

OO Boot Camp Explained

In C++

Arthur Yuan @ ThoughWorks

ABSTRACT

[Type the abstract of the document here. The abstract is typically a short summary of the contents of the document.]

Table of Contents

| | |
|------------------|-----------|
| 第一部分：设计原则 | 5 |
| 为什么要做软件设计？ | 6 |
| 成本 | 7 |
| 软件设计中的“偶发成本” | 8 |
| 质量 | 10 |
| 没有失败的项目 | 12 |
| 结论 | 12 |
| 重用 | 13 |
| 修改 | 13 |
| 拷贝粘贴 | 13 |
| 重用 | 14 |
| 可重用 | 14 |
| 重用的范围 | 14 |
| 可重用性 | 15 |
| 结论 | 15 |
| 高内聚，低耦合 | 16 |
| 内聚 | 16 |
| 耦合 | 17 |
| 内聚和耦合的关系 | 19 |
| 正交 | 20 |
| 正交策略 | 21 |
| 结论 | 21 |
| 策略1：消除重复 | 22 |
| 什么是重复 | 22 |
| 重复的邪恶性 | 24 |
| 重复类型 | 26 |
| 产生原因 | 29 |
| 重复和重用 | 30 |
| DRY | 31 |
| 结论 | 31 |
| 策略2：分离关注点 | 32 |
| 粒度与重用 | 32 |
| 功能分解 | 33 |
| 关注点 | 34 |
| 例子 | 34 |
| 动机 | 38 |
| 手段 | 39 |
| 粒度 | 39 |
| 必要性 | 40 |
| 单一职责 | 41 |

| | |
|-----------------------------|------------|
| 策略3：缩小依赖范围 | 42 |
| 依赖点 | 42 |
| 减少依赖点 | 42 |
| 策略4：向着稳定的方向依赖 | 44 |
| 简单设计 | 45 |
| 第二部分：练习一 | 46 |
| 需求一 | 47 |
| 查询函数 | 51 |
| 构造和SETTER | 55 |
| C++ 封装手段 | 57 |
| 逻辑依赖 & 物理依赖 | 60 |
| INLINE | 62 |
| 需求二 | 64 |
| 模块 | 68 |
| 直觉 | 71 |
| 命名 | 72 |
| 需求三 | 74 |
| 抽象的动机 | 74 |
| 继承 | 76 |
| 第三部分：练习二 | 77 |
| 需求一 | 78 |
| STRUCT vs. CLASS | 79 |
| YAGNI | 85 |
| 需求二 | 87 |
| 基类命名 | 91 |
| 需求三 | 96 |
| 常量 | 100 |
| GETTER vs. TDA | 104 |
| 知识的显性化 | 107 |
| 基于分析的设计 | 108 |
| 开放封闭原则 | 111 |
| ENUM, IF-ELSE & SWITCH-CASE | 114 |
| 需求四 | 118 |
| SLUG IN C++ | 119 |
| SLUG & SINGLETON | 121 |
| 需求五 | 125 |
| NULL OBJECT | 132 |
| DELEGATE POINTER | 135 |
| PIMPL | 141 |
| 需求六 | 146 |
| 测试与重用 | 159 |
| 预防胜于治疗 | 160 |
| 需求七 | 161 |
| 可测试性设计，抽象& 测试替身 | 163 |

| | |
|------------------------|------------|
| ITERATOR vs. TDA | 170 |
| 笛米特法则 & TDA..... | 182 |
| 依赖注入..... | 186 |
| 需求八 | 187 |
| SPIKE..... | 190 |
| 对象的哲学..... | 194 |
| STRATEGY & TDA..... | 196 |
| 价值观, 原则, 模式与实践..... | 200 |

第一部分：设计原则

为什么要做软件设计？

“Common sense is not so common”

—— Voltaire

在讨论为什么要做软件设计这个问题之前，我们需要先回答一个更加根本的问题：对于一个企业而言，为什么要开发软件？

当被问起这个问题，相信绝大多数人都会同意，开发软件为了“**创造客户价值**”，或者“**满足客户需求**”，无论这个客户是外部客户还是内部客户。然后，通过“满足客户需求”为自己或自己所在的企业直接或间接的获取**收益**。

但软件开发的过程还要产生**成本**：雇员的薪水，设备的购买，办公室的租赁，等等等等。

而一个简单的商业逻辑是：

$$\text{收益} - \text{成本} = \text{利润}$$

所以，为了让利润最大化，除了需要放大收益之外，亦需要压缩成本。

为了放大收益，公司需要有好的市场战略，提高市场份额，提高行业门槛，提高产品售价等等等等。

但这不是我们要讨论的重点。企业战略，市场营销，商业运作等相关的事情，是与软件开发技术无太大关系的另外一套知识体系。

开发团队需要关注的是：**是否有能力快速交付客户真正需要的产品**。而这样的能力将会对企业收益的提升很有帮助。

问题的另外一个方面则是成本。但我们也不打算在这里讨论更大范围的成本压缩问题，比如是否降低管理成本，是否购买不那么必要的设备，是否租赁更便宜的办公室等等。那属于管理学的范畴。

我们仅仅讨论，开发团队可以做什么，能够帮助企业降低成本。

成本

为了进一步讨论，我们需要先明确两个概念：

- 内在成本（Essential Cost）
- 偶发成本（Accidental Cost）

内在成本，指的是解决一个问题的最简单方案所付出的成本。

偶发成本，则是指为解决一个问题，一个解决方案相对于最简单方案多出的那部分成本。



为了更好的理解这两个概念，我们来看一个需求：使用C++语言编写一个Hello, World程序。

```
std::cout << "Hello, World" << std::endl;
```

不知是否还有成本更低的实现方法，但这是我现在所能想到的最简单写法。我们暂且把它定义为解决这个问题的“内在成本”。

但还有很多种方案都可以实现这个需求，比如：

```
for(int i=0; i<strlen("Hello,World\n"); i++)
{
    putchar("Hello,World\n"[i]);
}
```

这个方案明显比之前的方案实现成本更高，但这还不是最高的：

```
void alien_say(char * p)
{
    while (putchar(*p += *(p + 1) - *p));
}
```

```
alien_say("BETHO! Altec oh liryom(a loadjudas!) dowd.");
```

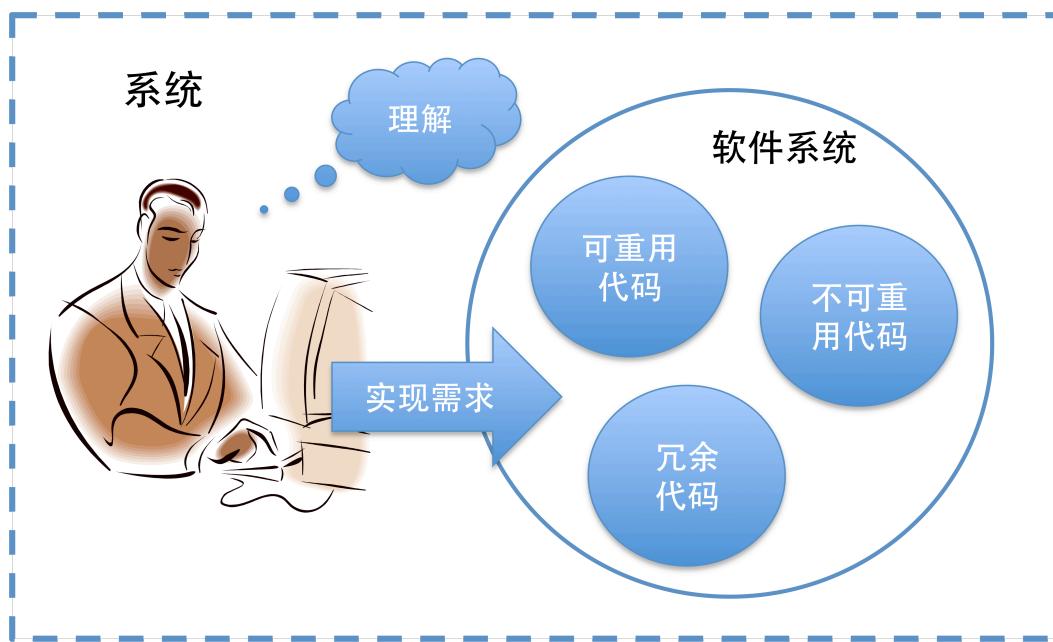
后两种方案相对于第一种方案都更加复杂，那么为之需要付出的成本也就越高。无论第一种方案是否是最简单的方案，但后两种方案无疑都存在一定的“偶发成本”。

所以，对于开发团队，如果能够尽量降低开发维护活动中“偶发成本”，那么我们就间接的为企业创造了利润。

软件设计中的“偶发成本”

为了降低“偶发成本”，我们需要首先识别它。

为了了解开发活动中的“偶发成本”生于何处，我们先来看看开发人员与软件系统所构成的系统中都包含那些元素和活动。



一个不难得出的结论是：当一个需求来临的时候，如果我们能够尽可能的利用系统中已有的元素，我们就可以付出尽可能小的努力来应对这个变化。这就是所谓的“重用”。但如果一些本来应该可以重用的部分，无法做到重用，则必须所有事情都要从头做起。这些工作量“重叠”的部分，就属于“偶发成本”。

但无论你的软件设计的多么易用重用，在很多时候，你仍然必须去修改原有系统。这属于“内在成本”的问题。为了能修改它，首先要理解它。它只有越容易理解，你才能更快、更有信心的去修改它。另外，即使原有系统易用重用，也只有在开发

人员理解它的情况下才可以重用。如果一个本来可以更容易理解的问题，被描述的可理解性更差，比如：混乱的逻辑，超高的圈复杂度，无序的排版，晦涩的命名等等。这就引入了“偶发成本”。

另外，对于一个问题，如果存在更简单的解决方案，但设计人员由于能力，或对于变化的恐惧，给出了一个复杂的方案。那么它相对于更简单方案所引入的复杂度，就是一种“冗余”。“冗余”的另外一种表现是：系统中充斥着大量已经完全无用的“死代码”。“冗余”的存在，会造成不必要的维护工作。而“冗余”毫无疑问会引入“偶发成本”。

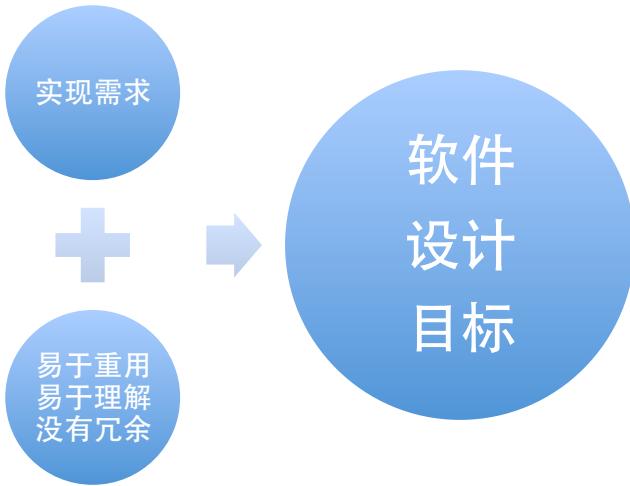
在这样一个系统中，应该不难得出如下结论：

- 在开发人员“实现需求”时，如果原有系统中对于要实现的需求，可重用的部分越少，那么开发人员为了实现需求所需要付出的成本就越高；
- 如果一个系统有效的代码部分被设计和实现的非常难以理解，则开发人员就必须花费很大的时间和精力来理解系统，从而付出更高的成本；
- 如果一个系统的冗余代码越多，开发人员就会把时间浪费在它们身上，从而可以产生不必要的成本；

然后，我们就可以得出进一步的结论：我们的软件在如下三个方面越出色，我们需要付出的“偶发成本”就越小。

- **易于重用**
- **易于理解**
- **没有冗余**

所以，我们将这三点与软件开发的根本目的——实现需求——放在一起，作为我们的软件设计目标。



质量

从传统的评价体系中，“质量”用来评价一个软件系统是否正确的满足了“客户需求”。

诚然，一个软件系统之所以存在，正是为了满足客户的需要。绝大多数情况下，客户根本不关心一个软件系统是如何设计的，客户关心的是：自己需要的功能是否在预算之内被正确的提供。

反过来，作为软件供应商，其开发者只要懂得相应的编程语言，具备必要的逻辑思维能力，就很可能实现出一个满足需求的软件。

这种现实造就了这样的结果：

没有一个工程技术行业像软件一样门槛低下，大量的从业人员来自于其它专业。

没有一个工程技术行业像软件行业一样，以30岁以下人群为主体，一旦过了这个年龄，大多数人就倾向于不再从事具体的开发工作。或者简单的说，岁月所积累的经验并不被认为有更大的价值。

但…

也没有一个工程技术行业像软件行业一样，只要一个项目达到一定规模，项目周期超过6个月，很大的概率会开始陷入“焦油坑”。

是的，一群懂得一门编程语言，具备必要逻辑思维能力的人可以交付一个可以工作的软件。

只要这个软件被客户接受和喜欢，“每个伟大的软件都有下一个版本”。在它随后的版本推进过程中，越来越多的修改导致成本快速上升，举步维艰，只好依靠没日没夜的加班来扩展功能，修复缺陷，很快将软件开发人员的热情和精力耗尽，过早失去对于软件开发的兴趣。

然后，下一批年轻的，缺乏设计能力的毕业生被补充进来，进一步恶化系统的状态，直到它彻底腐烂消亡，只能完全推到重来——而这不过是新一轮恶性循环的起始。

造成这种结果的原因在于，绝大多数团队对于软件设计的理解仅限于“功能实现”。即，从客户的观点来看，只要一个系统是稳定的，正确的，健全的，那么软件设计就达到了它的目标。

这就是所谓的“外部质量”，是每个人都能观测到，并能感受到其重要性的质量。它意味着软件存在的原因，它意味着“收入”。

但“外部质量”和“软件设计能力”的关系并不大。“软件设计能力”的高低对于“外部质量”好坏的影响并不是决定性的。只要你在这个行业呆的足够久，你就能不断看到设计能力匮乏的团队同样能够交付具备良好“外部质量”的软件。

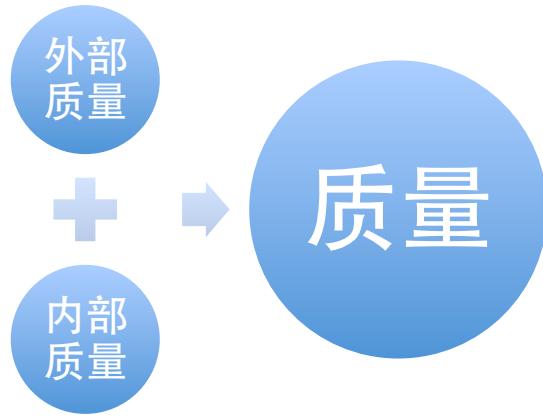
但是，相对于软件设计能力更好的团队，软件设计能力差的团队在相同的努力下，交付的速度更慢。换句话说，为了能够交付具有相同“外部质量”的软件，软件设计能力匮乏的团队需要付出更多的努力。

这些努力对于团队而言意味着“成本”。付出更多的努力，意味着付出更高的“成本”。

所以，单纯的“外部质量”并不能全面的衡量一个软件系统的价值所在。因为，如果一个软件开发企业需要为之付出的开发维护成本长期超过它所能带来的受益时，那么它为软件开发企业所带来的就是负价值。在正常情况下，这个软件也就走到了它生命周期的尽头。

另外，“内部质量”和“外部质量”并非各自独立的事物，“内部质量”的优劣最终会反映到“外部质量”上。更低的“内部质量”，在相同的条件下，会导致更慢的交付速度，更多的缺陷。并且这种情况，越往长期发展，就体现的越明显。

所以，两种质量合在一起，才能最终决定一个软件给企业创造利润的能力。



没有失败的项目

曾有一次与一位前同事聊天，他看到一个蒸蒸日上的企业的软件系统，其内部质量已经到了无法更差的地步，几乎每一行代码都是对良好设计原则的违反。他对此满腹疑问，“为什么一个这样的系统还能成功？内部质量真的重要吗？”

我当时的回答是：“没有失败的项目，只有被取消的项目”。

尽管一个具有糟糕内部质量的系统，需要付出高昂的维护成本；但只要它为企业创造的收益能够承担这样的成本，或远大于维护成本，那么企业就可以很好的活下去。但这并不意味着那部分成本并不存在。

只要一个软件的功能从技术上是可行的，如果企业愿意付出无尽的成本，软件总是可以被做出来。但企业在正常情况下是理性的，那些最后被取消的项目，无非是因为其成本超出其所能带来的收益。而这样的项目并不少见。

这是“做正确的事情”和“正确的做事”之间的关系。一个错误的软件，无论内部质量有多好，如果不能给企业带来足够的收益，那么最终也是一个失败的项目。反过来，一个正确的软件，尽管错误的方法也可能导致成功，但正确的方法只会让它更成功，或者具备更高的成功概率。

结论

对于软件设计，Kent Beck总结道 “Design is there to enable you to keep changing the software easily in the long term”。

之所以强调“长期”，是因为“对于一个不需要维护的一次性项目，有着完全不同的方法学”。

重用

“Don't reinvent the wheel, just realign it.”

—— Anthony J. D'Angelo

我们上一章已经得出结论，一个系统的越易于重用，则为开发者带来的“偶发成本”越少。

重用是整个软件开发社区几十年来一直努力的重要课题。由此产生了各种设计范式，原则与模式。许多出色的软件设计师，尤其是传统的设计师，在“满足需求”这个目标之外，可能并不过度关心软件的表现力和冗余问题，而是将所有的才华和注意力都放在了对“重用”的追求上。

修改

当要实现一个新的需求时，一种常见的“重用”方式是：在原有函数里塞入几行新的代码逻辑。

这样做的动机，绝对是为了“重用”原有流程中已存在的代码。而这样的做法，最不需要思考。在很多情况下，也确实是成本最小的实现方式。

但如果所有人都使用这样的策略，那么原有函数的逻辑很快会变得复杂，进而晦涩难懂。从长期来看，团队需要付出更多的成本来理解和维护它。

另外，一个经过修改的软件单位，必须重新进行验证；无论是它自身，还是调用它的代码，都必须这么做。如果一个软件单位的修改包括接口的部分，还会造成所有使用它的代码的修改。这就会造成更大范围的重新验证。

拷贝粘贴

另外一种常见方式是：为了不影响原有功能，将原有代码进行“copy-paste”，得到一份拷贝。然后在拷贝代码上修改了其中的一部分，得到另外一段代码。两段代码对应于两种不同的需求，同时存在于同一系统。

这种方式的动机也是“复用”，但造就的结果却只是“重复”。它的危害甚至比前一种方式还要大。

重用

所以，重用方式虽然有很多种，但其中一些会比另外的方式更合理。虽然目的都是想通过“重用”来达到降低effort，但不合理的重用方式只是降低了眼前的effort，从整个项目的生命周期来看，effort反而增加了。

一个已经存在的软件单位，无须任何修改，无须拷贝粘贴，就可以用于帮助实现某些需求。这样的重用方式无疑是最合理的。我们仅仅将这样的重用方式，定义为“重用”。

可重用

当我们谈到“重用”，句法结构是A“重用”B。反过来，则是B对于A“可重用”；如果A无法“重用”B，则B对于A“不可重用”。

一个已经存在的软件单位，无须任何修改，无须拷贝粘贴，就可以用于帮助实现某些需求；则我们称“这个软件单位”对于“这些需求”是“可重用的”。

如果一个软件单位必须被修改，才能满足新的需要，即便你只修改了其中的1%，看起来你好像“复用”了其中的99%。我们仍然称它对于这个需求“不可重用”。

如果这个软件单位的无须修改的部分中，能够与需要修改的部分进行分离，那么无须修改的部分对于这个需求就是“可重用”的。

重用的范围

一个“可重用”实体，只能在可见的范围内被重用。

它描述了这样一些事实：

- 如果一个常量在一个函数内被定义，那么它就只能在当前函数内被重用；
- 如果一个类方法被定义为私有的，则它只能在当前类中被重用。
- 如果一个类被隐藏在一个模块内部，那么它就只能在一个模块内被重用。
-

这看似一句废话。但它所描述的约束却对于设计决策影响重大。

当你定义一个实体的时候，你希望它在多大的范围内被重用，也就决定了它应该被部署的位置，进而影响系统的类设计和结构设计——

如果一个可在更大范围内重用的单位被控制在较小的范围内，那么势必会造成重复。

如果一个可重用单元U，被模块A所定义，其初衷是仅供本模块内部使用，但却没有对U进行访问范围控制。由于模块A被定义了明确的职责，现在由于U的公开，事实上造成了模块A提供了更多的职责。

随后，另一个模块B正好需要这个功能，所以直接进行了调用。然后就造成了B对A不恰当的依赖。

可重用性

没有绝对可重用的东西。只有对合适的问题，一个软件单位才可以被重用。一个软件单位适用的范围越广泛，那么它被重用的可能性就越高。我们把重用的可能性，定义为“可重用性”。

比如，一般而言，STL库的“可重用性”要优于一个领域中间件。

结论

高内聚，低耦合

"The only thing that does not change is change itself."

——Leif Ericsson Leo Veness

我们已经讨论过，“重用”对于降低成本的意义。但问题是，如何才能让我们的软件具备更好的“可重用性”？

1975年，Larry Constantine和Ed Yourdon在其著作《Structured Design》里，正式提出了著名的设计原则：“高内聚，低耦合”。而它则是解决“可重用性”问题的最高原则。

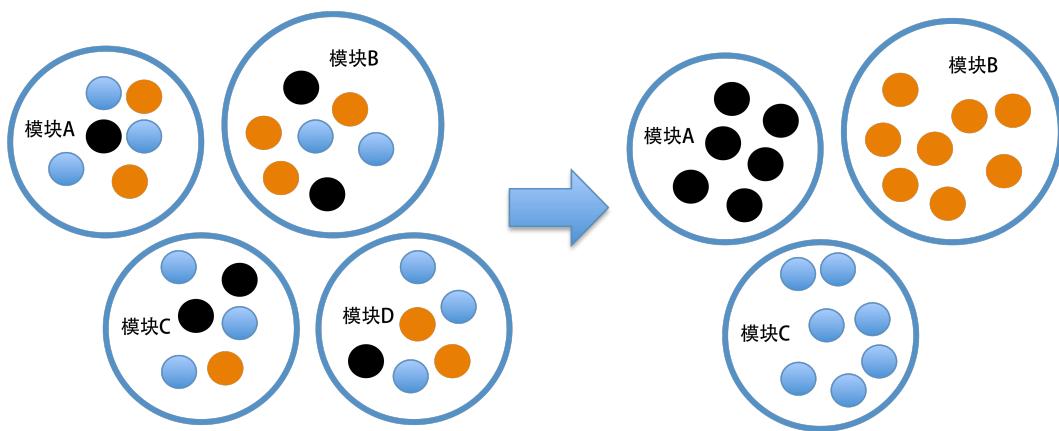
内聚

内聚，用来衡量一个单位（函数、类、结构体、模块、子系统）内部各种元素之间的关联紧密程度。元素之间关联度越高，则其内聚度越高。

一方面，高内聚强调，**紧密关联的事物应该放在一起**。这会增强一个软件单位的可重用性和可理解性。

假设一个系统，有四个模块，提供三种功能。但三种功能的代码分散在四个模块内。如下图所示，其中不同的颜色代表不同的功能的代码块。

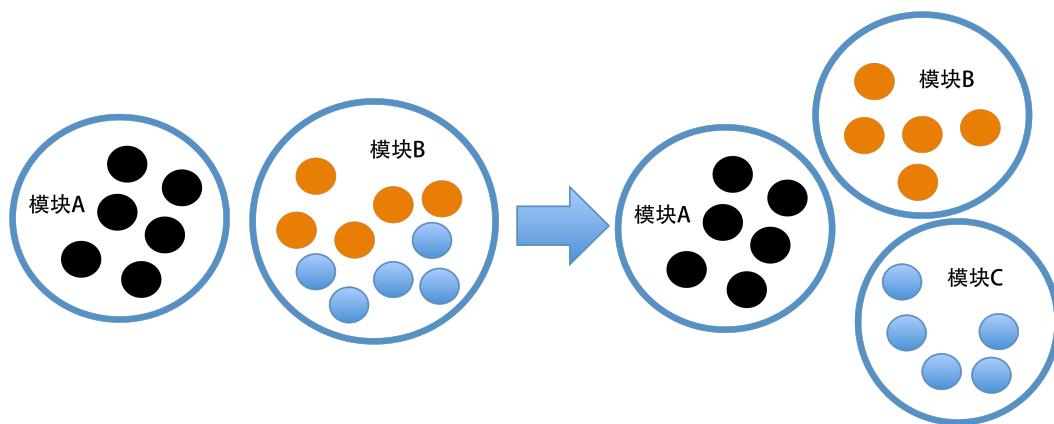
由于这种分散性，为了重用某个功能，我们必须要把四个模块一起拿来，才可以重用。由于每个模块中都存在自己的环境和假设，比如对于具体操作系统的调用，对于某个具体硬件平台的依赖，或对于某种配置的依赖等等，这就会降低其中任何一个模块的“可重用性”。



