腐烂。

粘滞性有两种表现形忒：软件的粘滞性和环境的粘滞性。

当面临一个改动时，开发人员常常发现会有多种改动的方法。其中，一些方法会保持设计；而另外一些会破坏设计（也就是生硬的手法）。当那些可以保持系统设计的方法比那些生硬手法更难应用时，就表明设计具有高的粘滞性。我们希望在软件设计中，可以容易地进行那些保持设计的变动。

当开发环境迟钝、低效时，就会产生环境的粘滯性。例如，如果编译所花费的时间很长，那么开发人员就会被引诱去做不会导致大规模重编译的改动，即使那些改动不再保持设计。如果源代码控制系统需要几个小时check in仅仅几个文件，那么开发人员就会被引诱去做那些需要尽可能少check in的改动，而不管改动是否会保持设计。

无论项目具有哪种粘滞性，都很难保持项目中的软件设计。我们希望创建易于保持设计的系统和项目环境。

### 不必要的复杂性Needless Complexity

设计中包含有不具任何直接好处的基础结构，过度设计。

如果设计中包含当前没有用的组成部分，它就含有不必要的复杂性。**当开发人员预测需求的变化，并在软件中放置了处理潜在变化的代码时，常常会出现这种情况（这点也正是我要强调的）。**起初，这样看起来是一件好事。毕竟，为将来的变化做准备会保持代码的灵活性，而且可以避免以后再进行痛苦的改动。

糟糕的是，结果常常正好相反。为过多的可能性作准备，致使设计中含有绝不会用到的结构，从而变得混乱。一些准备也许会带来回报，但是更多的不会。期间，设计背负着这些不会用到的部分，使软件变得复杂，而且难以理解。

### 不必要的重复Needless Repetition

设计中包含有重复的结构，而该重复的结构本可以使用单一的抽象进行统一。出于自己的目的复制相同的代码，然后各自修改。

想复用A模块，但A牵扯BCDE…，牵扯太多，不敢被复用；不同模块中的重复功能，重复实现，发现问题，重复修改。

复制（Copy）和粘贴（paste）也许是有用的文本编辑（text-editing）操作，但是它们却是灾难性的代码编辑（code-editing）操作。时常，软件系统都是构建于众多的重复代码片断之上。

当系统中有重复代码时，对系统进行改动会变得困难。在一个重复的代码体中发现的错误必须要在每个重复体中一一修正。不过，由于每个重复体之间都有细微的差别，所以修正的方式也不总是相同的。

### 晦涩性（Opacity）

很难阅读、理解。没有很好地表现出意图。

晦涩性是指模块难以理解。代码可以用清晰、富有表现力的方式编写，或者可以用晦涩、费解的方式编写。代码随着时间而演化，往往会变得越来越晦涩。为了使代码的晦涩性保持最低，就需要持续地保持代码淸晰并且富有表现力。

## 设计范畴的特征

衡量软件设计质量的首要标准是该设计方案是否能满足软件的功能性需求。不过，仅仅满足功能性需求的设计不见得就是好的设计。除了功能性需求以外，还有很多衡量软件设计质量的标准，例如：

### 最小的复杂度（Minimal complexity）

设计的首要目标就是要让复杂度最小。要避免做出“聪明的”设计，因为“聪明的”设计常常都是难以理解的，应该做出简单易于理解的设计。如果你的设计方案不能让你专注于程序的一部分时安心地忽视其他部分的话，这一设计就没有什么作用了。

### 易于维护（Ease of maintenance)

软件维护（主要指软件错误的修改、遗漏功能的添加等）的难易程度。

易于维护意味着在设计时为做维护工作的程序员着想。请时刻想着这些维护程序员可能会就你写的代码而提出的问题。把这些程序员当成你的听众，进而设计出能自明的（self-explanatory）系统来。

### 松散耦合（loose coupling）

松散耦合意味着在设计时让程序的各个组成部分之间关联最小。通过应用类接口中的合理抽象、封装性及信息隐藏等原则，设计出相互关联尽可能最少的类。减少关联也就减少了集成、测试与维护时的工作量。

### 可扩展性（extensibility）

软件面对变化的需求时，进行功能或性能扩展的难易程度。

可扩展性是说你能增强系统的功能而无须破坏其底层结构。你可以改动系统的某一部分而不会影响到其他部分。越是可能发生的改动，越不会给系统造成什么破坏。

### 可重用性（reusability）

可重用性意味着所设计的系统的组成部分能在其他系统中重复使用。

### 高扇入（high fan-in）

高扇入就是说让大量的类使用某个给定的类。这意味着设计出的系统很好地利用了在较低层次上的工具类（utility classes）。

### 低扇出（low fan-out）

低扇出就是让一个类里少量或适中地使用其他的类。 高扇出（超过约7个）说明一个类使用了大量其他的类，因此可能变得过于复杂。研究发现，无论考虑某个子程序调用其他子程序的量，还是考虑某个类使用其他类的量，低扇出的原则都是有益的。

### 可移植性（portability）

可移植性是说应该这样设计系统，使它能很方便地移植到其他环境中。

### 精简性（leanness）

精简性意味着设计出的系统没有多余的部分。伏尔泰曾说，一本书的完成，不在它不能再加入任何内容的时候，而在不能再删去任何内容的时候。在软件领域中，这一观点就更正确，因为任何多余的代码也需要开发、复审和测试，并且当修改了其他代码之后还要重新考虑它们。软件的后续版本也要和这些多余代码保持向后兼容。要问这个关键的问题：“这虽然简单，但把它加进来之后会损害什么呢？”

### 层次性（stratification）

层次性意味着尽量保持系统各个分解层的层次性，设计出来的系统能在任意层次上观察而不需要进入其他层次。

举例来说，假设你正在编写一个新系统，其中用到很多设计不佳的旧代码，这时你就应该为新系统编写一个负责同旧代码交互的层。在设计这一层时，要让它能隐藏旧代码的低劣质量，同时为新的层次提供一组一致的服务。这样，你的系统的其他部分就只需与这一层进行交互，而无须直接同旧代码打交道了。在这个例了中，层次化设计的益处有：1、它把低劣代码的烂泥潭禁闭起来；2、如果你最终能抛弃或者重构旧代码，那时就不必修改除交互层之外的任何新代码。

### 标准技术（Standard techniques）

—个系统所依赖的外来的、古怪的东西越多，别人在第一次想要理解它的时候就越是头疼。要尽量用标准化的、常用的方法，让整个系统给人一种熟悉的感觉。

### 内聚度、耦合度

两个更为具体的衡量标准：**内聚度、耦合度**。如果一个软件的内聚度和耦合度都符合要求，它也就自然具备了比较好的可复用性、可扩展件和可维护性。"低耦合、高内聚"是所有优秀软件的共同特征。

**内聚度：**表示一个模块、类或函数所承担职责的自相关程度。内聚度高的模块通常很容易理解，很容易被复用、扩展和维护。

有一个简单的方法可以粗略地判断系统单元的内聚度：如果个函数可以用简单的"动词十名词"的形式，如"LoadFile"、"RefreshWindow"来命名，或者，如果一个类可以用准确的名词，如"Folder"、"Mouse"来命名，那么，这样的函数或类多半都是内聚度较高的系统单元；反之，如果函数或类的名字必须包含"和(And)"、"或(Or)"等字样才能准确反映其功能特性的话，这些函数和类的内聚度就一定不高。

**耦合度：**表示模块和模块之间、类和类之间、函数和函数之间关系的亲密程度。耦合度越高，软件单元间的依赖性也就越强，软件的可维护性、可扩展性和可复用性就会相应地降低。

在结构化程序设计语言中，如果两个函数访问了同一个全局变量，它们之间就具有了非常强的搞合度；如果它们都没有访问全局变量，它们彼此的耦合度就由二者互相调用时传递的信息量来决定。函数调用时，函数参数包含的信息越多，函数和函数之间的稿合度就较大。

在面向对象的程序设计语言中，类与类之间的祸合度由它们为了完成自己的职责而必须相互发送的消息及消息的参数来决定。

# 实体（类、模块、函数等）设计原则

## 单一职责原则SRP

### 定义

Robert.C Martin给出了一个著名的定义：所谓一个类的一个职责是指引起该类变化的一个原因。

简述：就一个类而言，应该仅有一个引起变化的原因。

单一职责不是说一个类就只有一个方法，而是单一功能。

类的职责主要包括两个方面：数据职责和行为职责，数据职责通过其属性来体现，而行为职责通过其方法来体现。

### 优缺点

**优点：**

优良的系统设计，强调模块间保持低耦合、高内聚的关系，所以面向对象的第一个设计原则就是：单一职责原则。单一职责原则是实现高内聚、低耦合的指导方针，它揭示了实现内聚性和耦合性的基本途径就是提高内聚性。

如果一个类承担的职责过多，那么这些职责就会相互依赖，一个职责的变化可能会影响另一个职责的履行。单一职责原则把相同的职责进行聚合，避免把相同的职责分散到不同的类之中，这样就可以控制变化，把变化限制在一个地方，防止因为一个地方的变动，引起更多地方的变动的“涟漪效应”。

单一职责原则实际上降低了对象之间的耦合，避免一个类承担过多的职责。

### 违背后果

一个类（或者大到模块，小到方法）承担的职责越多，它被复用的可能性越小。

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，当其中一个职责变化时，可能会影响其他职责的运作。这种设计会导致脆弱的设计，当变化发生的时候，设计会遭到意想不到的破坏。

如果一个类具有一个以上的职责，那么就会有多个不同的原因引起该类变化，而这种变化将影响到该类不同职责的使用者（不同用户）：

1、如果一个职责使用了外部类库，则使用另外一个职责的用户却也不得不包含这个未被使用的外部类库。

2、某个用户由于某个原因需要修改其中一个职责，另外一个职责的用户也将受到影响，他将不得不重新编译和配置。

这违反了设计的开闭原则。

### 如何划分职责

单一职责原则是最简单但又最难运用的原则，需要设计人员发现类的不同职责并将其分离，而发现类的多重职责需要设计人员具有较强的分析设计能力和相关重构经验。

怎么能知道该拆分还是不应该拆分呢？原则很简单：需求决定。

以“示例：计算器”为例，如果你所需要的计算器，永远都没有外观和处理器变动的可能性，那么就应该把它抽象为一个整体的计算器；如果你所需要的计算器，外壳和处理器都有可能发生变动，那么就必须把它拆离为外壳和处理器。只能有一个原因可能引起计算器的变化。

怎么判断一个职责是不是一个对象的呢？你试着让这个对象自己来完成这个职责，比如：“书自己阅读内容”，阅读的职责显然不是书自己的。

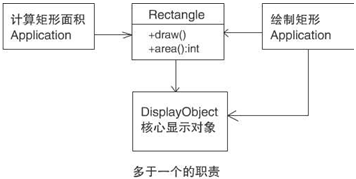
维护类的简单性，类不应承担一个以上令其变化的原因，否则应考虑分离并重新构造类，但如果应用的变化方式总是导致类中的职责同时变化，就没必要分离他们。实际上，分离他们就会具有不必要的**复杂性臭味**。

分离的原则不是教条性的，需要应实际需求而定。在没有需求变化征兆的情况下，应用SRP或其他原则是不明智的，因为这样会使系统变得很复杂，系统由一堆细小的颗粒组成，这纯属于没事找抽。

### 示例

**示例1：Retangle类**

考虑下图设计。Retangle类具有两方法，一个方法把矩形绘制在屏幕上，另一个方法计算矩形的面积。



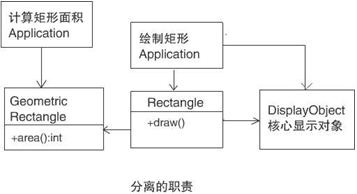
有两个不同的Application使用Rectangle类。一个计算几何面积，另一个Application绘制一个在舞台上显示的矩形。

Rectangle类具有了两个职责，第一个职责是提供一个矩形形状几何数据模型；第二个职责是把矩形显示在屏幕上。

对于SRP的违反导致了一些严重的问题。首先，我们必须在计算几何应用程序中包含核心显示对象的模块。其次，如果绘制矩形Application发生改变，也可能导致计算矩形面积Application发生改变，导致不必要的重新编译，和不可预测的失败。

一个较好的设计是把这两个职责分离到下图所示的两个完全不同的类中。这个设计把Rectangle类中进行计算的部分移到GeometryRectangle类中。

现在矩形绘制方式的改变不会对计算矩形面积的应用产生影响了。



**示例2：划分职责的示例**

既然一个类不能有多个职责，那么怎么划分职责呢？

Single Responsibility Principle（SRP）的原文里举了一个Modem的例子来说明怎么样进行职责的划分，这里我们也沿用这个例子来说明一下：

SRP违反例：

public interface Modem {

public void dial(String pno); //拨号

public void hangup(); //挂断

public void send(char c); //发送数据

public char recv(); //接收数据

}

上述Modem接口，大多数人会认为这个接口看起来非常合理，该接口声明了4个函数确实是Modem所具有的功能。然而，该接口却显示出了两个职责，一个是连接管理（dial+hangup），第二个是数据通信（send+recv）。

这两个职责应该被分离开么？这依赖于应用程序的变化，是按照实际情况决定的。如果应用程序的变化会影响连接管理，那么设计就具有僵化的臭味。因为，调用send和recv的类必须要重新编辑。在这种情况下，这两个职责应该被分离，这样做会避免这两个职责耦合在一起。

另一方面，如果应用程序的变化总是导致这两方面职责同时变化，那么就不必分离他们。实际上，分离他们就会具有不必要的复杂性臭味。

很多情况下，这2个职责没有任何共通的部分，它们因为不同的理由而改变，被不同部分的程序调用。所以它违反了SRP原则。

下面的类图将它的2个不同职责分成2个不同的接口，这样至少可以让客户端应用程序使用具有单一职责的接口：

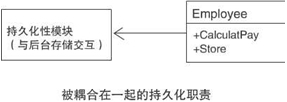
Connection接口：包含dial和hangup方法

DataChannel接口：包含send和recv方法

ModemImplementation类：实现这两个接口。

让ModemImplementation实现这两个接口。我们注意到，ModemImplementation又组合了2个职责，这不是我们希望的，但有时这又是必须的。通常由于某些原因，迫使我们不得不绑定多个职责到一个类中，但我们至少可以通过接口的分割来分离应用程序关心的概念。

**示例3：耦合的持久化职责**



上图展示了一种常见的违反SRP的情况，Employee类包含了业务逻辑和对于持久层的控制，这两个职责在大多数情况下决不应该混合在一起。

业务规则往往会频繁的变化，而持久化的方式却不会如此频繁的变化，并且变化的原因也是完全不同的。把业务规则和持久模块绑定在一起的做法是不妥的。当僵化性和脆弱性的臭味变得强烈，那么就应该使用FACADE和PROXY模式对设计进行重构，分离这两个职责。

**示例4：计算器**

在有些人眼里，计算器是一个整体，所以它把这个需求进行了抽象，最终设计为一个Calculator类。

另外，还有一部分人认为：计算器是一个外壳Appearance和一个处理器Processor的组合。

为什么这么做呢？因为外壳和处理器是两个职责，都是很容易发生需求变动的因素，所以把他们放到一个类中，违背了单一职责原则。

比如，用户可能对计算器提出以下要求：

1、目前已经实现“加法”、“减法”、“乘法”和“除法”，以后还可能出现“乘方”、“开方”等运算。

2、现在人机界面太简单了，还可能做个Windows计算器风格的界面或者Mac计算器风格的界面。

所以，把一个类Calculator 拆分为两个类Appearance和Processor，更容易应对需求变化。如果界面需要修改，那么就去修改Appearance类；如果处理器需要修改，那么就去修改Processor类。

## 开闭原则OCP

### 定义

简述：一个软件实体（**模块，类，函数等**）应当对扩展开放（可以扩展），对修改关闭（不可修改）。

在设计一个模块的时候，应当使这个模块可以在不被修改的前提下被扩展，即不修改模块的现有源代码，而改变该模块的行为。从另外一个角度讲，就是所谓的“对可变性封装原则”。

### 分析

**开闭原则是面向对象设计原则的基石，其它的设计原则可以作为实现开闭原则的手段。**

在对每一个模式进行优缺点评价时都会以开闭原则作为一个重要的评价依据，以判断基于该模式设计的系统是否具备良好的灵活性和可扩展性。

几乎所有的设计模式都是对不同的可变性进行封装，从而使系统在不同角度上达到开闭的要求。其他的设计原则，很多时候是为实现这一目标服务的，例如以LSP实现最佳的、正确的继承层次，就能保证不会违反开放封闭原则。

### 用途

本原则紧紧围绕变化展开，OCP的动机很简单：软件是变化的，不论是优质的设计还是低劣的设计都无法回避这一问题。OCP说明了软件设计应该尽可能地使架构稳定而又容易满足不同的需求。

软件的维护和升级是一个永恒的话题，在维护的过程中，可能要不断地增加一些小功能；在升级的过程中，要增加一些较大的功能。这种功能的扩展，就要求我们改变原有的代码。

但是，对原代码的修改就会深刻地影响到原来的功能，在对系统功能进行扩展时承受一定代价。换句话说，我们设计出来的系统，一定要是扩展性良好的系统。如何才能够设计出扩展性良好的系统呢？这就需要在软件系统设计时遵守开闭原则。

### 优缺点

软件设计本身所追求的目标就是封装变化、降低耦合，开放封闭原则正是对这一目标的最直接体现。

满足OCP的设计给系统带来两个无可比拟的优越性：

1、通过扩展已有的软件系统，可以提供新的行为，以满足对软件的新需求，使变化中的软件系统有灵活的可扩展性。

2、已有的软件模块，特别是最重要的抽象层模块不能再修改，这可以让软件系统在变化中保持稳定。

遵循开闭原则的系统设计，可以让软件系统可复用，并且易于维护。

### 如何实现开闭原则

**“抽象化”是OCP的关键**，也就是面象接口编程，具体一点就是声明的变量的类型、函数的参数类型、函数的返回类型等要尽量使用抽象类和接口。在C++中，这个抽象是指抽象基类或纯抽象基类。

OCP的遵循关键在于抽象，其主要实现方式有：定义接口描绘所需的操作，client只需关注接口的调用，子类型可以以任何其选择的方式实现接口，即所谓的stategy模式；或者定义抽象类并于其中实现公共操作，个性操作定义为abstract或virtual，由子类型负责个性化实现。通过此两法，将功能的通用部分和实现细节分离出来。当然，设计人员应该确定（猜测或凭经验）系统对哪种变化做到封闭，因为不可能对所有变化做到封闭。

如果我们预测到某种变化，或者某种变化发生了，我们应当创建抽象类来隔离以后发生的同类变化。

给系统定义出一个稳定的抽象设计层，允许尽可能多的行为在实现层被实现。抽象层相对稳定，在扩展软件行为时，抽象层不需要修改，满足了开闭原则中“对修改关闭”的原则。同时，从抽象层导出的具体新类可以改变系统的行为，又是对扩展开放的。

开闭原则实际也是“对可变性的封装原则”，那就是“考虑设计中什么可能发生变化，找到一个系统的可变因素，将它封装起来”。把变化封闭起来也就是编写一个抽象类或接口，然后用子类来继承它。

为了满足开闭原则的对修改关闭（closed for modification）原则以及扩展开放（open for extension）原则，应该对软件系统中的不变的部分加以抽象，在面向对象的设计中：

1、可以把这些不变的部分加以抽象成不变的接口，这些不变的接口可以应对未来的扩展；

2、接口的最小功能设计原则。根据这个原则，原有的接口要么可以应对未来的扩展；不足的部分可以通过定义新的接口来实现；

3、模块之间的调用通过抽象接口进行，这样即使实现层发生变化，也无需修改调用方的代码。

接口是稳定的，关闭的，但接口的实现是可变的，开放的。可以通过对接口的不同实现以及类的继承行为等为系统增加新的或改变系统原来的功能，实现软件系统的柔软扩展。

简单地说，软件系统是否有良好的接口（抽象）设计是判断软件系统是否满足开闭原则的一种重要的判断基准。现在多把开闭原则等同于面向接口的软件设计。

**开闭原则的相对性**

软件系统的构建是一个需要不断重构的过程，在过程中，模块的功能抽象，模块与模块间的关系，不会一开始就非常清晰明了，所以构建100%满足开闭原则的软件系统是很困难的，这就是开闭原则的相对性。

但在设计过程中，通过对模块功能的抽象（接口定义），模块之间的关系的抽象（通过接口调用），抽象与实现的分离（面向接口的程序设计）等，可以尽量接近满足开闭原则。

一个软件系统的所有模块不可能都满足OCP，我们应该努力最小化这些不满足OCP的模块数量。

对于应用程序中的每个部分都肆意地进行抽象同样不是一个好主意。开发人员应该仅仅对程序中呈现出**频繁变化**的那些部分作出抽象，拒绝不成熟的抽象和抽象本身一样重要。

但需要注意的是，开闭原则只是一个理想化的准则，在真正的面向对象设计里，我们很难百分之百地满足开闭原则的规定。通常，我们会选择模块中最容易发生变化的因素，使系统对于这些因素符合开闭原则，以适应复用、扩展等变化的需求。

**做到开闭原则，就注意以下两点：**

1、多使用抽象类

在设计类时，对于拥有共同功能的相似类进行抽象化处理，将公用的功能部分放到抽象类中，所有的操作都调用子类。这样，在需要对系统进行功能扩展时，只需要依据抽象类实现新的子类即可。在扩展子类时，不仅可以拥有抽象类的共有属性和共有函数，还可以拥有自定义的属性和函数。

2、多使用接口

与抽象类不同，接口只定义子类应该实现的接口函数，而不实现公有的功能。大多数软件开发中，都会为实现类定义接口，这样在扩展子类时实现该接口。如果要改换原有的实现，只需要改换一个实现类。

**对可变性的封装原则**

“开-闭”原则也就是“对可变性的封装原则”。即“考虑设计中什么可能发生变化，找到系统中的可变因素，将它封装起来”。“对可变性的封装原则”意味着：

1、一种可变性不应当散落在代码的许多角落，而应当被封装到一个对象里面。同一可变性的不同表象意味着同一个继承等级结构中的具体子类。因此，此处可以期待继承关系的出现。继承是封装变化的方法，而不仅仅是从一般的对象生成特殊的对象。

2、一种可变性不应当与另一种可变性混合在一起。

**OCP背后的机制就是抽象和多态**

该原则要求我们的代码模块应能很容易地扩展，但在扩展的过程中，无须改动已有的代码。怎样在不改变源代码（关闭修改）的情况下更改它的行为呢？

在使用面向过程的设计方法时，我们很难做到这一点一一一惟一可行的就是尽量利用回调函数。但是，在面向对象的语言中，利用多态性可以很容易地满足上述原则的要求。OCP背后的机制就是抽象和多态。

当需要扩展服务器类的功能时，我们只要从服务器类中派生出新的服务器类，并实现新的"服务()"函数就可以了一一这是基于继承关系的实现方法。

另一种基于聚合关系的扩展方式：客户类中聚合一个接口的实例，该实例所属的类是由某个特定的接口类派生的，这种实现方式可以方便地把接口组合在其他的客户类中，并可以动态的改变接口的实现方式。

这里都是通过动态多态性来实现开闭原则的，利用静态多态性，也同样可以获得类似的效果。

### 示例

**示例1：totalPrice**

考虑下面某类的方法：

public double totalPrice(Part[] parts) {

double total = 0.0;

for (int i=0; i<parts.length; i++) {

total += parts[i].getPrice();

}

return total;

}

以上函数的工作是在制订的部件数组中计算各个部件价格的总和。若Part是一个基类或接口且使用了多态，则该类可很容易地来适应新类型的部件，而不必其进行修改。其将符合OCP。

但在计算总价格时，若财务部颁布主板和内存应使用额外费用，如何做？下列的代码是如何做的呢？

double total = 0.0;

for (int i=0; i<parts.length; i++) {

if (parts[i] instanceof Motherborad)

total += (1.45 \* parts[i].getPrice());

else if (parts[i] instanceof Memery)

total += (1.27 \* parts[i].getPrice());

else

total += parts[i].getPrice();

}

return total;

这符合OCP吗？当每次财务部提出新的计价策略，我们都不得不要修改totalPrice()方法！这并非对更改是封闭的。显然，策略的变更便意味着我们不得不要在一些地方修改代码的，因此我们该如何去做呢？

为了使用我们第一个版本的totalPrice()，我们可以将计价策略合并到Part的getPrice()方法中。

这里是Part和ConcretePart类的示例：

public class part {

private double basePrice;

public void setPrice(double price) {basePrice = price;}

public double getPrice() {return basePrice;}

}

public class ConcretePart extends Part {

public double getPrice() { return (0.90 \* basePrice); }

}

但是现在每当计价策略发生改变，我们就必须修改Part的每个子类！

一个更好的思路是采用一个PricePolicy类，通过对其进行继承以提供不同的计价策略：

public class Part {

private PricePolicy pricePolicy;

public void setPricePolicy(PricePolicy policy) { pricePolicy = policy; }

public void setPrice(double price) { pricePolicy.setPrice(price); }

public double getPrice() { return pricePolicy.getPrice(); }

}

public class PricePolicy {

private double basePrice;

public void setPrice(double price) { basePrice = price; }

public double getPrice() { return basePrice; }

}

public class SalePrice extends PricePolicy {

private double discount;

public void setDiscount(double discount) { this.discount = discount; }

public double getPrice() { return (basePrice \* discount); }

}

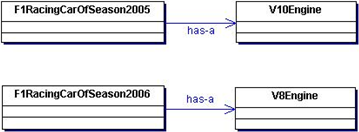
看起来我们所做的就是将问题推迟到另一个类中。但是使用该解决方案，我们可通过改变Part对象，在运行期间动态地来设定计价的策略。

另一个解决方案是使每个ConcretePart从数据库或属性文件中获取其当前的价格。

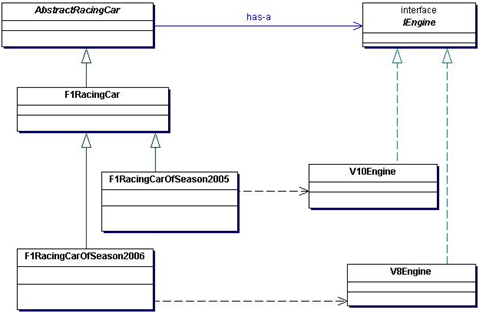
**示例2：赛车和引擎**

如何在OO中引入OCP原则？把对实体的依赖改为对抽象的依赖。下面的例子说明了这个过程：

05赛季的时候，一辆F1赛车有一台V10引擎。但到了06赛季，国际汽联修改了规则，一辆F1赛车只能安装一台V8引擎。车队很快投入了新赛车的研发，不幸的是，从工程师那里得到消息，旧车身的设计不能够装进新研发的引擎。我们不得不为新的引擎重新打造车身，于是一辆新的赛车诞生了。但是麻烦的事接踵而来，国际汽联频频修改规则，搞得设计师在赛车上改了又改，最终变得不成样子，只能把它废弃。



为了能够重用这辆昂贵的赛车，工程师们提出解决方案：首先，在车身的设计上预留出安装引擎的位置和管线。然后，根据这些设计好的规范设计引擎（或是引擎的适配器）。于是，新的赛车设计方案诞生了。



## 里氏代换原则LSP

### 定义

**完全替换原则：派生类应该能完全替换掉基类。**

简述：在一个软件系统中，子类可以替换任何基类能够出现的地方，并且经过替换以后，代码还能正常工作。

即在软件中把基类都替换成它的子类，程序将不会产生任何错误和异常。反过来的代换不成立，如果一个软件实体使用的是一个子类的话，那么它不一定适用于基类。

### 分析

里氏代换原则是继承复用的基石。只有当衍生类可以替换掉基类，软件单元的功能不受影响时，基类才能真正被复用。

里氏代换原则是实现开闭原则的重要方式之一，正是子类型的可替换性，才使得使用基类型模块无需修改就可扩充。由于使用基类对象的地方都可以使用子类对象，因此在程序中尽量使用基类类型来对对象进行定义，而在运行时再确定其子类类型，用子类对象来替换父类对象。

**与“开-闭”原则的关系**

实现“开-闭”原则的关键步骤是抽象化。基类与子类之间的继承关系就是抽象化的体现，因此里氏代换原则是对实现抽象化的具体步骤的规范。

违反里氏代换原则意味着违反了“开-闭”原则，反之未必。

**抽象出公共部分作为抽象基类的设计：**提取公共职责放入超类中，稍后添加的新的子类型可能会以新的方式支持同样的职责，此时原来的超类可能会是一个抽象类。

### 如何实现此原则

里氏代换原则要求，凡是基类适用的地方，子类一定适用。因此，子类必须具备基类的所有接口，可以更宽，但不能少。

要想使派生类能完全替换基类，派生类和基类就必须拥有相同的接口定义。此外，对接口中的每一个方法，派生类和基类还必须拥有相同的前提条件和后置条件，因为当前提条件和后置条件不同时，如果用派生类替换基类，就有可能会造成错误的调用。这种要求派生类和基类的同一个方法必须具有相同的前提条件和后置条件的理论，正是伯特兰德·梅耶为Eiffel 语言引入的一个重要特性一一面向契约的设计（Design by Contact）。即便我们使用的不是Eiffel语言，我们也完全可以使用断言Assert和注释等技术手段来限定方法的前提条件和后置条件。

满足完全替换原则的类可以完全透明地被客户调用。这时，客户程序可完全不必关心对象的类型，随意地使用类中的方法。但在一棵继承树非常庞大的情况下，程序员就很难保证这棵树的所有类都是可以完全替换的，这也是继承树过大时的缺陷之一。

当我们在实现子类时必须要谨慎对待，以确保我们不会无意中违背了LSP。若一个函数未能满足LSP，那么可能是因为它显式地引用了超类的一些或所有子类。

**IS-A是关于行为的**

LSP关注的是怎样良好的使用继承。LSP清晰的指出，OOD的IS-A关系是就行为方式而言。

当前存在的主要现象：**滥用继承。面向对象程序设计中的继承概念，与通常的数学法则与生活常识是不同范畴的事物，有不可混淆的区别。子类与基类的行为必须一致。**

究竟是怎么会使Square和Rectangle这个显然合理的模型为什么会有问题呢？毕竟，Square应该就是Rectangle。难道他们之间不存在IS-A关系吗？

因为Square对象的行为方式和调用者所期望的Rectangle对象的行为方式不相容。从行为方式的角度来看，Square不是Rectangle，对象的行为方式才是软件真正所关注的问题。LSP清楚地指出，OOD中IS-A关系是就行为方式而言的，是客户程序所依赖的。

**从派生类中抛出异常**

另一种LSP的违规形式是在派生类的方法中添加了其他基类不会抛出的异常。如果基类的使用者不期望这些异常，那么把它们添加到派生类的方法中就会导致不可替换性。此时要遵循LSP，要么就必须改变使用者的期望，要么派生类就不应该抛出这些异常。

**有效性并非本质属性**

在考虑一个特定设计是否恰当时，不能完全孤立地来看这个解决方案，必须要根据设计的使用者做出的合理假设来审视它。

有谁知道设计的使用者会做出什么样的合理假设呢？大多数这样的假设都很难预测。事实上，如果试图去预测所有这些假设，我们所得到的系统很可能会充满不必要的复杂性的臭味。因此，像所有其他原则一样，通常最好的方法只预测那些最明显的对于LSP的违反情况而推迟所有其他的预测，知道出现相关的脆弱性的臭味时，才去处理它们。

### Design by Contract

根据Bertrand Meyer氏提出的Design by Contract（DBC：基于契约的设计）概念的描述，对于类的一个方法，都有一个前提条件以及一个后续条件。前提条件说明方法接受什么样的参数数据等，只有前提条件得到满足时，这个方法才能被调用；同时后续条件用来说明这个方法完成时的状态，如果一个方法的执行会导致这个方法的后续条件不成立，那么这个方法也不应该正常返回。

现在把前提条件以及后续条件应用到继承子类中，子类方法应该满足：

1、前提条件不强于基类；

2、后续条件不弱于基类。

换句话说，通过基类的接口调用一个对象时，用户只知道基类前提条件以及后续条件，因此继承类不得要求用户提供比基类方法要求的更强的前提条件，即继承类方法必须接受任何基类方法能接受的任何条件（参数）。同样，继承类必须顺从基类的所有后续条件，即继承类方法的行为和输出不得违反由基类建立起来的任何约束，不能让用户对继承类方法的输出感到困惑。

这样，我们就有了基于契约的LSP，基于契约的LSP是LSP的一种强化。

Liskov替换原则从基于契约的设计演化而来，契约通过为每个方法声明“先验条件”和“后验条件”；定义子类时，必须遵守这些“先验条件”和“后验条件”。当前基于契约的设计发展势头正劲，对实现“软件工厂”的“组装生产”梦想是一个有力的支持。

为遵循LSP，可借用DBC（design by contract）的操作前置条件和后置条件，“要使操作得以执行，前置条件必须为真，执行完毕后，该操作要确保后置条件为真”（为每个方法注明其前置和后置条件十分有帮助），如此，则“在重新声明派生类中操作时，只能使用相等或更弱的前置条件来替换原始的前置条件，只能用相等或更强的后置条件来替换原始的后置条件”（interface和其实现类间抽象方法和其实现此二者一定满足前述条件）。同时亦可用前述OCP遵循所用的二模式使设计符合LSP，另外子类型中的异常抛出应考虑在遵循LSP的范围内。

### 示例

**示例1：正方形不是长方形**

违反里氏代换原则一个最经典的例子便是把正方形设计成长方形的子类，虽然在数学意义上是的。在数学领域里，正方形毫无疑问是长方形，它是一个长宽相等的长方形。

但是从行为上来说，一个Square对象不是一个Rectangle对象，Square对象的行为与一个Rectangle对象的行为是不一致的！一个Square对象与一个Rectangle对象之间不具有多态的特征。

由于数学意义上是一致的，所以在我们开发的一个与几何图形相关的软件系统中，让正方形继承自长方形是顺利成章的事情。现在，我们截取该系统的一个代码片段进行分析：

class Rectangle

{

double length;

double width;

public :

double getLength() { return length; }   
void setLength(double height) { this.length = length; }     
double getWidth() { return width; }  
void setWidth(double width) { this.width = width; }

}

class Square : public Rectangle

{

public :

void setWidth(double width)

{

Rectangle ::setLength(width);

Rectangle :: setWidth(width);     
}  
void setLength(double length)

{

Rectangle ::.setLength(length);

Rectangle ::.setWidth(length);

}

}

由于正方形的度和宽度必须相等，所以在方法setLength和setWidth中，对长度和宽度赋值相同。类TestRectangle是我们的软件系统中的一个组件，它有一个resize方法要用到基类Rectangle，resize方法的功能是模拟长方形宽度逐步增长的效果：

测试类TestRectangle：

class TestRectangle

{

public:

void resize(Rectangle &objRect)

{

while(objRect.getWidth() <= objRect.getLength() )

{

objRect.setWidth( objRect.getWidth () + 1 );

}

}

}

我们运行一下这段代码就会发现，假如我们把一个普通长方形作为参数传入resize方法，就会看到长方形宽度逐渐增长的效果，当宽度大于长度，代码就会停止，这种行为的结果符合我们的预期；假如我们再把一个正方形作为参数传入resize方法后，就会看到正方形的宽度和长度都在不断增长，代码会一直运行下去，直至系统产生溢出错误。所以，普通的长方形是适合这段代码的，正方形不适合。

我们得出结论：在resize方法中，Rectangle类型的参数不能被Square类型的参数所代替，如果进行了替换就得不到预期结果。因此，Square类和Rectangle类之间的继承关系违反了里氏代换原则，它们之间的继承关系不成立，正方形不是长方形。

**示例2：鸵鸟不是鸟**

“鸵鸟非鸟”也是一个理解里氏代换原则的经典的例子。“鸵鸟非鸟”的另一个版本是“企鹅非鸟”，这两种说法本质上没有区别，前提条件都是这种鸟不会飞。生物学中对于鸟类的定义：“恒温动物，卵生，全身披有羽毛，身体呈流线形，有角质的喙，眼在头的两侧。前肢退化成翼，后肢有鳞状外皮，有四趾”。所以，从生物学角度来看，鸵鸟肯定是一种鸟。

我们设计一个与鸟有关的系统，鸵鸟类顺理成章地由鸟类派生，鸟类所有的特性和行为都被鸵鸟类继承。大多数的鸟类在人们的印象中都是会飞的，所以，我们给鸟类设计了一个名字为fly的方法，还给出了与飞行相关的一些属性，比如飞行速度（velocity）。

class Bird

{

double velocity;

public :

void fly() { //I am flying; };

void setVelocity(double velocity) { this.velocity = velocity; };  
double getVelocity() { return this.velocity; };

}

鸵鸟不会飞怎么办？我们就让它扇扇翅膀表示一下吧，在fly方法里什么都不做。至于它的飞行速度，不会飞就只能设定为0了，于是我们就有了鸵鸟类的设计。

鸵鸟类Ostrich:

class Ostrich :public Bird

{

public fly() { //I do nothing; };

public setVelocity(double velocity) { this.velocity = 0; };

public getVelocity() { return 0; };

}

好了，所有的类都设计完成，我们把类Bird提供给了其它的代码（消费者）使用。现在，消费者使用Bird类完成这样一个需求：计算鸟飞越黄河所需的时间。

对于Bird类的消费者而言，它只看到了Bird类中有fly和getVelocity两个方法，至于里面的实现细节，它不关心，而且也无需关心，于是给出了实现代码：

测试类TestBird:

class TestBird

{

public:

void calcFlyTime(Bird bird)

{

try

{

double riverWidth = 3000;

cout.<<riverWidth / bird.getVelocity()<<endl;

}

catch(…) { cout<<"An error occured!"<<endl; }

};

}

如果我们拿一种飞鸟来测试这段代码，没有问题，结果正确，符合我们的预期，系统输出了飞鸟飞越黄河的所需要的时间；如果我们再拿鸵鸟来测试这段代码，结果代码发生了系统除零的异常，明显不符合我们的预期。

对于TestBird类而言，它只是Bird类的一个消费者，它在使用Bird类的时候，只需要根据Bird类提供的方法进行相应的使用，根本不会关心鸵鸟会不会飞这样的问题，而且也无须知道。它就是要按照“所需时间 = 黄河的宽度 / 鸟的飞行速度”的规则来计算鸟飞越黄河所需要的时间。

我们得出结论：在calcFlyTime方法中，Bird类型的参数不能被Ostrich类型的参数所代替，如果进行了替换就得不到预期结果。因此，Ostrich类和Bird类之间的继承关系违反了里氏代换原则，它们之间的继承关系不成立，鸵鸟不是鸟。

产生这种混乱有两方面的原因：

**原因一：对类的继承关系的定义没有搞清楚。**

面向对象的设计关注的是对象的行为，它是使用“行为”来对对象进行分类的，只有行为一致的对象才能抽象出一个类来。

**类的继承关系就是一种“Is-A”关系，实际上指的是行为上的“Is-A”关系，可以把它描述为“Act-As”。**

我们再来看“正方形不是长方形”这个例子，正方形在设置长度和宽度这两个行为上，与长方形显然是不同的。长方形的行为：设置长方形的长度的时候，它的宽度保持不变，设置宽度的时候，长度保持不变。正方形的行为：设置正方形的长度的时候，宽度随之改变；设置宽度的时候，长度随之改变。所以，如果我们把这种行为加到基类长方形的时候，就导致了正方形无法继承这种行为。我们“强行”把正方形从长方形继承过来，就造成无法达到预期的结果。

“鸵鸟非鸟”基本上也是同样的道理。我们一讲到鸟，就认为它能飞，有的鸟确实能飞，但不是所有的鸟都能飞。问题就是出在这里。如果以“飞”的行为作为衡量“鸟”的标准的话，鸵鸟显然不是鸟；如果按照生物学的划分标准：有翅膀、有羽毛等特性作为衡量“鸟”的标准的话，鸵鸟理所当然就是鸟了。鸵鸟没有“飞”的行为，我们强行给它加上了这个行为，所以在面对“飞越黄河”的需求时，代码就会出现运行期故障。

**原因二：设计要依赖于用户要求和具体环境。**

继承关系要求子类要具有基类全部的行为，这里的行为是指落在需求范围内的行为。

假如鸟类提供4个对外的行为：1、我会飞；2、我有羽毛；3、我有翅膀；4、我是卵生的。A系统需求需要“我会飞”，B系统需求需要“我有羽毛”。

A需求期望鸟类提供与飞翔有关的行为，即使鸵鸟跟普通的鸟在外观上就是100%的相像，但在A需求范围内，鸵鸟在飞翔这一点上跟其它普通的鸟是不一致的，它没有这个能力，所以，鸵鸟类无法从鸟类派生，鸵鸟不是鸟。

B需求期望鸟类提供与羽毛有关的行为，鸵鸟在这一点上跟其它普通的鸟一致的。虽然它不会飞，但是这一点不在B需求范围内，所以它具备了鸟类全部的行为特征，鸵鸟类能够从鸟类派生，鸵鸟就是鸟。

所有派生类的行为功能必须和使用者对其基类的期望保持一致，如果派生类达不到这一点，那么必然违反里氏替换原则。

在实际的开发过程中，不正确的派生关系是非常有害的。伴随着软件开发规模的扩大，参与的开发人员也越来越多，每个人都在使用别人提供的组件，也会为别人提供组件。最终，所有人的开发的组件经过层层包装和不断组合，被集成为一个完整的系统。每个开发人员在使用别人的组件时，只需知道组件的对外裸露的接口，那就是它全部行为的集合，至于内部到底是怎么实现的，无法知道，也无须知道。所以，对于使用者而言，它只能通过接口实现自己的预期，如果组件接口提供的行为与使用者的预期不符，错误便产生了。里氏代换原则就是在设计时避免出现派生类与基类不一致的行为。

## 依赖倒转原则DIP

### 定义

**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。

**依赖倒置原则的2个重要方针：**

1、高层模块不应该依赖于低层模块，两者都应该依赖于抽象。这就是著名的Hollywood原则：Don't call us, we'll call you。

2、抽象不应该依赖于细节（实现），细节应该依赖于抽象。具体点就是要依赖于抽象类和接口，不要依赖具体类。**另一种表述为：要针对接口编程，不要针对实现编程。**

### 来历

传统过程开发，设计方法倾向于创建高层模块依赖于低层模块；抽象层次依赖于具体层次（策略依赖于细节）的软件结构。实际上这些方法的目的之一就是要定义子程序层次结构，该层次结构描述了高层模块怎样调用低层模块。

本应该是高层的策略设置模块去影响低层的细节实现模块的，包含业务规则的模块应该优先于并独立于包含实现细节的模块，无论如何高层模块都不应该依赖于低层模块。这是错误的，依赖倒转原则就是要把这种错误的依赖关系倒转过来。

在一般的设计中，基类的任务是为派生类提供服务，派生类通过调用基类的服务来扩展基类的行为，这种设计方法是正向的依赖关系。而依赖倒置原则强调的是一种反向的依赖关系，通过依赖倒置，我们可以很容易地满足开闭原则。这是因为，在反向的依赖关系中，客户程序不会直接调用派生类的任何方法，而只是调用基类的方法，基类通过调用自己的虚函数来把控制权传递给应用程序实现的具体的派生类，即基类依赖于特定的派生类。

**问题的提出：**

Robert C. Martin氏在原文中给出了**Bad Design的定义**：

1、系统很难改变，因为每个改变都会影响其他很多部分。

2、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。

3、系统很难被另外一个应用重用，因为你很难将要重用的部分从系统中分离开来。

导致Bad Design的很大原因是高层模块过分依赖低层模块。

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。

比如，一个Copy模块，需要把来自Keyboard的输入复制到Print，即使对Keyboard和Print的封装已经做得非常好，但如果Copy模块里直接使用Keyboard与Print，Copy任很难被其他应用环境（比如需要输出到磁盘时）重用。

**问题的解决：**

为了解决上述问题，Robert C. Martin氏提出了OO设计的Dependency Inversion Principle(DIP)原则。

DIP给出了一个解决方案：在高层模块与低层模块之间，引入一个抽象接口层。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，高层模块不直接依赖低层模块，高层模块与低层模块都依赖抽象接口层。

当然，抽象也不依赖低层模块的实现细节，低层模块依赖（继承或实现）抽象定义。

### 分析

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。

抽象层（高层模块）：包含了一个应该程序中的重要的策略选择和业务模型，包含的是应用系统的商务逻辑和宏观的、对整个系统来说重要的战略性决定。是一个系统的本质的概括，它相对稳定，是必然性的体现。

具体层（底层模块）：含有一些次要的与实现有关的算法和逻辑，以及战术性的决定，带有很大的偶然性选择。它相对易变，具体层次的代码经常会变动，不能避免出现错误。

抽象层是宏观逻辑的、战略决策的、带有必然选择性的，应该决定具体层次的具体算法、战术决策、偶然性选择。

如果高层依赖于低层，那么对低层模块的改动就会直接影响到高层模块，从而迫使它们依次做出改动。

抽象层次依赖于具体层次，使许多具体层次的细节的算法变化立即影响到抽象层次的宏观的商务逻辑，导致微观决定宏观，战术决定战略，偶然决定必然。

通常（传统）软件设计中采用结构化设计，用高层模块直接调用底层模块，这样高层模块将严重依赖于底层模块的变动。

**与“开-闭”原则的关系**

由于抽象耦合总要涉及具体类从抽象类继承，并且需要保证在任何引用到某类的地方都可以改换成其子类，因此，LSP是DIP的基础，是实现OCP的主要手段。

实现开闭原则的关键是抽象化，并且从抽象化导出具体化实现，如果说开闭原则是面向对象设计的目标的话，那么依赖倒转原则就是面向对象设计的主要手段。

开闭原则与依赖倒转原则是目标和手段的关系。开闭原则是目标，依赖倒转原则是到达开闭原则的手段。如果要达到最好的开闭原则，就要尽量的遵守依赖倒转原则，依赖倒转原则是对抽象化的最好规范。

里氏代换原则是依赖倒转原则的基础，依赖倒转原则是里氏代换原则的重要补充。

### 用途：复用

我们更希望能够重用的是高层的策略设置模块。我们已经非常擅长于通过子程序库的形式来重用低层模块。如果高层模块依赖于低层模块，那么在不同的上下文中重用高层模块就会变得非常困难。然而，如果高层模块独立于低层模块，那么高层模块就可以非常容易的被重用。该原则是框架（framework）设计的核心原则。

**传统过程性设计中的复用性：**

传统的过程性设计中，侧重于具体层次模块的复用，如算法的复用、数据结构的复用、函数库的复用等，都局限于具体层次模块里的复用。较高层次的结构依赖于较低层次的结构，较低层次的结构又进一步依赖于更低层次的结构。如此，较低层次上的修改，往往导致较高层次的修改，直到高层次逻辑的修改。

同样，传统的做法，也强调具体层次上的可维护性，包括一个函数、数据结构等的可维护性，而不是高级层次上的可维护性。

**面向对象编程中的复用性：**

从复用的意义上讲，抽象层次有一个应用系统最重要的宏观逻辑，是做战略判断和决策的地方，抽象层次应当是较为稳定的，应当是复用的重点。

传统复用侧重于具体模块和细节的复用，因此，“倒转”，是指应当将复用的重点放在抽象层次上。

如果抽象层次上的模块相对独立于具体层次的模块的话，抽象层次的模块复用就较为容易。

同样的，宏观逻辑是维护的重点，而不是相反。

### 如何实现依赖倒转

著名的Hollywood原则：Don't call us we'll call you，底层模块实现了在高层模块声明并被高层模块调用的接口。在OOD中通过为高层模块定义所需使用的服务接口，底层模块现实这样的接口，高层模块通过抽象接口使用下一层（strategy模式所声明的）。