另外，为了修改一个功能，必须要到所有的模块内进行维护。这就意味着，你只有理解了所有模块，才可能对一个功能进行良好的维护。同时，由于每个模块都有所有功能的代码，对一个模块的修改，很可能会影响所有的功能。但如果我们把不同功能的代码放入各自的模块，则上述问题都得到改善。

另一方面，高内聚强调，**只有紧密关联的事物才应该被放到一起**。这同样会提高软件单位的可重用性和可理解性。

同样是上述的例子，如果我们把其中两个功能的代码都放在模块B内，那么为了重用一个功能，却被迫引入另外一个功能的代码。并且，由于模块B的功能不单一，我们必须理解两个功能的代码，才可以对模块B进行有效的维护。



把上述两个方面结合起来，就是著名的Unix设计哲学：*Do One Thing, Do It Well.*

耦合

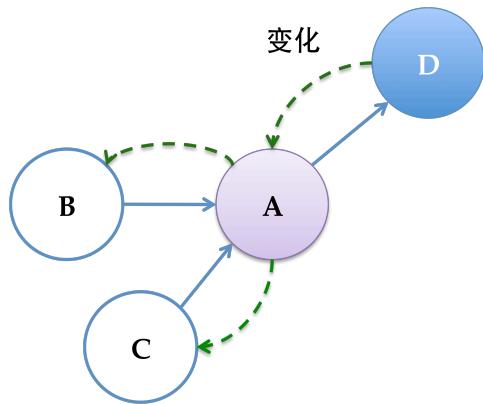
耦合，用来衡量单位之间的关联程度。单位之间的关联度越高，则它们之间的耦合度就越高。

耦合产生耦合的原因是依赖。对于任意两个代码元素，当一方依赖了另外一方，则两者之间就产生了耦合。所以耦合和依赖可以理解为同义词。

耦合会带来的问题是，对于耦合的双方，当被依赖的一方发生变化的时候，会导致依赖的一方也跟着变化。

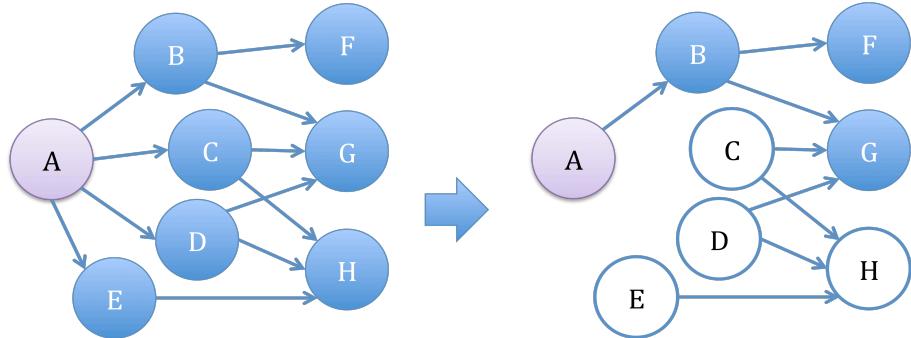


这就会导致可重用性的降低。例如，在下面的系统中，由于一个需求的来临，需要修改D，由于D的改变引起了A的变化，由于B和C依赖了A，然后A的变化又引起了B和C的变化。一个需求的发生会导致整个系统都发生变化的时候，那么原有系统对于这个新需求的可重用性就很差。



同样是这个例子，如果中间任何一个依赖关系得到减弱，那么D变化的时候，引起整个系统变化的概率就会降低，那么系统的“可重用性”也就得到增强。

另外，高耦合会降低系统重用性的原因在于，一个模块直接或间接依赖的其它代码元素越多，为了重用它，需要引入的元素也就越多，而这些元素会进一步的依赖其它元素，而那些元素中的很多东西可能是你并不需要的，但你必须为之付出更多的代价，比如，为之建立起相关的配置，环境等，或者引入一大批你根本不需要的东西。我们应该都经历过为了使用一个库的其中一个很小的功能，而不得不把整个库都引入的痛苦经历。如果，把依赖关系降低，就有可能得到一个更加轻量级的，么有冗余的，没有更复杂环境配置的，也就是更容易重用的东西。如下图所示：



另外，一个系统的耦合越强，那么系统的可理解性也就越差。比如，一个函数直接引用了另外一个模块的全局变量，你必须了解，都有谁可能会修改这个全局变量，其它模块都在以什么样的方式使用这个全局变量，当前的函数行为，以及它所可能导致的结果才能够被理解。

内聚和耦合的关系

为了理解“内聚”和“耦合”的关系，我们来看一个例子。

如图1所示，一个城市分为两个行政区A和B，中间由一条河隔开，河上有一座桥，方便于两个行政区之间的通行。

每个行政区都有一个居住区和一个公司。每个居住区里，都有部分居民在公司A上班，另外一部分居民在公司B上班。

所以，每天早晨和傍晚，居住在行政区A但在公司B上班的人，以及居住在行政区B但在公司A上班的人，都要跨越行政区区上班。朝九晚五的时候，两区之间的桥上，交通会特别拥挤。

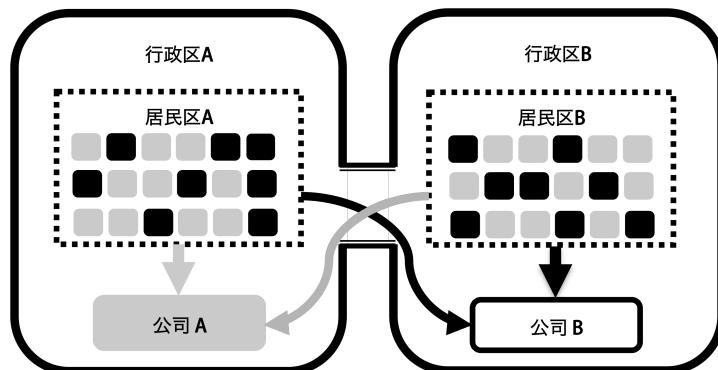


图1-原有系统

现在，我们将所有在A公司工作的居民安排到行政区A居住，同样的，将所有在B公司工作的居民安排到行政区B居住。然后，两区之间的交通状况就大为改善。如图2所示。

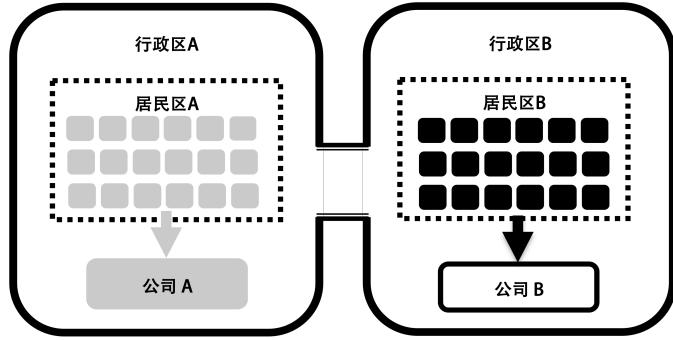


图2一调整后系统

在这个例子中，我们并没有改变任何本质性的事情。所有的人都还在原来的公司上班，没有人失业。没有人流离失所，只是改变了居住地。

但，仅仅由于居民居住区域的改变，两个行政区的依赖关系就大为减弱。事实上，对于这个理想模型，两个行政区之间已经没有任何关系，它们之间桥梁完全可以拆除。

从这个例子可以看出，通过将关联程度更高的元素控制在一个单位内部，就可以达到降低单位间关联的目的。

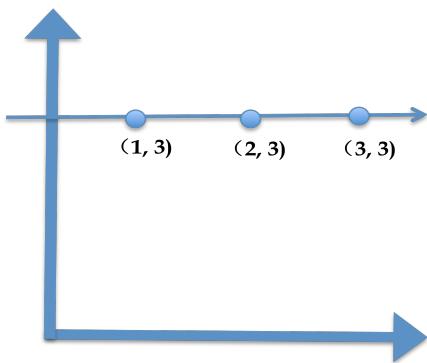
所以，内聚和耦合是紧密关联的两个概念：提高内聚度，往往会降低耦合。反之，降低耦合的过程，往往会提高内聚性。

正交

一个低内聚的系统，通常是高耦合的系统。高度耦合会造成糟糕的可重用性，一块代码的变更很容易会导致大面积的修改。

而我们希望的情况恰恰相反，即一块儿代码的修改不会引起其它代码的修改。这样的状态被称做“正交”（Orthogonal）。

正交来自于一个数学概念：两个向量的内积为零，则认为它们是“正交”的。简单的说，就是两个向量是垂直的。在一个正交系统里，沿着一个方向的变化，其另外一个方向的值不会发生变化。



一个系统的“高内聚，低耦合”的程度越强，则其“正交性”越强，其“可重用性”亦越好。所以，我们也可以用“正交性”来衡量一个系统的“可重用性”。

正交策略

为了达到更好的“正交性”，我们可以采取如下策略：

- 消除重复
- 分离关注点
- 缩小依赖范围
- 向着稳定的方向依赖

结论

策略1：消除重复

“*Duplication is root of all evil*”
——Anonymous

Larry Wall曾经在《Programming Perl》第二版里提到“懒惰，骄傲，缺乏耐心”是“程序员的三大美德”（Three Virtues of a Programmer: Laziness, Impatience, Hubris）。

接着，他对“懒惰”进行解释：

(Laziness is) The quality that makes you go to great effort to reduce overall energy expenditure. It makes you write laborsaving programs that other people will find useful, and document what you wrote so you don't have to answer so many questions about it.

尽管并没有直接提到重复，但事实上，他的每一句话都暗示着对于重复的厌恶。

什么是重复

代码是对知识的描述。“重复”则是不同的代码元素对同一知识进行了多次描述，无论它们的描述方式是否一致。

比如，某个系统存在一个约束：“允许最多1000个连接”。在同一系统中存在如下三个定义：

```
=====
.....
const unsigned int max_num_of_allowed_connections = 1000;

.....
#define MAX_ALLOWED_CONNECTIONS ((unsigned int)1000)

.....
unsigned int get_max_num_of_allowed_connections()
{
    return 1000;
}

=====
```

这三个不同的代码单元的形式完全不同，但它们却都是对同一知识的定义。所以，这就是一种重复。

有些重复并非那么明显：

```
=====
.....
strcpy(buf, packet->src_address);
buf += strlen(packet->src_address) + 1;

strcpy(buf, packet->dest_address);
buf += strlen(packet->dest_address) + 1;

memcpy(buf, &packet->user_type, sizeof(packet->user_type));
buf += sizeof(packet->user_type);
=====
```

事实上，这段代码的三个段落是重复的。每个段落都间接描述了同一个算法：缓冲区拷贝方法。如果把代码重构一下，事实就会更加明显：

```
=====
.....
size_t size;
void* p;

p = (void*)packet->src_address;
size = strlen(packet->src_address) + 1;
memcpy(buf, p, size);
buf += size;

p = (void*)packet->dest_address;
size = strlen(packet->dest_address) + 1;
memcpy(buf, p, size);
buf += size;

p = (void*)&packet->user_type;
size = sizeof(packet->user_type);
memcpy(buf, p, size);
buf += size;
=====
```

有很多办法可以消除这个重复。一个最初级的方法是定义宏。

```
=====
#define APPEND_TO_BUF(content, size) do { \
    memcpy(buf, (void*)content, size); \
    buf += size; \
} while(0)

///////////////////////////////
APPEND_TO_BUF(packet->src_address, strlen(packet->src_address)+1);
APPEND_TO_BUF(packet->src_address, strlen(packet->src_address)+1);
APPEND_TO_BUF(packet->user_type,   sizeof(packet->user_type));
=====
```

然后我们就能发现，代码中依然有重复，每个调用中的两个参数，都使用了同一表达式。这个重复说明两个参数都依赖了知识：要拷贝的数据源。

然后我们进一步消除重复：

```
=====
#define APPEND_TO_BUF(content, size) do { \
    memcpy(buf, (void*)content, size); \
    buf += size; \
} while(0)

#define APPEND_STR_TO_BUF(str) APPEND_TO_BUF((str), strlen(str) + 1)
#define APPEND_DATA_TO_BUF(data) APPEND_TO_BUF((data), sizeof(data))

///////////////////////////////
APPEND_STR_TO_BUF(packet->src_address);
APPEND_STR_TO_BUF(packet->src_address);
APPEND_DATA_TO_BUF(packet->user_type);
=====
```

重复的邪恶性

令人生厌

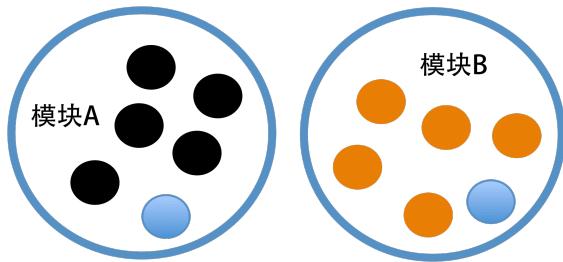
重复，首先意味着：同样一件事情，需要被反复的，机械的执行；这无疑另人生厌。

冗余

增加了代码量，从而增加了不必要的理解成本和维护成本。

低内聚

假设一个系统包含两个模块，每个模块负责完成一个功能。两个模块完全独立，相互之间没有任何关系。但两个模块都需要某个子功能，而这个子功能在两个模块中分别单独进行了实现。

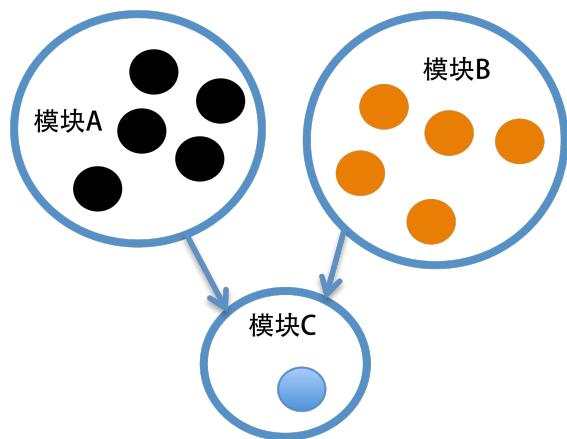


根据“高内聚”原则，关联紧密的事物应该被放在一起。

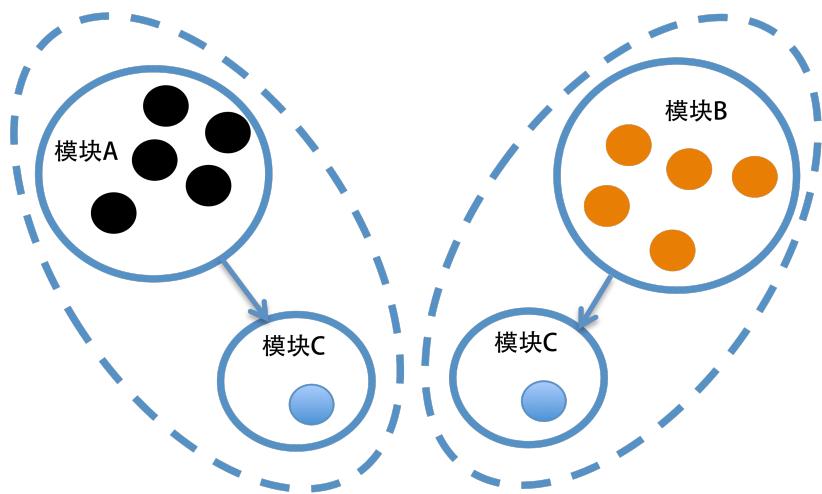
在此例中，两个重复的实现由于功能完全重合，所以它们之间的关联紧密程度是最高的；但现在它们被放置在同一系统的不同模块内，这就违背了“高内聚”原则。

注意，此例子中，两个模块的“内聚度”都非常高，但重复子功能的“内聚度”却是低的。它妨害了重复子功能的可重用性，也会导致对此子功能维护工作量的增加。

如果我们把这个子功能放在一个独立的模块内，则此子功能的内聚度得到了提高。

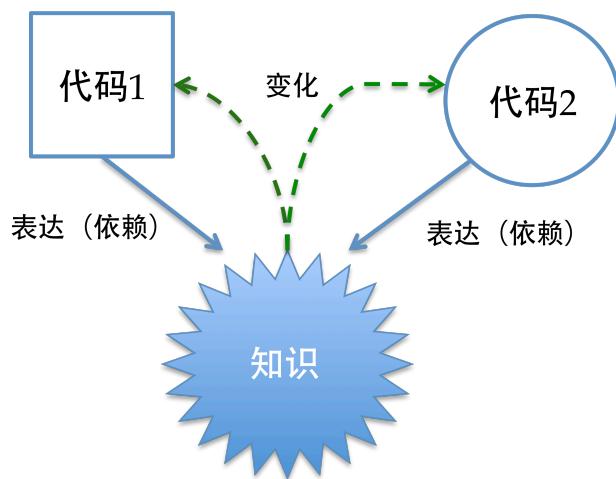


而两个模块的内聚度似乎看起来降低了一些，但这并没有改变原有功能的重用粒度。因为，你仍然可以把模块A和模块C放在一起，以重用功能A，把模块B和模块C放在一起，以重用功能B。



高耦合

由于一个代码单元定义或描述了某种知识，那么这个代码单元就对这个知识产生了耦合。如果只存在一个定义，则系统对这个知识只存在一个耦合点，N次定义则对同一知识产生了N个耦合点。当这个知识发生变化的时候，则必须在N个地方同步修改。所以，重复会导致更高的耦合。



掩盖问题

表面的重复，往往意味着背后存在的共性概念没有得到发掘。共性和差异的充分发掘，很有可能会得到更为合理的概念划分，从而得到更加合理的设计。

重复类型

重复有很多类型，比如结构型重复，逻辑型重复等。我们以逻辑型重复为例。在C++语言里，逻辑有关的最小可重用单位是函数（或许你认为是宏，但一般而言逻辑型的宏都可以转化为函数）。不同函数的“重用范围”不一样，比如模块内函数的“重用范围”为模块，而全局自由函数的“重用范围”为整个系统。

完全重复

假定现在有一个函数F1，可以完成某个子功能S。随后，在其“重用范围”内，有另外一个需求实现也需要子功能S，如果开发者并不知道它的存在，就需要重写另外一个函数F2。或者开发者即便知道F1的存在，但由于某种无法说明的原因，仍然通过copy-paste得到了函数F2。这就造成了“重复”，并且由于F1和F2都实现了子功能S，所以两者属于功能型“完全重复”。

对于完全重复的事情，没什么可说的。直接删除其中一个，保留另外一个即可。

但大多数情况下，重复都是以“部分重复”的形态出现的。两个“部分重复”的函数，其重复形式可以归纳为三种：参数型重复，调用型重复和回调型重复。

参数型重复

参数型重复是指，两个函数的算法相同，只是处理的数据不同。

```
void* f1(void* buf, const Packet* packet)
{
    strcpy(buf, packet->src_address);
    return (void*) ((char*)buf + strlen(packet->src_address) + 1);
}

void* f2(void* buf, const Packet* packet)
{
    strcpy(buf, packet->dest_address);
    return (void*) ((char*)buf + strlen(packet->dest_address) + 1);
}
```

消除这样的重复，只需要差异的数据“参数化”。

```
void* appendStrToBuf(void* buf, const char* str)
{
    strcpy(buf, str);
    return (void*) ((char*)buf + strlen(str) + 1);
}

void* f1(void* buf, const Packet* packet)
{
    return appendStrToBuf(buf, packet->src_address);
}

void* f2(void* buf, const Packet* packet)
{
    return appendStrToBuf(buf, packet->dest_address);
}
```

调用型重复

如果两个函数的“重复部分”完全相同，可以将重复的部分提取为函数F，然后原函数各自对F直接进行调用。

```
void* f1(void* buf, const Packet* packet)
{
}

void* f2(void* buf, const Packet* packet)
{
}
```

经过转化：

```
void F()
{
}

void* f1(void* buf, const Packet* packet)
{
    F();
}

void* f2(void* buf, const Packet* packet)
{
    F();
}
```

回调型重复

如果两个函数的“重复部分”完全相同，可以将重复的部分提取为函数F，将差异的部分形成原型相同的两个函数s1和s2，然后通过F分别对s1和s2进行调用。

```
void* f1(void* buf, const Packet* packet)
{
}

void* f2(void* buf, const Packet* packet)
{
}
```

经过转化：

```
void s1()
{
}

void s2()
{
}

void F(void (*s)())
{
    s();
}

void f1()
{
    F(s1);
}

void f2()
{
    F(s2);
}
```

}

回调型重复和参数型重复的唯一差别是，一个参数化的是数据，一个参数化的是行为。两者在某个层面可以抽象为一种概念。但为了更具指导意义，我们仍然认为这是两种类型。

产生原因

很多因素都会导致重复，下面列出一些常见原因：

低成本

编辑器“拷贝粘贴”的便捷性，让制造重复的成本太低。人天生是厌恶重复的，但制造重复的过程是如此容易，让其显得不再那么令人讨厌。

对于变化的恐惧

消除重复的过程，往往意味着对于原有代码的修改，以提高其可重用性。但是，基于对破坏原有功能的担心，“另起炉灶”往往是更加“理智”的选择。

尤其是当原有代码的可维护状况越糟糕，重复就越容易产生。

不易识别

重复的模式，完全被混乱的代码所掩盖。想理解它已经很困难，更不用说去识别重复。造成开发人员“有心杀敌，无力回天”。

技能上的不足

即便重复明确的摆在眼前，开发人员技能上的欠缺，比如对于语言特性的了解不够，对于消除重复的手段知之甚少，导致重复“合理”的产生，“合理”的存在。

庞大的代码库

当开发人员开发一个功能时，即便类似的问题曾经被解决过，但由于代码库过于庞大，开发人员很难知道它的存在，只好独立的重新解决解决，然后造就更加庞大的代码库。

知识共享的不足

一个大型团队往往会分成多个子团队，这些团队往往按照模块进行划分。团队之间由于模块的隔离，根本不存在内在需要，在团队之间分享信息和共享代码。事实上，一些公司基于“信息安全”的考虑，团队之间根本没有权限查看对方的代码。

即便一个团队内部，每个成员也有明确的职责划分，不同的功能长期由不同的团队成员进行维护，大家各自为战，也会造成轮子不断被重新发明。

价值导向

是的，重复让人生厌。但“消除重复”大多时候要比“制造重复”更困难。它需要个人更多的努力，更好的感觉，和更高的技能，更需要团队提供良好的机制。

当企业或团队并不鼓励对于“内部质量”的重视或改善时，一个开发人员即便发现了重复，也知道如何消除重复，但其为之付出的努力并不会得到认可，甚至会被管理者和同事认为是在浪费时间。尤其是当“消除重复”的过程引入任何问题时，反而会受到责难和惩罚。

这是最糟糕的造成重复的因素。在这样的价值导向下，所有其它会导致重复的因素都不会得到改善，只会进一步恶化，最终让一个系统迅速的充满重复。

认知差异

有些开发人员认为重复是合理的。在他们看来，重复的双方本来就是不同的事物，只是它们现在恰巧有相似的流程，恰巧使用相同的算法。重复的存在，让各自独立的变化，相互不干扰。这反而是一种优势。

这样的看法并非错误，在一些情况下确实如此。对于这类的问题，可以通过调整不同的抽象层次来发现内在的一致性，以此消除重复。如果从任何层面二者都是完全不同的事物，那重复就是合理的。

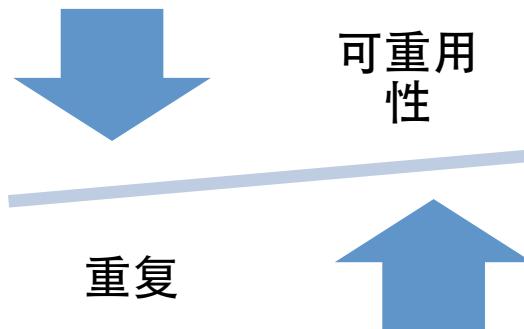
只是，识别重复的真伪，需要具备敏锐的洞察力。否则，将会导致很多内在的重复得以光明正大的保留。

重复和重用

假设，现在要实现一个新需求。原有系统中已经存于此需求所需要的某个子功能的代码，

- 如果它可以“重用”，则开发人员无须“重复”编写这部分代码。
- 否则，开发人员就需要对此子功能进行再次实现，从而造成“重复”。
- 但是，当重复产生之后，如果开发人员通过重构消除了“重复”，那么得到的那个子功能代码就已经被“重用”，这就提高了原有系统的“可重用性”。

所以，“消除重复”的过程就是一个提高系统“可重用性”的过程；反过来，提高系统的“可重用性”，也就降低了未来产生“重复”的可能性。两者是“此消彼长”的关系。



DRY

对于任何一项知识，在一个系统内，只应该存在一个明确而权威的表示。

DRY原则致力于消除开发过程中所有可能产生的重复。从注释，文档，到跨系统的开发。

结论

重复是邪恶的，与“高内聚，低耦合”背道而驰，从而导致可维护性相关的一系列问题。通过“消除重复”，可以提高系统的“可重用性”。

除了DRY之外，还有一些其它类似的原则。比如OAOO (Once And Only Once), Singular Responsibility Principle (不是Single Responsibility Principle) 等。

策略2：分离关注点

"I'm not a great programmer, I'm just a good programmer with great habits."

——Kent Beck

“消除重复”是提高可重用性的“被动型”策略。即在“重复”出现之后，在不影响“需求实现”的前提下，尽可能的消除它。而消除了“重复”，既让整个系统更加干净，又提高了系统内部元素的可重用性。

但问题的另外一面是，如果当前系统中没有“重复”，我们如何通过“主动出击”来提高系统的可重用性，在早期就让系统更加干净，更易于重用？

粒度与重用

一个不好充分证明却显而易见的事实是：粒度越小的事物越容易重用。

到任何一个大型遗留系统里，都会看到这个现象：你很难看到两个完全重复的函数，但部分重复的例子却比比皆是。而那些部分重复的代码一旦提取为更小粒度的函数，它们就已经具备了更好的“可重用性”。至少它们已经被“重用”，不是吗？

而硬件中的例子是，“二极管”要比“处理器”更容易重用，而“处理器”要比“计算机”更容易重用。

而问题的另外一面是，尽管大粒度的事物更难以重用，但一旦被重用，其产生的价值也就更大。因为你重用了更多的功能，节省了更多的劳动。这也正是为何企业往往热衷于平台重用，架构重用，甚至产品重用（解决方案供应商借此赚取了大量的利润）。

而小粒度的重用却经常被轻视。对于管理层而言，它看不见摸不到；而开发人员，则受限于技能、经验的缺乏，或开发进度和价值导向的影响。

就个体而言，一个小粒度的重用价值确实不如一个大粒度重用。但它庞大的数量，和更高的重用频度，决定了它所累积的价值不容忽视。

功能分解

尽管系统存在着各种各样的分割方法，但可以肯定的是，任何一个非玩具系统都不会只有一个函数。无论你采用自顶向下，还是自低向上的设计方式，一个系统的最终形态总是由多个层次，多种粒度的软件单位逐级构成。

比如，一个网络系统中有不同网元，每个网元内部有多个单板，每个单板上有多个任务，每个任务上有多个组件，每个组件里有多个类，每个类里有多个函数。

同时，一个设计良好的系统，必然是符合“高内聚，低耦合”原则的。谈到高内聚，它包含两个方面的约束：

1. 关联紧密的事物应该被放到一起；
2. 只有关联紧密的事物才应该被放到一起。

所以，“关联紧密度”也存在着不同粒度。从大粒度的角度看，首先把一个复杂的问题按照关联的紧密程度进行分割，然后我们得到了一个个网元，每个网元的复杂度相对于最初的复杂性已经得到降低，而可重用性却得到提高。然后我们在一个网元内部进一步按照内部元素的“关联紧密程度”进行划分，我们就得到了一个个单板...一步步分割下去，直到我们得到最原子层面的元素——函数。

每次分割出来的元素，无论你是一个网元，单板，组件，类，或函数，都应该在自己所处的层次符合“高内聚”原则。

由于“只有关联紧密的事物才应该被放到一起”，所以，每个元素所应该包含的下层元素数量都不会很多。即便你是一个网元，它所包含的细节数量可能极其庞大，但其所包含的单板数量，单板职责，以及单板之间的关系要简单的多。

有些软件并没有这么多层次，它们往往直接把一个软件分为多个类，然后通过类与类之间的协作来完成整个软件的功能，而类的数量可能非常多。如果这些类之间的关系也是按照“高内聚，低耦合”的原则来设计的，那么不同的类之间的紧密关联程度也一定是不同的，它们从概念上仍然可以被分离为不同的单元。当使用Java的时候，“包”可以作为这一级别单元的管理工具；而对于C++，尽管语言并没有提供明确的机制，但事实上仍然有办法来进行类似的分组管理。

关注点

为什么不同事物会呈现不同的“关联紧密的程度”？不难得出的结论是“功能”或“职责”。“功能”的描述也分为很多层级，从高层的角度来看，一个网元的功能是可以用不多的文字来描述的。一旦深入细节，一个高层的“功能”就需要分为更多的“子功能”；而每个“子功能”又可进一步细分为更小粒度的“子功能”，越往上层，“功能”的描述越贴近于“问题域”，越往下层，“功能”的描述越接近“实现域”。

所以，所谓“关注点”，从行为的角度看，是一个个“功能”，即在某个层面，你可以清晰的描述它在做一件什么具体的事情。只要能够描述出来，那就是一个“关注点”。而从概念的角度看，“关注点”是则是一个承担某种职责，代表某种角色，提供某类服务的实体。

这两种不同的角度，区别并非泾渭分明。

例子

为了更加直观的理解“关注点”，以及如何“分离不同的关注点”，我们来看一个需求：将所有的学生按照身高从低到高排序。

首先我们可以从简单的问题描述中得知：存在一个学生集合，其中，每个学生都有一个身高属性；需求是将一个集合中的所有学生按照他们的身高进行正向排序。

现在我们不借助任何外部库，通过冒泡排序算法直接在一个函数内进行了实现：

```
=====
struct Student
{
    char           name[MAX_NAME_LEN];
    unsigned int height;
};

void sort_students_by_height( Student students[]
                               , size_t num_of_students)
{
    for(size_t y=0; y < num_of_students-1; y++)
    {
        for(size_t x=1; x < num_of_students - y; x++)
        {
            if(students[x].height > students[x-1].height)
            {
                SWAP(students[x], students[x-1]);
            }
        }
    }
}
```

```
        }
    }
}
```

仅仅从需求实现的角度来看，我们干的相当不错。

为了发现其中的“关注点”，我们来臆测一下，如果随后出现另外一个需求：将所有老师按照年龄从小到大排序。

这个需求对我们不构成任何挑战。因为这个需求和上个需求几乎是一样的，差别仅仅在于：排序的对象是所有老师，排序依据是年龄。

于是我们快速的将需求实现为：

```
struct Teacher
{
    char           name[MAX_NAME_LEN];
    unsigned int   age;
};

void sort_teachers_by_age( Teacher teachers[]
                         , size_t num_of_teachers)
{
    for(size_t y=0; y < num_of_teachers-1; y++)
    {
        for(size_t x=1; x < num_of_teachers - y; x++)
        {
            if(teachers[x].age > teachers[x-1].age)
            {
                SWAP(teachers[x], teachers[x-1]);
            }
        }
    }
}
```

两份代码的重复给出了明确的指示，我们从实现中发现了三个关注点：

- 冒泡排序算法（重复的部分）
- 排序的对象（差异的部分）
- 排序对象的对比规则（if语句的条件部分）

所以我们需要把两者分离开来。首先是冒泡排序算法：

```

template <typename T>
void bulb_sort( T objects[]
                , size_t num_of_objects)
{
    for(size_t y=0; y < num_of_objects - 1; y++)
    {
        for(size_t x=1; x < num_of_objects - y; x++)
        {
            if(objects[x] > objects[x-1])
            {
                SWAP(objects[x], objects[x-1]);
            }
        }
    }
}
=====
```

我们将其实现为模版，让其与“排序的对象”解开耦合；将if语句的条件设为通过操作符“>”来对比对象的大小关系，让其与“排序对象的对比规则”解耦。“冒泡排序算法”已经由原来专门针对Students的实现，变为一个通用实现，其“可重用性”有了质的飞跃。

为了使用这个算法，我们需要让Student实现“>”操作符。如下：

```

struct Student
{
    bool operator>(const Student& another) const
    { return height > another.height; }

    private:
        char name[MAX_NAME_LEN];
        unsigned int height;
};
```

此时Student也具备了更好的可重用性。如果由于“性能”问题需要换一个排序算法，它只需要提供建立和“冒泡排序”函数一样的机制，Student则无须任何修改就可以使用新的排序算法。

```

template <typename T>
void qsort( T objects[]
            , size_t num_of_objects)
{
    // 快速排序的实现
```

```
.....  
    if(objects[i] > objects[i+1])  
    .....  
}
```

如果现在又出现一个这样的需求：对于所有学生，也可以按照年龄进行排序。我们的现有机制将受到破坏。

这是因为，现有的实现却将“排序对象”和“排序规则”两个关注点耦合到了一起。无论是Student的实现，还是排序算法的机制，都没有将这两个关注点分开。

现在，我们通过引入一个谓词GreaterThan，将这两个关注点进行分离。

```
template <typename T, typename GreaterThan>  
void bulb_sort( T objects[]  
                , size_t num_of_objects  
                , const GreaterThan& greater_than )  
{  
    for(size_t y=0; y < num_of_objects -1; y++)  
    {  
        for(size_t x=1; x < num_of_objects - y; x++)  
        {  
            if(greater_than(objects[x], objects[x-1]))  
            {  
                SWAP(objects[x], objects[x-1]);  
            }  
        }  
    }  
}
```

然后不同的“排序规则”各自实现自己的谓词GreaterThan。

```
struct Student  
{  
    char name[MAX_NAME_LEN];  
    unsigned int age;  
    unsigned int height;  
};  
  
bool student_older_than(const Student& lhs, const Student& rhs)  
{  
    return lhs.age > rhs.age;  
}
```

```
bool student_taller_than(const Student& lhs, const Student& rhs)
{
    return lhs.height > rhs.height;
}
```

事实上，大多数类型（包括整数，浮点数等）的“排序规则”都是基于操作符“`>`”。为了避免让这些类型各自重复的实现，我们可以给出一个默认谓词。

```
template <typename T>
struct GreaterThan
{
    bool operator()(const T& lhs, const T& rhs) const
    { return lhs > rhs; }
};

template <typename T>
void bulb_sort( T objects[]
                , size_t num_of_objects )
{
    bulb_sort(objects, num_of_objects, GreaterThan<T>());
}
```

到目前为止，我们所有的实现都基于这样的假设：可以进行排序的集合都必须以数组的方式存储。如果一个数据集合使用了其它数据结构，比如链表，那么为了使用现有的排序算法，则必须首先将集合内的数据存储到一个数组中。

此时，一个新的“关注点”就出现了：集合的存储形式。

为了分离这样的“关注点”，就必须引入其它的抽象。比如：容器概念，迭代器等等。这就是STL库现有的设计。这里就不再详述。

动机

“分离关注点”的动机很多，首先是为了“管理复杂度”。当把一个复杂的事情分为一个个的“关注点”时，每个关注点都更简单，由此而更容易理解。

其二，是为了“可重用性”，在一个大型网络系统的里，某个网元承担了某个职责后，其它网元就不再需要各自完成相应的职责，而是“重用”此网元的功能实现。至于“弹性”，“可扩展性”等等目的，都属于“可重用性”的范畴，只是它们更具体。

“实现需求”可能也会被认为“分离关注点”的重要动机。但事实上，用户并不关心你是否“分离了关注点”。它唯一关心的是“功能”和“成本”。即便在一个大型电信网络系统中，现在的网络结构和网元构成，在最初也只是基于“成本”和“功能”的考虑，工程师们通过设计而提供的一个“解决方案”而已。

手段

在“面向过程”的方法学里，“关注点”用“过程”或“函数”作为承载的手段。一个大的“关注点”被分离为一个个“函数”，而每个函数则承载了更小，更低层次的关注点。

在“面向对象”方法学里，“关注点”主要由“对象”来承载。而对象则是面向对象编程中的最小单位。对于一个承载更高层关注点的对象，需要通过进一步“分离”出通过“聚合”或者“组合”手段来分离出更底层的“关注点”。比如，飞机作为一个独立的高层对象，则需要进一步细分成机翼，机舱，滑轮等更小的“聚合对象”。

即便在“面向对象”领域，每个对象的方法，仍然需要分离关注点，尽管这些关注点可能体现为私有方法。但为了清晰的表达，以及在类内部进行“重用”，也需要进行分离。有些时候，这些以“成员函数”形式存在的关注点，很可能会转化成“对象”。

而作为混合了“面向过程”，“面向对象”，“范型”等范式的语言，用C++进行“分离关注点”也可以有多种手段。但如果，你主要在利用其“面向对象”特性，那就仍然以对象为主，除非某些情况下，“函数”是最简洁的方式。

还有一种方兴未艾的方法学叫做AOP（面向方面的编程），它试图把一个个关注点分离为一个个称为Aspect的描述。然后通过“编织”(Wave)的方式把不同的Aspect插程序的的cutpoint，来实现软件的功能。AOP有语言级的解决方案，也有框架级别的解决方案。

粒度

如果“关注点”被分离的足够好，那么大多数函数都不会超出10行（据统计，ThoughtWorks某个项目的代码，平均方法长度为1.7行）。

当然在一些C/C++语言项目中，由于没有更好的异常处理手段，会存在很多条件语句以应对异常。比如，

```
if (NULL == pointer)
{
    log("null pointer blah blah blah...");
    return NULL_POINTER_ERROR;
}
```

这会大大增加代码的数量。这个问题一直让我头痛，但除了使用宏来解决之外，一直都没有更加干净的手段（C/C++的AOP什么时候可以成熟，这个问题就会好很多）。

但事实上很多C/C++程序的问题并不在于这些异常处理，即便一段正常的逻辑也实现的超级复杂，通常包含混乱的分支逻辑，多层的嵌套，和几百甚至上千的代码行。出现这样的结果，通常是因为疏于对关注点的识别和分离。

我经常能够听到一些关于“函数调用成本”的借口。但如果我们提议：暂时忘掉性能问题，只是基于试验的目的来把一个函数重构的更加清晰，通常这些程序员也很少独立的把一个函数的不同关注点正确识别和分离出来。原因是长期缺乏这方面的意识，进而缺乏这方面的锻炼，对于软件的内聚性和耦合性反映迟钝。简单的说，没有形成良好的习惯。

必要性

从上面的例子可以看出，在重复出现后，“分离关注点”可以作为消除重复的一种手段。在重复出现前，“分离关注点”则可以防备一些未来的“重复”。

从逻辑上来看，我们并没有必要在重复出现之前就未雨绸缪。因为，只要重复出现，我们总是可以通过“消除重复”来分离出相应的关注点。而事前做，“重复”却可能永远不会到来，“重用”也就无从谈起。

这是一个不错的观点。但放在现实的情况下，“分离关注点”仍然有它的必要性。

因为，通过分离关注点，可以让原有软件单位的复杂性得到降低，更加容易理解和维护。而良好的“可理解性”是“识别重复”的前提，而“识别重复”是“消除重复”的前提。

事实上，对于很多保持良好习惯的程序员来说，很多情况下，“分离关注点”的动机并非是为了重用，而是为了降低复杂度，提高表现力。而在清晰的表达下，重复无处躲藏。

反过来，每次当我开始重构一个内部质量糟糕的遗留系统时，首先映入你眼帘的，是一大块混乱的，让人发狂的复杂逻辑，从中你看不到任何明显的模式。这种情况下，我总是先把代码梳理干净，尝试者分离出一个个关注点。在这个过程中，重复的模式就逐渐显现出来。

单一职责

“分离关注点”要分到什么程度？“单一职责”或许是解答这个问题的答案。

在Uncle Bob的大作《敏捷软件开发：原则，设计与模式》里，“单一职责原则”被定义为针对“类”设计的原则。但事实上它可以推广到软件设计的各个层面：

- 包（“共同封闭原则”就是包设计的“单一职责原则”）
- 头文件（降低物理耦合）
- 函数／方法
- 模块
- 组件

之所以要“单一职责”，是因为它完全符合“高内聚，低耦合”：让每个职责高度内聚，不同职责松散耦合，从而让各个职责尽可能独立的变化。这也正是为何Uncle Bob将“职责”定义为“变化的原因”。

策略3：缩小依赖范围

在我们将一个问题分离为多个“关注点”之后，由于每个关注点都是单一职责的，所以关注点之间必然以某种方式耦合在一起，才可以通过协作完成更大的职责。

而我们希望能够把耦合降到最低，这样能够让某个关注点上的变化带来的影响尽量小，使系统处于尽量好的正交性。

依赖点

所谓耦合，是指双方或多方通过某种方式产生了依赖关系。所以，“依赖”可以被理解为“耦合”的代名词。

当谈起一个模块依赖于另外一个模块时，并非意味着一个模块依赖了对方的所有功能或代码。这种依赖往往是局部的。尽管其依赖关系可能非常复杂和混乱，但如果仔细梳理，我们仍然能够识别出，在被依赖的模块里，究竟是什么元素，一旦发生了变化，就会导致依赖方的变化，那么这些元素就被称为“依赖点”。

比如，模块A调用了模块B的某个API，那么当这个API的（原型，事先条件，事后条件）发生变化的情况下，将会导致模块A的代码变化，那么此API的（原型，事先条件，事后条件）就是模块A对模块B的一个“依赖点”。

如果此API的参数列表或者返回值使用了某个类型T，如果这个类型属于模块B，那么模块A对模块B的依赖就增加了一个“依赖点”：类型T。如果类型T定义在模块A，那么A和B之间就产生了相互依赖，其中类型T是B对A的依赖点。

通俗的说，“依赖点”就是一个个知识，一个模块了解另外一个模块的知识越多，则它对另外一个模块的“依赖点”就越多。

减少依赖点

任意一个依赖点的变化都会导致依赖方的变化。所以两个模块之间存在的依赖点越多，则依赖方变化的概率就越高，这会导致糟糕的正交性。所以我们必须尽可能的减少依赖点的数量。

策略4：向着稳定的方向依赖

简单设计

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

—— C. A. R. Hoare

第二部分：练习一

需求一

问题描述

实现一个几何意义的矩形(Rectangle)，以像素为单位来表示宽和高，能够计算它的面积。

需求实现

既然要求要提供一个Rectangle的类，我们可以先构造第一个测试用例：

```
TEST(能够计算一个矩形的面积)
{
    Rectangle rect;

    unsigned int width = 3;
    unsigned int height = 4;

    ASSERT_EQ(12, rect.calcArea(width, height));
}
```

从这个测试用例所提供的UI来看，Rectangle提供了一个计算面积的方法calcArea，当用户需要计算一个矩形的面积的时候，则通过提供一个矩形的宽和高，Rectangle的calcArea方法就会帮助用户计算相关的面积。根据这样的UI，可以得出如下两种实现方式：

```
struct Rectangle
{
    unsigned int calcArea(unsigned int width, unsigned int height);
};
```

或者，

```
struct Rectangle
{
    static unsigned int
    calcArea(unsigned int width, unsigned int height);
};
```

很明显，在两种实现中，Rectangle都没有任何数据（属性）。既然自身没有数据，那么Rectangle存在的目的就是“提供算法”。

以“提供算法”为目的的类，如果以实体类（可被实例化的类）的形式出现，就被称为“算法类”；如果其成员全是“静态函数”，则被称为“工具类”。

在上述的实现中，前者是一个“算法类”，后者则是一个“工具类”。

刚刚从C语言世界转过来的工程师，很容易得出这样的设计。C语言工程师由于长期浸染“程序=数据结构+算法”的思路，非常习惯于“将数据和算法进行分离”的设计方式。

这样的做法并非全无道理。对于某些通用算法（比如排序算法）而言，将算法和可计算数据进行解耦反而可以提高可重用性。

无论如何，我们先假设这个设计是合理的。但根据需求，用户不仅仅需要计算一个矩形的长度和面积，还需要有个地方可以存储矩形的宽和高，那么这些数据应该存在哪里？

这并不能难倒C语言工程师。既然矩形包含两个数据，那自然应该建一个结构体。

```
///////////
struct RectangleData
{
    unsigned int width;
    unsigned int height;
};

///////////
struct Rectangle
{
    static unsigned int
        calcArea(unsigned int width, unsigned int height);
};
```

这样的命名让人不容易理解，即便是C语言工程师也知道保存数据的那个结构体更应该改叫做Rectangle，而下面的类既然是一个工具类，那莫不如干脆就叫RectangleUtils。

```
///////////
struct Rectangle
{
    unsigned int width;
    unsigned int height;
};

///////////
struct RectangleUtils
```

```

{
    static unsigned int
    calcArea(unsigned int width, unsigned int height);
};

```

然后用例就修改为：

```

TEST(能够计算一个矩形的面积)
{
    Rectangle rect;

    rect.width = 3;
    rect.height = 4;

    ASSERT_EQ(12, RectangleUtils::calcArea(rect.width, rect.height));
}

```

消除重复

然后我们发现，每次调用函数RectangleUtils::calcArea都需要让客户提供两个参数，显得很啰嗦。如果把它的原型改为：

```
static unsigned int calcArea(Rectangle* rect)
```

将会消除用户在这个问题上的重复。

但机敏的你马上指出，这样会降低RectangleUtils::calcArea的可重用性，因为如果Rectangle改了名字，或Rectangle的属性改了名字，那么这个函数要被迫修改。

在佩服你敏锐的洞察力之余，我想到了一个两全其美的方案：架一个中间层。

```

///////////
struct Rectangle
{
    unsigned int width;
    unsigned int height;
};

///////////
struct RectangleCaculator
{
    static unsigned int calcArea(Rectangle* rect)
    {
        return RectangleUtils::calcArea(rect->width, rect->height);
    }
};

///////////
struct RectangleUtils
{
    static unsigned int

```

```
    calcArea(unsigned int width, unsigned int height);  
};
```

消除冗余

但RectangleCaculator有些多余，因为，如果我们把calcArea函数移到Rectangle里，并不会影响Rectangle原有的使用方式。并且RectangleCaculator本来就是专门为Rectangle而创建的，既然它们关联如此紧密，莫不如干脆合并。这也是“高内聚”所倡导的。

```
//////////  
struct Rectangle  
{  
    unsigned int width;  
    unsigned int height;  
  
    static unsigned int calcArea(Rectangle* rect)  
    {  
        return RectangleUtils::calcArea(rect->width, rect->height);  
    }  
};  
  
//////////  
struct RectangleUtils  
{  
    static unsigned int  
    calcArea(unsigned int width, unsigned int height);  
};
```

但根据C++语义模型，非静态成员函数存在一个潜在的this指针，如果我们把Rectangle::calcArea改为非静态的，那么它的唯一参数也可以省去了：

```
//////////  
struct Rectangle  
{  
    unsigned int width;  
    unsigned int height;  
  
    unsigned int calcArea(/* Rectangle* this */)  
    {  
        return RectangleUtils::calcArea(this->width, this->height);  
    }  
};  
  
//////////  
struct RectangleUtils  
{  
    static unsigned int  
    calcArea(unsigned int width, unsigned int height);  
};
```

而用例则修改为：

```
TEST(能够计算一个矩形的面积)
{
    Rectangle rect;

    rect.width = 3;
    rect.height = 4;

    ASSERT_EQ(12, rect.calcArea());
}
```

现在我们注意到用户已经完全不再关注RectangleUtils。是的，我们承认它可以更加通用，但，既然用户已经不再直接使用它，那它的通用性也就暂时没有了价值。等用户直接需要这种通用性，或者等重复出现，我们需要借助它来消除重复的时候再说。

```
struct Rectangle
{
    unsigned int width;
    unsigned int height;

    unsigned int calcArea()
    { return width * height; }
};
```

另外，既然函数calcArea()不应该修改系统状态，那么它就是一个查询函数，我们通过const把这个关系明确化。

```
struct Rectangle
{
    unsigned int width;
    unsigned int height;

    unsigned int calcArea() const
    { return width * height; }
};
```

查询函数

正如它的名字所暗示的，一个“查询函数”的计算仅仅会“查询”对象的当前状态，而不会修改它。而会修改对象状态的函数则被称做“命令函数”。

但“查询函数”这个名字也会造成一些误解，容易被理解为Getter函数。但事实上，并非任何查询函数都包含Getter语义，比如类的一个Visitor。所以，“常态函数”或许是个更贴切的名字。

之所以要提出这个概念，是因为“查询函数”没有副作用。无论用户对一个“查询函数”进行多少次调用，都不会对对象的状态产生影响。

这是一种用户和类之间的契约关系。当你复用某些框架，或者重载某些操作符时，相关的函数必须遵从这种契约；当对象以const参数传递给一个函数的时候，对方只可能调用你的“查询函数”。另外，在这种契约关系的保证下，客户可以采用更加灵活的编码方式。比如，重构手法“Replace Temp With Query”。

C++提供了语言机制来明确这种契约关系，即在一个成员函数的后面使用关键字const来修饰，比如：

```
class Foo
{
    .....
    int getBar() const;
    .....
};
```

但需要注意的是，既然这是一种契约关系，就必须能够经受住考验。否则，一旦这种契约关系被打破，客户的假设就会全部失效，从而可能引起系统缺陷。只有当一个成员函数从概念层面应该是“查询函数”时，才应该建立这种契约关系。一个函数的实现手段没有修改对象的状态，并不能确定它的“常态”性。否则，一旦实现手段发生变化，就会打破这种契约关系。