DIP实施重点：从问题的具体细节中分离出抽象，以抽象方式对类进行耦合，客户端依赖于抽象耦合。

依赖倒转原则的常用实现方式之一是在代码中使用抽象类，而将具体类放在配置文件中。“将抽象放进代码，将细节放进元数据”---《程序员修炼之道：从小工到专家》

我们自己的具体类大多是不稳定的，通过把他们隐藏在抽象接口后面可以隔离不稳定性。

如果一个类很稳定，那么依赖于它不会造成伤害。有一些具体类可能是相当稳定不会变化，使用这个具体类实例的应用完全可以依赖于这个具体类型，而不必为此创建一个抽象类型。

在抽象层次上的耦合虽然有灵活性，但也带来了额外的复杂性，如果一个具体类发生变化的可能性非常小，那么抽象耦合能发挥的好处便十分有限，这时可以用具体耦合反而会更好。如：专门教鹦鹉唱歌的软件，不必使用鸟类做基类。

**抽象不应该依赖于细节，细节应该依赖于抽象**，依赖于抽象，不依赖于具体类，尽量做到：

1、程序中所有的依赖关系都应该终止于抽象类和接口。

2、针对接口而非实现编程。

3、任何变量都不应该持有一个指向具体类的指针或引用。

4、任何类都不应该从具体类派生。

5、任何方法都不应该覆写他的任何基类中的已经实现了的方法。

每个程序都会有违反该规则的情况，有时必须要创建具体类的实例，而创建这些实例的模块将会依赖于它们。

**基于针对接口编程而不是实现的原则，设计类结构的方式应该是从上层模块到底层模块：**

1、上层类 --> 抽象层 --> 底层类。

2、高层模块 -> 抽象接口层 --> 低层模块。

**以依赖倒转原则为指导原则的设计模式包括：**1、工厂方法模式；2、模板方法模式；3、迭代子模式。

### 依赖倒转原则的缺点

1、实现困难。依赖倒转原则虽然很强大，但却最不容易实现。

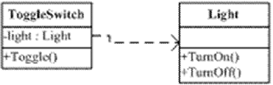
2、因为依赖倒转的缘故，对象的创建很可能要使用对象工厂，以避免对具体类的直接引用。

3、此原则的使用可能还会导致产生大量的类。

4、对不熟悉面向对象技术的工程师来说，维护这样的系统需要较好地理解面向对象设计。

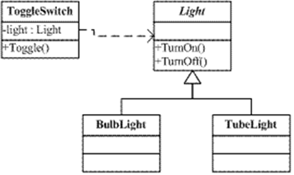
1. 在抽象层次上的耦合虽然有灵活性，但也带来了额外的复杂性。

### 示例：Light类



缺点：耦合太紧密，Light发生变化将影响ToggleSwitch。

**解决办法一：**

****

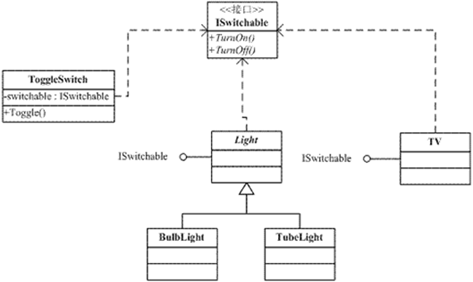
将Light作成Abstract，然后具体类继承自Light。

优点：ToggleSwitch依赖于抽象类Light，具有更高的稳定性，而BulbLight与TubeLight继承自Light，可以根据“开放－封闭”原则进行扩展。

只要Light不发生变化，BulbLight与TubeLight的变化就不会波及ToggleSwitch。

缺点：如果用ToggleSwitch控制一台电视就很困难了。总不能让TV继承自Light吧。

**解决办法二**



## 针对接口编程，而非（接口的）实现

### 相关概念

接口表示“…像…”（LikeA）的关系，继承表示“…是…”（IsA）的关系，组合表示“…有…”（HasA）的关系。）

**变量的静态类型和真实类型：**

静态类型：变量被声明时的类型，也叫做明显类型；

实际类型：即所引用的对象的真实类型。

**接口**

接口是一个对象在对其它的对象进行调用时所知道的方法集合，一个对象可以有多个接口（实际上，接口是对象所有方法的一个子集）。

类型是对象的一个特定的接口。不同的对象可以具有相同的类型，而且一个对象可以具有多个不同的类型。

接口，类似角色，包装与该角色相关的操作和属性，而实现这个接口的类便是扮演这个角色的演员。一个角色可以由不同的演员来演；而不同的演员之间，除了可以扮演一个共同的角色之外，并不要求有其他任何共同之处。

接口是实现插件化（pluggability）的关键。

**实现继承和接口继承**

实现继承（类继承）：一个对象的实现是根据另一个对象的实现来定义的。

接口继承（子类型化）：描述了一个对象可在什么时候被用来替代另一个对象。

C++的继承机制既指类继承，又指接口继承。C++通过继承纯虚类来实现接口继承。

Java对接口继承具有单独的语言构造方式——Java Interface，Java接口构造方式更加易于表达和实现那些专注于对象接口的设计。

### 如何实现

针对接口编程：应当使用接口和抽象类进行变量类型声明、参数类型声明、方法返还类型说明，以及数据类型的转换等；

不要对实现编程：不要用具体类进行变量的类型声明、参数类型声明、方法返还类型说明，以及数据类型的转换等。

要做到针对接口编程，一个具体类应当只实现接口和抽象类中声明的方法，而不要给出多余的方法。程序在需要引用一个对象时，应当尽可能的使用抽象类型作为变量的静态类型。

### 优缺点

**什么是面向接口编程呢？**

我个人的定义是：在系统分析和架构中，分清层次和依赖关系，每个层次不是直接向其上层提供服务（即不是直接实例化在上层中），而是通过定义一组接口，仅向上层暴露其接口功能，上层对于下层仅仅是接口依赖，而不依赖具体类。

好处：首先对系统灵活性大有好处。当下层需要改变时，只要接口及接口功能不变，则上层不用做任何修改，甚至可以在不改动上层代码时将下层整个替换掉。就像我们将一个WD的60G硬盘换成一个希捷的160G的硬盘，计算机其他地方不用做任何改动，而是把原硬盘拔下来、新硬盘插上就行了，因为计算机其他部分不依赖具体硬盘，而只依赖一个IDE接口，只要硬盘实现了这个接口，就可以替换上去。

使用接口的另一个好处，就是不同部件或层次的开发人员可以并行开工，就像造硬盘的不用等造CPU的，也不用等造显示器的，只要接口一致，设计合理，完全可以并行进行开发，从而提高效率。

**优点：**

1、Client不必知道其使用对象的具体所属类。

2、Client无需知道特定类，只需知道他们所期望的接口。

3、一个对象可以很容易地被（实现了相同接口的）的另一个对象所替换。

4、对象间的连接不必硬绑定（hardwire）到一个具体类的对象上，因此增加了灵活性。

5、松散耦合（loosens coupling）。

6、增加了重用的可能性。

7、提高了（对象）组合的机率，因为被包含对象可以是任何实现了一个指定接口的类。

**缺点：**

带来设计的复杂性，特别对于没有丰富经验的设计人员。

### 示例

**示例1：人和黑人**

一般做法：黑人 aMan = new 黑人();

针对接口的定义：人 aMan = new 黑人()；

二者的区别，前者使用具体类作为变量的类型，后者使用一个抽象类型（接口）作为类型。

后者的好处在于：在决定将黑人类型换为白人类型时，改动很少。如：人 aMan = new 白人()；

如此，程序具有更好的灵活性，因为除调用构造子的一行语句之外，程序的其他部分感觉不到变化；特别是存在许多其他程序调用此对象的情况下，优势更为明显。

结论：只要一个被引用的对象存在抽象类型，就应当在任何引用此对象的地方使用抽象类型，包括参量的类型声明、方法返回类型的声明、属性变量的类型声明等。

**示例2：移动存储设备**

定义：现在我们要开发一个应用，模拟移动存储设备的读写，即计算机与U盘、MP3、移动硬盘等设备进行数据交换。

上下文（环境）：已知要实现U盘、MP3播放器、移动硬盘三种移动存储设备，要求计算机能同这三种设备进行数据交换，并且以后可能会有新的第三方的移动存储设备，所以计算机必须有扩展性，能与目前未知而以后可能会出现的存储设备进行数据交换。

各个存储设备间读、写的实现方法不同，U盘和移动硬盘只有这两个方法，MP3Player还有一个PlayMusic方法。

下面，列举几个典型的方案。

方案一：分别定义FlashDisk、MP3Player、MobileHardDisk三个类，

实现各自的Read和Write方法。然后在Computer类中实例化上述三个类，为每个类分别写读、写方法。例如，为FlashDisk写ReadFromFlashDisk、WriteToFlashDisk两个方法。总共六个方法。

方案二：定义抽象类MobileStorage，在里面写虚方法Read和Write，三个存储设备继承此抽象类，并重写Read和Write方法。Computer类中包含一个类型为MobileStorage的成员变量，并为其编写get/set器，这样Computer中只需要两个方法：ReadData和WriteData，并通过多态性实现不同移动设备的读写。

方案三：与方案二基本相同，只是不定义抽象类，而是定义接口IMobileStorage，移动存储器类实现此接口。Computer中通过依赖接口IMobileStorage实现多态性。

方案四：定义接口IReadable和IWritable，两个接口分别只包含Read和Write，然后定义接口IMobileStorage接口继承自IReadable和IWritable，剩下的实现与方案三相同。

下面，我们来分析一下以上四种方案：

首先，方案一最直白，实现起来最简单，但是它有一个致命的弱点：可扩展性差。或者说，不符合开闭原则。当将来有了第三方扩展移动存储设备时，必须对Computer进行修改。这就如在一个真实的计算机上，为每一种移动存储设备实现一个不同的插口、并分别有各自的驱动程序。当有了一种新的移动存储设备后，就要将计算机大卸八块，然后增加一个新的插口，在编写一套针对此新设备的驱动程序。

另一个缺点在于，冗余代码多。如果有100种移动存储，那Computer中要至少写200个方法！

再来看方案二和方案三，之所以将这两个方案放在一起讨论，是因为他们基本是一个方案（从思想层面上来说），只不过实现手段不同，一个使用了抽象类，一个使用了接口，最终达到的目的是一样的。

我们先来评价这种方案：首先它解决了代码冗余的问题，因为可以动态替换移动设备，并且都实现了共同的接口，不管有多少种移动设备，只要一个Read方法和一个Write方法，多态性就帮我们解决问题了。而对第一个问题，由于可以运行时动态替换，而不必将移动存储类硬编码在Computer中，所以有了新的第三方设备，完全可以替换进去运行。这就是所谓的“依赖接口，而不是依赖与具体类”，Computer类只有一个MobileStorage类型或IMobileStorage类型的成员变量，至于这个变量具体是什么类型，它并不知道，这取决于在运行时给这个变量的赋值。如此一来，Computer和移动存储器类的耦合度大大下降。

那么这里该选抽象类还是接口呢？还记得对抽象类和接口选择的建议吗？看动机。这里，我们的动机显然是实现多态性而不是为了代码复用，所以当然要用接口。

最后再来看方案四，它和方案三很类似，只是将“可读”和“可写”两个规则分别抽象成了接口，然后让IMobileStorage再继承它们。这样做，显然进一步提高了灵活性，但是，这有没有设计过度的嫌疑呢？我的观点是：这要看具体情况。如果我们的应用中可能会出现一些类，这些类只实现读方法或只实现写方法，如只读光盘，那么这样做也是可以的。如果我们知道以后出现的东西都是能读又能写的，那这两个接口就没有必要了。其实如果将只读设备的Write方法留空或抛出异常，也可以不要这两个接口。总之一句话：理论是死的，人是活的，一切从现实需要来，防止设计不足，也要防止设计过度。

在这里，我们姑且认为以后的移动存储都是能读又能写的，所以我们选方案三。

## 接口隔离原则ISP

### 定义

简述：客户端不应该依赖那些它不需要的接口。在该定义中的接口指的是一个类型所具有的所有方法特征的集合，而不是类似于JAVA中的Interface结构。

可以把接口理解成角色，一个接口只是代表一个角色，每个角色都有它特定的一个接口，这个原则可以叫做角色隔离原则。

### 分析

本原则是单一职责原则用于接口设计的自然结果。

一个接口相当于剧本中的一个角色。而此角色在舞台上，由哪个演员来演，则相当于接口的实现。为避免混淆，本书将这种角色划分的原则叫做角色隔离原则。

ISP接口隔离原则，如SRP维护类的简单性一样。ISP用于维护接口的简单和必要性，因为接口是为客户调用的，因此其应该是“大小尺寸合适的”，“胖”接口显然对调用者造成累赘，ISP则用于将“胖”接口分离成多个合适的接口。

如果类的接口不是内聚的（cohesive），就表示该类具有“胖”接口。类的“胖”接口可以分解成多组方法，每一组方法都服务于一组不同的客户程序。这样，一些客户程序可以使用一组成员函数，而其他客户程序可以使用其他组的成员函数。

ISP承认存在有一些对象，它们确实不需要内聚的接口：但是ISP建议客户程序不应该看到它们作为单一的类存在。相反，客户程序看到的应该是多个具有内聚接口的抽象基类。

### 违背后果

过于臃肿的接口就是对接口的污染，接口污染就是为接口添加了不必要的职责。

一个没有经验的设计师往往想节省接口的数目，将一些功能相近或相关功能的接口合并，并将这看成是代码优化的一部分。在接口中加一个新方法只是为了给实现类带来好处，以减少类的数目。持续这样做，接口就被不断污染，变胖。

实际上，类的数目根本不是什么问题，接口污染会带来维护和重用方面的问题。最常见的问题是为了重用被污染的接口，被迫实现并维护不必要的方法。如果一个客户程序依赖于一个含有它不使用的方法的类，但是其他客户程序却要使用该方法，那么当其他客户要求这个类改变时，就会影响到这个客户程序，这导致客户程序之间产生不必要的耦合关系，牵一发而动全身。胖类（fat class）导致它们的客户程序之间产生不正常的并且有害的耦合关系。当一个客户程序要求该胖类进行一个改动时，会影响到所有其他的客户程序。

因此，客户程序应该仅仅依赖于它们实际调用的方法。通过把胖类的接口分解成多个特定于客户程序的接口，可以实现这个目标。每个特定于客户程序的接口仅仅声明它的特定客户或者客户组调用的那些函数。接着，该胖类就可以继承所有特定于客户程序的接口，并实现它们。这就解决了客户程序和它们没有调用的方法间的依赖关系， 并使客户程序之间互不依赖。

分解胖接口，使客户程序只依赖它需要的方法，从设计上讲，简单易维护，重用度也高。应用了ISP之后，客户程序看到的是多个内聚的接口。

**与迪米特法则的关系：**

迪米特法则要求任何一个软件实体，除非绝对必要，不然不要与外界通信。即使必须进行通信，也应当尽量限制通信的广度和深度。

定制服务原则拒绝向客户提供不需要的行为，符合迪米特法则。

**接口隔离原则涉及的设计模式：**1、备忘录模式；2、迭代子模式。

### 优缺点

优点：

1、结构整洁清晰；

2、系统的可维护性。向客户端提供public接口是一种承诺，一个public接口一旦提供，就很难撤回。对于不必要的过多的承诺，会给系统的维护造成不必要的负担。

3、如果这些接口仅仅是内部使用，将接口隔离，也可以降低维护成本，因为一旦提供的服务出现变化，设计师知道哪些客户端会受到影响，哪些不会。

4、一个接口应该保证，实现该接口的实例对象可以只呈现为单一的角色；这样，当某个客户程序的要求发生变化，而迫使接口发生改变时，影响到其他客户程序的可能生性小。

### 如何实现

一个类对另一个类的依赖应该表现成依赖尽可能小的接口，不应该强迫客户依赖于它们不用的方法。如果客户端仅仅需要某一些方法，那么就应该向客户提供这些所需要的方法，而不要提供不需要的方法。

接口是对外的承诺，过多的承诺会给系统的维护造成不必要的负担。承诺的越少，越利于开发；但是开发的过程中也要注意一个度的概念，否则接口太多也不利于维护；

在我们进行设计的时候，一个重要的工作就是恰当地划分角色和角色对应的接口。

接口隔离原则是指使用多个专门的接口，而不使用单一的总接口。每一个接口应该承担一种相对独立的角色。

1、一个接口只代表一个角色，每个角色都有它特定的一个接口，此时这个原则可以叫做“角色隔离原则”。

2、接口仅仅提供客户端需要的行为，即所需的方法，应当为客户端提供尽可能小的单独的接口，而不要提供大的总接口。

3、使用接口隔离原则拆分接口时，首先必须满足单一职责原则，将一组相关的操作定义在一个接口中，且在满足高内聚的前提下，接口中的方法越少越好。

4、可以在进行系统设计时采用定制服务的方式，即为不同的客户端提供宽窄不同的接口，只提供用户需要的行为。

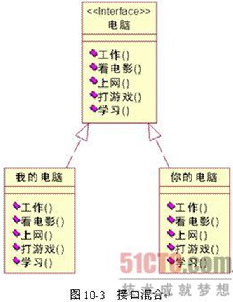
分离接口的方式一般分为两种，委托和多继承。前者把请求委托给别的接口的实现类来完成需要的职责，后者则是通过实现多个接口来完成需要的职责。两种方式各有优缺点，通常我们应该先考虑后一个方案，如果涉及到类型转换时则选择前一个方案。

### 示例

**示例1：电脑**

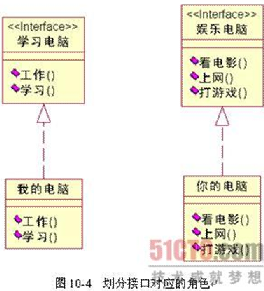
1、接口对应的角色

例如，我们将电脑的所有功能角色集合为一起，构建了一个接口，如下图所示：



此时，我的电脑和你的电脑要实现该接口，就必须实现所有的接口函数，显然接口混乱，并不能够满足实际的需求：

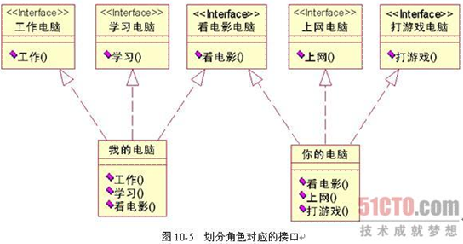
我的电脑可能是用来工作和学习的，你的电脑可能是用来看电影、上网和打游戏等娱乐活动的，那我们就可以将电脑的角色划分为两类，如下图所示：



2、角色对应的接口

对不同的客户端，同一个角色提供宽窄不同的接口，也就是定制服务，仅仅提供客户端需要的行为，客户端不需要的行为则隐藏起来。

对于图10-4中的接口定义，如果我的电脑除了工作和学习之外，还想上网，那就没办法了，必须实现娱乐电脑的接口，这样就必须实现它的所有接口函数了。此时我们需要将对应角色中的接口再进行划分，如图10-5所示。



这样，经过以上的划分，如果我的电脑想增加某一项功能，只需要继承不同的接口类即可。

由此可见，对接口角色的划分，是从大的类上进行划分的；对角色的接口进行的划分，是对类的接口函数的划分。它们两者由粗到细，实现了接口的完全分离。

**示例2：全文检索引擎的系统设计**

**反面例子**

一个接口负责所有的操作，从搜索功能到建立索引的功能，甚至包括搜索结果集合的功能均在一个接口内提供。

该解决方案违反了角色分割原则，把不同功能的接口放在一起，由一个接口给出包括搜索器角色、索引生成器角色以及搜索结果集角色在内的所有角色。

**不良结果**

子类需要全部实现这些接口方法，而对于具体子类，这些方法可能是不需要的。

由于接口大而全，修改的可能性大大增加。而一旦修改，其所有子类都需要修改，对子类的影响很大。

**搜索引擎的功能被分割为三个角色**

搜索器角色、索引生成器角色、搜索结果集角色。

索引生成器角色：由于索引生成因数据的格式不同而不同，故分为RdbIndexer和FileIndexer两种实现。FileIndexer类代表对诸如\*.txt、\*.html、\*.doc等文件类型的数据生成全文索引，而RdbIndexer则针对关系数据库的数据进行全文索引生成。这两个实现的都是索引生成器角色，即实现同一接口。

搜索器角色则是与索引生成器角色完全不同的角色，它提供用户全文搜索功能。用户传进一些关键字，搜索器角色则返回一个ResultSet对象。

搜索结果集角色就是ResultSet。它提供给客户对集合进行迭代走访的功能，如first()将光标移到集合的第一个元素；last()将光标移到集合的下一个元素。

## 迪米特法则 LOD

### 定义

一个软件实体应当尽可能少的与其他实体发生相互作用。这是对软件实体之间通信的限制，它要求限制软件实体之间通信的宽度和深度。

没有任何一个其他的OO设计原则象迪米特法则这样有如此之多的表述方式，如下几种：

1、只与你直接的朋友们通信。

2、不要跟“陌生人”说话。

3、最少知识原则（Least Knowledge Principle,LKP）。一个软件单位（比如对象）应当对其他的单位（比如对象）有尽可能少的了了解，而且局限于那些与本单位密切相关的软件单位。

迪米特法则可分为狭义法则和广义法则。

**狭义的迪米特法则：**如果两个类不必彼此直接通信，那么这两个类就不应当发生直接的相互作用。如果其中的一个类需要调用另一个类的某方法，可以通过第三者转发这个调用。与依赖倒转原则互补使用。

朋友圈的确定“朋友”条件：

1、当前对象本身（this）

2、以参量形式传入到当前对象方法中的对象

3、当前对象的实例变量直接引用的对象

4、当前对象的实例变量如果是一个聚集，那么聚集中的元素也都是朋友

5、当前对象所创建的对象任何一个对象

如果满足上面的条件之一，就是当前对象的“朋友”，否则就是“陌生人”

**广义的迪米特法则：**一个模块设计的好坏的一个重要标志就是该模块在多大程度上将自己的内部数据与实现的有关细节隐藏起来。它指对对象之间的信息流量、流向以及信息的影响的控制，主要是对信息隐藏的控制。

广义迪米特法则：

1、封装（信息隐藏）。

2、模块将它所有的实现细节隐藏起来，彻底地将提供给外界的API和自己的实现分割开来。

3、模块与模块间仅通过彼此的API相互通信，而不会理会模块内部的工件细节。

**著名的得墨忒耳律（Hie Law of Demeter）：**模块不应了解它所操作对象的内部情况。更准确地说，得墨忒耳律认为，类C的方法f只应该调用以下对象的方法：

1、C

2、由f创建的对象;

3、作为参数传递给f的对象；

4、由C的实体变量持有的对象。

方法不应调用由任何函数返回的对象的方法。换言之，只跟朋友谈话，不与陌生人谈话。

### 分析

**迪米特法则与设计模式**

门面（外观）模式、调停者（中介者）模式实际上就是迪米特法则的具体应用。

### 优缺点

1、迪米特法则的初衷在于降低类之间的耦合。由于每个类尽量减少对其他类的依赖，因此，很容易使得系统的功能模块功能独立，相互之间不存在（或很少有）依赖关系。

2、一个和其它实体关系很少的实体，在系统扩展时受的影响就会很小。当一个模块修改时，就会尽量少的影响其他的模块，扩展会相对容易。

接口隔离原则也是同样的道理，小的单独的接口，在系统扩展时受到的影响也会很小。不同的是，迪米特法则（广义迪米特）限制的是相互作用的深度和广度，而接口隔离原则限制的只是相互作用的广度。

**狭义的迪米特法则：**

1、可以降低类之间的耦合，但是会在系统中增加大量的小方法并散落在系统的各个角落。这些方法仅仅是传递间接地调用，因此与系统的业务逻辑无关。对于开发人员，从一张类图了解总体架构的时候，这些小的方法，往往会造成迷惑和困扰。

2、它可以使一个系统的局部设计简化，因为每一个局部都不会和远距离的对象有直接的关联，但是也会造成系统的不同模块之间的通信效率降低，使得系统的不同模块之间不容易协调。