所谓从概念层面来判断一个函数是否是“查询函数”，名字是个重要的指示。首先，带有询问语气的名字应该属于“查询函数”，比如 isSuccess, hasChild；另外，以get, calculate, (convert)To, find等为前缀的名字也大体属于“查询函数”，比如，getValue, toString等。

总的原则是，在不考虑实现的情况下，这个操作从概念逻辑上是否应该改修改对象的状态，如果答案是Yes，就将其定位“查询函数”。如果最终在实现中发现你必须修改对象的状态，再反过来问一下，这个函数的概念语义是否错了，或者函数实现是否承担了过多的职责。

最后，需要特别指出的是，在C++语言中，“查询函数”仅仅能保证类实例状态的不变性。如果你在“查询函数”里修改了全局变量，或者类的静态变量，编译器无法给出任何警告，但每次对“查询函数”的调用都在修改系统状态。所以，不要在一个“查询函数”里进行任何可能修改系统状态的操作。

缩小依赖范围

封装字段

相信你应该听到过这样的建议：在面向对象编程时，应该将所有的属性都变为私有的，然后通过Setter方法来设置它们的值。因为函数比数据更有弹性，当使用函数时，你可以加入更多的逻辑，比如值的合法性检查。而直接给成员变量赋值则过于刚性，难以修改。

```
struct Rectangle
{
    void setWidth(unsigned int width)
    { m_width = width; }

    void setHeight (unsigned int height)
    { m_height = height; }

    unsigned int calcArea() const
    { return m_width * m_height; }

private:
    unsigned int m_width;
    unsigned int m_height;
};
```

而用例则变为：

```
TEST(能够计算一个矩形的面积)
{
    Rectangle rect;

    rect.setWidth(3);
    rect.setHeight(4);

    ASSERT_EQ(12, rect.calcArea());
}
```

但从用例中我们可以看出，为了构建一个合法的Rectangle，用户总是要调用两个Setter，而类是有构造函数的，为什么不通过构造来设置宽和高？这样至少用户就可以少写一些代码。

```
TEST(能够计算一个矩形的面积)
{
    unsigned int width = 3;
    unsigned int height = 4;

    Rectangle rect(width, height);

    ASSERT_EQ(12, rect.calcArea());
}

struct Rectangle
{
    Rectangle(unsigned int width, unsigned int height)
        : m_width(width),
        , m_height(height)
    {}

    void setWidth(unsigned int width)
    { m_width = width; }

    void setHeight (unsigned int height)
    { m_height = height; }

    unsigned int calcArea() const
    { return m_width * m_height; }

private:
    unsigned int m_width;
    unsigned int m_height;
};
```

这样，用户的使用方法就简洁了许多：

```
TEST(能够计算一个矩形的面积)
{
    Rectangle rect(3, 4);

    ASSERT_EQ(12, rect.calcArea());
}
```

从用例中，我们看不到用户使用两个Setter方法。但你的直觉告诉你，它们仍然应该存在，因为它们会让用户具备更大的灵活性：用户可以随时重新设置矩形的宽或高。

我完全同意你的判断。但灵活性不是免费的。它们的存在，增加了代码的数量，从而增加了维护的成本。而从现有的需求中，我们看不到任何对于这种灵活性的需求。所以我们可以先将其删除，等需要的时候再提供。

```
struct Rectangle
{
    Rectangle(unsigned int width, unsigned int height)
        : m_width(width),
          m_height(height)
    {}

    unsigned int calcArea() const
    { return m_width * m_height; }

private:
    unsigned int m_width;
    unsigned int m_height;
};
```

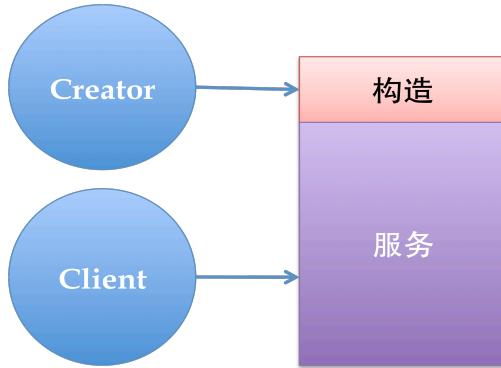
构造和SETTER

在本例中，构造函数和Setter方法，看起来都是为了给成员变量赋值。它们似乎是可以相互替代的赋值手段，其差异不过是，构造看起来要比Setter更简洁。

这是一种完全错误的看法。两者从任何一个角度看，都是完全不同的两类事物。

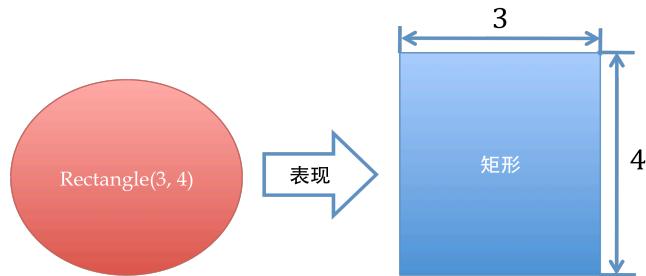
首先，构造函数面向的用户是Creator，而其它成员函数的用户则是类服务的需要者，即类真正的Clients。

尽管可以由单个类同时承担，但Creator和Clients本质上是两种完全不同的角色。在需要分离关注点的时候，Creator和Client经常由不同的类来扮演。



其次，当一个类实例被构造完成时，就应该能够表现它所表现的对象。比如，当一个下面的表达式执行结束后，实例rect就应该能够代表一个宽为3，高为4的矩形：

```
Rectangle rect(3, 4);
```

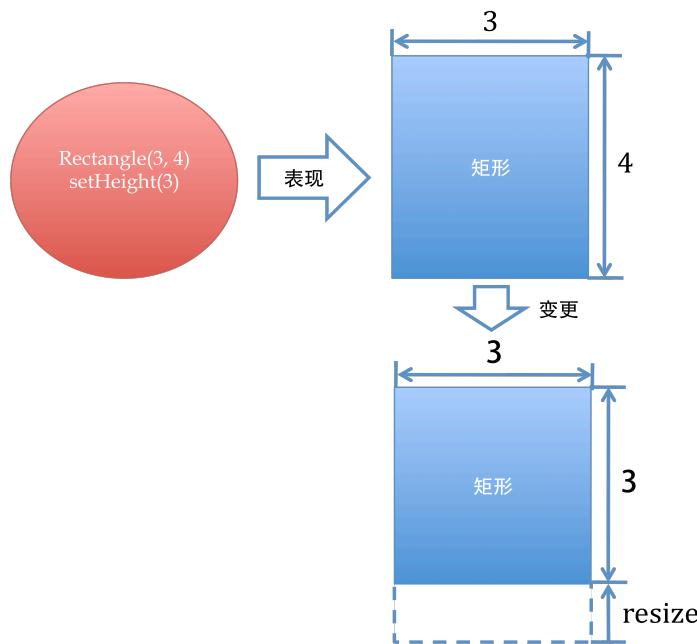


而下面的表达式执行结束后，rect其实什么都不是。它无法履行它的职责，即无法表现一个合法的矩形。

```
Rectangle rect;
```

所以，构造函数的目的，是为了创造出一个合法的，能够履行职责的，能够代表它要表现的概念实例。

而Setter函数，则是提供给Client的一个状态变更接口。它的存在，完全取决于客户是否需要。从这个意义上，Setter和任何其它服务函数没有任何差别。



所以，构造和Setter并非二选一的关系，而是为不同目的存在，完全没有任何关系的两套接口。

以Rectangle为例，通过构造函数来得到合法的Rectangle实例是必须的。如果没有任何状态变更的需要，那么在构造结束后，我们就得到了一个“常量”。

如果，随后Clients要求一个状态变更接口，无论是否是Setter，只要可能改变实例的状态，我们构造出的就是一个“变量”。

消除inline

我们发现，现在所有的实现都在头文件里，这会带来不必要的“物理耦合”。所以，我们将其移入源代码文件。

```
struct Rectangle
{
    Rectangle(unsigned int width, unsigned int height);

    unsigned int calcArea() const;

private:
    unsigned int m_width;
    unsigned int m_height;
};

Rectangle::Rectangle(unsigned int width, unsigned int height)
    : m_width(width),
      m_height(height)
{ }

unsigned int Rectangle::calcArea() const
{ return m_width * m_height; }
```

C++ 封装手段

封装的目的是为了“信息隐藏”，而“信息隐藏”的目的是为了降低系统的耦合度，局部化影响。

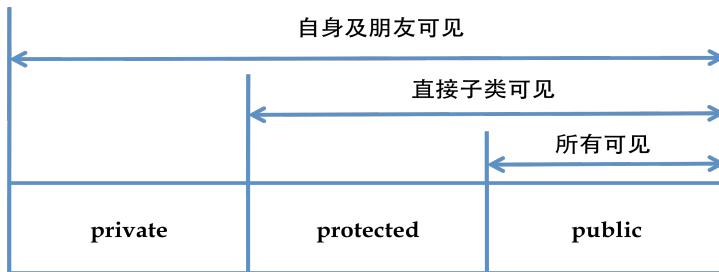
尽管有着相同的目的，但不同语言，即便都是面向对象语言，封装的手段却不尽相同。

具体到C++, 当考虑封装的时候, 就必须结合语法和编译过程来综合考虑这个问题。

首先从语法角度, “类”成员的可访问性 (Accessibility) 是进行封装的一种手段。在类中你可以定义和声明:

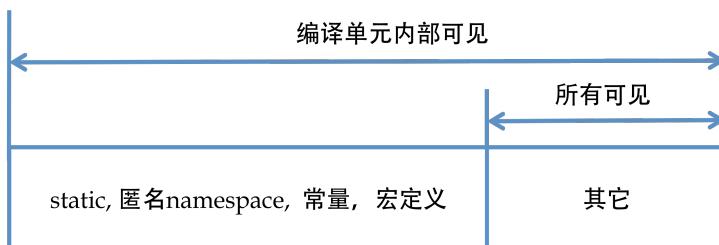
1. 类型 (类, 枚举, 别名)
2. 成员 (实例成员和类成员: 变量及函数)
3. 友元 (类, 函数)

除了友元关系之外, 其它都可以通过可访问性控制来进行不同范围的封装。当你使用private来修饰一个类型或成员时, 只有在当前类范围内可访问, 而protected类型或成员, 则可以由直接子类访问。



这是只要对C++略有了解, 就应该了解的封装手段。但在现实项目中, 对于这种封装手段的运用, 却差强人意。但这属于另外一个话题。

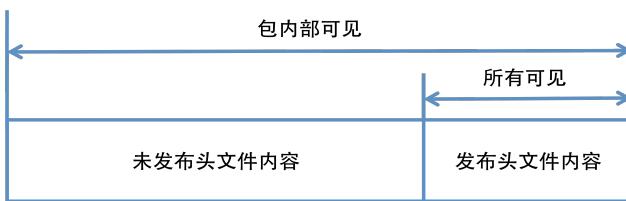
第二种封装手段是“编译单元”级别的, 即封装所定义符号的链接可见性。具体手段是在源文件内部, 通过static, 匿名namespace等手段来屏蔽一个“编译单元”内部所定义的符号, 对于其它编译单元在链接时的可见性。这种封装控制是仍然由语言的语法来保障。



相对于第一种手段, 这种手段在现实项目中被了解和应用的程度要低一些, 它的重要程度也被不正常的低估。

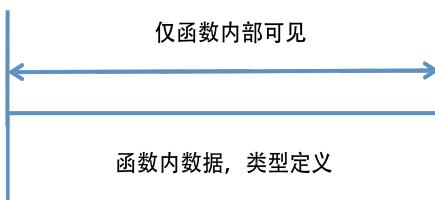
第三种封装手段是“头文件隐藏”，它利用的是C++预处理和编译过程的可见性。如果你想使用一个在其它模块定义的宏，类型，类，就必须包含相应模块的头文件。如果对方没有提供相应的头文件，你就没有办法使用。（你可以通过前导声明来声明一个类，但如果不知道它的接口声明，则无法对它的成员进行访问）。

C++没有包(package)的概念。但程序员却可以利用上述的第三种方式来定义逻辑上的包。对于包内可见，包外不可见的宏定义，类型定义等等，均可以通过“头文件隐藏”，即“包内头文件”的方式进行封装。这对应了Java的internal关键字。



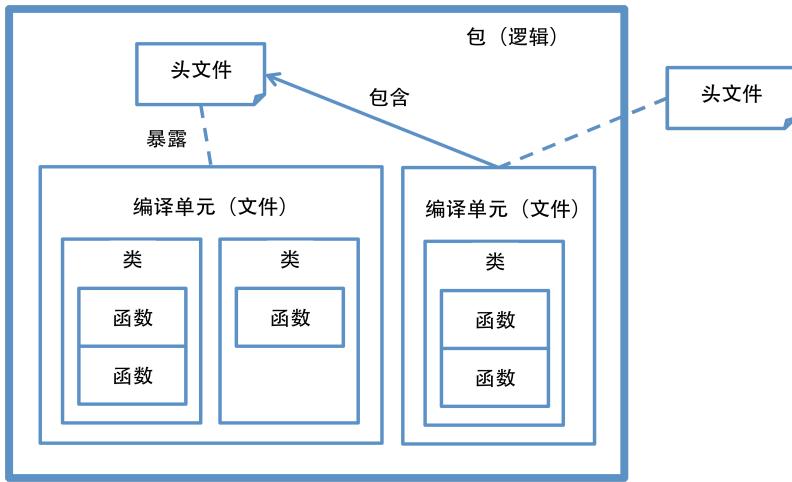
而对于仅应在“编译单元”内可见的定义，则可以通过第二种方式来封装。这对应了Java用来修饰类的private关键字。

其实还有第四种封装手段，即将类定义在一个函数的内部。对于局部变量（静态和非静态），大家应该已经很熟悉。但对于类型定义，其应用场景则相对少见。主要原因在于，它会让一个函数的定义显得不干净，于是将其控制在“编译单元”一级即可。但对于追求纯粹语义的程序员而言，这也是一种有效的手段。



所以，C++一共直接或间接的提供了四种级别的封装机制（从低到高）：

- 局部（函数）
- 类
- 编译单元
- 包



在设计和开发过程中，我们需要合理的利用这四级封装机制。原则是：

- 尽量隐藏可以隐藏的一切信息
- 根据信息重用的范围，尽量将其隐藏在更低的级别

最后需要指出的是，非匿名的namespace并不是一种进行封装的工具。正像它的名字所暗示的，它只是一个用来避免不同模块名字冲突，从而避免每个模块的开发者都不得不使用带有一个或多个前缀——既充满重复，也影响阅读——的名字。

逻辑依赖 & 物理依赖

在程序设计过程中，如果一段代码定义了一个知识，则这段代码就对此知识产生了依赖；如果一段代码调用了一个函数，则这段代码就对此函数声明（而不是实现），以及相关的调用契约产生了依赖。

一个依赖关系一旦建立，那么被依赖方的变化就会导致依赖方的设计变更。所以，出色的软件设计者会付出很大的精力来管理这些依赖，以将变化控制在尽量小的范围内。

但C++的编译模型会导致另外一种依赖：被依赖方的变化，并不会导致依赖方的设计变更，却会引起被依赖方的重新编译。这就是头文件所导致的依赖。

我们把前一种依赖定义为“逻辑依赖”，而后一种则定义为“物理依赖”。

由于编译看起来是机器做的事情，程序员无须修改代码，所以对“物理依赖”的管理常常被忽略，至少没有引起足够的重视。

但如果你工作在一个大型C++项目上，一次完整的编译需要几个小时，甚至几天。即便使用增量式编译，你偶然做的一个改动，也经常需要等待半个小时甚至更久，你就会了解编译不仅仅占用机器的时间，其“反馈效率”会直接影响程序员的工作效率。

既然头文件里的内容会导致“物理依赖”的产生，而严重的“物理依赖”会导致令人无法忍受的编译时间，那么我们就需要认真的来管理头文件。总的策略仍然是：高内聚，低耦合：

1. 每个头文件职责单一
2. 在“编译完备”的前提下，头文件之间尽量不要包含
3. 将尽量多的东西放入源文件

对于第二点，主要是为了用户使用上的方便性。所谓“编译完备”指的是，头文件自身可以通过编译。

由此导致了一些具体实践，例如：

- 尽量使用“前导声明”，而不是直接包含头文件；
- 尽量不使用**inline**函数
- PIMPL模式

“逻辑依赖”是我们主要要管理的依赖，这种依赖是超越语言的。它的任何变化，都需要依赖它的所有客户进行修改，以适应这种变化。

“物理依赖”要优于“逻辑依赖”。因为“逻辑依赖”必然带来“物理依赖”，反之则不然。更重要的是，“逻辑依赖”所带来的代码变更，让团队为之付出的风险和成本，在大多数情况下，要远远高于“物理依赖”。

所以，永远不要为了改善“物理依赖”而增加“逻辑依赖”。

INLINE

在很多C++程序中，总能看到一些小函数或构造函数被定义为inline的，理由是这样可以提高性能。

好吧，假设这样做确实可以增加性能，但并非全无代价。首先它是以“空间”换取“时间”。其次，阅读和修改其源代码需要在两个文件中跳来跳去；其三，对于只想关注类接口的阅读者，增加了“噪音”；最后，它引入了“物理依赖”，inline函数的代码作为一种不稳定的内部实现细节，被放置在头文件里，其变更所导致的大面积的重新编译是个大概率事件。

另外，究竟以什么样的标准来界定一个函数是否是inline的？以行数为标准？好吧，我们极端一些：一行代码以内的函数就inline，其它的就放入源代码文件。

但代码行数并不是静态的，它会由于重构或者需求变更而变化。重要的是，一个设计合理的面向对象程序，每个函数的代码长度都很短，一般都在两三行，连七八行的都很少见。经常性的，一不小心就把一个函数重构为一行的了，或者把一个本来一行的函数重构为两行了，是不是我们就应该把这个函数一会儿移动到头文件，一会儿移动到源代码文件？相信我，没有人会不厌其烦的在两个文件中移来移去。最可能得结果是：你往往会只顾得改代码，而忘掉移动位置。

你可能会说，一个函数最初在什么位置，就依然保持它的位置，除非其行数发生特大的变化。

如果采取这样的策略，说明你希望通过inline提高性能，同时不大幅增加空间消耗的初衷并不重要。那何必自找麻烦？

最重要的是，就连你所设想的“性能提高”也是站不住脚的。算法专家，图灵奖获得者Donald Knuth曾经说过“过早的性能优化是万恶之源，95%的情况都是如此”。按照“二八原则”，一个系统八成的时间运行在两成的代码上；并且“二八原则”还是递归的，也就是说，在剩下的两成代码中，又是20%的代码耗去那八成中80%的时间。

在这种情况下，只有事后根据profiling的结果，对那些关键的热点代码的进行性能调优，才会真正获取回报。为了这些可疑的，little tiny的性能提高，却带来一系列的麻烦，结果只能是得不偿失。

所以，对于**inline**问题，最好的解决办法是，把所有函数实现都放入源代码文件。如果最后证明将某个函数**inline**确实对性能有所帮助，再把它单独移动到头文件，并且通过注释来强调这是一个确定的优化，以防止被它人不小心移回源代码文件。

但我们忽视了**inline**函数一个重要优点——

C++的编译模型决定了，一个类如果需要被多个“编译单元”所见，就必须将声明放在头文件，然后将非**inline**成员函数的实现放到源文件中。这会导致让人厌烦的重复，每次你添加，修改，删除任何一个成员函数，都必须在头文件和源文件重复操作。

我们讨厌重复。为了消除它，一个简单的解决方案是，将所有的函数都定义为**inline**的。但这会大幅增加目标文件的尺寸，以及严重的“物理耦合”。这不是我们想要的结果。

但对于在“编译单元”内部定义的类而言，则无须承担这种代价。因为它的客户数量是确定的，往往只有一个。另外，由于它本来就定义在源代码文件中，因此并没有增加任何“物理耦合”。

所以，对于这样的类，我们大可以将其所有函数都实现为**inline**的，像写Java代码那样，Once & Only Once。

需求二

问题描述

现在需要一个可以用定点小数来记录边长的矩形。可以求它的面积和周长。

规则如下：

- 边长的精度为小数点后两位。如果用户在设置边长时，给出的数字小数部分超过两位，那么从第三位（包括第三位）之后的小数完全忽略。例如：
1.236 => 1.23。
- 面积也用2位小数的定点数来表示。当求面积时，如果得出的结果小数点后超过2位，那么从第三位（包括第三位）按照“四舍五入”的方法舍去。例如：1.236 => 1.24。
- 宽的取值范围为(0, 100]，高的取值范围为(0, 75)。当用户设置的值不在此范围内，则认为矩形为非法。非法矩形的面积和周长为0。

需求实现

```
struct Rectangle
{
    Rectangle(double width, double height);

    double getArea() const;
    double getPerimeter() const;

private:
    double processInput(double value) const;
    double processOutput(double value) const;
    double processWidth(double value) const;
    double processHeight(double value) const;

private:
    double m_width, m_height;
};

Rectangle::Rectangle(double width, double height)
: m_width(processWidth(width))
, m_height(processHeight(height))
{ }

double Rectangle::processInput(double value) const
{
    double factor = ::pow(10, 2);
    return ::floor(value * factor) / factor;
}
```

```

double Rectangle::processWidth(double value) const
{
    double result = processInput(value);
    return (result > 0.00 && result <= 100.00) ? result : 0;
}

double Rectangle::processHeight(double value) const
{
    double result = processInput(value);
    return (result > 0.00 && result < 75.00) ? result : 0;
}

double Rectangle::processOutput(double value) const
{
    double factor = ::pow(10, 2);
    return ::floor(value * factor + 0.5) / factor;
}

double Rectangle::getArea() const
{
    return processOutput(m_width * m_height);
}

double Rectangle::getPerimeter() const
{
    return 2 * (m_width + m_height);
}

```

从实现代码来看，每个函数的有效代码都不超过3行。现实项目的代码如果能够被控制在这样的复杂度，维护成本就不会是个大问题。

但，这份实现真的没有任何问题吗？

提高表达力

消除魔数

代码中存在着大量的“魔鬼数字”（magic number），有的来自于需求中的约束，有些出自于算法的需要。如果不给它们一个有意义的名字，阅读者不得不花更多的时间来理解它们。

下面是对两个魔鬼数字2的重构。虽然它们的值都是2，但事实上它们在需求中表示完全不同的含义：一个是对边长输入的精度，一个是面积输出的精度。所以我们应该用两个不同的名字。

```

const unsigned int INPUT_RECISION = 2;
const unsigned int OUTPUT_PRECISION = 2;

```

```

double Rectangle::processInput(double value) const
{
    double factor = ::pow(10, INPUT_PRECISION);
    return ::floor(value * factor) / factor;
}

double Rectangle::processOutput(double value) const
{
    double factor = ::pow(10, OUTPUT_PRECISION);
    return ::floor(value * factor + 0.5) / factor;
}

```

在这个实现里，我们将常量集中式的定义在了前面。有些人可能更喜欢把它们放在最接近它们使用的地方。比如：

```

double Rectangle::processInput(double value) const
{
    const unsigned int INPUT_PRECISION = 2;

    double factor = ::pow(10, INPUT_PRECISION);
    return ::floor(value * factor) / factor;
}

double Rectangle::processOutput(double value) const
{
    const unsigned int OUTPUT_PRECISION = 2;

    double factor = ::pow(10, OUTPUT_PRECISION);
    return ::floor(value * factor + 0.5) / factor;
}

```

我个人对此并无特别偏好，而是根据不同的情况采取不同的策略。一是看这个定义和使用此定义算法之间的关系有多紧密——如果这个常量定义和算法之间可以各自独立变化，那我就会把它们分开；另外，如果一个常量在多处被使用，我也会把它们分开。所以，对于这两个常量，我会倾向于分开定义。

但对于另外的两个魔数，如果提取常量的话，我会肯定会把它们放在一起。因为它们的存在完全是出于算法的需要。

```

double Rectangle::processOutput(double value) const
{
    double factor = ::pow(10, OUTPUT_PRECISION);
    return ::floor(value * factor + 0.5) / factor;
}

```

但对于它们，我倾向于不去消除它们。因为它们是算法决定的必然，以数字方式存在，反而对于理解算法更有帮助。我们不妨对照一下：

```

double Rectangle::processOutput(double value) const
{
    const double BASE = 10;
    double factor = ::pow(BASE, INPUT_PRECISION);
    return ::floor(value * factor) / factor;
}

```

本来函数pow的第一个参数的含义就是base，这样的定义完全是画蛇添足。代码阅读者在这一刻对于“base值”的关注要远大于“它是base”这件事。有些团队强制规定不允许存在魔数，所以团队成员只好把上述定义改为：

```

double Rectangle::processInput(double value) const
{
    const double TEN = 10;
    double factor = ::pow(TEN, INPUT_PRECISION);
    return ::floor(value * factor) / factor;
}

```

这不是一个玩笑，而是来自于真实发生过的事情。

剩下的两个魔数也是来自于需求中的约束，所以我们照猫画虎：

```

const double MIN_WIDTH = 0.00;
const double MIN_HEIGHT = 0.00;
const double MAX_WIDTH = 100.00;
const double MAX_HEIGHT = 75.00;

.....
return (result > MIN_WIDTH && result <= MAX_WIDTH) ? result : 0;

```

消除重复

提取函数

函数processInput和processOutput的实现有很大的相似性。这是一种“参数型”重复，所以，我们将公共部分进行“提取函数”操作。

```

double Rectangle::processPrecision
( double value, unsigned int precision, double compensation) const
{
    double factor = ::pow(10, precision);
    return ::floor(value * factor + compensation) / factor;
}

double Rectangle::processInput(double value) const
{
    return processPrecision(value, INPUT_PRECISION, 0);
}

```

```
double Rectangle::processOutput(double value) const
{
    return processPrecision(value, OUTPUT_PRECISION, 0.5);
}
```

模块

在生活中，每个人就应该有过这样的体验：两个人就一个话题展开讨论，过程中，对方的观点让你迷惑，抗拒，当你开始在心中将其定论为“不可理喻”的时候，发现对方脸上也尽现茫然和不屑。最终发现，两个人虽然都在谈一个名词，各自对它的定义却是天壤之别。

“模块”就是一个极易造成上述场景的名词。当不同的人在谈论软件“模块”时，有人脑海里漂浮的是一个多任务环境的进程，有人想的是一个组件，有人指的是一个文件，还有人并不知道自己谈的是什么…

上网搜了一下之后，找到的对于“模块”在计算机科学领域的定义也都相当笼统。比如，我在XX百科看到的“模块”定义如下：

在程序设计中，为完成某一功能所需的一段程序或子程序；或指能由编译程序、装配程序等处理的独立程序单位；或指大型软件系统的一部分。

而在www.thefreedictionary.com看到的定义如下：

A portion of a program that carries out a specific function and may be used alone or combined with other modules of the same program.

既然这个领域并没有对“模块”进行明确的定义，而我又无法做到“难得糊涂”，所以我要对它进行清晰的界定。首先是因为本文中将多次提到“模块”一词，需要避免不必要的困扰。

在本文的任何地方，只要看到“模块”一词，指的都是一个可通过编译的源代码文件，加上一个或多个用来放置其“对外用户接口”的头文件；如果一个模块是以模版或宏的方式来定义，则可能没有源文件，只有头文件。

这样的定义，并未超出字典对“模块”定义的范围。它只是更明确。

更重要的是，这样的定义，对于指导我如何对文件进行管理，如何将代码合理的部署在不同的文件中，是非常有帮助的。我不喜欢似是而非，不喜欢模棱两可；那样会让我无所适从。只有在清晰定义的指导下，我才可以安心的做事。

文件对我而言，不应该是一个可任意堆放代码的杂物箱，而应该是一个个与逻辑设计具有精确映射关系的档案袋。本属于别人的一份表格出现在我的档案袋里，就说明一定有什么地方除了错。同理，当我无法确定一个代码元素应该放在哪个文件的时候，也一定是我逻辑设计出问题的时候。

有人可能会疑惑，既然你谈的模块就是文件，那你为何要还要把它称做模块。

我也希望对于同一事物，应该只有一个名字。而一个名字也只应该描述一个事物，这样这个世界就会省去很多麻烦。

小说《1984》（乔治·奥威尔）里面的“老大哥”也在进行类似的努力。不过他的动机在于“洗脑”和控制。而我的理由却完全不同。

由于C++语言的特点，我们不得不把代码分成两部分：对外接口和内部实现；其中对外接口以头文件的方式发布，而内部实现则放置在源文件中。所以，一个模块，只要它可能被外部使用，就会分为至少两个文件。虽然用模版和宏定义的库也可能只有头文件，但无论那种情况都说明了文件和模块不是一个完全对等的概念。

更重要的是，既然文件内容和逻辑设计有着精确的映射关系，所以一个“模块”事实上指的是一组逻辑设计实体，它职责单一，边界明确，依赖清晰。而文件，则只是那个袋子。

降低耦合

将工具函数改为模块内函数

然后我们发现这三个函数均没有访问成员变量，所以，它们应该属于“类静态函数”，而不应该是“类成员函数”。

另外这三个函数都属于“私有的”。那么它们作为“私有类静态函数”，被放在头文件则没有任何价值，反而增加了很多麻烦：

- 定义和声明的重复；
- 增加物理耦合；

- 头文件的阅读者仅仅关注public和protected的成员，私有成员增加了类声明的“噪音”

所以，任何“私有静态成员”，无论是“变量”还是“函数”都应该移入“模块内部”，放置在匿名namespace内，如下：

```
namespace
{
    double processPrecision
        ( double value, unsigned int precision, double compensation) {...}
    double processInput(double value){...}
    double processOutput(double value) {...}
}
```

提高函数可重用性

然后我们发现processPrecision如果进行合适的参数化，则会变成一个用来处理浮点数的通用函数：其中一个为“舍弃算法” floor，另外一个是“四舍五入算法” round。所以，我们首先将它们封装起来，给予准确的命名。

```
double floor(double value, unsigned int precision)
{ return processPrecision(value, precision, 0); }

double round(double value, unsigned int precision)
{ return processPrecision(value, precision, 0.5); }
```

一个清晰直观的名字，是团队范围内进行代码重用的基础。大多数人都没有耐心去代码库中，查看一个个类或函数的实现细节，以找出适合自己重用的代码。

分离模块

尽管系统中当前并没有任何其它对于它们的需要，但将其抽取为具备更好可重用性的函数，仍然非常有价值和意义。尤其在大型团队，这样的工作一旦不及早去做，就会将其淹没在实现细节之中，当别人需要类似功能的时候，由于无法直接看到可重用代码，就会重新再实现一遍。

所以，对于这类本身就是具备更好可重用性的概念或功能，应该及早将其提取出去，放置到公用库的合适模块里。

对于这三个函数，我们应该将其移动到名字包含Math的头文件中，其中floor和round作为外部接口，而它们共同依赖的实现函数则作为内部实现细节，在模块内进行封装。

```
namespace
{
    double processPrecision
        ( double value, unsigned int precision, double compensation) {...}

    double floor(double value, unsigned int precision) {...}
    double round(double value, unsigned int precision) {...}
}
```

杜绝潜规则

不要投机取巧（不要将非法长度设为0），不要把偶然当必然（非本质的关系不稳定）

直觉

尽管没有做过详细的调查，但通过和出色的程序员交流，发现他们都有一个共同的结论：相信直觉。

具体点说，是指如果一个设计让你总是觉得别扭或者不安，那么这个设计就基本上肯定是有问题的。

Kent Beck在他的演讲“响应式设计”里提到，任何设计决定都应该是“直接”的（Straitforward），而不是拿着一把大锤把代码强行砸入它不应该呆的地方。

所谓直觉，并非指没有道理可讲，纯粹感觉上的事情。直觉的背后，往往是你经验所形成的常识，或对逻辑一致性长期追求所形成的思维习惯。

所以，直觉又是严重依赖于个体的。对于同样一件事情，一些人会觉得违背直觉，而另外一些人的反应则是毫无感觉。

用对象取代数据值

自完备的头文件

命名

在讨论如何进行命名之前，我们先来回顾一下相关历史。

在早期，由于存储设备的稀缺性，编译器往往会对名字长度有一定的约束。另外，对于那个时代的程序员而言，函数逻辑的重要性要远远大于命名。即便你给一个无人能够理解的名字，只要名字没有冲突，编译器是不会抱怨的，运行时更是毫无影响。而一旦语法有错，编译器将会报错，而如果逻辑出了错，则是更加严重的问题。

逐渐程序员们意识到代码可理解的重要性。然后出现了两个应对的方法：强调注释的重要性；匈牙利命名法。但最终证明，它们的应用效果可以评价为“杀敌一千，自损八百”。

注释带来信息重复，而在现实中人们总是忘了维护，以至于它不仅不能帮助理解，反而会造成困惑和误导。Norm Schryer的名言“假如代码和注释不一致，那么很可能两者都是错的”，准确的描述了不一致的注释给维护者带来的困扰。更加让人哭笑不得的是，很多注释完全是对代码行为的描述，而不是对代码意图的解释。这样的注释，只是在浪费所有人的时问。

但注释本身并非问题，把它当作提高代码可理解性的主要手段才是问题，把注释密度当作考核指标更是百害而无一利。代码可理解性的提高只能依靠代码自身，任何其它手段都是在代码自身无法表达的情况下，才应该拿来作为辅助工具。

而匈牙利命名法一度被社区奉为神物。以前缀的方式清晰描述出一个类型或变量的类型信息，让程序员可以直观的随时了解更多细节，从而避免误操作。在大函数非常普遍，而“重构”不是一种普遍行为的前提下，它不失为一种解决问题的方法。但随着开发环境和工具的完善，以及程序员意识和习惯的改变，匈牙利命名法对于细节的耦合让其反而变成了维护的负担。

在C语言缺乏名字空间的情况下，为了避免重大项目中的名字冲突，前缀就变成了一种解决问题的重要手段。而前缀会让函数的名字更长，有些大型团队的编程规范甚至让一个函数的名字有三个前缀。而每个前缀都平均有三到五个字母，再加上下划线，一个名字首先已经有了十到十五个字母；如果一个函数再承担

过多的职责，真正的名字部分就会再增加至少十五个字母。最终的结果让一个函数的名字几乎没有什么可理解性可言。每次一看到一大块字母的堆积，绝大多数人都会失去阅读的愿望。而这样的长名字，每被调用一次，都会让调用方的代码更加拥挤和晦涩。

是的，我喜欢让人一目了然，望文生义的短名字。在不伤害可理解性的前提下，名字越短越好。但必须注意它的前提——不要伤害可理解性。否则，我们就失去了根本。

基于此，名字的所有前缀，后缀等用来表现细节而不是意图的部分都是一种无谓的伤害，统统不应该使用。所以，不要给一个类型加上诸如ST，EN，TY等前缀或后缀，能够让阅读者知道这是一个类型就够了。否则，除了影响阅读之外，更关键的问题是，谁能保证一个枚举随后不会变为类，一个unsigned int的别名定义最后不会变为一个类？这样的反映不稳定细节的命名只会增加麻烦。要知道Java社区的主流系统从来都没有在命名时使用过任何前后缀，但得到的只是更清晰的代码，却从未听说引起过什么问题。

另外，起一个好名字的前提是，每个类或者函数都应该是单一职责的。一旦需要一个很长的名字才能描述一个函数或类，尤其是当名字里包含“*And*”，“*Or*”等连词时，基本上就可以肯定这个函数或类已经承担了太多的责任。

但是，能够通过命名发现函数／类多重职责的前提是，你真心希望能够用自然语言来清晰而完整的描述它的职责，否则，你不但不能通过命名识别出多重职责，反而会给出让人困惑或误导的名字。比如，当看到一个成员函数的名字是isValid时，阅读者会认为这是一个验证对象状态的查询接口；但一阅读代码，却发现它事实上做了很多事情，甚至修改了对象的状态。这样的结果还不如给一个包含“*And*”的长名字，至少它反映了事实。

但必须承认的是，起一个好名字很困难。即便对英语是母语的程序员，一个好名字也不是件“唾手可得”的事情。但命名如此重要，我们不应该就此认命，而是通过询问同事，查询字典，尽自己所能给出准确的名字。

需求三

问题描述

现在需要一个可以用定点小数来记录边长的正方型。可以获取它的面积和周长。

规则如下：

- 边长的精度为小数点后两位。如果用户在设置边长时，给出的数字小数部分超过两位，那么从第三位（包括第三位）之后的小数完全忽略。例如：
1.236 => 1.23。
- 面积也用2位小数的定点数来表示。当求面积时，如果得出的结果小数点后超过2位，那么从第三位（包括第三位）按照“四舍五入”的方法舍去。例如：1.236 => 1.24。
- 边长的取值范围为(0, 75)。当用户设置的值不在此范围内，则认为正方形为非法。非法正方形的面积和周长为0。

抽象的动机

在OO训练营进行的过程中，我多次看到经验丰富的程序员们会对正方形和长方形给出一个公共的抽象基类，名字一般是Shape或者Polygon。比如：

```
struct Shape
{
    virtual ~Shape() {}

    virtual double getArea() const = 0;
    virtual double getPerimeter() const = 0;
};
```

然后，Rectangle和Square都继承自Shape：

```
struct Rectangle : public Shape {...};
struct Square : public Shape {...};
```

问及原因，得到的答案是：它们两个确实都是Shape，如果未来添加一个圆形，还可以让它继承自Shape。另外，还可以强迫每个新添加的形状都必须实现getArea()或者getPerimeter()方法。当然，还有其它的想法：如果所有形状之间有一些公共代码，它们可以被放置在Shape里。

虽然面向对象强调的是对“物理世界实体”进行建模，但这并非意味着，我们需要对其物理世界的特征和行为进行完整的描述。

软件之所以存在，不是为了对现实世界进行完整的描述，而是为了满足特定的需求。其每一行代码都应该由需求直接或间接驱动而来。

Rectangle和Square为什么存在？因为客户需要我们提供一套库来描述和记录它们。我们为何没有定义Circle，因为客户没有要求。

同理，它们的行为也是由于客户的明确要求而被创建的。如果用户仅仅要求求面积，我们就肯定不应该提供求周长的成员函数，尽管周长也是它在物理世界所具备的必然属性。如果你觉得反正多提供一个函数，就算用户用不到也没关系，因为那是它的本质。那我们还要不要提供求对角线长度的函数？难道那不是Rectangle和Square的本质吗？

而Shape作为一个抽象基类，如果它必须存在，一定是因为某段客户代码确实需要依赖这样的抽象，因为那段客户代码需要屏蔽掉不同形状的差别。如果客户代码完全不需要对它们进行无差别访问，而是直接使用Rectangle和Square，那么这个抽象就是完全是一种毫无价值的冗余。

你可能还会争辩说，虽然现在没有任何代码需要用到这个抽象，并不意味着未来也不需要，而现在给出这个抽象，无非就是多定义了一个没有任何实现代码的抽象类而已，并没有增加多少成本。

如果这个理由成立的话，那么Shape究竟应该具备什么样的行为好呢？在没有客户代码使用模式的要求下，靠盲目猜测给出的任何行为都有可能在需要真正发生的那一天失效，更别说那一天很有可能永远也不会到来。

我们不妨来设想一下：未来某一天，客户要求增加一个类来表示不规则多边形，但无需求面积和周长，而是提供一个Draw方法；另外，Rectangle和Square也需要提供Draw方法。而客户希望无差别的对待各种形状，想利用运行时多态来对它们进行绘制。

这种情况下，Shape确实应该存在，但它却只应该存在抽象的Draw方法，而不应该提供求面积和周长。

然后你会站出来，求面积和周长是任何形状的本质属性，所以保留也无妨。那好吧，这时候你就不得不花大力气来实现不规则多边形的求面积和周长和算法，要知道它们并不像Rectangle和Square的算法那么简单。

好容易千辛万苦完成之后，客户又提出一个需求，需要增加一个不规则的弧线多边形，只需要Draw，无需求面积和周长。于是，再一次，为了提供不被需要的一致抽象，你必须额外付出更多没有产生任何商业价值的努力。

所以，任何一行没有被需求直接或间接需要的代码元素，都是一种冗余。更多的代码，意味着更多的理解和维护，而这些都意味着成本。如果说混乱的代码至少还在创造用户价值的话，那么冗余代码则是彻底的只赔不赚。

因此，任何时候都不要主动给一个类增加额外的属性或方法，无论它是不是一个类所代表概念的本质行为或特征。更不要主动创造任何概念，如果一个设计中没有任何结构，逻辑或约束需要使用这种概念。无论你用何种方法学，代码中的每一个元素都应该是需求驱动的产物，而不是对现实世界的完备描述——更何况，你也不可能对其真正完备的进行描述——因为，“所有抽象都是谎言”。

继承

第三部分：练习二

需求一

问题描述

实现一个库：通过这个库，用户可以以Mile为单位来表示一个长度，精度为1 Mile。并且，可以对比两个长度的相等性：

- 3 Mile == 3 Mile
- 3 Mile != 2 Mile
- 3 Mile != 4 Mile

需求实现

根据需求的直观描述，我们可以构造出下面的用例：

```
TEST(两个数量相同的以Mile为单位的长度应该相等)
{
    ASSERT_TRUE(Mile(3) == Mile(3));
}

TEST(两个数量不同的以Mile为单位的长度应该不等: 3 Mile != 2 Mile)
{
    ASSERT_TRUE(Mile(3) != Mile(2));
}

TEST(两个数量不同的以Mile为单位的长度应该不等: 3 Mile != 4 Mile)
{
    ASSERT_TRUE(Mile(3) != Mile(4));
}
```

根据用例所定义的UI¹，我们将类定义和实现如下：

```
struct Mile
{
    Mile(double amount) : m_amount(amount) {}

    bool operator==(const Mile& rhs) const
    { return abs(m_amount - rhs.m_amount) < DBL_EPSILON; }

    bool operator!=(const Mile& rhs) const
    { return abs(m_amount - rhs.m_amount) >= DBL_EPSILON; }

private:
    double m_amount;
};
```

¹ User Interface (用户接口)

将这个实现编译链接，并运行所有测试。All Tests Pass。

STRUCT vs. CLASS

你应该已经注意到了Mile是一个struct，而不是一个class。为什么要这么做？

关键字struct是C++继承自C语言的一项遗产；作为更加贴切的词汇，词汇class被C++引入，用来表现“类”。这个决策造成的结果是，一种语言提供了两个关键字来表示完全一致的概念，在什么情况下应该使用谁，社区内并无定论，甚至C++的发明者Bjarne Stroustrup也无法给出毫不含糊的建议。

如果只是名字的差别，那么毫无疑问，在任何时候都应该使用class，毕竟它更直观，准确。

很多C++开发者可能并不同意这一点。从他们的逻辑和经验出发，struct仍然应该用来定义那些只有数据，没有行为的“类”，因为它们事实上是C语言中的结构体；而class只应该用来定义真正的“类”——那些有行为的家伙。

作为一门多范式的语言，C++可以支持面向过程风格和面向对象风格的编程。C++程序员也有足够的底气来宣称自己在使用混合风格。

我偶尔也会使用混合风格。但对于一致性的追求会让我以一种风格为主，那就是面向对象，只有在极个别场合，我才会基于成本的考虑和务实的选择而采用过程风格，它占的比例可能连1%也不到，并且都是在定义自由函数。

而在面向对象设计中，有数据无行为的类通常是设计问题的征兆。尽管这样的类在社区，尤其是C++社区内非常普遍，但这也正是许多项目都在“焦油坑”中挣扎的原因。我已经很久没有机会定义这样的类了。所以，struct不会基于这样的原因而被我使用。

除了名字不同之外，class和struct唯一的差别是：默认可见性。这体现在定义时和继承时。struct在定义一个成员，或者继承时，如果不指明，则默认为public；而class则默认为private。

但这不是我要讨论的重点，介绍语言的基础特性并不是本文的目标。重点是这样的差别会产生出不同的代码。

比如，现在要定义一个纯虚类，用两个不同的关键字，会导致如下不同的结果：

```
class Invokable
{
public:
    virtual Any& invoke() = 0;
    virtual ~Invokable() {}
};

struct Invokable
{
    virtual Any& invoke() = 0;
    virtual ~Invokable() {}
};
```

两者差别很小，你或许并不在意。但对我而言，一个纯虚类，从逻辑上本来就是一个只有公开方法声明，没有实现细节的“接口”类，它所声明的一切都应该是公开的。在这样的契约关系下，再通过public指明其公开性，就是画蛇添足。

懒惰的我讨厌冗余，讨厌重复。更何况从平衡和美感的角度，那个横立的public像洁白墙面上的一沫蚊子血，显得格外刺眼。

对于实体类，由于存在私有数据（没有数据，只有实现的实体类也往往意味着坏味道），而class的默认私有性，让这种场景变成了它的show time。

```
class Foo
{
    int a;
    double b;

public:
    Foo(int);
    void doSomething();
};
```

这个类把私有数据定义在前面，把公开方法定义在后面，所以可以利用class的默认私有性。

但这样的定义布局并不只是顺序上的差异。我们的认知习惯和阅读顺序决定了我们总是希望把更重要的、更希望人们了解的信息摆在一目了然的位置。而不是让别人穿越重重迷雾才能找到自己的关注点。我们希望别人更容易的理解我们的意图，而不是试图挑战别人的智商和耐心。

所以，信息摆放的顺序就成了一件有所谓的事情。你如果认为私有实现细节是更重要的，那就把私有数据摆在前面。否则，就把公开方法置于前列。

对于把程序理解为“数据结构+算法”的程序员，尽管正在使用面向对象的元素——“类”，会依然认为“理解一个程序的前提是理解它的数据结构”。在这样的价值驱动下，把私有数据摆在前面就是完全合情合理的。在数年前，我就曾在一本C++相关的书中（忘了名字）读到过这样的建议。

但对于越来越确信“抽象”和“信息隐藏”对于软件之重要性的我，则更倾向于认为公开接口才是理解一个模块最关键的知识。当试图去使用一个模块时，我总是会优先查看它的测试用例（如果有的话），然后去看它的公开接口声明。一般而言，这对于理解它能做什么以及如何使用它已经足够，接口声明处的私有元素反而会干扰我对一个模块的理解。只有在好奇心的驱使下，我才会更进一步，去看看它的实现。

基于这样的认知，当定义一个类的时候，我会把公开方法定义在前面。至于私有的内容，我总是会竭尽所能的不希望引起人们的注意，宁愿付出一些代价也要把它们藏到别人在类声明里看不到的地方，更不会放在前面扰乱视听。

所以，`struct`的默认公开性在定义实体类的时候也更有价值。虽然这个价值相对于用它定义接口类从比例上小一些（但绝对价值是一样的）。

罗嗦了半天，我真正的目的不在于讨论到底应该用`struct`还是`class`来定义一个类，而是想明确，究竟是`public`接口更重要，还是实现细节更重要。在明确了这一点之后，我们才会清楚，我们该如何应对头文件里类定义所带来的物理耦合问题，但那是后续的故事。

另外需要强调的是，无论你使用`class`还是`struct`，都应该只选择其中一个，而不是混合使用（我过去的做法就是，定义接口类是使用`struct`，定义实体类时用`class`）。否则，在大量使用前导声明的情况下，一旦一个使用`struct`的类改为`class`，所有的前导声明都需要修改。编译器或许并不认为这种不一致是一种错误，但那些不断骚扰你的警告也会让你不胜其烦。

消除重复

我们检视一下代码，很快会发现：操作符“`==`”和“`!=`”各自进行了独立实现。

尽管两份代码不尽相同，但事实上却是一种重复——

从数学逻辑上，两者之间具备必然的互斥关系；因此，两者的实现逻辑也必须具备互斥关系。如果各自独立实现，则两个函数都会对此“互斥约束”产生依赖。其中一个逻辑发生变化，另外一个必须跟着改变。

通过将此约束进行“显性化”，则可以消除这个重复。

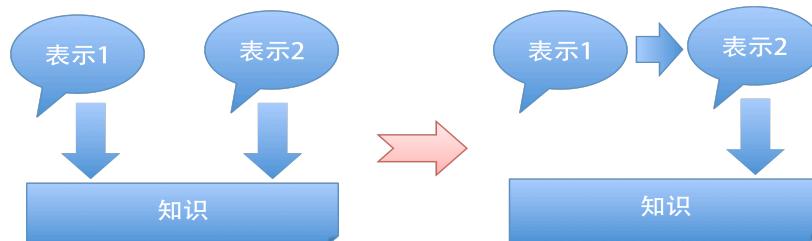
```
bool Mile::operator==(const Mile& rhs) const
{
    return abs(m_amount - rhs.m_amount) < DBL_EPSILON;
}

bool Mile::operator!=(const Mile& rhs) const
{
    return !operator==(rhs);
}
```

在约束“显性化”的过程，我们将函数operator!=对于“对象相等性判断算法”这个知识的直接耦合，变成了通过依赖operator==来间接依赖“对象相等性判断算法”这个知识。

这是一个解决重复定义的常用手段。如果一个系统由于某些原因，确实需要通过不同的“表示”来表达同一项知识。如果各自独立对这项知识进行表达，则会造成重复问题，从而带来维护上的问题。

这种情况下，我们应该只用一种“表示”来表达一项知识，其它“表示”则直接或间接的依赖这个“表示”。由此解决重复问题。



这并没有违背DRY原则，因为DRY原则的定义“在一个系统内，任何一项知识都只应该存在一个明确而权威的表示”，并没有禁止对一项知识存在多个表示。事实上，无论存在有多少个表示，只要仅存在一个直接的明确的表示，它就是那个权威的表示。

消除冗余

在需求描述中，明确的指明：用户所表示的长度精度为1 Mile。

我们的实现用了浮点数。而浮点数存在精度问题，由此引入了更复杂的代码逻辑和更多的代码元素。而这些复杂度，对于需求是不必要的。事实上，更简单的无符号整数就够了。

```
struct Mile
{
    Mile(unsigned int amount) {...}

    bool operator==(const Mile& rhs) const
    { return m_amount == rhs.m_amount; }

    bool operator!=(const Mile& rhs) const {...}

private:
    unsigned int m_amount;
};
```

我们知道，使用double要比unsigned int具备更好的通用性。如果我们无须为之付出任何代价，我们会毫不犹豫的选择double。但对于当前需求而言，我们必须处理double所带来的精度问题。

客户指明了精度为1 MILE。如果我们的接口使用了double类型，客户就可能误传以double类型的数值来构造MILE对象²。那么我们就要面临MILE(9.8)和MILE(10)是否相等，MILE(9.2)应该等于MILE(9)还是MILE(10)之类的问题。虽然由此带来的绝对复杂度并不惊人，但相对unsigned int，复杂度却是成倍的增加。而我们当前无须为之买单。

但这并非意味着，我们对于未来可能的变化完全采取鸵鸟策略。我们确实需要考虑另外一个成本，那就是将来一旦这样的变化发生，我们需要付出什么样的代价。否则，我们就可能面临不可承受之重。

一旦未来用户将精度改为0.1 MILE，我们就必须把所有的unsigned int改成double。我们自己的代码尚可控制，但客户的代码却已经遍布各处，这不是我们愿意看到的。

两者之间的平衡点，就是不要让客户依赖具体的类型，而是提供一个“抽象”。只是抽象背后的具体实现，仍然是unsigned int。

² 墨菲法则：一件事只要可能出错，那就一定会出错。

```
typedef unsigned int Amount;

struct Mile
{
    Mile(Amount amount);

    .....

private:
    Amount m_amount;
};
```

数量现在已经由一个抽象数据类型Amount来表示，它变化的可能是否只有double？它是否还可能变成一个更复杂的数据类型，比如类？

由于Amount一定是一个数值，我们似乎无须担心它变成一个类，别名技术应该已经可以解决它变化的问题。不过，以double来表示数量就完全没可能是一个类吗？double的精度控制问题难道不应该是Amount的职责吗？

既然Amount向类变化的可能性是存在的，那么我们就需要考虑：如果它变成类，我们现在需要付出什么样的代价来应对它？

我们似乎无须为之做任何事情。即便Amount是一个类，但既然它从本质上应改是个数值，那么它就应该支持所有的数值计算。

除了一点小问题：Amount相关的参数是传值，还是传引用常量的问题。但即便我们不解决它，将来也只会带来一些可能的性能问题。而过早的性能优化是邪恶的。

只是，即便我们现在就以引用常量的方式来传递它，也没有引入什么成本。那何乐而不为呢？

```
struct Mile
{
    Mile(const Amount& amount);

    .....

private:
    Amount m_amount;
};
```

现在的方案，是否已经防备了在Amount类型上所有可能的变化？

坦率的说，我不知道，也没有人能够知道。我们努力防备变化，变化却经常以超出所有人预期的方式发生。这种特征并非软件行业的特质，而是这个世界的本质。

但我们也已经做了我们所能做的和应该做的，这就够了。对于不确定的未来，我们没有必要现在就杞人忧天。我们必须坦然接受我们不是“拥有超能力的巫师”这个现实。

YAGNI

像大多数喜欢软件设计的Geeks一样，在我最初的软件生涯里，我总是倾向于尽自己所能，让设计尽量的灵活。

加入ThoughtWorks之后，每次和Pair讨论一个设计，当Pair问我，这样做的好处是什么的时候，我的答案往往是：“如果将来...”