**广义的迪米特法则，封装的意义在于：**

1、信息的隐藏可以使各个子系统之间脱耦，从而允许它们独立地被开发、优化、使用和修改。

2、信息的隐藏，可以促进软件的复用。由于每一个模块都不依赖于其他模块而存在，因此每一个模块都可以独立地在其他的地方使用。一个系统的规模越大，信息的隐藏就越重要，封装的效能也越明显。

3、脱耦，由于可以独立的同时开发，因而可以有效的加快系统的开发过程。而且，是维护过程变得容易，而不必担心对其他模块的影响。

**缺点：**

1、过度使用迪米特法则，也会造成系统的不同模块之间的通信效率降低，使系统的不同模块之间不容易协调等缺点。

2、因为迪米特法则要求类与类之间尽量不直接通信，如果类之间需要通信就通过第三方转发的方式，这就直接导致了系统中存在大量的中介类，这些类存在的唯一原因是为了传递类与类之间的相互调用关系，这就毫无疑问的增加了系统的复杂度。

### 如何实现

迪米特法则的主要用途在于控制信息的过载。在将迪米特法则运用到系统设计中时，要注意下面几点：

1、在类的划分上，应当创建有弱耦合的类。类之间的耦合越弱，就越有利于复用。即一个弱耦合的类一旦被修改，不会对有关系的类造成波及。

2、在类的结构设计上，每一个类都应当尽量降低成员的访问权限。即：一个类包装好各自的private内容。这样想要了解一个类的意义，就无需了解很多别的细节。一个类，不应当public自己的属性，而应当提供取值和赋值的方法让外界间接访问自己的属性。

3、在类的设计上，只要有可能，一个类应当设计成不变类。

4、在对其它类的引用上，一个对象对其实例的引用应当降到最低。

5、尽量限制局部变量的有效范围。

**迪米特法则与依赖倒转原则互补使用：**

为了克服狭义迪米特法则的缺点，可以使用依赖倒转原则（通俗的讲就是要针对接口编程，不要针对具体编程），引入一个抽象的类型引用“抽象陌生人”对象，使“某人”依赖于“抽象陌生人”。即：将“抽象陌生人”变为朋友。

这样调用方和被调用方之间有了一个抽象层，被调用方在遵循抽象层的前提下就可以自由的变化，此时抽象层成了调用方的朋友。

某人现在与一个抽象角色建立了朋友关系，这样做的好处是“朋友”可以随时将具体的陌生人替换掉。只要新的具体“陌生人”具有相同的抽象类型，则“某人”就无法区分他们。这就允许“陌生人”的具体实现可以独立于“某人”而变化。

**广义的迪米特法则在代码层次上的实现：在任何地方都可以声明变量，具有深远的意义：**

1、程序更易理解，更易维护

如果变量都在开头声明，在后面看到变量，需要回头去确认，不易理解；如果涉及变量的语句被删除，相应的变量往往会被遗漏；

2、限定有效范围

如果一个变量是在程序块外部被声明，这该程序块被执行前，该变量就已经被分配了内存；而当该程序块执行完毕后，这个变量所占据的内存空间还不能被释放，这显然不合理。

## 合成/聚合复用原则CARP（复用）

### 定义

黑盒BBP（Black Box Principle）原则

Composite/Aggregate Reuse Principle或CARP

在面向对象设计中，可以通过两种基本方法在不同的环境中复用已有的设计和实现，即通过组合/聚合关系或通过继承。

合成复用原则就是指在一个新的对象里通过关联关系（包括组合关系和聚合关系）来使用一些已有的对象，使之成为新对象的一部分；新对象通过委派调用已有对象的方法达到复用其已有功能的目的。

简言之：要尽量使用组合/聚合关系，少用继承。

### 分析

组合与继承都是重要的重用方法。

在OO开发的早期，继承被过度地使用。随着时间的发展，我们发现优先使用组合可以获得重用性与简单性更佳的设计。

如果为了复用，便使用继承的方式将两个不相干的类联系在一起，这样的方式是违反合成/聚合复用原则的，更进一步的后果便是违反里氏代换原则。

合成/聚合复用和里氏代换原则相辅相成，合成/聚合复用原则要求在复用时首先考虑合成/聚合关系，而里氏代换原则是要求我们在使用继承时，必须满足一定的条件。

**什么是合成和聚合**

对象的组合是一种通过创建一个组合了其它对象的对象，从而获得新功能的复用方法。将功能委托给所组合的一个对象，从而获得新功能。

合成（Composition）和聚合（Aggregation）都是关联（Association）的特殊种类。

聚合表示整体和部分的关系，表示“拥有”。如奔驰S360汽车，对奔驰S360引擎、奔驰S360轮胎的关系是聚合关系。在设计中，聚合不应该频繁出现，这样会增大设计的耦合度。

合成则是一种更强的“拥有”，部分和整体的生命周期一样。换句话说，合成是值的聚合（Aggregation by Value），而一般说的聚合是引用的聚合（Aggregation by Reference）。

**继承**

（类）继承是一种通过扩展一个已有对象的实现，从而获得新功能的复用方法。

泛化类（超类）可以显式地捕获那些公共的属性和方法。

特殊类（子类）则通过附加属性和方法来进行实现的扩展。

### 组合复用优缺点

组合/聚合可以使系统更加灵活，类与类之间的耦合度降低，一个类的变化对其他类造成的影响相对较少，因此一般首选使用组合/聚合来实现复用；其次才考虑继承，在使用继承时，需要严格遵循里氏代换原则，有效使用继承会有助于对问题的理解，降低复杂度，而滥用继承反而会增加系统构建和维护的难度以及系统的复杂度，因此需要慎重使用继承复用。

**优点：**

1、新对象存取成分对象的唯一方法是通过成分对象的接口

2、支持封装，这种复用是**黑箱复用**，因为成分对象的内部细节是新对象所看不见的

3、这种复用所需的依赖较少，选择性地调用成员对象的操作，被包含对象与容器对象之间的依赖关系比较少

4、通过获取指向其它的具有相同类型的对象引用，可以在运行期间**动态**地定义（对象的）组合

5、作为复用手段可以应用到几乎任何环境中去

6、聚合反映的是类之间"hasA"的关系，通过获取指向其它的具有相同类型的对象引用，可以在运行期间**动态**地定义（对象的）组合。被聚合对象的类型可以很容易地在运行期间发生变化，只要保证它们的接口相同，满足完全替换原则即可。

7、使用聚合可以更好地封装对象，使每一个类集中在单个职能上，类的继承层次也会保持较小的规模，不会造成类数量的爆炸。

**缺点：**

1、系统中会有较多的对象需要管理

2、为了能将多个不同的对象作为组合块（composition block）来使用，必须仔细地对接口进行定义。

3、它并不是面向对象语言直接支持的一个特性，用户必须编写一些代码来完成聚合功能。被聚合对象的接口必须遵从聚合类的要求，这种消息转发的方式又被称为"委托(Delegation)"。一般来说，聚合的结构比继承更难理解一些。

### 继承复用优缺点

**优点：**

1、新的实现较为容易，因为超类的大部分功能可以通过继承的关系自动进入子类

2、修改和扩展继承而来的实现较为容易

3、使用起来比较简单，因为面向对象的语言直接支持继承机制，对设计人员来说比较容易理解

**缺点：**

1、基类中往往定义了部分的实现，基类的实现暴露给派生类后，继承机制就会破坏数据和操作的封装，使派生类对基类产生较强的依赖。继承在某种程度上破坏了封装性，子类父类耦合度高，因为继承将超类的实现细节暴露给子类。由于超类的内部细节常常是对于子类透明的，所以这种复用是透明的复用，又称**白箱复用**。而对象组合则只要求被组合的对象具有良好定义的接口，耦合度低。

2、很难处理超类的变化。如果超类的实现发生改变，那么子类的实现也不得不发生改变。

3、继承只能在有限的环境中使用。

4、过大的继承树会严重降低软件的可读性，当客户程序使用继承树上的某个类时，它不仅需要了解该类的接口，还要了解所有直接和间接基类的接口，在一棵庞大的继承树上，类与类之间有很强的隐性耦合关系。

5、继承是在编译期间通过静态绑定实现的，仅使用继承机制，我们没有办法在程序运行期间动态改变属性或方法的绑定关系，这对于一些需要支持动态变化的需求来说，显然不能满足要求。

6、失去灵活性，使用具体类会给底层的修改带来麻烦。

### 如何使用

在C++等面向对象的语言中，继承机制极其强大，以至于很多软件工程师都有意无意地忽略了继承机制本身的缺陷。

继承固然很好，既封装了变化又实现了复用，但却不能滥用，应当把继承仅仅当做封装变化的方法，而不应当被用于从一般的对象生成特殊的对象。

**优先使用聚合，而不是继承**

继承反映的是类之间的isA的关系，聚合反映的是类之间hasA的关系。在不违反这个关系前提下，应该优先使用聚合而不是继承。应首先使用合成/聚合，合成/聚合则使系统灵活，其次才考虑继承，达到复用的目的。而使用继承时，要严格遵循里氏代换原则。

使用聚合时，必须遵循针对接口编程的设计原则，不能聚合某一个具体的派生类对象，而应该聚合该类的抽象接口，即通过指向接口类的引用或指针来访问对象，这种实现方法其实综合了聚合与继承两种机制的方式。

有效地使用继承会有助于对问题的理解，降低复杂度，而滥用继承会增加系统构建、维护时的难度及系统的复杂度。要避免在系统设计中出现，一个类的继承层次超过3层，否则需考虑重构代码，或者重新设计结构。当然最好的办法就是考虑使用合成/聚合原则。

**错误使用继承的情况**

“Is-A”是严格的分类学意义上定义，“Is－A”代表一个类是另外一个类的一种；“Has-A”代表一个类是另外一个类的一个角色，而不是另外一个类的特殊种类。

错误的使用继承而不是合成/聚合的一个常见原因是错误的把“Has-A”当成了“Is－A”。如果两个类是“Has-a”关系那么应使用合成、聚合，如果是“Is-a”关系那么可使用继承。

如果我们把“人”当成一个类，然后把“雇员”，“经理”，“学生”当成是“人”的子类。这个的错误在于把“角色”的等级结构和“人”的等级结构混淆了。“经理”，“雇员”，“学生”是一个人的角色，一个人可以同时拥有上述角色。

继承，是静态的，如果按继承来设计，会使得一个人在成为雇员后，永远是雇员，而无法成为经理、雇员、学生等，导致角色无法互换，这是不合理的。

正确的设计是有个抽象类“角色”，“人”可以拥有多个“角色”（聚合），“雇员”，“经理”，“学生”是“角色”的子类。

另外一个就是只有两个类满足里氏代换原则的时候，才可能是“Is－A”关系。也就是说，如果两个类是“Has-A”关系，但是设计成了继承，那么肯定违反里氏代换原则。

**什么时候使用继承**

要正确的选择合成/复用和继承，必须透彻的理解里氏代换原则和Coad法则。Coad法则由Peter Coad提出，只有当以下的Coad条件全部被满足时，才应当使用继承作为复用的工具：

1、子类表达了“是一个…的特殊类型”，而非“是一个由…所扮演的角色”，子类是超类的一个特殊种类，而不是超类的一个角色，也就是区分Has-A和Is-A。只有Is-A关系才符合继承关系，Has-A关系应当用聚合来描述。

2、子类的一个实例永远不需要转化（transmute）为其它类的一个对象。如果不能肯定，就不要使用继承。

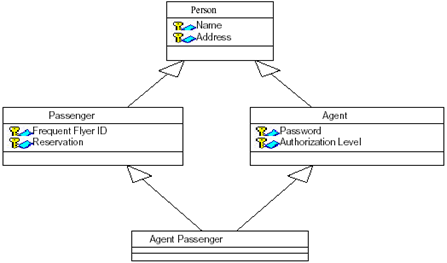
3、子类是对其父类的职责（responsibility）进行扩展，而非重写（override）或废除（nullify）。如果一个子类需要大量的置换掉超类的行为，那么这个类就不应该是这个超类的子类。

4、不要从工具类继承。子类没有对那些仅作为一个工具类（utility class）的功能进行扩展。

5、对于一个位于实际的问题域（Problem Domain）的类而言，其子类特指一种角色、交易或设备。只有在分类学角度上有意义时，才可以使用继承。

### 示例

**示例1：继承/组合示例**



1、“是一个…的特殊类型”，而非“是一个由…所扮演的角色”。失败，乘客是人所扮演的一种角色，代理人亦然。

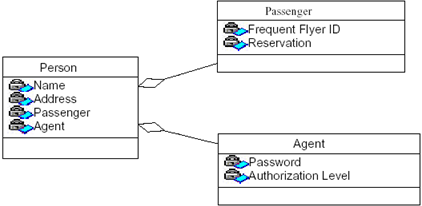
 2、永远不需要转化。失败，随着时间发展，一个Person的子类实例可能会从Passenger转变成Agent，再到Agent Passenger。

3、扩展，而非重写和废除。通过。

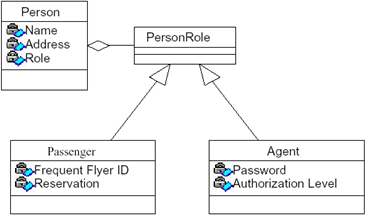
4、不要扩展一个工具类。通过。

5、在问题域内，特指一种角色，交易或设备。失败，Person不是一种角色，交易或设备。

继承并非适用于此处！使用组合进行挽救！



**示例2：继承/组合示例**



1、“是一个…的特殊类型”，而非“是一个由…所扮演的角色”。通过，乘客和代理人都是特殊类型的人所扮演的角色。

2、永远不需要转化。通过，一个Passenger对象将保持不变；Agent对象亦然。

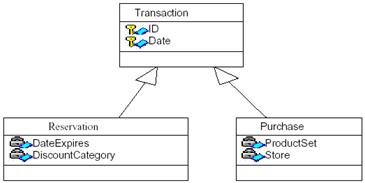
3、扩展，而非重写和废除。通过。

4、不要扩展一个工具类。通过。

5、在问题域内，特指一种角色，交易或设备。通过，PersonRole是一种类型的角色。

继承适用于此处！

**示例3：继承/组合示例**



1、“是一个…的特殊类型”，而非“是一个由…所扮演的角色”。通过，预订和购买都是一种特殊类型的交易。

 2、永远不需要转化。通过，一个Reservation对象将保持不变；Purchase对象亦然。

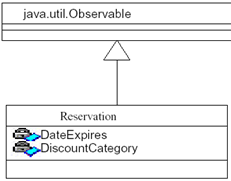
 3、扩展，而非重写和废除。通过。

 4、不要扩展一个工具类。通过。

 5、在问题域内，特指一种角色，交易或设备。通过，是一种交易。

继承适用于此处！

**示例4：继承/组合示例**



1、 “是一个…的特殊类型”，而非“是一个由…所扮演的角色”。失败，预订不是一种特殊类型的observable。

2、永远不需要转化。通过，一个Reservation对象将保持不变。

3、扩展，而非重写和废除。通过。

4、不要扩展一个工具类。失败，Observable就是一个工具类。

5、在问题域内，特指一种角色，交易或设备。不适用，Observable是一个工具类，并非一个问题域的类。

继承并非适用于此处！

# 包的设计原则

**PCRF或OCS的一个子目录（会生成一个so），就是这里的“包”的概念。**

随着应用程序的规模和复杂度的增加，需要在更高的层次对它们进行组织。类对于小型应用程序来说是非常方便的组织单元，但是对于大型应用程序来说，如果仅仅使用类作为唯一的组织单元，就会显得粒度过细。因此，需要比类大的东西来辅助大型应用程序的组织，这个东西就是包 （package）。

在UML的概念中，包可以用作包容一组类的容器。通过把类组织成包，我们可以在更高层次的抽象上来理解设计。我们也可以通过包来管理软件的开发和发布，目的就是根据一些原则对应用程序中的类进行划分，然后把这些划分后的类分配到包中。

类经常会和其他类之间存在着依赖关系，这些依赖关系还经常会跨越包的边界。因此，包之间也会产生依赖关系。包之间的依赖关系展现了应用程序的高层组织结构，我们应该对这些关系进行管理。

## 粒度：包（类库、DLL）内聚原则

3个关于包的内聚性原则，可以帮助开发者决定如何把类划分到包中。这些原则依赖于这样的事实：至少已经存在一些类，并且它们之间的相互关系也已经确定。因此，这些原则是根据“自底向上”的观点对类进行划分的。

我们习惯于认为内聚性只不过是指一个模块执行一项并且仅仅一项功能。然而，3个关于包内聚性的原则描述了有关内聚性的更丰富的变化。在选择要共同组织到包中的类时，必须要考虑可重用性与可开发性（develop ability）之间的相反作用力，在这些作用力和应用的需要之间进行平衡不是一件简单的工作。此外，这个平衡几乎总是动态的，今天看起来合适的划分到了明年也许就不再合适了。因此，当项目的重心从可开发性向可重用性转变时，包的组成很可能会变动并随时间而演化。

### 重用发布等价原则REP

定义：重用的粒度就是发布的粒度。一个包中的软件要么都是可重用的，要么都是不可重用的。

当你重用一个类库时，对这个类库的作者有什么期望呢？你想得到好的文档，可以工作的代码，规格清晰的接口等。但是，你还会有其他的期望。

首先，你希望代码的作者能保证为你维护这些代码，这样才值得你在重用这些代码上花费时间。其次，你希望代码的作者在计划对代码的接口和功能进行任何改变时，提前通知你。但是仅仅通知是不够的，代码的作者必须尊重你拒绝使用任何新版本的权力。否则，当你处在开发进度中的一个关键时刻时，他可能发布了一个新的版本。或者他对代码进行了改变，之后就干脆再也无法与你的系统兼容了。

无论在哪种情况下，如果你决定不接纳新版本，作者必须保证对于你所使用的旧版本继续提供一段时间的支持。这段时间也许只有3个月，或者长达1年，你们两个人之间必须就这些事情进行磋商。但是，他不能和你断绝关系并且拒绝对你提供支持。如果他不同意对你使用的稍旧一点的版本提供支持，那么你就应该认真的考虑一下是否情愿忍受对方反复无常的变化，而继续使用他的代码。

这个问题主要是行政问题。如果有其他的人将要重用代码，就必须要进行行政和支持方面的工作，但是这些行政上的问题对于软件包的结构具有深刻的影响。为了给重用者提供所需的保证，代码的作者必须把它们的软件组织到一个可重用的包中，并且通过版本号对这些包进行跟踪。

REP指出，一个包的重用粒度（granule of reuse）可以和发布粒度（granule of release）一样大。重用的任何东西都必须同时被发布和跟踪。简单的编写一个类，然后声称它是可重用的做法是不现实的。只有在建立一个跟踪系统，为潜在的使用者提供所需的变更通知、安全性以及支持后，重用才有可能。REP带给了我们关于如何把设计划分到包中的第一个提示。

那么，关于包的内部结构方面，我们学到了什么呢？我们必须从潜在的重用者的角度去考虑包的内容。如果一个包中的软件是用来重用的，那么它就不能再包含不是为了重用的目的而设计的代码。一个包中的代码要么都是可重用的，要么都是不可重用的。

可重用性不是唯一的标准，我们也要考虑重用这些软件的人。一个容器类库是可重用的，一个金融方面的框架也是可重用的，但是我们不希望把它们放进同一个包中，很多希望重用容器类库的人可能对于金融框架根本不感兴趣。因此，我们希望一个包中的所有类对于同一类用户来说都是可重用的，我们不希望一个用户发现包中所包含的类中，一些是他所需要的，另一些对他却完全没用。

### 共同封闭原则CCP

简述：包所有类应该对于同一类性质的变化应该是共同封闭的。一个变化若对一个包产生影响，则将对该包中的所有类产生影响，而对于其他的包不造成任何影响。

这是单一职责原则（SRP）对于包的重新规定。正如SRP规定的一个类不应该包含多个引起变化的原因那样，这个原则规定了一个包不应该包含多个引起变化的原因。

在大多数的应用中，可维护性的重要性是超过可重用性的。如果一个应用中的代码必须更改，那么我们宁愿更改都集中在一个包中，而不是分布在多个包中。如果更改集中在一个单一的包中，那么我们仅仅需要发布那一个更改了的包，不依赖于那个更改了的包的其他包则不需要重新验证或重新发布。

CCP鼓励我们把可能由于同样的原因而改变的所有类共同聚集在同一个地方。如果两个类之间有非常紧密的绑定关系，不管是物理上的还是概念上的，那么它们总会一同进行变化，因而它们应该属于同一个包中，这样做会减少软件的发布、重新验证、重新发行的工作量。

这个原则和开放封闭原则（OCP）密切相关，本原则中“封闭”这个词和OCP中的具有同样的含义。OCP规定了类对于修改应该是封闭的，对于扩展应该是开放的。但是，100％的封闭是不可能的，应当进行有策略的封闭。我们所设计的系统应该对于我们经历过的最常见的变化做到封闭。

OCP通过把对于一些确定的变化类型开放的类共同组织到同一个包中，从而增强了上述内容。因而，当需求中的一个变化到来时，那个变化就会很有可能被限制在最小数量的包中。

### 共同重用原则CRP

简述：一个包中的所有类应该是共同重用的。如果重用包中的一个类，就应该重用包中所有类。相互之间没有紧密联系的类不应该在同一个包（类库、DLL）中。

类很少会孤立地重用。一般来说，可重用的类需要与作为该可重用抽象一部分的其他类协作，CRP规定了这些类应该属于同一个包。在这样的一个包中，我们会看到类之间有很多的互相依赖。

一个简单的例子是容器类以及与它关联的迭代器类。这些类彼此之间紧密耦合在一起，因此必须共同重用，所以它们应该在同一个包中。

但是，CRP告诉我们的不仅仅是什么类应该共同放入一个包中，它还告诉我们什么类不应该放入同一个包中。当一个包使用了另一个包时，它们之间会存在一个依赖关系。也许一个包仅仅使用了另外一个包中的一个类，然而，那根本不会削弱这两个包之间的依赖关系，使用者包依然依赖于被使用的包。每当被使用的包发布时，使用者包必须进行重新验证和重新发布，即使发布的原因仅仅改变了一个使用者包根本不关心的类，也必须要这样做。

此外，包也经常以共享库、DLL、JAR等物理表示的形式出现。如果被使用的包以JAR的形式发布，那么使用这个包的代码就依赖于整个JAR。对JAR的任何修改——即使所修改的是与用户代码无关的类，也会造成这个JAR的一个新版本发布。这个新JAR也要重新发行，并且使用这个JAR的代码也要进行重新验证。

因此，我想确信当我依赖于一个包时，我将依赖于那个包中的每一个类。换句话说，我想确信我放入一个包中的所有类是不可分开的，仅仅依赖于其中一部分的情况是不可能的。否则，我将要进行不必要的重新验证和重新发行，并且会白费相当数量的努力。

因此，CRP告诉我们更多的是，什么类不应该放在一起。CRP规定相互之间没有紧密联系的类不应该放在同一个包中。

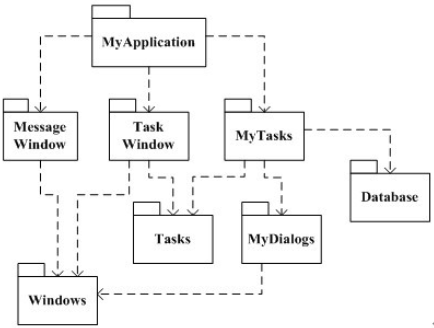
## 稳定性：包（类库、DLL）耦合原则

接下来的3个原则用来处理包之间的关系。这里，我们会再次碰到可开发性与逻辑设计之间的冲突力（tension）。来自技术和行政方面的作用力都会影响到包的组织结构，并且这种作用力还是易变的。

### 无环依赖原则ADP

简述：在包的依赖图中，不允许存在环。

下图展示了组成一个应用程序的非常典型的包结构。请注意，该机构是一个有向图（directed graph）。其中，包是结点（node），依赖关系是有向边（directed edge）。