然后，我的Pair会很淡定的回应：“YAGNI”。

随后的日子里，我不断的听到有人在谈论“简单设计”，问起什么是“简单设计”，得到的答案往往是：“YAGNI”，“Do The Simplest Thing That Possibly Work”等口号，或者给出“如果一个问题能够用数组解决，就坚决不用链表”这样简单的例子。

这样的答案让我对“简单设计”有了朦胧的感觉，但却不能在我需要做出设计决策的时候，给出一个清晰的标尺：这个设计是“简单设计”，还是“过度设计”？

带着困惑，和一位资深的同事讨论，结论是“如果未来一种变化发生的时候，你必须付出很大的代价才能应对，那么你眼前就可以进行一定的过度设计来应对它；否则，Leave it alone”。

后来，去参加ThoughtWorks Immersion，和培训老师再次讨论起这个问题，得到的答案是：“如果现在你有两个方案A和B，A比B更加有弹性，并且A的实现和维护成本和B很接近，甚至更小，那么选择A”。

两者结合起来，基本上描述了在“简单设计”的价值观下，进行设计决策的依据。它们比之前听到的简单口号更具体。

所以，YAGNI并非不考虑未来，而是考虑：

- 如果未来一个预期的变化发生，我们将来需要付出怎样的代价来应对？

- 如果未来一个预期的变化没有发生，我们现在付出了怎样的成本（浪费）？

也就是说，首先找到当前所能想到的能够应对这种变化的最廉价的方案。看看它眼前需要付出的额外成本（可能的浪费）和未来变化发生时需要付出的代价（可能的成本）之间的比例关系。比值越小，就越值得去做。

但事实上，这个比值是无法准确计算的，只能粗略的估量。这就给决策留下了很大的空间，不同的人很可能做出不同的判断和决策。但这就是软件开发的现实，本来软件工程就不是一门精确的学科。

但这并不意味着我们就此悲观。正如我们并不需要精确测量，也能准确判断一些人的胖瘦一样；很多情况下，其实我们不需要精确的估量，也能识别明显比例失调的情况，并制定相应的应对策略：

如果，未来一个预期中的变化发生，我们需要付出的代价很小，我们眼前可以不用考虑它。

如果，未来一个预期中的变化发生，我们需要付出的代价很大，我们眼前需要建立一些机制（比如抽象，约束，甚至是纪律）来应对它。注意，我们只是建立机制，在实现层面，我们仍然会给出对于当前需求刚刚好的功能。除非——

你找到一个额外复杂度非常小的方案，让我们在实现当前功能的时候，正好把未来的一种情况也实现了（确保没有任何一行代码是针对未来具体需求的）。

简单的说就是：尝试预测未来，可能应对未来，绝不实现未来。

需求二

问题描述

用户除了可以使用Mile为单位来表示长度之外，还可以使用Yard为单位来表示长度，其中：

- 当以Mile为单位来表示一个长度时，精度为1 Mile；
- 当以Yard为单位来表示一个长度时，精度为1 Yard；
- 能够对比任意两个长度的相等性：
 - 1 Mile == 1760 Yard
 - 3 Yard == 3 Yard
 - 1 Mile != 1761 Yard
 - 3 Yard != 4 Yard

需求实现

在上一个版本，我们将Mile实现为一个类。很自然的，Yard也可以实现为一个类。两者之间的关系，在Mile用例中表现如下：

```
TEST(数量符合1:1760比例的，以Mile为单位的长度和以Yard为单位的长度应该相等)
{
    ASSERT_TRUE(Mile(2) == Yard(3520));
}

TEST(数量不符合1:1760比例的，以Mile为单位的长度和以Yard为单位的长度应该不等)
{
    ASSERT_TRUE(Mile(2) != Yard(3521));
}
```

同样的，Yard的测试用例如下：

```
TEST(数量符合1760:1比例的，以Yard为单位的长度和以Mile为单位的长度应该相等)
{
    ASSERT_TRUE(Yard(3520) == Mile(2));
}

TEST(数量不符合1760 :1比例的，以Yard为单位的长度和以Mile为单位的长度应该不等)
{
    ASSERT_TRUE(Yard(3521) != Mile(2));
}
```

两个不同单位的长度该如何对比相等性？不难得出的算法是：我们两者都转换成同一单位的数量，才能通过对比数量的相等性来确定对象的相等性。

基于这个算法，我们给出了下列实现：

```
struct Mile
{
    Mile(const Amount& amount) {...}

    bool operator==(const Mile& rhs) const
    { return m_amount == rhs.m_amount; }

    bool operator==(const Yard& yard) const
    { return m_amount * 1760 == yard.m_amount; }

    bool operator!=(const Mile& rhs) const {...}
    bool operator!=(const Yard& yard) const {...}

private:
    Amount m_amount;
};
```

为了能够和Yard对比相等性，Mile将自己的数量转换成了以Yard为单位的数量。但问题是：Yard的数量现在由其私有数据m_amount保存，怎么办？

我们至少有3个选择：

- 将Yard的属性m_amount设为public的；
- Yard提供一个额外的方法：比如
 - unsigned int getAmount() const，或
 - bool isAmountEquals(unsigned int amountInYard) const。
- 将Mile设置为Yard的friend；

第一种方法，将会导致所有Yard的客户都可以访问Yard的这个属性。而m_amount只是作为Yard的一种具体实现方式。我们完全可以让Yard换一种实现方式，比如记录以Mile为单位的数量，而不是以Yard为单位的数量，同样可以满足需求。所以，作为一个用户直接可见的类，Yard暴露出自己基于实现的属性，将会增加不必要的耦合度。

第二种方法比第一种方法略好一些。至少，无论Yard的内部实现细节如何，总是可以将其转换为以Yard为单位的数量返回出来。

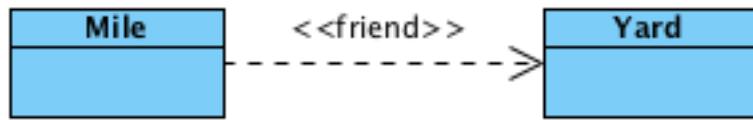
```
bool Mile::operator==(const Yard& yard) const
{ return m_amount * 1760 == yard.getAmount(); }
```

或者，

```
bool Mile::operator==(const Yard& yard) const
{ return yard.isAmountEquals(m_amount * 1760); }
```

但同样的问题是：Mile是直接暴露给用户的类，从需求上看，用户并不需要这些额外的接口，它们的存在仅仅出于实现的需要，由此而增加客户的依赖是不恰当的。

所以，第三种方式似乎更优一些。至少，这种friend关系仅仅会造成Mile和Yard的强依赖，对于Yard的其它客户却不会增加任何耦合。



按照第三种方式，我们对Yard进行了实现：

```
struct Yard
{
    Yard(const Amount& amount) { ... }

    bool operator==(const Yard& rhs) const
    { return m_amount == rhs.m_amount; }

    bool operator==(const Mile& mile) const
    { return mile.operator==(*this); }

    bool operator!=(const Yard& rhs) const { ... }
    bool operator!=(const Mile& mile) const { ... }

private:
    Amount m_amount;

private:
    friend class Mile;
};
```

消除重复

从最初我们开始构造Mile和Yard的测试用例时，我们就已经能够感受到重复：

- Mile和Yard的最后两个测试用例几乎是完全重复的；
- 两个类所有的公开方法也都是重复的；
- 更重要的是，当增加一个新的单位时，三个类都需要添加与另外两个类对比的“==”方法，这样的重复会随着单位的增加以指数级快速膨胀。

- 两个类为了相互比较，必须相互知道对方；再增加一个单位的话，三者之间也必须相互知道对方，这些类之间的耦合度会越来越强。

这样的重复方式表明，这并不仅仅是一个重复问题，我们的设计已经误入歧途。

但无论如何，我们先从消除重复入手，来看看会发生什么。

提取基类

观察一下Mile和Yard的类声明。你就会发现两个类的声明方式几乎完全一样。所以，我们可以通过“提取基类”来消除这个重复。那么提取出来的类应该叫什么名字？

首先，观察一下它的公开方法：`==`, `!=`，都是为了对比相等性。那么到底是什么可以对比相等性？再仔细阅读一下需求，很快就能得知：是任意的两个“长度”可以对比相等性。

所以，这个类的职责是“代表一个长度”，所以它应该叫做Length。而原来的四个方法也压缩为两个方法。

```
struct Length
{
    bool operator==(const Length& rhs) const;
    bool operator!=(const Length& rhs) const;
};
```

在我们之前的设计中，Mile和Yard都是用来表示长度的对象，所以，它们应该属于Length的子类。

```
struct Mile : public Length
{
    Mile(const Amount& amountInMile);
};

struct Yard : public Length
{
    Yard(const Amount& amountInYard);
};
```

基类命名

参加OO训练营活动的大多是程序员，或曾经是程序员。经过多年的理工科思维训练，程序员们对于“重复”都有不错的识别能力。只要提出明确的要求，也基本上都能以正确的方式来消除重复。

当基于“消除重复”的目的而提取出一个基类时，类的属性和方法看起来都是正确的，但却给类取了一个匪夷所思，难以理解的名字。

比如，在本例中，这个本该命名为Length的类，却被称做NumericLogic或者Compare。一个带有唯一公开方法move的基类，则被命名为Move。

这并非是因为英语不是我们的母语，因而我们无法找到一个准确的词汇。从上面的例子可以看出，这些命名事实上是根据类所具备的行为而产生的，从这个角度看，这些命名还是挺贴切的。

但行为，只是一个概念根据需求而定义的；在概念不变的情况下，它的行为会随着需求的变化而变化。

所以类的定义应该是先有概念，再定职责，然后才有行为。先看行为特征，然后基于对行为的抽象来给类取名，是一种本末倒置。由此而产生千奇百怪的名字也就毫不意外。

类表示一个实体概念，所以类名一定是名词（“接口”的名字可以是形容词）。没有任何理由用动词给类命名。

听到这个建议之后，Move类的作者马上把类名改成了Movement。

简单地把动词名词化并没有解决真正的问题。真正的问题是，无论用名词或动词，它所代表的概念是否正确。而概念又从哪里找？

首先尝试从“问题域”或者“隐喻模型”中寻找。对于本例，当这样一组重复代码被抽取成一个类时，

```
struct ???  
{  
    bool operator==(const Length& rhs) const;  
    bool operator!=(const Length& rhs) const;  
};
```

不能因为它们具备相等性判断行为，就认为这是一个与“问题域”无关的东西，就可以自己根据它行为创造出Compare这样的概念。而是要考虑在问题域中，哪个概念具备这样的行为，那么那个概念就应该是这个类的名字。而本例中“长度”则是一个这样的概念。

这并非意味着不能创造概念。在很多情况下，问题域都是以具体的概念来描述问题的，并没有进行抽象。这种情况下，我们需要通过创造概念来描述抽象。

必须注意的是，“问题域”的类和抽取出来的公共基类所代表的概念之间具备IS-A的关系。所以，你创造出来的概念名字，也必须经得起“祖母法则”的考验。

现在的问题是，amount应该放在哪里？Mile在构造的时候，传入的是amountInMile，而Yard在构造的时候，传入的是amountInYard；如果放在Length那里，这个amount代表什么含义？

在决策之前，先来考虑一下，我们如何实现Length里的“==”和“!=”操作符：

由于任意两个长度对象之间都可以进行相等性比较，为了能够比较，必须要有一个基准。根据公式 $1\text{ Mile} = 1760\text{ Yard}$ ，我们知道，将amountInMile统统转化为amountInYard，两者即可比较。

所以，我们在Length里存储的amount的含义就是amountInYard。如下：

```
struct Length
{
protected:
    Length(const Amount& amountInYard) {...}

public:
    bool operator==(const Length& rhs) const
    { return m_amountInYard == rhs.m_amountInYard; }

    bool operator!=(const Length& rhs) const {...}

private:
    Amount m_amountInYard;
};
```

建立约束

我们将Length的构造函数设为protected的， why?

因为按照需求，以及我们当前的设计，既然存在Mile和Yard等具体的可以用来表现长度的对象，所以，我们并不希望用户直接使用这个我们为了消除重复而抽取出的基类来构造长度对象。

至于Yard和Mile的实现，只需要通过构造函数将各自都amount都转换成Length所要求的amountInYard即可。

```
Mile::Mile(const Amount& amountInMile)
    : Length(amountInMile * 1760) {}
```

```
Yard::Yard(const Amount& amountInYard)
    : Length(amountInYard) {}
```

通过这次重构，我们了解到：重复，在很多时候并不仅仅是“重复”这么简单。其背后往往意味着重大的设计缺陷，一些核心的本质概念没有得到发现。

如果这样的重复得不到消除，未来的演进会越来越别扭。会进一步带来更强的耦合，快速增长的工作量，更加难以理解的设计和实现，更多的出现bug的机会。

提高表现力

到目前为止，在我们的代码中，大多数命名都看起来不错。

如果仔细追究的话，Mile的实现里有一个magic number。应该定义一个常量来让它的语义更明确。如下：

```
const unsigned int YARDS_PER_MILE = 1760;

Mile::Mile(const Amount& amountInMile)
    : Length(amountInMile * YARDS_PER_MILE)
{}
```

消除冗余

对于Yard和Mile这两个类，它们唯一的职责就是负责把“以自己为单位的数量”，转换成“以Yard为单位的数量”。

如果我们把Yard的代码也重新整理一下，就会得到下面的结果。

```
const unsigned int YARDS_PER_YARD = 1;
```

```

Yard::Yard(const Amount& amountInYard)
    : Length(amountInYard * YARDS_PER_YARD)
{ }

```

现在，把Yard和Mile的实现放在一起对比，就会发现，它们之间只有转换系数的不同。

所以，相对于“类”这种重量级别的元素，我们可以把Mile和Yard转换成更加轻量级的实现：Length构造的一个参数。

```

struct Length
{
    Length(const Amount& amount, unsigned int conversionFactor)
        : m_amountInYard(amount * conversionFactor) {}

    .....

private:
    Amount m_amountInYard;
};

const unsigned int YARDS_PER_MILE = 1760;
const unsigned int YARDS_PER_YARD = 1;

```

提高UI的表现力

我们将用例按照最新的定义重构为：

```

TEST(两个数量相同的以Mile为单位的长度应该相等)
{
    ASSERT_TRUE( Length(3, YARDS_PER_MILE) ==
                    Length(3, YARDS_PER_MILE));
}

```

简单阅读一下用例，就会发现这样的UI存在如下问题：

- 用户需要知道实现策略是向Yard转换；
- Length(3, YARDS_PER_MILE) 不容易理解；

前一点增加了不必要的耦合，后一点则造成糟糕的表现力。

解决耦合问题，我们可以使用策略：“向着稳定的方向依赖”。什么稳定？一个更抽象的名字，而不是与实现细节有关的更具体，更不稳定的名字。

当然，一个好的名字，亦会更有表现力。

所以，如果我们将转换系数定义为更有表现力的常量，上述问题均便会迎刃而解。

```
const unsigned int __YARDS_PER_MILE__ = 1760;
const unsigned int __YARDS_PER_YARD__ = 1;

const unsigned int MILE = __YARDS_PER_MILE__;
const unsigned int YARD = __YARDS_PER_YARD__;
```

用例则调整为：

```
TEST(两个数量相同的以Mile为单位的长度应该相等)
{
    ASSERT_TRUE(Length(3, MILE) == Length(3, MILE));
}

TEST(两个数量不同的以Mile为单位的长度应该不等: 3 Mile != 4 Mile)
{
    ASSERT_TRUE(Length(3, MILE) != Length(4, YARD));
}

TEST(两个数量相同的以Yard为单位的长度应该相等)
{
    ASSERT_TRUE(Length(3, YARD) == Length(3, YARD));
}
```

这次重构，看起来是在消除不必要的元素，但其根本动机，仍然是消除重复。然后，我们进一步通过将参数常量化，改善了UI的表达力。

需求三

问题描述

增加两个新的长度单位，Feet和Inch，用户可以使用它们为单位来表现一个长度。其中：

- 1 Yard == 3 Feet
- 1 Feet == 12 Inch
- 当用Feet为单位来表示长度的时候，精度为1 Feet。
- 当用Inch为单位来表示长度的时候，精度为1 Inch。

需求实现

在这里，我们只列出部分新用例：

TEST (数量符合1:3比例，以Yard为单位的长度和以Feet为单位的长度应该相等)

```
{  
    ASSERT_TRUE(Length(2, YARD) == Length(6, FEET));  
}
```

TEST (数量不符合1:3比例，以Yard为单位的长度和以Feet为单位的长度应该不等)

```
{  
    ASSERT_TRUE(Length(2, YARD) != Length(5, FEET));  
}
```

TEST (数量符合1:12比例，以Feet为单位的长度和以Inch为单位的长度应该相等)

```
{  
    ASSERT_TRUE(Length(2, FEET) == Length(24, INCH));  
}
```

TEST (数量不符合1:12比例，以Feet为单位的长度和以Inch为单位的长度应该不等)

```
{  
    ASSERT_TRUE(Length(2, YARD) != Length(25, FEET));  
}
```

在原来的设计基础上，增加两个长度单位，看起来是件很容易的事情。只需要增加两个向Yard进行转换的系数常量即可。

但，由于Feet和Inch比Yard更小，其转换系数都小于0，所以必须引入浮点数。而浮点数计算最终会因为精度问题而变得更复杂，晦涩。而只有重新设置更小的基准，才可以避免浮点数的引入。

对于当前的需求，我们可以把基准设置为1 Inch。

```
const unsigned int MILE = 1760 * 3 * 12;
const unsigned int YARD = 3 * 12;
const unsigned int FEET = 12;
const unsigned int INCH = 1;
```

提高表现力

准确命名

在增加了两个新的长度单位之后，Length定义里的命名已经有误。

```
struct Length
{
    .....
private:
    Amount m_amountInYard;
};
```

之前的设计中，我们设定的基准是Yard，Length的属性名m_amountInYard就是用来表达这个事实。在当时的情况下，做出这样的选择是为了增加表现力。现在我们的基准已经改成了Inch，这样的命名则变成了一种误导。

一种显而易见的改法是，把它命名为m_amountInInch。但之前的变化已经表明：这样的命名方式，依赖了一个不稳定的事。

向着稳定的方向依赖。我们应该给它一个更反应本质，更加抽象的名字。由于基准单位的变化，这个名字一直在变化。所以m_amountInInch和m_amountInYard其实代表着“以基准单位计数的量”。

```
struct Length
{
    Length::Length(const Amount& amount, unsigned int conversionFactor)
        : m_amountInBaseUnit(amount * conversionFactor) {}

    bool Length::operator==(const Length& rhs) const
    { return m_amountInBaseUnit == rhs.m_amountInBaseUnit; }

private:
    Amount m_amountInYard;
    Amount m_amountInBaseUnit;
};
```

准确命名

现在，我们再来看看这些常量定义，会发现：每个常量都定义了一个单位向基准单位的转换系数，但其名字却与“转换系数”没有任何关系。

这会让阅读者感到困惑。这种不直观背后隐藏着由于设计方式所带来的“**隐性知识**”。为了消除这种困惑，我们要把知识“**显性化**”：让命名更加准确。

```
const unsigned int INCH_CONV_FACTOR = 1;
const unsigned int FEET_CONV_FACTOR = 12;
const unsigned int YARD_CONV_FACTOR = 3 * 12;
const unsigned int MILE_CONV_FACTOR = 1760 * 3 * 12;
```

由于Length类构造函数的第二个参数也明确的说明了那是一个“**转换系数**”，所以把这些常量当作Length的第二个构造参数，也就顺应一致，直观易懂。

此时，用户在构造一个Length对象的时候，就变成了这样：

Length(12, MILE) → **Length(12, MILE_CONV_FACTOR)**

而这样的表现形式，相对于之前的方式，要差的多。

不仅仅是因为多了几个字母，更重要的是，这样的表达让用户了解到一个设计细节：第二个参数是一个转换系数。你的设计细节与用户有什么关系？

而是原来的方式更加直观：这里创造了一个长度为12 Mile的长度对象。这才是用户真正要表达的意图。

改回原来的名字，就要忍受那些名字“**名不副实**”。使用现在的名字，则会造成用户表达的“**名不副实**”。

由于后者的名字来自需求，前者的名字基于设计，所以这种矛盾的产生，是由于“**问题域**”和“**设计域**”没有实现无缝的对接。

为了能够解决这一矛盾，我们需要分析一下这个问题，看看如何能够把二者通过某种机制关联起来。

首先，我们要保证UI的合理性，即提供Length(12, MILE)这样的写法。在这种情况下，MILE究竟意味着什么？答案是“**长度单位**”。

那么如何把用户所关注的概念“**长度单位**”，与实现所需要的概念“**转换系数**”联系在一起？

以类取代类型码³

在“面向对象”范式下，我们很容易想到类：增加一个用来代表“长度单位”概念的类，再把“转换系数”当作类的内部实现隐藏起来，就可以自然的把两个概念联系在一起。

```
struct LengthUnit
{
    LengthUnit(unsigned int conversionFactor);

    unsigned int getConversionFactor() const;

private:
    unsigned int m_conversionFactor;
};
```

Length的实现：

```
struct Length
{
    Length( const Amount& amount
           , const LengthUnit& unit)
        : m_amountInBaseUnit(amount * unit.getConversionFactor()) {}

    bool operator==(const Length& rhs) const {...}
    bool operator!=(const Length& rhs) const {...}

private:
    Amount m_amountInBaseUnit;
};
```