现在，请注意另外一件事情。无论从哪个包开始，都无法沿着依赖关系而回绕到这个包。该结构中没有环，它是一个有向无环图（DAG）。

当负责MyDialogs的团队发布了该包的一个新版本时，会很容易找出受影响的包：只需逆着依赖关系指向寻找即可。因此，MyTasks和MyApplication都会受到影响。当前工作于这两个包的开发人员就要决定何时应该和MyDialogs的新版本集成。

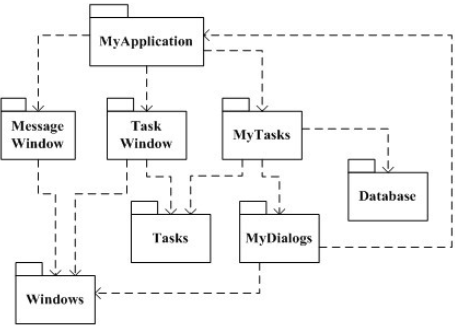
还要注意，当MyDialogs发布时，完全不会影响到系统中许多其他包。它们不知道MyDialogs，并且也不关心何时对MyDialogs进行了更改。这很好。这意味着发布MyDialogs的影响相对较小。

当工作于MyDialogs包的开发人员想要运行该包的测试时，只需把他们的MyDialogs版本和当前正使用的Windows包的版本一起编译、链接即可，不会涉及到系统中任何其他包。这意味着工作于MyDialogs的开发人员只需较少的工作即可建立一个测试，而且他们要考虑的变数（variable）也不多。

在发布整个系统时，是自底向上进行的。首先编译、测试已经发布Windows包。接着是MessageWindow和MyDiaolgs。在它们之后是Tasks，然后是TaskWindow和Database。接着是MyTasks，最后是MyApplication。这个过程非常清楚并且易于处理。我们知道如何去构建系统，因为我们理解系统各个部分的依赖关系。

**包依赖关系图中环造成的影响**

如果一个新需求迫使我们更改MyDialogs中的一个类去使用MyApplication中的一个类，这就产生了一个依赖关系环，如下图：



这个依赖环会导致一些直接后果。例如，工作于MyTasks包的开发人员知道，为了发布MyTasks包，它们必须得兼容Task、MyDialogs、Database以及Windows。然而，由于依赖关系的存在，它们现在必须也兼容MyApplication、TaskWindow、以及MessageWindow。也就是说，现在MyTasks依赖于系统中所有其他的包，这就致使MyTasks非常难以发布。MyDialogs有着同样的问题，该依赖关系会迫使MyApplication、MyTasks以及MyDialogs总是同时发布，它们实际上已经变成了同一个大包。于是，在这些包上工作的所有开发人员，他们彼此之间的发布行动要完全一致，因为它们必须都要使用彼此间完全相同的版本。

这还只是部分问题。考虑一下在想要测试MyDialogs包时会发生什么。我们必须要链接进系统中所有其他的包，包括Database包。这意味着仅仅为了测试MyDialogs就必须做一次完整的构建。

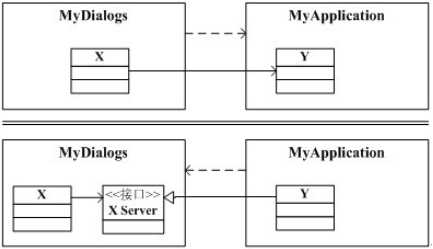
如果想知道为何必须要链接进这么多不同的库，以及这么多其他人的代码，只需运行一个某个类的简单单元测试即可，或许这是因为依赖关系图中存在环的缘故，这种环使得非常难以对模块进行隔离。单元测试和发布变得非常困难且易于出错。而且，在C++中，编译时间会随模块的数目成几何级数增长。

此外，如果依赖关系图中存在环，就很难确定包构建的顺序。事实上，也许就不存在恰当的顺序。对于像Java一样要从编译过的二进制文件中读取它们的声明的语言来说，这会导致一些非常讨厌的问题。

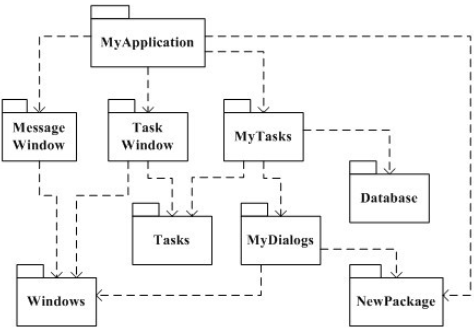
**解除依赖环**

任何情况下，都可以解除包之间的依赖环并把依赖关系图恢复为一个DAG。有两个主要的方法：

1、使用依赖倒置原则（DIP）。针对上图这样的情况，可以创建一个具有MyDialogs需要的接口的抽象基类。然后，把该抽象基类放进MyDialogs中，并使MyApplication中的类从其继承。这就倒置了MyDialogs和MyApplication间的依赖关系，从而解除了依赖关系环。参见下图：



2、新创建一个MyDialogs和MyApplication都依赖的包，把MyDialogs和MyApplication都依赖的类移到这个新包中。参见下图：



第2个解决方案意味着，在需求改变面前，包的结构是不稳定的。事实上，随着应用程序的增长，包的依赖关系结构会抖动（jitter）和增长。因此，必须始终要对依赖关系结构中环的情况进行监控。如果出现了环，就必须要使用某种方法把其解除。有时这意味着要创建新的包，致使依赖关系结构增长。

**自顶向下设计**

讨论到现在，我们可以得出一个必然的结论：不能自顶向下设计包的结构。这意味着，包结构不是设计系统时首先考虑的事情之一。事实上，包结构应该时随着系统的增长、变化而逐步演化的。

也许你会认为这是违反直觉的，我们已经认为像包这样的大粒度分解同样也是高层的功能分解。当我们看到一个像包依赖关系结构这样的大粒度分组时，就会觉得包应该以某种方式描绘了系统的功能。然而，这可能不是包依赖关系图的一个属性。

事实上，包的依赖关系图和描绘应用程序的功能之间几乎没有关系。相反，它们是应用程序可构建性的映射图。这就是为何不在项目开始时设计它们的原因。在项目开始时，没有软件可构建，因此也无需构建映射图。但是，随着实现和设计初期累积的类越来越多，对依赖关系进行管理的需要就不断增长。此外，我们也想尽可能地保持更改的局部化，所以我们开始关注SRP和CCP，并把可能会一同变化的类放在一起。

随着应用程序的不断增长，我们开始关注创建可重用的元素。于是，就开始使用CRP来指导包的组合。最后，当环出现时，就会使用ADP，从而，包的依赖关系图会出现抖动以及增长。

如果在设计任何类之前试图去设计包的依赖关系结构，那么很可能会遭受惨败。我们对于共同封闭还没有多少了解，也还没有觉察到任何可重用的元素，从而，几乎当然会创建产生依赖环的包。所以，

### 稳定依赖原则SDP

简述：朝着稳定依赖的方向进行依赖。

应该把封装系统高层设计的软件（比如抽象类）放进稳定的包中，不稳定的包中应该只包含那些很可能会改变的软件（比如具体类）。

设计不能是完全固定的。要使设计可维护，某种程度的易变性是必要的。我们通过遵循共同封闭原则（CCP）来达到这个目标，使用这个原则，可以创建对某些变化类型敏感的包。这些包被设计成可变的，我们期望它们的变化。

对于任何包而言，如果期望它是可变的，就不应该让一个难以改变的包依赖于它！否则，可变的包同样也会难以更改。

你设计了一个易于更改的包，其他人只要创建一个对它的依赖就可以使它变得难以更改，这就是软件的反常特性。没有改变你的模块中任何一行代码，可是它突然之间就变得难以更改了。通过遵循SDP，我们可以确保那些打算易于更改的模块不会被那些比它们难以更改的模块所依赖。

**并非所有的包都应该是稳定的**

如果一个系统在所有的包都是最大程度稳定的，那么该系统就是不能改变的，这不是所希望的情形。事实上，我们希望所设计出来的包结构中，一些包是不稳定的而另外一些是稳定的。

可改变的包位于顶部并依赖于底部稳定的包。把不稳定的包放在图的顶部是一个有用的约定，因为任何一个向上的箭头都意味着违反了SDP。

系统中的某些软件不应该经常改变，该软件代表着系统的高层构架和设计决策，我们希望这些构架决策是稳定的。因此，应该把封装系统高层设计的软件放进稳定的包中，不稳定的包中应该只包含那些很可能会改变的软件。

然而如果把高层设计放进稳定的包中，那么体现高层设计的源代码就会难以更改，这会使设计变得不灵活。怎样才能让一个具有最高稳定性的包足够灵活，可以经受得住变化呢？在OCP中可以找到答案。OCP原则告诉我们，那些足够灵活可以无需修改即可扩展的类是存在的，并且是所希望的。哪种类符合OCP原则呢？——抽象类。

### 稳定抽象原则SAP

简述：包的抽象程度应该和稳定的程度一致。

一个稳定的包应该也是抽象的，一个不稳定的包不应该是抽象的。

该原则把包的稳定性和抽象性联系起来。它规定，一个稳定的包应该也是抽象的，这样它的稳定性就不会使其无法扩展。另一方面，它规定，一个不稳定的包应该是具体的，因为它的不稳定性使得其内部具体代码易于更改。

因此，如果一个包是稳定的，那么它应该也要包含一些抽象类，这样就可以对它进行扩展。可扩展的稳定包是灵活的，并且不会过分限制设计。

SAP和SDP结合在一起形成了针对包的DIP原则。这样说是准确的，因为SDP规定依赖应该朝着稳定的方向进行，而SAP则规定稳定性意味着抽象性。因此，依赖应该朝着抽象的方向进行。

然而，DIP是一个处理类的原则。类没有灰度（the shades of grey）的概念。一个类要么是抽象的，要么不是。SDP和SAP的结合是处理包的，并且允许一个包是部分抽象、部分稳定的。

# 其它扩展原则

**缺省抽象DAP(Default Abstraction Principle)原则：**在接口和实现接口的类之间引入一个抽象类，这个类实现了接口的大部分操作。

**接口设计IDP(Interface Design Principle)原则：**规划一个接口而不是实现一个接口。

**DCSP(Don't Concrete Supperclass Principle)：**不要构造具体的超类原则，避免维护具体的超类。

## 依赖注入DI

我们先看一些生活中的例子，帮助理解依赖注入：

**主机和内置硬盘**

对于一体机而言，一旦出现了问题，我们可能无法准确地判断到底是什么零部件出现了问题，可能是CPU坏了，也有可能是主板烧了，还有可能是内存松动了，还有可能是电脑硬盘出了问题。

从这个例子，我们可以看到部件之间“紧密耦合”的产生的问题：无法准确的定位和诊断故障所在。这种情形，在软件工程的理论中，称之为可理解性和可测试性差。

如果修理电脑的硬盘，在修理过程中就必须小心翼翼，不要把其它的部件再搞坏，比如不慎把内存碰松动了，硬盘是修好了，但整台电脑仍然无法使用。这种情形，在软件工程的理论中，称之为可修改性差。

可理解性、可测试性、可修改性组成了系统的可维护性，一体机的可维护性就表现得比较差。

**主机和USB设备**

大家对USB接口和设备应该都很熟悉。自从有了USB接口，给我们使用电脑带来了很大的方便，现在有很多的外部设备都支持USB接口。从软件工程角度，我们分析一下USB带来的好处：

1、USB设备作为主机的外部设备，在插入主机之前，与主机没有任何的关系，两者都可独立进行测试，无论两者中的任何一方出现什么的问题，都不会影响另一方的运行，所以可维护性比较好。

2、同一个USB设备可以插接到不同的支持USB的任何主机，也就是USB设备可以被重复利用，所以可复用性比较好。

3、支持热插拔，只要是支持USB接口的设备都可以接入，所以可扩展性比较好，非常灵活。

**依赖注入**

2004年，Martin Fowler从另一个角度来思考这个问题，提出了“哪些方面的控制被反转了？”这样一个问题，并给出了答案：“依赖对象的获得被反转”。于是，他给“控制反转”取了一个他认为更合适的名字叫做“依赖注入（Dependency Injection）”。他的这个答案，实际上点明了实现IoC理论的解决方法。所谓依赖注入，就是由IoC容器在运行期间，动态地将某种依赖关系注入到对象之中。

依赖注入(DI)和控制反转(IoC)是从不同的角度的描述的同一件事情，都是指通过引入第三方，即IoC容器，实现软件系统中对象之间的解耦。

控制反转能够带给系统开发的好处，与USB机制带来的好处基本类似，而且依赖注入的实现跟USB机制也完全一样。USB机制是现实中依赖注入的很好的案例。我们用一个实际的例子，分析一下USB机制：

**任务：主机通过USB接口读取一个文件。**

思路：首先，必须制定一个USB接口标准，主机对USB设备的访问严格按照USB接口标准，USB设备提供的功能也必须符合USB接口标准。

当主机需要获取一个文件，它直接去读取USB接口，根本不会关心USB接口上连接的是什么设备。

如果我给主机连接上一个U盘，那么主机就从U盘上读取文件；如果我给主机连接上一个外接硬盘，那么主机就从外接硬盘上读取文件。选取何种外部设备的权力由我说了算，也就是控制权归我。

至此，依赖注入的思路已经非常清楚：当主机需要读取文件的时候，我就把它所要依赖的外部设备，挑出来一个，帮他挂接上。这个过程就是一个被依赖的对象在系统运行时被注入另外一个对象内部的过程。在这个过程中，我就起到了IoC容器的作用。

**我们再把依赖注入应用到软件工程中：**

Class A依赖于Class B，当Class A需要用到Class B的时候，IoC容器就会立刻创建一个Class B送给Class A使用。IoC容器就是一个类制造工厂，你需要什么，直接来拿，直接用就可以了，而不需要去关心你所用的东西是如何制成的，也不用关心最后是怎么被销毁的，这一切全由IoC容器包办。

# 设计过程

## 分层设计

## 模块设计

## 包设计

so、DLL、JAR，可以算作为包。具体见面向对象设计原则。

包将相关组件的集合聚集为一个逻辑上内聚的**物理**单位。

类是程序员们实现模块化（modularity）的最佳方式。不过模块化是个很庞大的话题，其影响范围要远远超出类。软件开发的进展在很大程度上要归功于我们在编程时进行工作的粒度的增长。首先是语句，这在当时算得上是自从机器指令以来迈进的一大步。接下来就是子程序，再后来则是类。

如果我们能有更好的工具来把对象聚合起来，我们就可能更好地朝着抽象和封装的目标迈进。Ada语言早在十多年前就已经支持包（package）的概念了，现今Java语言也支持包了。如果你所用的编程语言不能直接支持包的概念，你也可以自行创建自己的包，并通过遵循下列编程标准来强制实施你的包。

随着应用程序的规模和复杂度的增加，需要在更高的层次对它们进行组织。类对于小型应用程序来说是非常方便的组织单元，但是对于大型应用程序来说，如果仅仅使用类作为唯一的组织单元，就会显得粒度过细。因此，就需要比类大的东西来辅助大型应用程序的组织，这个东西就是包（package）。

## 组件设计

一个组件的物理接口就是它的头文件中的所有东西。

组成一个组件的.c文件和.h文件的根名称应该严格匹配。

一个组件是由一个.c文件和一个.h文件组成的物理实体，它具体表达了一个逻辑抽象的具体实现。一个组件一般包含一个、两个甚至多个类，以及需要用来支持全部抽象的适当的自由运算符。

如果同样的子系统着眼于用最小CCD的方法来设计，则大多数（如果不是全部的话）的循环依赖会被消除。因此有可能在隔离的情况下研究子系统的部件，以便测试、检验、调整甚至替换它们，而无论在主观上还是在物理上都不必涉及整个子系统。换句话说，有效地减少组件间的相互依赖（通过CCD来量化）可以提高可理解性，因而也提高了可维护性。

可理解性是几个难以量化的性质之一，但是它对于最小化组之间的依赖非常有好处。

## 类设计

### 抽象

### 封装

封装是指把数据，以及操作数据的相关方法组合在同一个单元中。通过封装，我们可以把类作为软件中的基本复用单元，提高其内聚度、降低其耦合度。

面向对象中最基本的封装单元一一类也是信息（数据和函数）隐藏的理想边界。信息隐藏将类的实现细节与类的客户接口严格分开，访问该类的客户程序只需要知道类的接口就可以完成各种功能操作，而无需了解类的内部细节。

封装强调的是对现实世界的模拟，软件中的每一个类实际上都反映了某一种事物的共性。在软件开发时，如果我们忘记了类的这个特点，把许多不相下的事物、不相干的属性都强行封装在一个类中，就会造成该类的内聚度降低，并进而影响到软件系统的设计质量。

封装是一个比抽象更强的概念。抽象通过提供一个可以让你忽略实现细节的模型来管理复杂度，而封装则强制阻止你看到细节——即便你想这么做。

这两个概念之所以相关，是因为没有封装时，抽象往往很容易被打破。依我的经验，要么就是封装与抽象两者皆有，要么就是两者皆失，除此之外没有其他可能。

**尽可能地限制类和成员的可访问性：**让可访问性（accessibility）尽可能低是促成封装的原则之一。

**不要公开暴露成员数据暴露成员：**数据会破坏封装性，从而限制你对这个抽象的控制能力。

#### 信息隐藏

信息隐藏是结构化程序设计与面向对象设计的基础之一。结构化设计里面的“黑盒子”概念就是来源于信息隐藏。在面向对象设计中，它又引出了封装和模块化的概念，并与抽象的概念紧密相关。

**找出容易改变的区域**

好的程序设计所面临的最重要挑战之一就是适应变化。目标应该是把不稳定的区域隔离出来，从而把变化所带来的影响限制在一个子程序、类或者包的内部。下面给出你应该采取的应对各种变动的措施：

1、找出看起来容易变化的项目。如果需求做得很好，那么其中就应该包含一份潜在变化的清单，以及其中每一项变化发生的可能性。在这种情况下，找出潜在的变化就非常容易了。如果需求中没有包括潜在的变化，请参考后面有关“在任何项目中都容易发生变化的区域”的讨论。

2、把容易变化的项目分离出来。把第一步中找出的容易变化的组件单独划分成类，或者和其他容易同时发生变化的组件划分到同一个类中。

3、把看起来容易变化的项目隔离开来。设法设计好类之间的接口，使其对潜在的变化不敏感。设计好类的接口，把变化限制在类的内部，且不会影响类的外部。任何使用了这个将会发生变化的类的其他类都不会察觉到变化的存在。类的接口应该肩负起保护类的隐私的职责。

**下面是一些容易发生变化的区域：**

业务规则：业务规则很容易成为软件频繁变化的根源。国会改变了税率结构，工会重新谈判合同的内容，保险公司改变了它的税率表……如果你遵循信息隐藏的原则，那么基于这些业务规则的逻辑就不应该遍布于整个程序，而是仅仅隐藏于系统的某个角落，直到需要对它进行改动，才会把它拎出来。

### 继承

通过继承，基类的属性和方法被遗传给派生类。在继承时，派生类可以主动选择继承的程度和范围，也可以选择对基类的某些操作进行变异。从设计初衷来看，继承机制是最大限度复用代码的一种手段。

除了继承以外，还可以使用聚合、委托、组合等关系来实现类与类之间的相关性。

**只有一个派生类的基类也值得怀疑：**每当我看到只有一个派生类的基类时，我就怀疑某个程序员又在进行“提前设计”了——也就是试图去预测未来的需要，而又没有真正了解未来到底需要什么。为未来要做的工作着手进行准备的最好方法，并不是去创建几层额外的、“没准以后哪天就能用得上的”基类，而是让眼下的工作成果尽可能地清晰、简单、直截了当。也就是说，不要创建任何并非绝对必要的继承结构。

**为什么有这么多关于继承的规则**

这一节给出了许多规则，它们能帮你远离与继承相关的麻烦。所有这些规则背后的潜台词都是在说，继承往往会让你和程序员的首要技术使命（即管理复杂度）背道而驰。从控制复杂度的角度说，你应该对继承持有非常歧视的态度。下面来总结一下何时可以使用继承，何时又该使用包含：

1、如果多个类共享数据而非行为，应该创建这些类可以包含的共用对象。

2、如果多个类共享行为而非数据，应该让它们从共同的基类继承而来，并在基类里定义共用的子程序。

3、如果多个类既共享数据也共享行为，应该让它们从一个共同的基类继承而来，并在基类里定义共用的数据和子程序。

4、当你想由基类控制接口时，使用继承；当你想自己控制接口时，使用包含。

### 多态

多态性是面向对象技术中最重要但也最难理解的一个概念。C++和大多数面向对象的语言一样，都提供了两种类型的多态性：一种是编译时绑定的静态多态性，另一种是运行时绑定的动态多态性。

**静态多态性：**两行代码调用的函数名完全一样，但函数参数类型不同，因此，两行代码调用的其实是两个不同的函数。这种静态多态性也被称为**函数多态性**，c++实现函数多态性的机制是函数重载Function Overloading。还有一种利用c++模板来实现静态多态性，这种多态性也是静态绑定的，被称为**模板多态性**。

与静态多态性相比，动态多态性对于面向对象设计的价值更大一些。只有理解了动态多态性的原理，才能参透面向对象设计理念的精髓。在c++语言里，虚函数和函数覆盖Function Overriding机制是实现动态多态性的重要手段。

只有采用"指针->函数()"或"引用变量.函数()"的方式调用C++类中的**虚**函数才会执行动态绑定。

C++语言中的**非虚**函数不具备动态绑定的特征；使用”类名::函数”或”对象名.函数”，一定是静态绑定。

### 应该避免的类

1、**避免创建万能类**（god class）：要避免创建什么都知道、什么都能干的万能类。如果一个类把工夫都花在用Get()方法和Set()方法向其他类索要数据（也就是说，深入到其他类的工作中并告诉它们该如何去做）的话，请考虑是否应该把这些功能组织到其他那些类中去，而不要放到万能类里。

2、消除无关紧要的类：如果一个类只包含数据但不包含行为的话，应该问问自己，它真的是一个类吗？同时应该考虑把这个类降级，让它的数据成员成为一个或多个其他类的属性。

3、避免用动词命名的类：只有行为而没有数据的类往往不是一个真正的类。请考虑把类似Databaselnitialization（数据库初始化）或StringBuilder（字符串构造器）这样的类变成其他类的一个子程序。

### 类设计的基本经验

1、加强内聚，松散耦合。

2、好的封装性。

3、类的粒度要合理。

4、实现类不能依赖它的使用类。

5、要考虑今后可能的变化，也就是可扩展性。

6、要考虑合理的复用。

7、要合理地考虑接口和抽象类的使用。优先使用接口，抽象类既要约束子类的行为，又要为子类提供公共的功能。

8、尽量减少类与协作类的交互次数和交互信息的量。

9、父类不应知道子类的信息，子类必须知道父类的信息。

10、更多地使用类的组合，而不是继承。

11、访问对象必须通过接口，不能绕过接口直接去访问。

**分析类的一些经验法则：**

**1、每个类大约3~5个职责：**典型地，类应该保持尽可能简单，这通常限制类能够支持3~5个职责。

**2、不存在独立的类：**好的OO分析和设计的精华是，类相互协作让用户受益。同样，每个类应该同小部分类协作以提供所期望的功能。类可以把它们的一些职责托付给专注于特定功能的其他“辅助”类。

**3、当心很多非常小的类：**有时很难取得正确的平衡。如果模型看起来具有大量的非常小的类，每个类都具有一个或者两个职责，那么你应该仔细查看以把一些小的类整合成更大的类。

**4、当心少数几个非常庞大的类：**上述的反面是，模型具有很少的类，但每个类都是具有职责数量（>5）的庞大的类。解决问题的策略是依次查看这些类，看看是否每个类都能够被分解成为两个或者多个能够承担恰当数目职责的、更小的类。

**5、当心“伪类”：**类其实是一般的过程函数，它伪装成类。

**6、当心万能类：**在似乎能够承担任何工作的类。看看名称为“system”和“controller”的类！处理这个问题的策略是看看万能类的职责是否能够分解成内聚的子集。如果是，每个这些内聚职责集合可能独立成类。这些更小的类协作以实现由原始万能类所提供的行为。

**7、避免深度继承树：**设计良好的继承层次中，每个抽象层次都应该具有良好定义。

通常的错误是使用继承来实现一种功能分解，其中每个抽象层次恰有一个职责！无论从哪个方面来讲，这都是无意义的，会导致复杂的、难以理解的模型。

在分析中，类代表业务事物，而业务事物趋向于形成更宽（不超过三层）的继承层次。我们把三层或者更多层次的继承认为是“深度”继承。

## 子程序（函数）设计

### 节制使用inline子程序

inline子程序违反了封装原则，因为C++要求把inline子程序的实现代码写在头文件里，从而也就把这些实现细节暴露给了所有使用该头文件的程序员。

inline子程序要求在调用子程序的每个地方都生成该子程序的全部代码，这样无论inline子程序是长是短，都会增加整体代码的长度。这也会带来其自身的问题。

和为追求性能而使用的其他编程技巧一样，为性能原因而使用inline子程序的底线是：剖测（profile）代码并衡量性能上的改进。如果预期获得的性能收益不能说明为“剖测代码以验证性能改进”操心是值得的，那也就没有理由再牺牲代码质量而使用inline子程序了。

## 变量

理想情况下，在靠近第一次使用变量的位置声明和定义该变量。

### 变量命名

名字对于代码读者的意义要比对作者更重要。

### 全局数据

如果你随意使用全局数据，或者认为不能随心所欲地使用它们是一种约束，那么你可能还没有充分理解信息隐藏和模块化的意义。模块化、信息隐藏并结合使用设计良好的类可能还算不上是绝对真理，但是它们能极大地提升大型程序的可理解性和可维护性。一旦明白了这一点，你就会努力去写出与全局变量和外界联系尽可能少的子程序和类来。

鉴于全局变量在耦合度、管理复杂性等方面存在致命缺陷，在大型软件项目中，滥用全局变量可能酿成的软件架构混乱、代码可读性、可维护性和可扩展性降低等问题，把全局变量仅存的那点儿简单、灵活、便捷的优点遮蔽殆尽，这也就是为什么大多数程序员都会对全局变量既爱又恨的原因所在了。

如果你发现一个全局变量真的被到处使用，这便意味着你的代码没有进行任何层次化设计。

**不要把所有的数据都放在一个大对象中并到处传递，以说明你没有使用全局（TratableEvent正是如此，语法上不是全局变量，事实上是携带几百上千个属性的全局变量）。**把所有一切都放在一个大对象里可能会满足不使用全局变量的要求，但是这样做纯粹是一种负担，它也无法真正带来封装所能带来的那些好处。如果你要使用全局数据，那么就公开地用，不要试图通过使用大对象来掩盖这一点。

### 全局变量的问题

**1、变量名冲突：**这个问题会使项目管理成本大幅增加。项目经理必须小心地维护变量命名规则，所有工程师在开发代码时，每遇到一个全局变量，都必须仔细分辨该变量究竟属于哪个模块、哪个程序。很显然，这种管理成本的增加是传统的结构化程序设计所不可避免的，但也是我们程序员最不愿意看到的。

**2、耦合度难题：**全局变量实际上大大增加了函数和模块之间的耦合度，需要访问某个特定全局变量的多个函数被该变量牢牢地耦合在一起，成为拆不散的一团乱麻。

常见的情况是，如果有一位新人来阅读我们的代码，当他看到某个函数访问了一个全局变量时，他无法明确地知道该全局变量会被其他哪些函数访问，会被哪些函数读取或改写。他必须找出所有读写该全局变量的代码，理解这些函数的操作流程，才能充分了解当前函数的功能以及函数之间相互影响、相互制约关系。或者，当我们要扩展某个函数的功能时，哪怕是只改变一、两行代码，我们也必须小心地检索、查找所有相关的全局变量和相关的函数操作，以确定代码的变更不会对耦合在一起的函数产生负面影响。

**3、单个实例问题：**全局变量不能阻止程序员定义一个类的多个对象实例。如果没有其他技术手段的帮助，保证一个类只有单个实例就全靠程序员的自觉了。在一个多人参与、并行开发的大型项目中，这也会增加很多管理上的负担。

**4、初始化顺序：**全局变量不可能保证相互之间遵循特定的初始化顺序，这完全由编译器决定。对于简单变量来说，这没什么麻烦的，但是对于类的对象实例，构造函数被调用的顺序有时就显得非常重要。

**5、多线程访问：**当多个并发的线程里都需要访问某些全局变量时，我们必须使用各种同步机制，小心地保护这些全局变量，以免陷入并发冲突的泥潭。

**6、全局数据阻碍代码重用：**要把一个程序里的代码应用于另一个程序，你必须能够把它从第一个程序里取出，然后插入到另一个程序里面。在理想状况下，可以把一个单一子程序或者类取出来，把它插到另一个程序里面去。

全局数据使这件事变得复杂了。如果想重用的类读或写了全局数据，那么就无法简单地把它插入到新的程序里。你将不得不修改新的程序或者旧类，以便让它们相容。

如果选择上策，应该去修改旧类，让它不再使用全局数据，这样下次再需要重用这个类的时候，就可以把它插入到新程序中而无须花费额外的力气。

如果选择下策，那就去修改新的程序，以创建旧有的类所需要使用的全局数据。这样做就像传播病毒，该全局数据不但影响了原来的程序，还传染到使用旧程序中的类的新程序里面。

**7、全局数据破坏了模块化和智力上的可管理性：**创建超过几百行代码的程序的核心便是管理复杂度。你能够在智力上管理一个大型程序的唯一方法就是把它拆分成几部分，从而可以在同一时间只考虑一部分。模块化就是你手中可以使用的把程序拆分成几部分的最强大工具。

全局数据使得模块化能力大打折扣。如果用了全局数据，你能够在同一时间只关注一个子程序吗？不能。你不得不关注一个子程序，以及使用了同样全局数据的其他所有子程序。尽管全局数据并没有完全破坏程序的模块化，但是却削弱了它，而这已经是很充分的理由，要求你去寻找问题的更好解决方案了。

### 单件(Sigleton)模式

使用单件类最易犯的一个错误就是按照原来结构化程序设计的思路考虑问题，把全局变量作为模块之间共享数据的一个桥梁，即把多个全局变量收集起来，放进一个单件类中，这完全是一种换汤不换药、新瓶装旧酒的做法，它虽然可以减少一些全局变量，部分地解决命名冲突问题，但是对系统的架构却没有任何好处，全局变量本身的风险依然存在。

**现有的TRatableEvent也是如此，实质就是全局变量。**

# 模块设计

**模块设计中耦合、内聚的概念，同样适用于包、类、函数。**

**专门针对包、类、函数的耦合、内聚的描述，更具直接的操作性。**

### 什么是模块

模块是可以组合、分解和更换的单元，是组成系统、易于处理的基本单元。系统中的任何一个处理功能都可以看成一个模块，也可以理解为用一个名字就可以调用的一段程序语句。

模块应具备以下四个要素：

1、输入和输出：模块的输入来源和输出去向都是同一个调用者，一个模块从调用者取得输入，加工后再把输出返回调用者。

2、功能：模块把输入转换成输出所做的工作。

3、内部数据：仅供该模块本身引用的数据；

4、程序代码：用来实现模块功能的程序。

其中，输入/输出和功能是模块的外部特性，反映模块的外貌；内部数据和程序代码是模块的内部特性。在结构化设计中，首先关心外部特性，其内部特性只做必要了解。

可以认为，模块就是指具有输入/输出、逻辑功能、运行程序和内部数据4种属性的一组程序。

### 耦合/聚合、模块独立性

全局变量对模块化是毒药，很容易使各模块轻率、混乱地互相泄漏信息。

模块独立性指每个模块只完成系统要求的独立子功能，并且与其他模块的联系最少且接口简单，两个定性的度量标准――耦合性和内聚性。

耦合与聚合都是模块独立性的定性标准，都反映模块独立性的良好程度。**但耦合是直接的主导因素，聚合则辅助耦合共同对模块独立性进行衡量。**

聚合与耦合密切相关，同其它模块强耦合的模块意味着弱聚合，强聚合模块意味着与其它模块间松散耦合。

耦合是影响系统复杂程度和设计质量的重要因素。

### 模块划分的目的

模块划分的目的主要是降低系统的开发难度，增加系统的可维护性等。

过大的模块常常是系统分解不充分，而过小的模块有可能降低模块的独立性，造成系统接口的复杂。

模块化原则就是：要编写复杂软件又不至于一败涂地的唯一方法，就是用定义清晰的接口把若干简单模块组合起来。这样，多数问题只会出现在局部，可以对局部进行改进或优化，而不至于牵动全身。

模块化设计的目标是：模块间的联系尽可能的少（低耦合）；而模块内的联系尽可能多（高聚合）。建立模块间耦合尽可能松散的系统。其优点在于：

1、在设计、编码、测试和维护系统中的任何一个模块时，不需要对系统中的其他模块有很多了解。

2、当某个模块出错时，对系统的其他部分影响小。

**正交性**

是从几何学中借来的木语，如果两条直线相交成直角，它们就是正交的。用向量术语来说，这两条直线互不依赖。

在计算技术中，该术语用于表示某种不相依赖性或是解耦性。如果两个或更多事务中的一个发生变化，不会影响其它事务，这些事务就是正交的。

**如何正交：**

1、让代码保持解耦。编写的模块，不必要地向其他模块暴露任何事情，也不依赖其他模块的实现。

2、避免使用全局数据。使用单例模式要小心，它们可能造成不必要的关联。

**正交的好处：**

正交性缩短了测试和开发的时间，因为那种既不产生副作用也不依赖其它代码副作用的代码校验起来要容易得多——需要测试的情况组合要少得多。如果正交性代码出现问题，把它替换掉而不影响系统其余部分也很容易做到。最后，正交性代码更容易文档化和复用。

正交系统更容易测试，因为可以在模块级进行测试，而不是集成测试，模块内部也可以进行单元测试。

我们想要设计自足的组件：独立。如果组件是相互隔离的，你就知道能够改变其中之一，而不用担心其余组件。只要不改变组件的外部接口，那么就不会造成波及整个系统的问题。

如果编写正交的系统，将得到两个好处：提高生产率与降低风险。

提高生产率：1、改动得以局部化，所以开发时间和测试时间得以降低；2、促进复用。

降低风险：1、有问题的代码区域被隔离开来，如果某个模块有毛病，它不大可能把病症扩散到系统的其余部分。2、系统更健壮，对特定区域做出小的改动和修正，导致的任何问题都降局限在该区域中。3、正交系统能得到更好的测试。因为设计测试，并针对组件运行测试更容易。

正交，可以降低系统的各个组件的相互依赖。系统会变得更灵活、易于理解、易于调试、测试和维护。否则每一处改动似乎都会造成别的东西出错。

**非正交的坏处：**

症状：

1、对某个模块的“简单”改动会传遍系统中的一些无关模块。

2、开发者害怕改动代码，因为他们不清楚哪些代码可能会受影响。

有许多不必要的依赖关系的系统非常难以维护，往往高度不稳定。

具有较大响应集的类更容易出错。

### 模块划分的方法

将初始系统结构图根据模块独立性原则进行精化，对模块进行合并、分解修改、调整，得到高内聚、低耦合模块，得到易于实现、易于测试和易于维护的软件结构，产生设计文档的最终系统结构图。

**模块划分的粒度**

模块分解得越彻底，每一块就越小，API的定义也就越重要。全局复杂度和受bug影响的程度也会相应降低。软件系统应设计成由层次分明的嵌套模块组成，而且每个层面上的模块粒度应降至最低。

然而，也可能因过度划分造成模块太小，模块过大或者过小都和更多的bug相关联。在模块很小时，bug发生率也出乎意料地增多。Hatton曾经提出过一个模型，将这种非线性同人类短期记忆的记忆块大小相比较。这种非线性的另一种解释是，模块小时，几乎所有复杂度都在于接口；想要理解任何一部分代码前必须理解全部代码，因此阅读代码非常困难。

模块大小需要适中。模块过大时，则可理解程度下降，需要考虑对该模块分解。模块过小时，则开销大于有效操作，系统接口复杂，需要考虑是否同它的调用模块合并。

几种情况下的小模块不合并：模块具有独立的功能；模块功能可能会发生变化；模块的复用性好，或者其调用模块很复杂。

降低模块接口的复杂性，接口传递信息应简单且和模块功能一致。

### 扇入和扇出

在软件设计中，扇入和扇出的概念是指应用程序模块之间的层次调用情况。按照结构化设计方法，一个应用程序是由多个功能相对独立的模块所组成。

扇入：是指直接调用该模块的上级模块的个数。扇入大表示模块的复用程序高。

扇出：是指该模块直接调用的下级模块的个数。扇出大表示模块的复杂度高，需要控制和协调过多的下级模块，这时应适当增加中间层次的控制模块；扇出太小时应把下级模块进一步分解成若干个子功能模块或合并到上级模块。

如果一个规模很小的底层模块的扇入数为1，则可把它合并到它的上层模块中，但扇入数较大时则不能向上合并（否则将导致对该模块做多次编码和排错）。如果一个模块具有多种功能（扇出数较大），应当考虑做进一步分解，使模块的扇出数控制在7以内。

扇出过大一般是因为缺乏中间层次，应该适当增加中间层次的模块。扇出太小时可以把下级模块进一步分解成若干个子功能模块，或者合并到它的上级模块中去。

设计的较好的系统结构，通常是顶层扇出较高，中层扇出较小，底层扇入较大。

扇出不宜过大，<=7；扇入尽可能大些。但case型模块扇出可>7，因为属自然分类情况，易理解。

良好的结构图往往呈**清真状**，即“顶为尖—下层逐渐加宽—底层收窄”。**金字塔**型是要避免的，底层模块复用度低。

结构图的形态应该均衡，即从顶到底层的路径长短不该过于悬殊。

## 内聚

### 定义

内聚，是一个模块内部各成分之间相关联程度的度量。更为专业的说法叫功能内聚，是对软件系统中元素职责相关性和集中度的度量。如果元素具有高度相关的职责，除了这些职责内的任务，没有其它过多的工作，那么该元素就具有高内聚性，反之则为低内聚性。

内聚性源于结构化设计，并且经常与耦合度结合在一起讨论。内聚性指的是类内部的子程序或者子程序内的所有代码在中心目标上的紧密程度——这个类的目标是否集中。包含一组密切相关的功能的类被称为有着高内聚性。

模块的内聚性和耦合性是两个相互对立且又密切相关的概念。模块内的内聚越高，模块间的耦合度就越低。只要做到各个模块的高内聚，模块间的耦合程度自然降低，模块的独立性也越高。

经实践证明，保证模块的高内聚性比低耦合性更为重要，在软件设计时应将更多的注意力集中在提高模块的内聚性上。

### 高内聚的好处

1、可读性

一个人写文章、讲事情，条理清晰才能易于理解，这同样发生在读写软件代码上。如果一堆代码写得一团乱麻，读它的人会感觉非常头疼。如果一段程序条理清晰，每个类通过名称或说明都能清楚明白它的意义，类的每个属性、函数也都是易于理解的它所应当完成的任务和行为，这段程序的可读性必然提高。

2、复用性

在软件开发中，最低等级的复用是代码拷贝，然后是函数的复用、对象的复用、组件的复用。代码复用使我们的代码在复用的过程中不断精化、不断健壮、提高代码质量。

软件中的一个对象如果能保证能完成自己职能范围内的各项任务，同时又不去理会与自己职能无关的其它任务，那么它就能够保证功能的相对独立性，也就可以脱离自己所处的环境而复用到其它环境中，这是一个具有内聚性的对象。

3、可维护性和易变更性

软件是在不断变更的，这种变更不仅来自于我们的客户，更来自于我们的市场。如果我们的软件通过变更能及时适应我们的市场需求，我们就可以在市场竞争中获胜。如何能及时变更以适应我们的市场，就是通过调整软件的结构，使我们每次的变更付出的代价最小，耗费的人力最小，这种变更才最快最经济。高内聚的软件，每个系统、模块、类的任务都高度相关，就使每一次的变更涉及的范围缩小到最小。如果我们能做到这一点，我们的系统当然是可维护性好、易变更性好的系统。

### 如何做到高内聚

以评审项目举例。我现在要为“评审表”对象编写一段填写并保存评审表的代码。评审表对象的职责是更新和查询评审表的数据，但是在显示一个要填写的评审表的时候，我需要显示该评审计划的名称、该评审计划有哪些评审对象需要评审。现在我如何编写显示一个要填写的评审表的代码？我在评审表对象的这个相应的函数中编写一段查询评审计划和评审对象的代码吗？假如你这样做了，你的代码就不是高内聚的，因为查询评审计划和评审对象的数据不是它的职责。正确的方法应当去请求“评审计划”对象和“评审对象”对象来完成这些工作，而“评审表”对象只是获取其结果。

与“低耦合”一样，高内聚也不是一个绝对，而是一个相对的指标，应当适当而不能过度。正如我们在现实生活中，如果在一个十来人的小公司，每个人的分工可能会粗一些，所分配的职责会广一些杂一些，因为其总体的任务少；而如果在一个一两百人的大公司，每个人的分工会细一些，所分配的任务会更加专一些，因为总体任务多，更需要专业化的分工来提高效率。

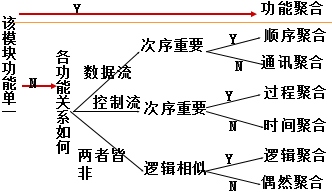
软件开发也是一样，如果“评审计划”对象完成的业务功能少，并且不复杂，它完全可以代理它的子表“评审对象”和“评审者”的管理。但是“评审计划”对象需要完成的“对评审计划表的管理”这个基本职责包含的业务功能繁多或者复杂，它就应当将“对评审对象表的管理”交给“评审对象”对象，将“对评审者表的管理”交给“评审者”对象。

同样，高内聚的可维护性好、易变更性好只能是一个相对的指标。如果一个变更的确是大范围的变更，你永远不可能通过内聚就不进行大范围的变更了。

同时内聚也是要付出代价的，所以不必要为了一个不太可能的变更去进行过度设计，应当掌握一个度。过度的内聚必将增加系统中元素之间的依赖，提高耦合度。

所以“高内聚”与“低耦合”是矛盾的，必须权衡利弊，综合地去处理。

### 模块聚合性的判断



### 内聚的类型

内聚按强度从高到低有以下几种类型：

#### 功能内聚

模块内所有元素共同完成一个单一的功能，联系紧密，缺一不可。这是最强的内聚。

如：计算实发工资，打印发票等。

#### 顺序内聚

一个模块的各个成分和同一个功能密切相关且必须顺序执行，而且前一个成分的输出数据作为下一个成分的输入数据，则称为顺序内聚。

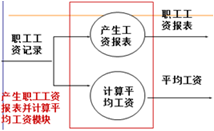
如模块1：读入 -> 存储。

又模块2：读数 -> 累加 -> 打印。

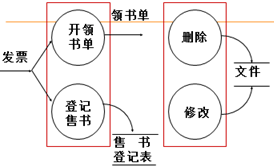
#### 通信内聚

一个模块中的各个部分都在同一个数据结构上操作（有时称之为信息内聚），或者模块中的各个部分使用相同的输入数据或产生相同的输出结果，则称为通信聚合。

**通信聚合模块例：**



**通信聚合模块例：**



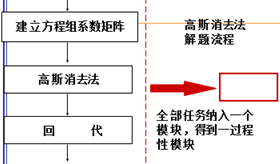
#### 过程内聚

一个模块内部的处理成分是相关的，而且这些处理必须以特定的次序执行，则称为过程内聚。

构件或者操作的组合方式是，允许在调用前面的构件或操作之后，马上调用后面的构件或操作，即使两者之间没有数据进行传递。

一个模块内各处理成分的动作各不相同，彼此也没什么关系，但它们都受一个控制流的支配，且决定了它们的执行次序。

**过程聚合模块例：**



#### 时间内聚

如果一个模块完成的功能必须在同一时间内执行，但这些功能只是因为时间因素关联在一起，则称为时间内聚。

例如，将多个变量的初始化放在同一个模块中实现，或将需要同时使用的多个库文件的打开操作放在同一个模块中，都会产生时间内聚的模块。

初始化系统模块、 系统结束模块、紧急故障处理模块等均是时间性聚合模块。

#### 逻辑内聚

几个逻辑上相关的功能被放在同一模块中，称为逻辑内聚。这种模块把几种相关的功能组合在一起，每次被调用时，由传送给模块参数来确定该模块应完成哪一种功能。如一个模块读取各种不同类型外设的输入。

尽管逻辑内聚比偶然内聚合理一些，但逻辑内聚的模块各成分在功能上并无关系，即使局部功能的修改有时也会影响全局，因此这类模块的修改也比较困难。

逻辑聚合缺点：1、不易修改；2、增强了耦合程度（控制耦合）；3、效率低。

#### 偶然内聚

如果一个模块的各成分之间毫无关系，则称为偶然内聚。

缺点：可理解性差， 可修改性差。

### 内聚的类型（子程序）

内聚性（cohesion）的概念是由Wayne Stevens、Glenford Myers和Larry Constantine在1974年发表的一篇论文中提出来的。其他一些更为现代的概念，如抽象和封装等，通常在类这一层次的设计中更为适用（**抽象和封装在类层次上已经很大程度上取代了内聚性**），但内聚性的概念仍然存在，而且在单个子程序这一层次上，仍是设计时常用的启发式方法。

对子程序而言，内聚性是指子程序中各种操作之间联系的紧密程度。

#### 功能内聚

最强也是最好的一种内聚，也就是让一个子程序仅执行一项操作。

例如sin()，GetCustomerName()、EraseFile()、CalculateLoanPayment()以及AgeFromBirthdate()这样的子程序都是高度内聚的。当然，以这种方式来评估内聚性，前提是子程序所执行的操作与其名字相符，如果它还做了其他的操作，那么它就不够内聚，同时其命名也有问题。

#### 顺序内聚(sequential cohesion)

是指在子程序内包含有需要按特定顺序执行的操作，这些步骤需要共享数据，而且只有在全部执行完毕后才完成了一项完整的功能。

举一个顺序上的内聚的例子，假设某个子程序需要按照给定出生日期来计算出员工的年龄和退休时间。如果子程序先计算员工的年龄，再根据他的年龄来计算退休时间，那么它就具有顺序的内聚性。而如果子程序先计算员工的年龄，然后再重新计算他的退休时间，两次计算之间只是碰巧使用了相同的出生日期，那么这个子程序就只具有通信上的内聚性communicational cohesion。

那么该怎样设计具有功能上的内聚性的子程序呢？你可以创建两个不同的子程序，它们能根据给定的生日分别计算员工的年龄和退休时间。其中，计算退休时间的子程序可以调用计算年龄的子程序。这样两者就都具有功能上的内聚性了，而其他的子程序则可以调用二者之一或全部。

#### 通信内聚(communicationc cohesion)

是指一个子程序中的不同操作使用了同样的数据，但不存在其他任何联系。例如某个子程序先根据传给它的汇总数据打印一份汇总报表，然后再把这些汇总数据重新初始化，那么这个子程序就具有通信上的内聚性：因为这两项操作只是因为使用了相同的数据才彼此产生联系。

要改善这个子程序的内聚性，应该让重新初始化汇总数据的操作尽可能靠近创建汇总数据的地方，而不是放在打印报表的子程序里。应该把这些子程序进一步拆分成几个独立的子程序：一个负责打印报表，一个负责在靠近创建或修改数据的代码的地方重新初始化数据。然后在原本调用那个具有通信内聚性的子程序的更高层的子程序中调用这两个子程序。

#### 过程内聚(procedural cohesion)

是指一个子程序中的操作是按特定的顺序进行的。

一个例子是依次获取员工的姓名、住址和电话号码的子程序。这些操作执行的顺序之所以重要，只是因为它和用户按屏幕提示而输入数据的顺序相一致。另一个子程序用来获取员工的其他数据，这段程序也具有过程上的内聚性，因为它把一组操作赋以特定的顺序，而这些操作并不需要为了除此之外的任何原因而彼此关联。