此时，那些常量定义就变为：

```
const LengthUnit& getINCH();
const LengthUnit& getFEET();
const LengthUnit& getYARD();
const LengthUnit& getMILE();

#define INCH (getINCH())
#define FEET (getFEET())
#define YARD (getYARD())
#define MILE (getMILE())
```

实现代码：

```
const unsigned int INCH_CONV_FACTOR = 1;
```

³ 《重构》(Martin Fowler) 8.13

```
const unsigned int FEET_CONV_FACTOR = 12;
const unsigned int YARD_CONV_FACTOR = 3 * 12;
const unsigned int MILE_CONV_FACTOR = 1760 * 3 * 12;

const LengthUnit& getINCH()
{
    static LengthUnit unit(INCH_CONV_FACTOR);
    return unit;
}

const LengthUnit& getFEET()
{
    static LengthUnit unit(FEET_CONV_FACTOR);
    return unit;
}
.....
```

常量

刚刚从C转到C++开发的程序员，除了手边需要放一本《C++ Primer》作为随时翻阅的参考书之外，一定还要有两本必读秘技《Effective C++》和《More Effective C++》。

而《Effective C++》的开篇条款，就是建议使用常量定义来代替同样性质的宏定义。比如把：

```
#define MAX_NUM (10)
```

替换为

```
const unsigned int MAX_NUM = 10;
```

从而避免宏所带来的tricky问题。

不过从外观看，常量定义只是一个不能修改的变量定义而已。没有static的修饰，程序员很容易把它误认为是一个全局变量。

而事实上每一个在函数和类之外定义的常量都默认以static修饰，即只在“编译单元”可见，并没有跨编译单元链接的可能。除非它带有extern的修饰，才会变成一个全局常量。

所以，如果你在源文件中定义一个常量，则无须担心符号污染的问题，并无必要把它放在匿名名字空间里（放进去也没有任何问题）。

重要的是那些放在头文件里的常量定义。由于头文件可被多个编译单元使用，而常量定义又具备static语义，那么同一常量定义就会在不同编译单元内生成不同的实例。如果你试图在不同编译单元内对同一个常量取地址，得到的结果则肯定是不一样的。如果你的设计对此做了像全局变量一样的假设，那么运行时就会遇到麻烦。

常量的这个性质，对于整数，浮点数等基本类型的常量，并没有什么太大问题。但如果你在头文件定义对一个对象，比如：

```
const Foo NIL_FOO = Foo(0);
```

就会让每个编译单元都会存在一份NIL_FOO的实例。这就会对内存产生不确定的消耗，而这样的消耗是没有必要的浪费，尤其当Foo是个很大的对象时。

所以，我们希望NIL_FOO（而不是Foo）在全局内只有一个实例。于是我们把它的定义移入了某个源文件中，并加上了extern修饰，这样就可把它变成了一个全局常量。

```
extern const Foo NIL_FOO = Foo(0);
```

然后，我们在头文件里进行了声明，以确保其它编译单元可以使用它。

```
extern const Foo NIL_FOO;
```

但这样做存在一个潜在的风险。如果另外一个编译单元定义了一个引用了NIL_FOO的常量，比如：

```
const Bar NIL_BAR = Bar(&NIL_FOO);
```

一旦Bar的构造函数调用了Foo的方法，那么系统的运行时行为就不可确定。因为NIL_FOO此时可能还没有构造。

C++规范仅仅保证在同一个编译单元内定义的常量会按照定义的顺序进行构造。跨编译单元常量的构造顺序，属于未定义行为。

为了确保一个常量被引用时已经被构造，我们可以引入“工厂函数”，实现机制可以利用局部静态变量语义：

```
const Foo& getNIL_FOO()
{
```

```
    static Foo nil(0);
    return nil;
}
```

然后在头文件里，进行如下的声明：

```
const Foo& getNIL_FOO();
const Foo& NIL_FOO = getNIL_FOO();
```

由于C++可以确保在同一编译单元内定义的常量的构造顺序，所以你不用担心在直接引用NIL_FOO时它尚未初始化。

不幸的是，如果你跨模块间接引用NIL_FOO时，仍然存在风险。比如，一个模块存在这个一个函数：

```
const Foo& getFoo()
{
    .....
    return NIL_FOO;
}
```

而在另外一个模块定义了这样一个常量：

```
const Foo& FOO = getFoo();
```

由于两个常量跨模块，因此没有人可以保证常量FOO的初始化一定在NIL_FOO之后，因此有相当的概率FOO被初始化时，getFoo函数返回一个非法的引用。

所以在这种情况下，宏才是我们的救世主。比如，以上述代码为例：

```
#define NIL_FOO getNIL_FOO()
```

由于函数getFoo引用了宏NILL_FOO，在预处理阶段即被更换为getNIL_FOO()，在随后的阶段，无论那个模块，直接或间接的调用getFoo，总是会引起getNIL_FOO的调用，这样就能保证一个合法的实例可以得到创建。

永远不要把一个常量赋值为“函数调用”；永远不要直接定义一个非原子类型的静态实例（函数内除外）或全局变量，切记切记！！

依赖管理

在LengthUnit的定义里，我们看到了Getter方法，这时一种破坏封装的强烈的坏味道。

```
struct LengthUnit
{
    .....
    unsigned int getConversionFactor() const;
};
```

之前，我们使用Getter方法，是为了快速的让这个重构可以工作。现在是重点解决它的时候了。

管理学中在分析问题的时候，经常用一个被称做“5 Why”的分析法。通过不断的问“为什么”，以找到问题发生的根源。

这个方法同样可以被用来解决Getter问题——

1. 问题：为什么Length需要从LengthUnit通过Getter方法得到“转换系数”？
答案：Length需要使用这个“转换系数”将原有单位的数量转换成“基准单位”的数量。
2. 问题：既然Length真正需要的是“转换”这件事，那为什么不让LengthUnit来提供“转换”这个服务，却要自己来通过Getter来获取“转换系数”，然后自己来做转换？
答案：Sounds Good...
3. 结论：向稳定的方向依赖。让Length来依赖一个更本质的需要：“转换”，相对于依赖“转换系数”这个细节，其更稳定。

于是，我们可以将代码重构为：

```
struct LengthUnit
{
    unsigned int getConversionFactor() const;
    Amount toAmountInBaseUnit(const Amount& amount) const;
    .....
};
```

而Length的实现则重构为：

```
Length::Length
( const Amount& amount
, const LengthUnit& unit)
: m_amountInBaseUnit(unit.toAmountInBaseUnit(amount)) {}
```

GETTER vs. TDA

一些C++程序员，甚至Java程序员在定义一个类的时候，只要定义一个属性，就习惯性的定义一对setter/getter。

这些人中，大抵都做过一些面向过程的程序，比如C语言的开发。所有的组合数据结构都是结构体／共用体，任何时候，只要需要，就直接操作访问结构体中的成员。带着这种根深蒂固的思维习惯，即便转到以其它范式来进行思考的语言，如果不让其可以自由的访问数据，这程序根本就没办法写下去。

即便为之穿上函数的外衣，也并没有改变想直接访问和操作成员的本质意图。尽管setter/getter确实比直接访问数据要有弹性，但它们的抽象层次并没有比数据成员高太多。

而对于数据的毫无限制的访问是完全违背“高内聚，低耦合”原则的。按照高内聚，关系紧密的事物应该放到一起，如果一段逻辑就是围绕一个类的数据成员展开的，那么这段逻辑就应该属于这个类。

我们以Rectangle为例，如果它最初的定义为：

```
struct Rectangle
{
public:
    Rectangle(unsigned int width, unsigned int height)
        : m_width(width), m_height(height) {}

    unsigned int getWidth() const
    { return m_width; }

    unsigned int getHeight() const
    { return m_height; }

private:
    unsigned int m_width, m_height;
};
```

如果现在一个客户需要计算矩形的面积，那么它的相关代码如下：

```
unsigned long long area = rect.getWidth() * rect.getHeight();
```

这样的写法，不仅降低了内聚型，还提高了耦合，因为客户更加本质的需要是求面积，而不是更加具体的width和height。相对于求面积，对于width和

height的依赖更加刚性一些，因为如果只是求面积，Rectangle可以有多种实现方式。

如果把这段代码移动到Rectangle，则提高了Rectangle的内聚性。

```
struct Rectangle
{
public:
    Rectangle(unsigned int width, unsigned int height)
        : m_width(width), m_height(height) {}

    unsigned int getWidth() const
    { return m_width; }

    unsigned int getHeight() const
    { return m_height; }

    unsigned long long getArea() const
    { return m_width * m_height; }

private:
    unsigned int m_width, m_height;
};
```

这样的设计方式，其背后的原则称为“Tell, Don't Ask”：一个对象的Client不要Ask对象的内部状态，然后自己根据得到的状态来做一件事情；而是要Tell对象，让它来做这件事情。

Tell, Don't Ask原则是保护封装，提高内聚，降低耦合的重要手段。毫不夸张的说，它是使用防备Getter问题的“必杀技”。本例只是这一原则的最简单使用场合。我们在随后的问题中还会不断探讨这一原则的其它运用模式。

另外，“5 Why”分析法，是对这一原则进行运用的具体操作方法。每次当一个类将要提供一个Getter接口的时候，就以“为什么对方需要知道这个内部状态？”作为第一个问题。

拿Rectangle为例，答案会是“需要计算它的面积”；然后下一个问题是“为什么不能把计算面积作为Rectangle的职责？”，一般而言，问题到此就结束了。

对于更加复杂的情况，用户可能有其它的原因，然后再问“为什么”，直到得出结论。如果确实得到一个合理的原因，那就提供Getter；否则，换一个更加合理的方案。在大多数情况下，都会发现Getter是一个更糟糕的选择。

消除重复

从下面的代码中，我们可以看到每个常量的表达式都是转换系数的连乘。而某些转换系数在不同的表达式里重复出现。

```
const unsigned int INCH_CONV_FACTOR = 1;
const unsigned int FEET_CONV_FACTOR = 12;
const unsigned int YARD_CONV_FACTOR = 3 * 12;
const unsigned int MILE_CONV_FACTOR = 1760 * 3 * 12;
```

为了消除这些重复，我们可以将表达式定义如下：

```
const unsigned int INCH_CONV_FACTOR = 1;
const unsigned int FEET_CONV_FACTOR = 12 * INCH_CONV_FACTOR;
const unsigned int YARD_CONV_FACTOR = 3 * FEET_CONV_FACTOR;
const unsigned int MILE_CONV_FACTOR = 1760 * YARD_CONV_FACTOR;
```

这种表达方法，不仅仅消除了重复，同时也提高了表现力：每一个表达式都是对需求中公式的直观表示。

然而，相对于之前的表达方式，新表达式却增加了耦合度。

但这样的耦合度可以为我们所接受。因为，一则，一个单位一旦被定义，再被删除的概率很低（因为很多用户已经使用）；其二，即便一个单位被删除，我们修改的成本也很低。

并且这样的耦合度反过来也给我们带来了一定的好处：如果我们现在添加了一个更加小的“基准单位”，除了INCH_CONV_FACTOR之外，其它的定义都不受影响。否则，每个转换系数都需要改变。

提高表现力

此时，衍生出来的一个问题是：“谁是基准单位”这个知识并没有在代码中得到明确的表示。代码的维护者，必须经过更多的理解之后，才能获取这个知识。

如果我们把常量定义进行如下修改，Inch作为“基准单位”的事实就得以彰显：

```
const unsigned int BASE_UNIT_CONV_FACTOR = 1;

const unsigned int INCH_CONV_FACTOR = BASE_UNIT_CONV_FACTOR;
const unsigned int FEET_CONV_FACTOR = 12 * INCH_CONV_FACTOR;
const unsigned int YARD_CONV_FACTOR = 3 * FEET_CONV_FACTOR;
const unsigned int MILE_CONV_FACTOR = 1760 * YARD_CONV_FACTOR;
```

知识的显性化

代码是对知识的表现。在软件开发过程中，当用语言或者文字来描述一个设计的时候，一件事物是存在的。但在实现代码中，却根本找不到任何与此有关的明确表现。这就会造成设计和实现的缝隙。而这部分在代码中没有得到直接体现，却隐藏在逻辑或者细节之中的知识，则被称为“隐性知识”。

“隐性知识”的概念来自于“知识管理”领域。它原本的意思是，有些真实存在的知识，但是难以用文字或语言来清晰的表达，所谓“只可意会，不可言传”。

在软件设计层面，我们借用了这个概念，指一个确定存在的，来自于“需求”或“设计”的概念，约束，逻辑等知识，但是没有用“代码语言”来进行明确的表述。

“隐性知识”会造成代码的阅读者需要花费更多的精力才能“猜测”或“推理”出它的存在。所以程序设计者需要把设计思路中的概念、假设以及来自需求的逻辑、约束，内在联系等都明确的表达出来，这个过程被称做“知识的显性化”。

“知识的显式化”是增强代码“表达力”的重要手段。你或许会认为程序员都有足够的逻辑思维能力，最终能够靠自己推理出这些“隐性知识”。但所谓“表达力”，就是希望阅读者可以更加快速的理解你的设计。如果能够让别人一秒种就可以理解，就不要让别人用两秒。

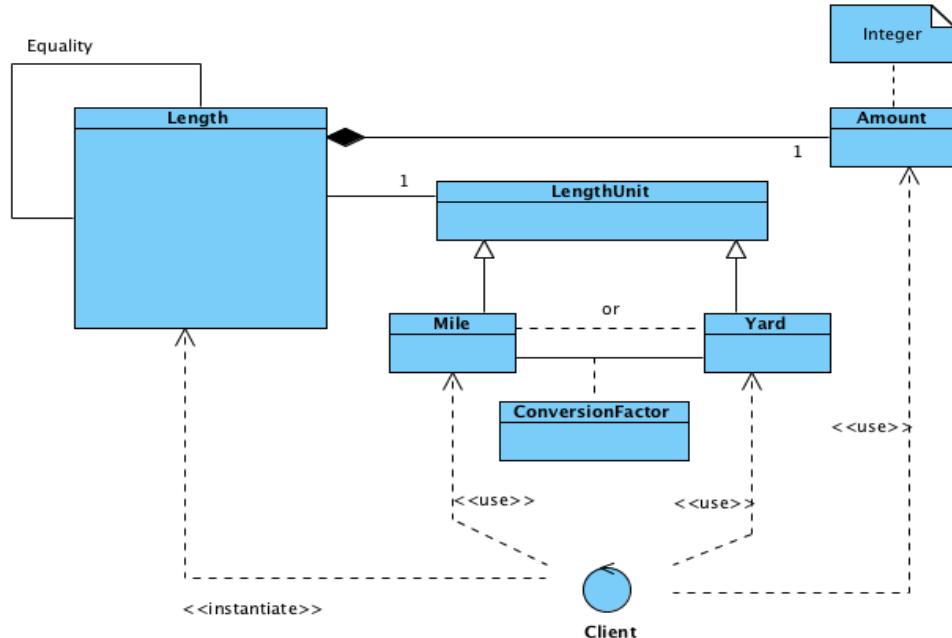
基于分析的设计

我们到目前为止，一直是通过“编码并重构”的方法，在“简单设计”原则的指导下，通过对“代码坏味道”的不断识别和消除，来确保设计的合理性。

而通过形式化的分析方法，得到正确的“概念模型”，并在“概念模型”的指导下进行设计，则有助于我们在早期就能得到正确的概念，避免较多的弯路。

这里并不打算介绍OOA。而是以一个已经得到的“概念模型”为起点，通过它来指导我们设计。

根据“需求二”得到的简要概念模型如下图所示：



从这个模型中我们得知，“用户”可见的概念是Length, Mile, Yard和Amount；所以，这四个概念是“边界类”。其它概念都应该隐藏于系统边界之后。

初步设计

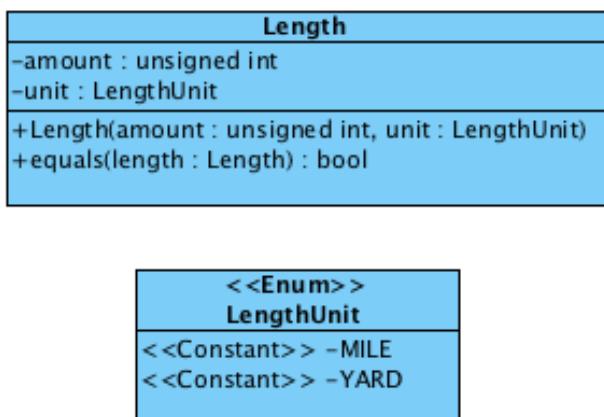
概念Amount存在一个“非负整数”约束，所以，Amount可以定义为unsigned int类型。

`Length`是用户要表示的概念，而它与另外两个概念`Amount`和`LengthUnit`都有关联，所以我们用类来表现它。

`LengthUnit`和`Mile`, `Yard`之间是“泛化关系”。在转换到“实现模型”时，对于C++程序员，一个简单的选择，是把`LengthUnit`实现为枚举类型，而`Mile`和`Yard`则作为枚举的实例。

为了构造一个`Length`对象，用户需要指明`Amount`和`LengthUnit`，所以，它们应该作为`Length`构造函数的参数。

这样，我们得到一个如下所示的设计：



初步实现

基于我们的设计模型，实现代码为：

```
enum LengthUnit
{
    MILE,
    YARD
};

struct Length
{
    Length(const Amount& amount, LengthUnit unit);

    bool operator==(const Length& rhs) const;
    bool operator!=(const Length& rhs) const;

private:
    Amount toYard() const;

private:
    Amount m_amount;
    LengthUnit m_unit;
```

```

};

namespace
{
    Amount mileToYard(const Amount& amountInMile)
    { return amountInMile * 1760; }
}

Amount Length::toYard() const
{ return (m_unit == MILE) ? mileToYard(m_amount) : m_amount; }

bool Length::operator==(const Length& rhs) const
{ return toYard() == rhs.toYard(); }

```

设计演进

“需求三”要求增加新的单位FEET和INCH。

按照我们之前的设计，只需要在枚举LengthUnit里加上新的实例FEET和INCH。然后在Length中加入对它们的处理即可。

```

enum LengthUnit
{
    MILE,
    YARD,
    FEET,
    INCH
};

unsigned int Length::getConversionFactor() const
{
    switch(m_unit)
    {
        case INCH: return INCH_CONV_FACTOR;
        case FEET: return FEET_CONV_FACTOR;
        case YARD: return YARD_CONV_FACTOR;
        case MILE: return MILE_CONV_FACTOR;
    }

    return BASE_UNIT_CONV_FACTOR;
}

```

我们的代码写的干净漂亮，作者得意，读者满意，一切似乎都很完美.....除了Length的稳定性问题——

任何时候只要扩展一个新的LengthUnit实例，Length的代码就不得不修改。

从“概念模型”可知，Length和LengthUnit是两个不同的概念。如果Length总是随着LengthUnit实例的变化而变化，那么Length就紧密的耦合了所有LengthUnit实例。

如果我们能把LengthUnit相关概念的逻辑移出Length，则Length就对当前系统的LengthUnit具体实例一无所知，那么它就无须随着LengthUnit实例的变化而变化。

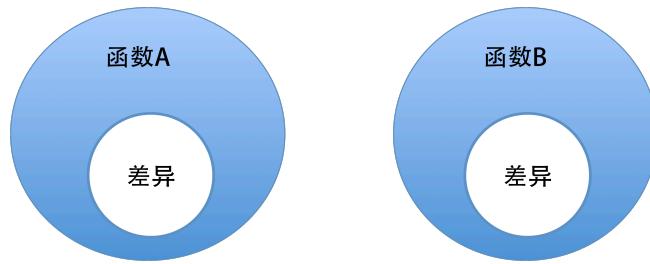
开放封闭原则

我们在开发过程中，经常能够碰到这样的情况，当来了一个新需求之后，只需要到原有的一个或多个类中，增加一些条件判断，并在条件判断中加入自己的逻辑即可完成。

在不想修改原有代码的情况下，程序员的选择则是：拷贝一份原来的代码，然后把中间与新需求不匹配的逻辑删除，把实现新需求的逻辑加上。

当碰到上述两种情况时，你很可能遭遇了“回调型重复”。

所谓“回调型重复”，是指两个逻辑单位（模块，类，函数）只有中间的一部分是不同的，其它代码完全重复。



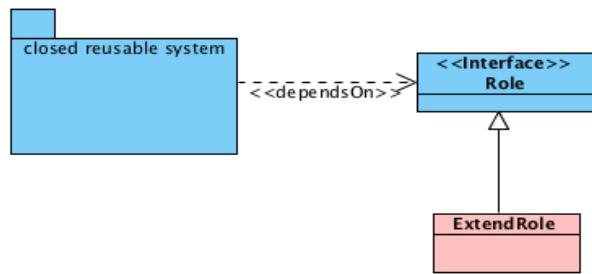
为了避免这样的重复，经常发生的应对方式是，到原有代码中加入新的条件判断，经常以if-else或者switch-case的方式存在。

之所以将其称之为“回调型重复”，是因为除了直接修改，copy-paste-modify之外，还可以将差异的部分以回调的方式，由用户在外部实现差异。

这就是“开放封闭原则”：一个设计，应该对于修改是封闭的，对于扩展是开放的。

所谓对于修改是封闭的，是指在“回调型重复”中，重复的部分应该可重用，不应该修改它。同时又提供了回调机制，让用户实现差异，这就是所谓的“对于扩展是开放的”。

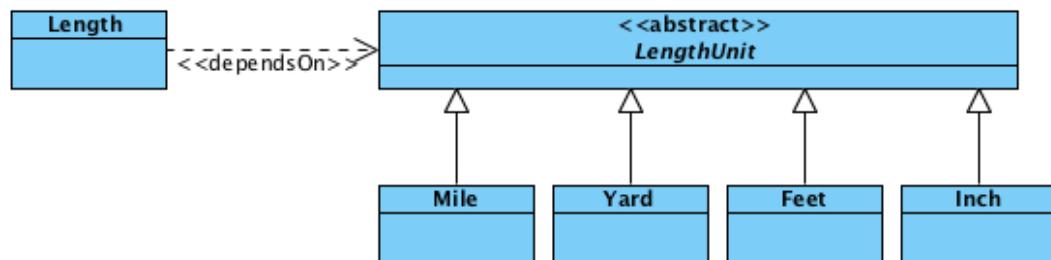
“分离关注点”是达到“封闭修改”的关键，而“抽象”则是“开放扩展”的手段。



“开放封闭原则”是解决可重用，可扩展问题的重要原则。其应用范围和粒度也非常广泛。`xUnit`测试框架，`jMock`，`eclipse`，大型项目的构建脚本，比如`makefile`的组织，都是对这一原则的应用。

设计模式中的策略，模版方法等即是对这一原则的直接体现；而观察者，状态等模式中也闪耀着这一原则的思想。

为了让我们的设计对于`LengthUnit`的扩展是开放的，那么`LengthUnit`必须是一个“抽象概念”。



然后我们将代码修改如下：

```
struct LengthUnit  
{  
    virtual Amount
```

```

        toAmountInBaseUnit(const Amount& amount) const = 0;

    virtual ~LengthUnit() {}

};

struct Length
{
    Length(const Amount& amount, const LengthUnit* const unit);

    Amount Length::toAmountInBaseUnit() const
    { return m_unit->getConversionFactor(m_amount); }

private:
    Amount m_amount;
    const LengthUnit* const m_unit;
};

```

上面的代码构成了一个“开闭”系统。其“开闭”的方向是LengthUnit实例的任何增删。然后，我们将现有的LengthUnit实例“扩展”到系统中。

```

struct Mile : public LengthUnit
{
    Amount toAmountInBaseUnit(const Amount& amount) const
    { return amount * BASE_UNITS_PER_MILE; }
};

struct Yard : public LengthUnit
{
    Amount toAmountInBaseUnit(const Amount& amount) const
    { return amount * BASE_UNITS_PER_YARD; }
};

struct Feet : public LengthUnit
{
    Amount toAmountInBaseUnit(const Amount& amount) const
    { return amount * BASE_UNITS_PER_FEET; }
};

struct Inch : public LengthUnit
{
    Amount toAmountInBaseUnit(const Amount& amount) const
    { return amount * BASE_UNITS_PER_INCH; }
};

```

这是一个教科书般标准的“开放封闭”实现。我们可以任意的扩展我们的LengthUnit子类，而不会影响“封闭”的部分。

消除重复

不幸的是，即便你粗略的审视一下四个LengthUnit子类的实现，也会发现它们的代码模式几乎一模一样。

唯一差别的是各自常量定义的值，而每个常量都是在定义各自向Inch转换的系数。如果把重复的方法toAmountInBaseUnit向上移动，就变成了LengthUnit的实现，其中的差异“转换系数”事实上可以通过LengthUnit构造时作为参数传入。

```
struct LengthUnit
{
    LengthUnit(unsigned int conversionFactor)
        : m_conversionFactor(conversionFactor) {}

    Amount toAmountInBaseUnit(const Amount& amount) const
    { return amount * m_conversionFactor; }

private:
    unsigned int m_conversionFactor;
};
```