为了得到更好的内聚性，可以把不同的操作纳入各自的子程序中，让调用方的子程序具有单一而完整的功能：GetEmployee()就比GetFirstPartOfEmployeeData()更为可取。你可能还需要修改用来读取其余数据的子程序。为了让所有的子程序都具有功能上的内聚性，对两个或更多的原有子程序进行修改是很常见的。

#### 临时内聚（temporal cohesion)

是指含有一些因为需要同时执行才放到一起的操作的子程序。典型的例子有：Startup()、 CompleteNewEmployee()、Shutdown()等。有些程序员认为临时的内聚性是不可取的，因为它们有时与不良的编程实践相关——比如说在Startup()子程序里塞进一大堆互不相关的代码等。

为避免这个问题，可以把临时性的子程序看做是一系列事件的组织者。前面提到的Startup()子程序可能需要读取配置文件、初始化临时文件、设置内存管理器，再显示启动画面。要想使它最有效，应该让原来那个具有临时内聚性的子程序去调用其他的子程序，由这些子程序来完成特定的操作，而不是由它直接执行所有的操作。

这个例子提出这样一个问题，即如何选择一个能够在恰当的抽象层次上描述子程序的名字。你可能决定把一个子程序命名为ReadConfigFilelnitScratchFileEtc()，它可以暗示该子程序只有巧合的内聚性（coincidental cohesion)。而如果你把它命名为startup()，那么很明显，这个子程序就只具有一个功能，且具有功能上的内聚性。

一般来说，其他类型的内聚性都是不可取的。它们都会导致代码组织混乱、难于调试、不便修改。如果一个子程序具有不良的内聚性，那最好还是花功夫重新编写，使其具有更好的内聚性，而不是再去花精力精确地诊断问题所在了。下面就给出一些不可取的内聚性。

#### 逻辑内聚(logical cohesion)

是指若干操作被放入同一个子程序中，通过传入的控制标志选择执行其中的一项操作。之所以称之为逻辑上的内聚性，是因为子程序的控制流或所谓“逻辑”是将这些操作放到一起的唯一原因——它们都被包在一个很大的if语句或case语句中，而不是因为各项操作之间有任何逻辑关联。认为是逻辑上的内聚性的标志性属性就是各项操作之间没有关联，因此，似乎更应称其为“缺乏逻辑的内聚性”。

这方面的一个例子是名为InputAllO的子程序，它根据传入的控制标志来决定是输入客户姓名、员工考勤卡信息，还是库存数据。类似的例子还有ComputeAll()、EditAll()、PrintAll()和SaveAll()。这种子程序的主要问题是你不该通过传入控制标志来控制另一个子程序的处理方式。相比之下，让三个子程序分别完成不同的操作，要比用一个“根据传入的控制标志选择执行三项不同的操作之一”的子程序清晰得多。如果这些操作中含有一些相同代码或共用了数据，那么应该把那些代码移入一个低层子程序中，这些子程序也应该包裹在一个类中。

如果子程序里的代码仅由一系列的if语句或者case语句，以及调用其他子程序的语句组成，那么创建这样一个具有逻辑上的内聚性的子程序通常也是可以的。在这种情况下，如果子程序唯一的功能就是发布各种命令，其自身并不做任何处理，这通常也是一个不错的设计。这类子程序的技术术语便是“事件处理器（event handler）”。事件处理器通常用在各种交互性的环境中，例如像 Apple Macintosh、Microsoft Windows 及其他一些 GUI（图形式用户界面）环境。

#### 巧合内聚(coincidental cohesion)

是指子程序中的各个操作之间没有任何可以看到的关联，也可称为“无内聚性”或“混乱的内聚性”。很难从巧合的内聚性转变为任何一类更好的内聚性，通常你需要深入地重新设计和重新实现。

## 耦合

### 定义

耦合（Coupling）是模块之间依赖程度的度量，是模块间相互联系紧密程度的一种度量。

对象之间的耦合度就是对象之间的依赖性，使用和维护对象的主要问题是对象之间的多重依赖性。对象之间的耦合性越高，维护成本越高，因此对象的设计应使类和构件之间的耦合最小。

耦合就是对某元素与其它元素之间的连接、感知和依赖的量度。这里所说的元素，即可以是功能、对象（类），也可以指系统、子系统、模块。假如一个元素A去连接元素B，或者通过自己的方法可以感知B，或者当B不存在的时候就不能正常工作，那么就说元素A与元素B耦合。

我们在看spring的书籍、MVC的数据、设计模式的书籍，无处不提到“低耦合、高内聚”，它已经成为软件设计质量的标准之一。

低耦合要求在我们的软件系统中，某元素不要过度依赖于其它元素。系统中低耦合不能过度，比如设计了一个类，它不与系统中的任何类发生耦合。如果有这样一个类，那么它必然是低内聚。耦合与内聚常常是一个矛盾的两个方面，最佳的方案就是寻找一个合适的中间点。

“低耦合”给软件项目带来的优点是：易于变更、易于重用。

### 高耦合的坏处

模块之间联系越紧密，其耦合性就越强，模块的独立性则越差。

假如元素A与元素B耦合，耦合带来的问题是，当元素B发生变更或不存在时，都将影响元素A的正常工作，影响系统的可维护性和易变更性。

同时元素A只能工作于元素B存在的环境中，这也降低了元素A的可复用性。

### 良性依赖原则

不会在实际中造成危害的依赖关系，都是良性依赖。

本原则的核心思想是“务实”，很好地揭示了极限编程中“简单设计”和“重构”的理论基础。可以帮助我们抵御“面向对象设计五大原则”以及设计模式的诱惑，以免陷入过度设计（Over-engineering）的尴尬境地，带来不必要的复杂性。

### 低耦合举例

目前已经有大量的框架帮助我们降低我们系统的耦合度。

比如，使用struts我们可以应用MVC模型，使页面展现与业务逻辑分离，做到了页面展现与业务逻辑的低耦合。当我们的页面展现需要变更时，我们只需要修改我们的页面，而不影响我们的业务逻辑；同样，我们的业务逻辑需要变更的时候，我们只需要修改我们的java程序，与我们的页面无关。

使用spring我们运用IoC（反向控制），降低了业务逻辑中各个类的相互依赖。假如类A因为需要功能F而调用类B，通常的情况下类A需要引用类B，因而类A依赖于类B，也就是当类B不存在的时候类A就无法使用。使用了IoC，类A调用的仅仅是实现了功能F的接口的某个类，这个类可能是类B，也可能是另一个类C，由spring的配置文件来决定。这样，类A就不再依赖于类B，耦合度降低，重用性提高了。

使用hibernate则是使我们的业务逻辑与数据持久化分离，也就是与将数据存储到数据库的操作分离。我们在业务逻辑中只需要将数据放到值对象中，然后交给hibernate，或者从hibernate那里得到值对象。至于用Oracle、MySQL还是SQL Server，如何执行的操作，与我无关。

### 如何做到低耦合

根据我的经验，以下一些问题我们应当引起注意：

1、根据可能的变化设计软件

设计中尽力做到“低耦合、高内聚”的一个非常重要的前提是，我们的软件是在不断变化的，如果没有变化我们就不用这么费劲。对于软件可能变更的部分，我们应当努力去降低耦合，使我们在适应或者更改这样的变化的时候，付出更小的代价。

注意，我们努力降低耦合的是那些可能发生变更的地方，因为降低耦合是有代价的，是以增加资源耗费和代码复杂度为代价的。如果系统中某些元素不太可能变更，或者降低耦合所付出的代价太大，我们当然就应当选择耦合。

2、合理的职责划分

合理的职责划分，让系统中的对象各司其职，不仅是提高内聚的要求，同时也可以有效地降低耦合。比如评审计划BUS、评审表BUS、评审报告BUS都需要通过评审计划DAO去查询一些评审计划的数据，如果它们都去直接调用评审计划DAO，则评审计划BUS、评审表BUS、评审报告BUS三个对象都与评审计划DAO耦合，评审计划DAO一旦变更将与这三个对象都有关。在这个实例中，实际上评审计划BUS是信息专家（关于信息专家模式我将在后面讨论），评审表BUS和评审报告BUS如果需要获得评审计划的数据，应当向评审计划BUS提出需求，由评审计划BUS提供数据。经过这样的调整，系统的耦合度就降低了。

3、使用接口而不是继承

通过对耦合的分析，我们不难发现，继承就是一种耦合。如果子类A继承了父类B，不论是直接或间接的继承，子类A都必将依赖父类B。子类A必须使用在存在父类B的环境中，父类B不存在子类A就不能使用，这样将影响子类A的可移植性。一旦父类B发生任何变更，更改或去掉一个函数名，或者改变一个函数的参数，都将导致子类A不得不变更，甚至重写。假如父类B的子类数十上百个，甚至贯穿这个项目各个模块，这样的变更是灾难性的。

**如何降低模块间耦合度：**

1、如模块必须存在耦合，选择适当的耦合类型。尽量使用数据耦合，少用控制耦合，限制公共耦合的范围，坚决避免使用内容耦合。

2、降低模块间接口的复杂性。

### 哪些是耦合

1、元素B是元素A的属性，或者元素A引用了元素B的实例（这包括元素A调用的某个方法，其参数中包含元素B）。

2、元素A调用了元素B的方法。

3、元素A直接或间接成为元素B的子类。

4、元素A是接口B的实现。

### 衡量耦合度的标准

由于模块间的联系是通过模块接口实现的，耦合强度，依赖于以下几个因素：

1、一个模块对另一个模块的调用方式；

2、一个模块向另一个模块传递的数据（传递的信息类型）；

3、模块之间接口的复杂程度。

**一些衡量模块之间耦合度的标准：**

**1、规模：**这里的规模指的是模块之间的连接数。对于耦合度来说，小就是美，因为只要做很少的事情，就可以把其他模块与一个有着很小的接口的模块连接起来。只有一个参数的子程序与调用它的子程序之间的耦合关系比有六个参数的子程序与它的调用方之间的耦合关系更为松散。包含4个定义明确的public方法的类与它的调用方的耦合关系，比包含37个public方法的类与它的调用方的耦合关系更为松散。

**2、可见性：**可见性指的是两个模块之间的连接的显著程度。开发程序与在中央情报局里工作不一样，你不能靠鬼鬼祟祟来获得信任，而是应该像登广告一样，通过把模块之间的连接关系变得广为人知而获取信任。通过参数表传递数据便是一种明显的连接，因而值得提倡。通过修改全局数据而使另一模块能够使用该数据则是一种“鬼鬼祟祟”的做法，因此是很不好的设计。如果把与全局数据之间的连接写入文档，那么就会使得这些连接相对明显些，因而会比上面的做法稍微好些。

**3、灵活性：**灵活性指的是模块之间的连接是否容易改动。理想状态下，你会更喜欢计算机上的热插拔USB连接器，而不喜欢用电烙铁焊接裸导线的连接方式。在一定程度上，灵活性是其他几个耦合特性综合作用的结果，但是也的确有所不同。

简而言之，一个模块越容易被其他模块所调用，那么它们之间的耦合关系就会越松散。这种设计非常不错，因为它更灵活，并且更易于维护。因此，在创建系统架构时，请按照“尽可能缩减相互连接”的准则来分解程序。如果把程序看做是一块木材，那么就请延着木材的纹理把它劈开。

### 耦合的类型

耦合按从弱到强的顺序可分为以下几种类型：

#### 非直接耦合

模块间没有信息传递，两个模块之间没有直接关系，它们之间的联系完全是通过主模块的控制和调用来实现的。

#### 数据耦合

模块间通过参数传递基本类型的数据，一模块调用另一模块时，被调用模块的输入、输出都是简单数据参数（不是控制参数、公共数据结构或外部变量），属松散耦合。

数据耦合是模块间必要的数据通信，数据耦合是模块设计的目标。数据耦合是松散的耦合，模块之间的独立性比较强。在软件程序结构中至少必须有这类耦合。

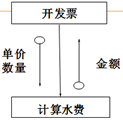
数据耦合的例子如下所示：

sum(int a,int b) { int c; c=a+b; return(c); }

main() { int x,y; printf("x+y= %d",sum(x,y)); }

/\*主函数与sum函数之间即为数据耦合关系\*/

**数据耦合举例如下：**

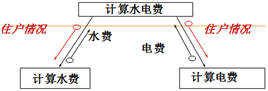


#### 标记耦合

模块间通过参数传递复杂的内部数据结构（不是简单数据，而是记录、数组等），而被调用模块只需要部分数据项，则称这两个模块之间存在标记耦合或特征耦合。

特征耦合传递的是数组之类的数据结构，使互不相关的模块建立了依赖关系，往往会造成侦错上的困难。在设计中应尽量避免这种耦合，它使在数据结构上的操作复杂化了。

**标记耦合举例**

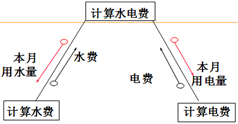


“住户情况”包含水费和电费、煤气费、电话费等。

“住户情况”是一个数据结构，图中模块都与此数据结构有关。

“计算水费”和“计算电费”本无关，由于引用了此数据结构产生依赖关系，它们之间是标记偶合。

**将标记耦合修改为数据耦合举例**



#### 控制耦合

如果一个模块通过传送开关、标志等控制信息（尽管有时控制信息是以数据的形式出现的），明显地控制选择另一模块的某项功能，就是控制耦合。

从分解的角度看，导致控制耦合的主要原因是分解不彻底，被调用模块不是执行单一的功能。

控制耦合是中等程度的耦合。由于在单一接口上选择多功能模块中的某项功能，因此对所控制模块的任何修改，都会影响控制模块，增加了模块之间的复杂性；另外，调用模块必须知道被调模块的内部逻辑，增加了相互依赖，降低了模块的独立性。

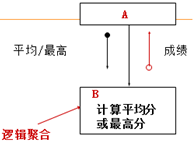
控制耦合的例子如下所示：

void output(flag) { if (flag) printf("OK! "); else printf("NO! "); }

main() { int flag; output(flag); }

/\*主函数与output函数之间即为控制耦合关系\*/

**控制耦合举例：**



调用逻辑性模块B时，须先传递控制信号（平均分/最高分），以选择所需的操作。

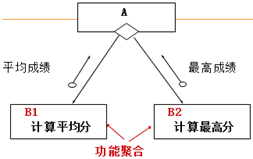
**改进的方法：**

去除模块间控制耦合的方法：

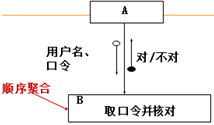
1、将被调用模块内的判定上移到调用模块中进行

2、被调用模块分解成若干单一功能模块

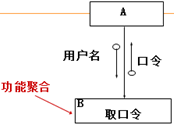
**改控制耦合为数据耦合：**



**控制耦合举例：**



**将B中核对口令功能划分到 A：**



#### 外部耦合

一组模块都访问同一全局简单变量而不是同一全局数据结构，而且不是通过参数表传递该全局变量的信息，则称之为外部耦合。

例如C语言程序中各个模块都访问被说明为extern类型的外部变量。

外部耦合引起的问题类似于公共耦合，区别在于在外部耦合中不存在依赖于一个数据结构内部各项的物理安排。

#### 公共耦合

若一组模块都访问同一个公共数据区，则它们之间的耦合就称为公共耦合。公共数据区可以是全局数据结构、共享的通信区、内存的公共覆盖区等。**例如，PCRF和OCS系统中的TRatableEvent类。**

这种耦合会引起下列问题：

1、所有公共耦合模块都与某一个公共数据环境内部各项的物理安排有关，若修改某个数据的大小，将会影响到所有的模块。

2、无法控制各个模块对公共数据的存取，严重影响软件模块的可靠性和适应性。

3、公共数据名的使用，明显降低了程序的可读性。

4、公共耦合的复杂程度随耦合模块的个数增加而显著增加。若只是两个模块之间有公共数据环境，则公共耦合有两种情况。若一个模块只是往公共数据环境里传送数据，而另一个模块只是从公共数据环境中取数据，则这种公共耦合叫做松散公共耦合；若两个模块都从公共数据环境中取数据，又都向公共数据环境里送数据，则这种公共耦合叫做紧密公共耦合。

只有在模块之间共享的数据很多，且通过参数表传递不方便时，才使用公共耦合。否则，还是使用模块独立性比较高的数据耦合好些。

**公共耦合存在的问题：**

1、系统可理解性降低。模块间存在错综复杂的连系。

2、系统可维护性差。如果要修改一个模块，很难确定哪些数据必须进行相应的修改；同样，如果要修改某一个数据，也很难确定哪些模块必须予以修改。

3、系统可靠性差。公共数据区及全程变量无保护措施。

4、在使用同一全局数据域的模块中，如果其中一个模块出现错误，则很可能该错误也出现在使用该数据域的其它模块中。

#### 内容耦合

如果两个模块之间出现下述情形之一，那么就发生了内容耦合，此时，被修改的模块完全依赖于修改它的模块：

1、一个模块对另一模块中的内容（包括数据和程序段）进行了直接的引用甚至修改；

2、通过非正常入口进入到另一模块内部；

3、一个模块具有多个入口；

4、两个模块共享一部分代码（只可能出现在汇编语言中）。

内容耦合是所有耦合关系中程度最高的，是模块独立性最弱的耦合，会使因模块间的联系过于紧密而对后期的开发和维护工作带来很大的麻烦。这是一种最高程度的耦合，应避免使用。

在内容耦合的情形，所访问模块的任何变更，或者用不同的编译器对它再编译，都会造成程序出错。大多数高级程序设计语言已经设计成不允许出现内容耦合。它一般出现在汇编语言程序中。

### 类耦合的种类

#### 简单数据参数耦合（simple'data-parameter coupling）

当两个模块之间通过参数来传递数据，并且所有的数据都是简单数据类型(primitive data type)，这两个模块之间的耦合关系就是简单数据参数耦合的。这种耦合关系是正常的，可以接受的。

#### 简单对象耦合（simple-object coupling）

如果一个模块实例化一个对象，那么它们之间的耦合关系就是简单对象耦合的。这种耦合关系也很不错。

#### 对象参数耦合（object-parameter coupling）

如果Object1要求Object2传给它一个Object3，那么这两个模块就是对象参数耦合的。与Object1仅要求Object2传递给它简单数据类型相比，这种耦合关系要更紧密一些，因为它要求0bject2了解Object3。

#### 语义上的耦合

最难缠的耦合关系是这样发生的：一个模块不仅使用了另一模块的语法元素（syntactic element），而且还使用了有关那个模块内部工作细节的语义知识（semantic knowledge）。这里是一些例子：

1、Module1向Module2传递了一个控制标志，用它告诉Module2该做什么。这种方法要求Modulel对Module2的内部工作细节有所了解，也就是说需要了解Module2对控制标志的使用。如果Module2把这个控制标志定义成一种特定的数据类型（枚举类型或者对象），那么这种使用方法还说得过去。

2、Module2在Modulel修改了某个全局数据之后使用该全局数据。这种方式就要求Module2假设Modulel对该数据所做出的修改符合Module2的需要，并且Modulel已经在恰当的时间被调用过。

3、Module1的接口要求它的Modulel.Initialize()子程序必须在它的 Modulel.Routine()之前得到调用。Module2 知道 Modulel.Routine()无论如何都会调用Modulel.Initialize()，所以它在实例化Modulel之后只是调用了 Modulel.Routine()，而没有先去调用 Modulel.Initialize()。

4、Modulel 把 Object传给Module2。由于Modulel知道Module2只用了Object 的7个方法（method）中的3个，因此它只部分地初始化Object——只包含那3个方法所需的数据。

5、Modulel 把BaseObject传给Module2。由于Module2知道Modulel实际上传给它的是DerivedObject，所以它把BaseObject转换成DerivedObject，并且调用了DerivedObject特有的方法（method）。

语义上的耦合是非常危险的，因为更改被调用的模块中的代码可能会破坏调用它的模块，破坏的方式是编译器完全无法检查的。类似这样的代码崩溃时，其方式是非常微妙的，看起来与被使用的模块中的代码更改毫无关系，因此会使得调试工作变得无比困难。

松散耦合的关键之处在于，一个有效的模块提供出了一层附加的抽象——一旦你写好了它，你就可以想当然地去用它。这样就降低了整体系统的复杂度，使得你可以在同一时间只关注一件事。如果对一个模块的使用要求你同时关注好几件事——其内部工作的细节、对全局数据的修改、不确定的功能点等——那么就失去了抽象的能力，模块所具有的管理复杂度的能力也削弱或完全丧失了。

类和子程序是用于降低复杂度的首选和最重要的智力工具。如果它们没帮助你简化工作，那么它们就是失职的。

### 类的耦合

#### 耦合/依赖的类型

依赖有几种，依赖也就是耦合，类之间的耦合分为下面三种：

1、零耦合关系。两个类没有依赖关系。

2、具体耦合（Concrete Coupling）关系。一个具体类直接引用另外一个具体类。

3、抽象耦合（Abstract Coupling）关系。这种关系发生在一个具体类和一个抽象类之间，这样就使必须发生关系的类之间保持最大的灵活性。

**如何把握耦合：**应该尽可能的避免实现继承，原因见CARP原则。

#### 保持松散耦合

耦合度表示类与类之间或者子程序与子程序之间关系的紧密程度。耦合度设计的目标是创建出小的、直接的、清晰的类或子程序，使它们与其他类或子程序之间关系尽可能地灵活，这就被称作松散耦合。松散耦合这一概念同时适用于类和子程序，所以在下面的讨论中会使用“模块”一词同时指代类和子程序。

模块之间的好的耦合关系会松散到恰好能使一个模块能够很容易地被其他模块使用。火车模型之间通过环钩彼此相连，把两辆列车连起来非常容易，只用把它们钩起来就可以了。如果必须要把它们用螺丝拧在一起，或者要连很多的线缆，或者只能连接某些特定种类的车辆，那么连接工作会是多么复杂。火车模型之间之所以能够相连，就是因为这种连接尽可能的简单。在软件中，也请确保模块之间的连接关系尽可能的简单。

请尽量使你创建的模块不依赖或者很少依赖其他模块，让模块之间的关系像商业合作者一样彼此分离，而不是像连体婴儿那样紧密相连。像sin()这样的子程序是松散耦合的，因为它需要知道的东西也就是一个传入的、代表角度的数值。而诸如InitVars(varl, var2, var3 , varN)这样的子程序则耦合得过于紧密了，因为对于调用端必须传入的各个参数，调用它的模块实际上知道在InitVars()的内部会做些什么。如果两个类都依赖于对方对同一个全局变量的使用情况，那么它们之间的稱合关系就更为紧密了。

#### 低耦合

这里的耦合主要指的是类和类之间的关联程度，即一个类知道其他类、依赖于其他类的强弱程度。类A 和类B 之间的相合包括以下几种情况：

1、类A 的一个方法的参数引用了类B，或方法内定义了类B 的一个局部变量；

2、类A 的一个属性引用了类B 的实例对象；

3、类A 的一个属性聚合了类B 的多个实例对象；

4、类A 是类B 间接或直接的派生类。

因此，类间关系就代表了类之间的耦合程度。以上几种耦合的情况，从上向下耦合程度越来越强，继承是最强的一种耦合，这是因为派生类没有任何声明，就悄悄地把基类的属性和方法全部包含进自己内部。因此，使用继承关系时一定要谨慎。

这些耦合关系给软件带来的问题是：类B 的变化会影响类A ，要求类A 也同时发生变化；程序员要想理解类A ，就必须首先理解类B；如果要想复用类A，就必须连类B 一起复用。鉴于这些原因，使整个系统的类之间尽量保持低耦合就是非常必要的一件事了。

面向对象系统就是通过对象之间的消息传递，实现整个系统正常运转。如果类与类之间没有关联，系统就不可能实现任何实际的功能。我们在这里讨论的，是如何**降低**耦合度，而不是如何**消灭**耦合关系。

#### 高内聚

这里的内聚指的是一个类的各个职责之间的相关程度或集中程度。一般说来，内聚度高的类应只包含很少的方法数，方法之间的关联程度很高，每个方法承担的工作量不是太大，任务也比较单一。

当我们在设计中发现一个类担负的任务太多、太杂，就应该主动把一些职责分配给其他的类，这样才能保证设计方案的高内聚。显然，内聚度高的设计方案更易理解、易维护、易重用。

#### 决定耦合的程度的依据

虽然我们总是希望类间的耦合度比较低，但是我们必须客观的评价耦合度。系统之间不可能总是松耦合的，那样肯定什么也做不了。而我们决定耦合的程度的依据何在呢？

简单的说，就是根据需求的稳定性，来决定耦合的程度。

对于稳定性高不容易发生变化的需求，完全可以把各类设计成紧耦合的（我们虽然讨论类之间的耦合度，但其实功能块、模块、包之间的耦合度也是一样的）。这样可以提高效率，而且我们还可以使用一些更好的技术来提高效率或简化代码，例如Java中的内部类技术。

如果需求极有可能变化，就需要充分的考虑类之间的耦合问题，有各种各样的办法来降低耦合程度，但是归纳起来，不外乎增加抽象的层次来隔离不同的类，这个抽象层次可以是具体的类，也可以是接口，或是一组的类。仔细观察GOF的23种设计模式，没有一种模式的思路不是从增加抽象层次入手来解决问题的。

设计不同的耦合度有利于实现透明和延展。对于类的客户（调用者）来说，他不需要知道过多的细节（实现），他只关心他感兴趣的（接口），目标类对客户来说就是一个黑盒子。如果接口是稳定的，那么实现再怎么扩展，对客户来说也不会有很大影响，那种牵一发而动全身的问题完全可以缓解甚至避免。

#### 怎样将大系统拆分成小系统

解决这个问题的一个思路是将许多类集合成一个更高层次的单位，形成一个高内聚、低耦合的类的集合。

耦合的目标是维护依赖的单向性，有时也会需要使用坏的耦合。这种情况下，应当小心记录下原因，以帮助日后该代码的用户了解使用耦合真正的原因。

# 依赖

## 定义

依赖的同义词：耦合（Coupling）。

依赖的概念：依赖是两个元素之间的一种关系，其中一个元素变化，导致另一个元素变化。

依赖的危害：如果被依赖元素发生变化，可能引起另一个元素不得不变化。

## 良性依赖原则

**从会不会造成头际危害的角度考察依赖**

如果被依赖元素发生变化，可能引起另一个元素不得不变化，这就是依赖的危害。但是，如果被依赖元素不发生变化呢？答案是：不会造成危害！于是，“冤案”产生了：由于需求分析上的偏差，设计中“在理论上”很稳定的耦合度低的依赖，可能“在实际中”恰恰是给我们造成危害的家伙；相反，“在理论上”声名狼藉的耦合度高的依赖，“在实际中”也可能并不给我们造成任何危害。

区分依赖的“实际危害”和“理论危害”是有实践意义的，从会不会造成“实际危害”的角度来考察依赖，将其分为良性依赖和恶性依赖两种类型。

1、恶性依赖：被依赖的元素“在实际中”，而不是“在理论上”，是“易变的”；

2、良性依赖：被依赖的元素“在实际中”，而不是“在理论上”，是“不易变的”。

**良性依赖原则**

不会“在实际中”造成危害的依赖关系，都是良性依赖；依赖的“理论危害”不一定成为“实际危害”，反之亦然。这就是良性依赖原则。

依赖是不可避免的，重要的是如何务实地应付变化。这就是良性依赖原则要做的。

**依赖是不可避免的**

OOD的实质，简而言之，就是妥善地为多个类进行职责分配，使这些类相互协作而构造出完成特定功能的系统。在软件设计中，依赖是不可避免的，就像人类社会不可能没有人与人之间的协作与依赖一样。

**重要的是如何务实地应付变化**

需求改变时常发生，而良性依赖是那些不会“在实际中”造成危害的依赖关系，所以，良性依赖是相对的——需求改变可能使先前不易变的元素变得易变起来，从而良性依赖也变成了恶性依赖。

Robert C. Martin在《敏捷软件开发》中提供的“只受一次愚弄”的策略很精辟。在我们最初编写代码时，假设变化不会发生；这时的设计很简洁，但对当时的需求，却是有效的、“不多不少的”。当变化发生时，我们就通过创建良性依赖，来隔离以后发生的同类变化；一般认为要通过创建抽象来隔离变化，而本章务实地认为只要是“不易变的”元素就可以。

## 循环依赖

允许两个组件通过#indude指令彼此“知道”，则隐含了循环物理依赖。

如果A 包引用了B 包中的对象，B 包引用了C 包中的对象，而C 包又引用了A 包中的对象，这种依赖关系就是循环依赖关系。

软件专业人员可能面临过这样一个局面：第一次看一个软件系统，似乎找不到一个合理的起点或者自身有完整意义的系统片段。不能理解或不能单独使用系统的任何部分就是一种循环依赖设计的征兆。

C++对象有一种显著的互相混杂的倾向，下面描述了这种潜在而危险的紧密物理耦合形态。

电路（circuit）是元件（elements）和电线（wires）的集合。因此，类Circuit知道Element和Wire的定义。Element知道它所属的Circuit，而且能分辨出是否与特定Wire相连，因此类Element也了解Circuit和Wire。一根电线可以连接到一个元件或一个电路的末端，为了完成这种连接，类Wire必须访问Element和Circuit 的定义。

这三种对象类型的定义存在于单独的物理组件（编译单元）中，以提高模块化程度。虽然这些单个类型的实现被它们的接口完全封装，然而，每个组件的.c文件都必须包含其他两个组件的头文件。这三个组件的最终依赖图是循环的，没有哪个组件可以在没有其他两个组件的情况下单独使用甚至单独测试。

草率构建的大型系统会因为循环依赖而变得紧密耦合，从而强烈地抗拒分解。支持这样的系统可能是一场噩梦，通常不可能对其进行有效的模块测试。

一个适当的案例是一种电子设计数据库的早期版本。那时，它的作者没有意识到在物理设计中避免循环依赖的必要性，其结果是创建了一个相互依赖的文件集合，其中包含了数千个函数的数百个类，最终导致没有办法使用甚至测试，而只能把它视作一个单一的模块。这个系统的可靠性非常差，并且不适于扩展和维护，最终不得不抛弃它而重新编写。

**循环依赖带来的问题**

层次化物理设计（即没有循环相互依赖）相对更容易理解、测试和重用，非循环物理依赖可以明显减少与开发、维护和测试大型系统相关的连接时开销。

组件之间的循环物理依赖，不仅使得组件难以测试，且不可能独立重用，而且还使得人们理解和维护这些组件更困难。一旦两个组件相互依赖，就需要同时理解两个组件，以便充分地理解其中的任何一个。

这个系统由7个组件（循环依赖）组成，每个组件直接或间接地依赖系统中的其他毎个组件。每个组件可以直接测试，但是没有一个组件可以在隔离的情况下测试或者独立于其他组件而重用。

循环依赖关系对于系统结构的危害非常大，它会使多个包之间的稿合度大幅度增强。在较大的开发团队里，循环依赖关系可能会使多个程序员的工作因为互相依赖而停滞不前。因此，在软件设计过程里，应严格杜绝循环依赖现象。

## 过多的编译时依赖

用C++开发一个多文件程序，那么改变一个头文件可能会潜在地引起多个编译中元要重新编译。

还有许多其他原因会引起不必要的编译时依赖。一个文件包含不必要的头文件，是造成C++中过多耦合的常见原因。过多地包含（伪）指令，不仅会增加编译客户程序的开销，而且改变一个低级头文件时会导致重新编译整个系统的时间会增加，甚至编译单个编译单元的时间也会增加，我们迟早会因为重编译的消耗太大而完全拒绝修改低级类。

过度编译时耦合，对于小型项目毫无影响，对于较大型的项目却可能成为支配开发时间的重要因素。如果忽略编译时依赖，有可能使得系统中的每一个编译单元包含系统中的几乎每一个头文件，从而使编译速度降低到跟爬行一样。

**一个过度编译时耦合的普通例子：**

开始时好像是一个好主意，随着系统容量的增长而变得槽糕。myerror组件中myerror.h定义了一个MyError结构，其中列举了所有可能的错误代码，每一个加进来的新组件都自然地包含这个头文件。但是，每一个新组件都可能有它自己的错误代码，这些代码并未标识在主列表中。

当组件的数量更多吋，我们将自己的错误信息代码加入到清单中的愿望就没有那么强烈了，我们对重用已有的错误信息代码感兴趣，因为不必修改myerror.h文件。最终，我们将放弃任何添加新的错误信息代码的想法，只是简单地返回ERROR或WARNING而不是修改myerror.h文件。到这个时候，我们就会陷入这样一个境地：所作的设计变得不可维护并且实际上己毫无用处了。

## 依赖解除的手段：层次化设计（循环依赖）

### 升级

原则：在庞大的低层子系统中的循环物理依赖将最大程度地增加维护系统的总开销。

从这个分析中得到的重要结论是，与低层次子系统有关的高度耦合可能会极大的增加开发和维护较高层次客户程序和子系统的开销。

总结：可以通过把相互依赖升级到更高的层次，将循环依赖转换成受欢迎的向下依赖。通过避免子系统本身内部组件之间不必要的依赖，可以显著降低子系统和它的所有客户程序的维护开销。同时，子系统也可以变得更灵活从而更可重用。这个改进设计的好处对于一个系统的较小版本也许并不显著。

### 降级

原则：同一层次的组件是循环依赖的，那么就有可能把互相依赖的功能从每一个组件降到一个潜在的新的较低级（共享）组件中，每一个原来的组件都依赖于这个新组件。

到现在为止，我们已经努力通过把互相依赖的功能推到物理层次结构的更高层来消除循环依赖。在这一节中，我们要探讨把公用的功能推到物理层次结构的更低层的技术，在那里公共功能可以被共享甚至也许可以被重用。这种把公用的功能移到物理层次结构的更低层的技术在本书中称为降级（demotion)。

升级和降级是相似的，因为在这两种情况下，都是通过把循环依赖功能移到物理层次结构的另一层的方法来消除组件之间的循环依赖。

### 不透明指针

原则：如果一个被包含的对象拥有一个指向它的容器的指针，并且要实现那些实质地依赖那个容器的功能，那么我们可以通过以下方法来消除相互依赖：1、让被包含类中的指针不透明；2、在被包含类的公共接口上提供对容器指针的访问；3、将被包含类的受影响的方法升级为容器类的静态成员。

定义：如果编译函数f的函数体时要求提前看到类型T的定义，则称函数f实质（in size）使用了类型T。

定义：如果编译函数f以及f可能依赖的任何组件时，不要求提前看到类型T 的定义，则称函数f只在名称上（hinameonly）使用了类型T。

上述定义，也可以扩展到类或组件。

如果一个指针所指向的类型的定义不包含在当前的编译单元中，这个指针就被称为是不透明的（opaque）。

如果一个函数使用了一个T类型对象，那么它以一种需要知道T的定义的方式使用T。也就是说，为了编译函数体，编译器需要知道它所用的对象的大小和布局。在C++中，一个编译器获悉一个对象的大小和布局的方法就是让使用这个对象的组件包含含有该对象类定义的组件的头文件。

如果一个函数体在只是看到类型T的声明（例如，class T;）的情况下就可以被编译，那么那个函数本身并不依赖于T的定义。

实质使用一个类型，会导致对定义T的组件的一种直接的编译时依赖。

虽然函数f一般只在名称上使用而不是实质使用类型T，但是如果f调用了其他组件中的一个或多个函数，这些函数依次地依赖T的定义，那么在这种情况下仍然有一个f对T的连接时依赖。

如果函数f和f依赖的所有组件在只看到了 T的声明（而不是定义）的情况下就能够编译和连接，那么f就被认为只在名称上使用了T。

只在名称上使用一个类型，则没有隐含的物理依赖——即使是在连接时。没有物理依赖，耦合也就几乎全部消除了。

### 哑数据

原则：哑数据可以用来打破in-name-only依赖，促进易测试性和减少实现的大小。但是，不透明指针可以同时保持类型安全和封装；而哑数据通常是不能的。

术语哑数据（dumb data）是对不透明指针概念的一种概括。哑数据是一个对象拥有但不知道如何解释的任何种类的信息。这样的数据必须用在另一个对象的上下文中，通常用在一个更高的层次上。

结论：哑数据是不透明指针的一种泛化，它有助于子系统的实现，在这此子系统中低层次的对象必须隐含地引用其他低层次对象。这种技术在这样的地方尤其有用：某些引用在子系统的较低层次不必解释，在某个（通常）较高层次的对象的上下文中解释。在这种受约束的上下文中，尽管会损害类型安全和封装，但实现可以更简洁。哑数据的使用是典型的低层次实现细节，通常不会暴露在较高层次子系统的接口中。

### 冗余

原则：与一些重用形式相关的额外耦合可能会超过从该重用获得的利益。

任何种类的重用都隐含着某种形式的耦合，在某些情况下耦合还可能很严重。在本书中，冗余指的是为了避免由重用导致的不需要的物理依赖而故意重复代码或数据的技术。

当功能在一个独立的物理单位中并且要重用的功能数量相对较小，而会导致的耦合的数量却是不合比例地大，以致于超过了重用的好处时，冗余是必要的。在重用的数最比较多的情况下，通常适合将共有代码降级到一个较低的层次，在那里可以共亨它。

甚至在单个的子系统内也有一个限度，在此限度之下外部功能的重用可能不是有益的。考虑两个独立的大组件，有可能其中一个组件实现了一小块功能（例如min、max等），而另一个组件可能重用它。把这一小块实现降级到一个独立组件中会不合道理地增加该子系统的物理复杂性，仅仅为了这样一点点重用而导致一个组件依赖另一个组件也会不合道理地增加子系统的CCD。若允许一个组件上支配另一个组件，那么会减少添加其他依赖关系的灵活性（即将来对组件的增强导致的添加其他依赖关系的灵活性）。有时候对重用来说一个可行的替代方案就是重复代码和避免耦合。

原则：提供少量的冗余数据可以使对一个对象的使用只是在名称上，从而消除连接到那个对象类型的定义的开销。

### 回调

一个回调（call back）是一个函数，由一个客户提供给一个子系统，它允许该子系统在该客户程序的上下文中执行一个特定的动作。

原则：不加选择地使用回调可能导致难以理解、调试和维护的设计。

回调是强有力的消除耦合的工具，但是应该只在必要的时候使用它们。由一对互相调用对方的成员函数的类引起的相互依赖是拙劣设计的一个症状。回调有时候可以用来打破循环，但通常这个问题最好通过对功能重新打包来处理。

原则：对回调的需求可能是不良的整体体系结构的一个症状。

但是有时候静态回调可以用来很好地消除对大型子系统的依赖。

总结：回调是一个函数，由一个客户提供，用以允许一个（通常是）较低层次的组件利用一个行为，这个行为需要一个（通常是）较高层次的上下文，虚函数可以用来实现一个类型安全回调机制。回调是打破协同操作类之间的依赖的强有力工具。回调对于图形学和基于事件程序设计是极其重要的。

如果不适当地使用，冋调可能会模糊低层次对象的职责并导致不必要的概念上的耦合。通常，回调（就像递归一样）可能比传统的函数调用更难以理解、维护和调试。它们的（伪）异步特件需要开发人员给予一种不同类型的关注。作为一个规则，回调应该被当作是最后求助的避难所。

### 分解

分解（factoring）的意思是提取小块的内聚功能，并把它们移到一个较低的层次，以便它们可以被独立地测试和重用。分解是减轻由循环依赖类强加的负担的一种非常普通而高效的技术。分解和降级类似，只是分解的行为不必然消除任何循环；取而代之的是它只减少参与到循环中的功能的数量。通过分解可以将循环依赖升级到一个更高层次（在那儿它们的不利影响较不显著）。

原则：将独立可测试实现细节分解出来并降级，能够减少维护一个循环依赖类集合的开销。

原则：在循环物理依赖不可避免的地方，将其升级到尽可能高的层次可减少CCD，甚至可以使循环能够由一个单个的、大小便于管理的组件代替。

总结：分解是一种通用技术，可用来减少有着固有循环依赖的设计的维护开销。通过将一些实现复杂事务重新安装到较低层次组件中，可以独立于余下的循环相互依赖代码对该功能进行测试（并可能重用）。一般的分解会得到更灵活的体系结构，又不会牺牲运行时效率。然而，在分解一个子系统的接口时，客户可能被要求使用子系统层次结构中较低层次的组件接口。

## 依赖解除的手段：绝缘

### 定义

好的物理设计的另一个重要方面是避免不必要的编译时依赖，过度的编译时耦合会严重地削弱我们维护一个系统的能力。一般来说，对驻留在一个组件的物理接口中的、通过编程不能访问的实现细节进行修改将强迫所有的客户程序重新编译。甚至对一定规模的大型项目，重新编译整个系统的开销也将抑制对低层次组件的物理接口的修改，甚至限制我们对它们的实现封装细节作局部的修改。

绝缘是一个物理设计问题，它的逻辑相似物一般称为封装。

绝缘与封装的过程类似，是指避免或消除不必要的编译时耦合的过程。

**从封装到绝缘**

在理想情况下，对一个组件的逻辑实现的修改，不应该影响客户程序。但是，在现实中，C++编译器依赖于一个头文件中的所有信息，包括私有的数据。如果一个人通过检查头文件来决定一个组件的实现策略，那么如果那个组件的实现策略发生了改变，该组件的客户程序很可能被迫重新编译。

即使在只有一个组件的实现作出了改变的情况下，迫使客户程序重新编译也不是一个组件的受欢迎的物理特性。越多的组件依赖那个组件，就会有越多这种不受欢迎的编译时耦合出现。不能将客户程序与逻辑实现的变化“绝缘”，会对开发大型项目的开销产生严重的影响。

想象一个有N个组件的系统，系统中每个组件在编译时都依赖其他所有的组件。也就是说，编译一个组件意味着要从所有N个组件的头文件中包含定义并对定义进行语法分析。对这样的系统中的任何一个头文件进行一次修改的编译吋开销都足惊人的。任何一个编译单元的编译开销都依赖于整个系统的大小，而不是与组件本身的大小成正比。随着整个系统规模的扩大，任一组件的编译开销会以一种不成比例的高速度增加。随着越来越多的头文件被读入每个编译单元，编译器数据结构的负荷越来越重。也就是说，若包含到一个编译单元中的行数翻倍，则用来对它进行语法分析的时间要比原来的两倍更多。

定义：一个被包含的实现细节（类型、数据或函数）如果被修改、添加或删除时不会迫使客户程序重新编译，则称这样的实现细节被绝緣了。

当一个大型系统的各种层次的内部版本之间发生错误时，修补绝缘组件（即，使客户程序与它们的实现绝缘的组件）要比修补没有绝缘的组件容易得多。就没有改变的接口而言，修改后的实现可以放在适当的位置，而不需耍重新编译其他组件或者担心头文件过时。

绝缘价值的一个最后的实证是它允许我们透明地替换动态装载的库。动态装载的库不是连接到个单个的可执行代码中，而是实时地链入到一个运行的程序中。假设你是一个C++应用程序库的销售商，如果你供应一个完全绝缘的库实现，那么你在增强性能和修复故障时完全不用打扰你的客户。给客户发送一个更新版本不会迫使用户程序重新编译或者连接，客户所要做的只是重新配置环境以指向新的动态装载的库。

### 什么时候不进行绝缘

原则：如果一个组件的用户是也限的，绝缘就不再是关键的了。

对于广泛使用的组件来说，尤其不希望由于封装细节的变化而引起客户程序重新编译。但是，绝缘有性能代价。小的、轻量级的类如Point、Stack等定义良好的具体的数据结构不太适合绝缘。对于那些经常重复调用的类（例如带有极小访问函数的类），使用间接和非内联的函数可能会使它们慢一个数量级，除非性能很明显不是一个问题。

当函数调用的运行时开销相对于调用本身的开销大时，绝缘不会引起严重的性能问题。  
原则：轻量级的、被广泛使用的并且通常通过值返回的对象，会显著地降低整体运行时性能。

更多参考《大规模C++程序设计》

### 编译时耦合的开销

这些耦合如何消除，参考《大规模C++程序设计》

### C++结构和编译时耦合

#### 继承（IsA）和编译时耦合

一个类无论何时派生自另一个类，即使是私有派生，也没有办法把客户程序与这个事实绝缘。即使私有继承被认为是派生类封装了的实现细节，派生对象的物理设计也会迫使包含派生类定义的每个客户程序都要见到基类的定义，因此对一个派生类而言，把含有基类的头文件显式地包含在该类的头文件中是合适的。无论何时修改基类的头文件（即使仅仅添加注释），在将该基类的派生类的客户程序连接为任何新的可执行程序之前，UNIX工具（如 make）会迫使重新编译所有客户程序。

#### 分层(HasA/HoldsA)和编译时耦合

当一个类在其定义中嵌入了另一个用户自定义类型的一个实例时（HasA），这个类的物理布局就会变得紧密依赖于该类型的布局。其结果是，如果没有见到嵌入到某个对象的每个分层子对象的定义，则一个客户程序不可能包含该对象的类定义。因此，若某些头文件包含了物理上嵌入到那个类的每个分层对象的定义，则一个复合对象显式地包含这些头文件是合适的。

与此相对的是，当一个类只拥有一个对象的地址时（HoIdsA），这个类不必依赖这个被拥有对象的物理布局。如果是这样，包含这个类的文件头不去包含被拥有对象的头文件而只是声明其类型是合适的。

#### 内联函数和编译时耦合

如果某函数要在当前组件之外替换为内联的，那么声明为inline的这个函数必须定义在头文件中，该需求迫使将内联函数体放在组件的物理接口上。由于除了通过其自身的逻辑接口来调用内联函数之外不能通过编程来访问它，所以说一个内联函数是封装的。但是，对象的此逻辑实现部分没有与客户程序绝缘，因此有以下后果：

1、能使用改组件的任何程序员都能看到内联实现。

2、改变一个内联函数的实现会迫使定义内联函数的组件的所有客户程序重新编译。

3、将一个函数改为内联函数或从内联函数改回来，也会迫使定义该函数的组件的所有客户程序重新编译。

4、从一个内联函数通过值返回的一个对象被实质地（in size）用在头文件中，因此决不会与客户程序绝缘（尽管从非内联函数通过值返回的一个对象可能会）。对一个在内联函数的函数体中被实质使用的对象也是这样。所以，当一个用户自定义对象传入，被使用在一个内联函数中或通过值从一个内联函数返回时，显式地将定义了被使用对象的文件头包含在定义了该内联函数的头文件中是合适的。

#### 私有成员和编译时耦合

一个类的每个私有数据成员——尽管封装好了——也没有与该类的客户程序绝缘。改变私有数据成员的类型，将迫使所有的客户程序重新编译。

私有成员函数是一个类的封装细节，但是它们并不是己绝缘的实现细节——甚至在它们没有声明为inline的时候。

#### 保护成员和编译时耦合

像私有成员一样，保护接口是在类定义中声明的，因而就一般用户而言，它不是个绝缘的实现细节。以任何方式修改一个基类的保护接口，将迫使以下三个方面重新编译：1、基类的全部客户程序；2、所有的派生类；3、派生类的所有客户程序。

#### 编译器产生的成员函数和编译时耦合

一些基本的成员函数是由编译器自动产生的，除非这些函数在一个类中被显式地声明。

在许多情况下，编译器产生的构造函数、赋值运算符以及析构函数所做的正是所需要的工作。但是，如果一个类的作者确定的需求与编译器产生的定义不同，那么有必要将合适的成员声明引入到类定义中。这样的声明引入不能认为是绝缘的，并且类的任何客户程序都将被被重新编译。

#### 包含命令和编译时耦合

每个头文件将包含一个或多个头文件，这些文件又依次包含一个或多个别的头文件，直到最后实际上在系统中的每个头文件都被包含了。

#### 默认参数和编译时耦合

默认值会随着接口一起编译，而对这些值的任何修改将强迫客户程序重新编译。

#### 枚举类型和编译时耦合