在这种设计下，我们封闭了更多的东西：LengthUnit的实现。

封闭部分提供的扩展机制则变为：通过定义不同的“转换参数”来实例化LengthUnit。所以，封闭部分当前所依赖的“抽象”是“转换参数”。

由此，我们可以明确：“抽象”并不意味着一定是一个“抽象类”或“接口”。一个常量定义也可以作为一种抽象。常量的赋值过程，就对这个“抽象”进行“具体化”的过程。

另外我们需要明确的是：在“概念模型”中的泛化关系，并不意味着在实现模型中，一定是类的继承关系。在实现模型中，枚举定义，字符串，整数，或者任意类型的常量定义都是一种“泛化关系”。

ENUM, IF-ELSE & SWITCH-CASE

C/C++中的枚举有一些很有趣的特性：潜在的整数类型，常量值的自动分配等等。尤其是第二点，让程序员可以摆脱对于“枚举常量”背后数值的维护，而将注意力集中在每个“枚举常量”所表现的概念本身（抽象和数据的分离）。另外，由于枚举本身是一个类型，所以，每个“枚举常量”都是所属“枚举类型”的具体实例。比下面的枚举定义：

```
enum LengthUnit
{
    INCH,
    FEET,
```

```
YARD,  
MILE  
};
```

它近似的等价于：

```
const int LengthUnit INCH = 0;  
const int LengthUnit FEET = 1;  
const int LengthUnit YARD = 2;  
const int LengthUnit MILE = 3;
```

所以INCH, FEET等枚举常量都是LengthUnit的实例。

这些特征，让枚举成为C/C++程序员的最爱，每次当需要对同一概念的不同实例进行区分的时候，枚举总是会首先蹦入C/C++程序员的脑海。尤其对于C程序员，枚举更是用来进行抽象表达为数不多的便利武器之一。

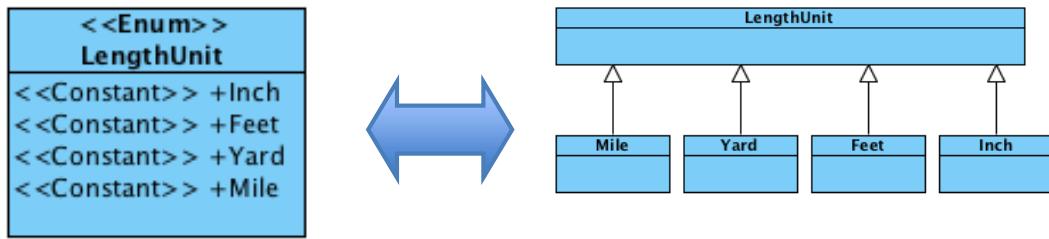
而与枚举伴生的常常是代码中的if-else结构。比如一个：

```
ConversionFactor getConversionFactor()  
{  
    if (m_unit == INCH) {...}  
    else if(m_unit == FEET) {...}  
    else if(m_unit == YARD) {...}  
    else if(m_unit == MILE) {...}  
    else {...}  
}
```

或者它的等价switch-case结构：

```
ConversionFactor getConversionFactor()  
{  
    switch(m_unit)  
    {  
        case INCH: ... break;  
        case FEET: ... break;  
        case YARD: ... break;  
        case MILE: ... break;  
        default: ...  
    }  
}
```

无论是if-else，还是switch-case，当其对同一枚举的不同枚举常量进行差异化处理的时候，那就说明每个枚举常量都有对应于自己的差异化行为。当把枚举类型看作概念，枚举常量对应差异行为的时候，那么这就面向对象领域中的“泛化关系”。



既然枚举和switch-case放在一起本质上是一个“泛化关系”，那么我们就可以将其转化为以类为基础的泛化实现方式：

1. 首先将枚举类型定义为基类
2. 将各个枚举常量作为子类
3. 然后整个switch-case结构所对应的函数作为虚方法放入基类；
4. 并把switch-case里每个枚举常量所对应的差异化行为放在各个子类的方法实现中。

两种设计从功能上是等价的，但其带来的影响却有本质上的差异。

首先，switch-case造成相关函数更加不稳定。因为新的枚举常量有可能会被加入出来；或者，某个枚举常量所对应的行为会发生变化。

其次，对于同一组枚举常量，当存在多个switch-case时，每次添加一个新的枚举常量，会造成多处查找和修改，不内聚。

而使用类的方式来解决，由于类实例和类行为天然的绑定关系，原来所有需要使用switch-case结构的地方都永久性的变成了一条语句：

```
instanceOfBaseClassType->doSth();
```

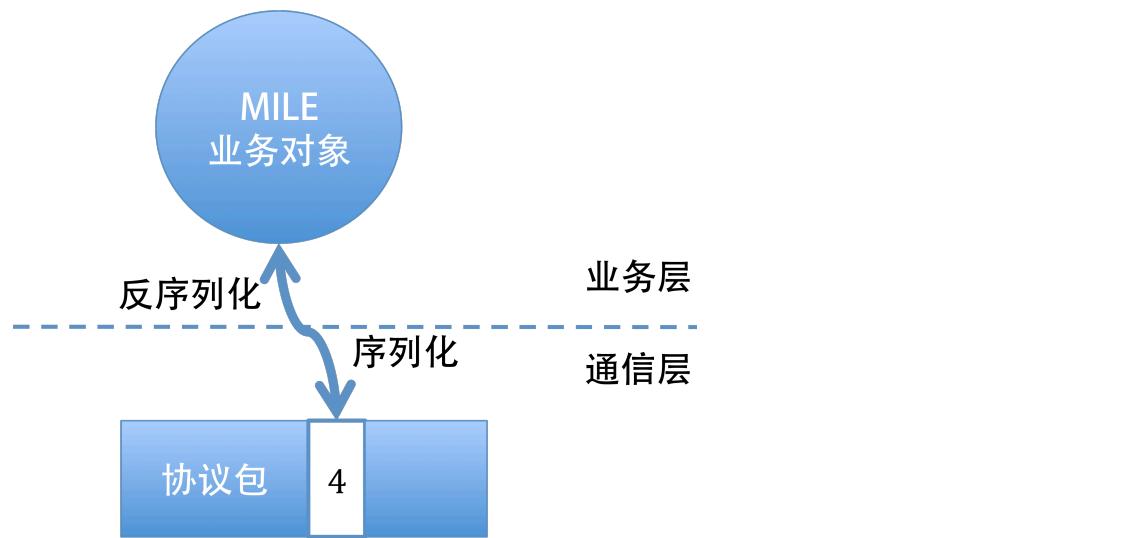
这将会提升代码的可重用性。

另外，当需要新增一种新的枚举常量时，类的方式则只需要增加一个新的类，然后在此类中实现所有的差异化，无须四处修改。

用枚举来表现概念，之所以会产生上述问题，是因为枚举的类型为整数，而它所代表的概念从本质上与整数没有任何关系，所以，必须通过switch-case或者if-else将两种类型之间进行映射。在这种情况下，枚举事实上是一种画蛇添足，有害无益的中间层。

这并非意味着应该禁止在C++编程中使用枚举。暂且不谈枚举与表现概念无关的用法；即便用于表现概念，枚举值的整数特性，表现概念的能力，在一些场合下也有其无可替代的优势。比如，网络协议，数据存储等。协议定义，数据存储都是用数据来表现概念的，而不是用带有行为的类。而枚举在软件层面用来表现协议中的数据定义则是一种很好的表现方式。

需要注意的是，既然枚举仅仅是为了表现网络协议定义，或者数据存储格式定义；那么在业务逻辑层面，就不应该再使用枚举，而是仍然使用业务对象，即类的实例来表现相应的概念。两者之间的转化则在系统边界通过类的“序列化”和“反序列化”来完成。



需求四

问题描述

这套库发布之后，用户只能使用既有单位（Mile, Yard, Feet, Inch），而不应该有能力创建新的单位；以避免用户由于某些原因创建一些现实中不存在的单位，从而让系统变得不可理解。

功能实现

这时一个非功能性需求，简单的说，就是不允许用户创建额外的LengthUnit对象。

这样的需求，当使用C++语言来实现的范式时：让LengthUnit的构造函数变成私有的。这样，在LengthUnit的外部，就无法再创建任何LengthUnit对象。即便是通过继承也不行。

```
struct LengthUnit
{
private:
    LengthUnit(unsigned int conversionFactor);

public:
    Amount toAmountInBaseUnit(const Amount& amount) const;

public:
    static const LengthUnit& getMILE();
    static const LengthUnit& getYARD();
    static const LengthUnit& getFEET();
    static const LengthUnit& getINCH();

private:
    unsigned int m_conversionFactor;
};

#define MILE (LengthUnit::getMILE())
#define YARD (LengthUnit::getYARD())
#define FEET (LengthUnit::getFEET())
#define INCH (LengthUnit::getINCH())
```

这里，之所以把MILE, YARD等常量定义在LengthUnit外面，是为了让UI仍然和过去保持一致。否则，用户就需要写成类似于Length(3, LengthUnit::MILE)这样更加罗嗦的方式。

SLUG IN C++

这种处理问题的方法，被称为Slug模式。中文可以翻译成“投币模式”。之所以被称做“投币模式”，你可以理解为自动售货机只接受它所指定的币种。

从实现的角度看，则是通过禁止用户创建某种类型的对象，而只能使用这个类提供的预先创建的静态实例。

这样的方法对于Java等语言，是一种非常有效的模式。因为Java发布的结果可以完全是二进制的。用户没有直接的办法来修改一个第三方库的任何实现。

但对于C++语言，这样的约束其实很容易被破解。原因在于，即便一个库以二进制的方式发布给用户，但用户可见的头文件也必须被一起发布。这样用户就有了直接修改头文件的权利。在可以修改原有类声明的前提下，恶意的用户可以在不破坏原有代码逻辑的前提下，通过修改类声明来达到自己的目的。

所以，“投币模式”对于C++语言有效的前提是，用户不会去修改别人发布的类声明。作为一个合格的程序员，必须通过实例化，继承和委托等合法方式来使用一个类，这是一种契约关系。不遵从契约，直接修改类声明的方式，是一种自做聪明，不负责任的“黑客”行为。

实现代码如下：

```
const unsigned BASE_UNIT_CONV_FACTOR = 1;

const unsigned INCH_CONV_FACTOR = BASE_UNIT_CONV_FACTOR;
const unsigned FEET_CONV_FACTOR = 12 * INCH_CONV_FACTOR;
const unsigned YARD_CONV_FACTOR = 3 * FEET_CONV_FACTOR;
const unsigned MILE_CONV_FACTOR = 1760 * YARD_CONV_FACTOR;

const LengthUnit& LengthUnit::getMILE()
{
    static LengthUnit unit(MILE_CONV_FACTOR);
    return unit;
}

const LengthUnit& LengthUnit::getYARD()
{
    static LengthUnit unit(YARD_CONV_FACTOR);
    return unit;
}
```

```
const LengthUnit& LengthUnit::getFEET() {...}
const LengthUnit& LengthUnit::getINCH() {...}
```

消除重复

可以看出，上述代码有着强烈的重复模式：除了名字的差别，各个Unit的定义是完全一样的。宏的标识符连接功能则是解决这类问题的灵丹妙药。

```
const unsigned int BASE_UNIT_CONV_FACTOR = 1;

///////////////
#define CONV_FACTOR(unit) unit ## _CONV_FACTOR

///////////////
#define DEFINE_UNIT(unit, eq, conversionFactor, refUnit) \
    const unsigned int CONV_FACTOR(unit) = \
        conversionFactor * CONV_FACTOR(refUnit); \
    const LengthUnit& LengthUnit::get##unit() \
    { \
        static LengthUnit unitInstance(CONV_FACTOR(unit)); \
        return unitInstance; \
    }

#define DEFINE_BASE_UNIT(unit) DEFINE_UNIT(unit, eq, 1, BASE_UNIT)
```

然后，我们就可以利用这套宏来定义所有的单位：

```
DEFINE_BASE_UNIT(INCH);
DEFINE_UNIT(FEET, =, 12, INCH);
DEFINE_UNIT(YARD, =, 3, FEET);
DEFINE_UNIT(MILE, =, 1760, YARD);
```

而头文件中的重复亦可以用宏的方式解决：

```
///////////////
#define __UNIT_TYPE const LengthUnit&
#define __UNIT_GETTER(unit) get##unit()

#define __UNIT_SLUG_DECL(unit) static __UNIT_TYPE __UNIT_GETTER(unit)
#define __UNIT_CONST_DECL(unit) (LengthUnit::__UNIT_GETTER(unit))

/////////////
struct LengthUnit
{
    .....
public:
    __UNIT_SLUG_DECL(MILE);
    __UNIT_SLUG_DECL(YARD);
    __UNIT_SLUG_DECL(FEET);
    __UNIT_SLUG_DECL(INCH);

};
```

```
//////////  
#define MILE __UNIT_CONST_DECL(MILE)  
#define YARD __UNIT_CONST_DECL(YARD)  
#define FEET __UNIT_CONST_DECL(FEET)  
#define INCH __UNIT_CONST_DECL(INCH)
```

SLUG & SINGLETON

如果你足够敏感，就会发现Slug模式和Singleton模式很相似。它们都属于“实例创建约束”相关的模式。尤其在C++的实现中，都是通过“静态工厂方法”来获取所需对象实例。

但它们的应用场景却有很大的不同：

- Slug模式用于创建允许客户使用的实例。实例通常是常量，每个实例都有特定的性质和明确的含义，性质上近似于枚举，但却比枚举强大的多。
- Singleton模式用于约束实例的数量，并不关注实例的可变性；如果可以创建多个实例，不同实例对于客户而言是透明而无差别的。

正是由于这样的性质差异，造成了客户在应用两个模式时，代码编写方式上有很大的差异。

对于Slug模式，用户通常无须直接调用某个具体Slug实例的方法，而是作为参数传递给与具体实例无关的函数，由其进行具有多态性质的计算。

对于Singleton，由于没有实例上的差别，客户更加关注的是自己所需要的服务，而不是哪个实体在提供服务；所以经常在客户代码中看到下面的写法：

```
Foo::getInstance() -> doSth();
```

理论上，这种写法的好处是，Singleton潜在的具备提供多个实例的可能性，每次调用getInstance()都可能返回不同的实例，而背后则是个对象池，Singleton负责选择返回对象实例的策略控制。

但事实上，这种场景及其罕见。绝大多数情况下，Singleton都是一个名副其实的Singleton，而不是Multiton。设计者之所以选择使用Singleton的内在原因，也只是想确保一个类只存在一个全局访问的实例，根本不存在创建多个实例的任何动机和可能。

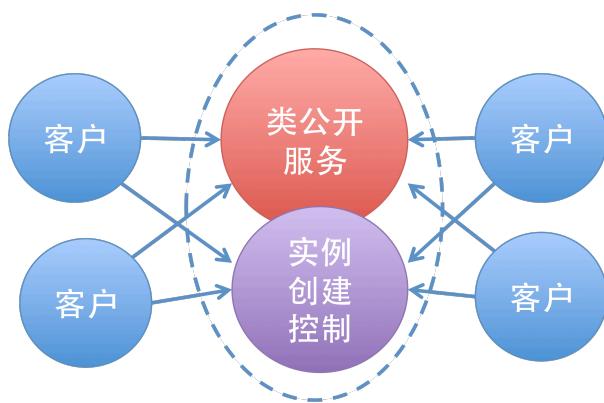
所以，如果Singleton确实只有一个实例的情况下，Singleton本质上是一个全局变量，但以上述使用方式却比直接使用全局变量还要糟糕。不仅造成了每处客户代码都要编写一个很长的表达式，还让客户代码对类型Foo产生了更多的依赖（多依赖了一个getInstance）。

Singleton是一个充满陷阱的模式。但也是看起来最“直观”的，进而最容易误用和滥用的模式（Singleton是我在现实C++项目中见到的应用最广泛的模式）。

首先，我们需要承认Singleton模式的价值：我们确实需要控制某个类型实例的数量。

但它与类本身的服务有着不同的关注点。而对于这个类型的用户，它关注的是类本身的服务，并不想关注提供服务的究竟是哪个实例，也并不关注类在系统中存在的实例数量。

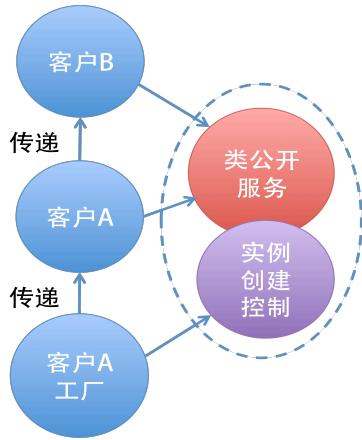
如果这两个关注点耦合在一起，那么所有Singleton的客户都将被迫耦合这两个关注点。



从分离关注点的角度，我们应该把两方进行隔离：

- 对于类的用户，我们应该以参数的形式把类的实例传入；
- 而类自身，则作为一个存在实例创建约束的实体；

经过这样的关注点分离，我们就可以让客户只关注类的公开服务，而实例的创建和引用则由工厂，或者客户类的客户通过参数传递的方式来完成。如下图所示：



另外，这样做的另外一个好处是让Singleton的全局性本质进行了“局部化”⁴，从而降低了“全局性”所带来的邪恶影响。

但上述所谈的是必须使用Singleton的场景下，我们可以通过分离关注点来降低Singleton所带来的负面作用。而现实中的大多数场景，Singleton的应用却往往是设计存在问题的征兆。

比如，一些系统往往是由一些大型甚至超大型职责不明的“上帝类”（God Object）组成（我见过的最大的类有1千多个方法），而这些的名字通常是XxxHandler, XxxProcessor, XxxManager, XxxLayer（由于职责不明，只好给予这样的名字，但并非所有这样命名的类都是职责不明的）。

由于这些类都具备过多的职责，几乎整个系统都会访问到它们，所以从内在本质上它们就具备“全局性”影响，这种情况下，Singleton反而是上佳的选择。

但这种“上佳”性是颇带有反讽的味道。因为Singleton的引入并非事物的“内在复杂度”而造成的，深层次设计问题所带来的“偶发复杂度”才让它承担起这种角色。

一般而言，这样的系统都是“挂羊头，卖狗肉”，披着“面向对象”的外衣，设计却处处体现着“过程”思想：那些“上帝类”所提供的公开方法，结合Singleton，其本质上就是想提供一个C语言函数而已。

但对于开发者而言，既然在使用C++，那么提供一个C语言函数就会显得很不专业，从而很不“体面”。然后，调用者就不得不承受更大的代价，每次都要编写类似于下面的代码：

⁴ 见Kent Beck《实现模式》3.2.1

```
Foo::getInstance() -> doSth();
```

事实上，用C++的面向对象元素，比如类，来进行面向过程的设计，会让系统更加僵化，更加难以维护和重构。因为类相对于自由函数，是个封闭性更强的结构。被不合理放置在一起的自由函数，可以很容易的被重新部署到不同的模块（文件），而无须改变函数名字。但对于C++，当一个类被拆分为不同的类的时候，Singleton的类名部分也必须被修改，所有依赖原有Singleton的客户代码，也必须分门别类的进行修改，这种维护上的代价会让重构者望而生畏。

并非在所有的情况下Singleton都是邪恶的，一些场合下客户代码直接使用Singleton反而是比较恰当和务实的选择。比如，Logger，由于系统每个地方都可能会用它，所以“全局性”和“单一实例”都是它的内在本质。而分离关注点则会带来大面积的参数传递，从而造成过高的成本，这就完全抵消并超出“关注点分离”所带来的益处。

最后，指明但不说明的一个结论是：Slug并不具备Singleton的邪恶特性。如果你感兴趣，可以思考一下其中的原因。

需求五

问题描述

任意两个长度对象可以进行加法运算。例如：

- 13 Inch + 11 Inch = 2 Feet
- 3 Feet + 2 Yard = 3 Yard

编写用例

下面列出的是Length的最新用例。

```
TEST(两个长度相加, 得到一个新的长度: 13 INCH + 11 INCH = 2 FEET)
{
    ASSERT_TRUE(Length(13, INCH) + Length(11, INCH) ==
                Length(2, FEET));
}

TEST(两个长度相加, 得到一个新的长度: 3 FEET + 2 YARD = 3 FEET)
{
    ASSERT_TRUE(Length(3, FEET) + Length(2, YARD) == Length(3, FEET));
}
```

满足功能

对于加法操作，在C++社区，已经有非常成熟的实现模式。代码如下：

```
struct Length
{
    .....

private:
    friend Length operator+(const Length& lhs, const Length& rhs);
};

Length operator+(const Length& lhs, const Length& rhs)
{
    unsigned int amountInBaseUnit =
        lhs.amountInBaseUnit +
        rhs.amountInBaseUnit;

    return Length(amountInBaseUnit, INCH);
}
```

消除重复

粗略的审视一下，新写的代码中似乎没有什么重复。

但如果更仔细一些，就会发现一个bad smell。在加法实现的最后一行里有这样一个表达式：

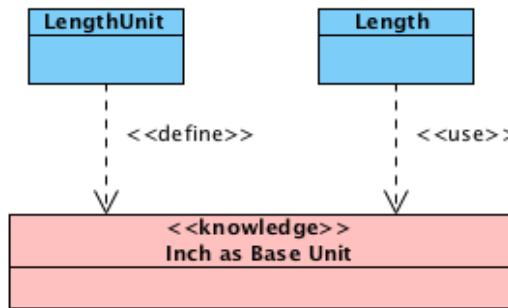
```
Length(amountInBaseUnit, INCH);
```

Length构造的第二个参数，使用了“当前基准单位”：INCH。

“Inch是Length的当前基准单位”是当前系统的一项知识。系统中现有两个地方对这个知识进行了直接的描述：

- LengthUnit
- “加法运算”所在的Length模块

二者都直接依赖了这个知识。当这个知识发生变化的时候，LengthUnit和Length都需要进行修改。所以，这是一种“重复”。



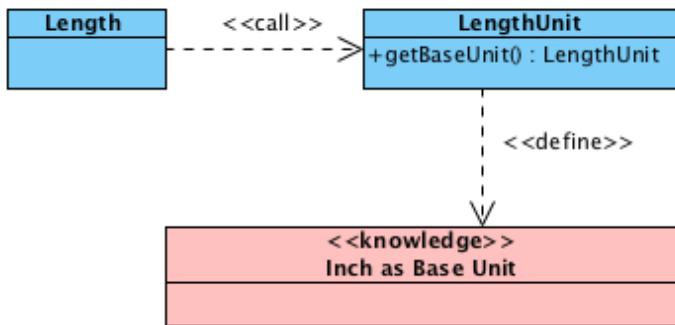
按照DRY原则，这个知识只应该存在“一个”明确而权威的表示。

在“slug模式”的实现里，`LengthUnit`负责提供所有合法的`LengthUnit`实例。所以，`LengthUnit`应该是这个知识的唯一归宿。

但“加法运算”确实需要这个知识，怎么办？

稍加考虑，就会明白：“加法运算”需要的是“当前基准长度单位”这个抽象概念，而不是“Inch作为当前基准长度单位”这个不稳定的知识。

向着稳定的方向依赖，`Length`应当依赖于`LengthUnit`提供的“当前基准长度单位”这个概念。



实现代码如下：

```

struct LengthUnit
{
    .....

public:
    static const LengthUnit& getBaseUnit();
    .....

};

Length operator+(const Length& lhs, const Length& rhs)
{
    unsigned int amountInBaseUnit =
        lhs.amountInBaseUnit +
        rhs.amountInBaseUnit;

    return Length(amountInBaseUnit, LengthUnit::getBaseUnit());
}

```

现在只有`LengthUnit`耦合了“当前基准长度单位是`Inch`”这个知识。也可以说，我们把这个知识“内聚”在`LengthUnit`模块。

下面我们看看`LengthUnit`的实现：

```

DEFINE_BASE_UNIT(INCH);

const LengthUnit& LengthUnit::getBaseUnit()
{ return getINCH(); }

```

尽管我们之前通过`LengthUnit::getBaseUnit()`方法将“`Inch`作为基准长度单位”这个知识内聚在`LengthUnit`模块内；但是，按照现有的实现，在`LengthUnit`模块内部，对这个知识，仍然存在着两种表达：

1. `INCH`对象作为“基准单位”；
2. `INCH_CONV_FACTOR=BASE_UNIT_CONV_FACTOR`

为了消除这种重复，我们必须想办法把两种表达关联起来，或者合二为一。

```
const unsigned int BASE_UNIT_CONV_FACTOR = 1;

const LengthUnit& LengthUnit::getBaseUnit()
{
    static LengthUnit unit(BASE_UNIT_CONV_FACTOR);
    return unit;
}
```

这种实现工作良好。它创造了一个新的对象，不是INCH，却完全等价于INCH。

提高表达力

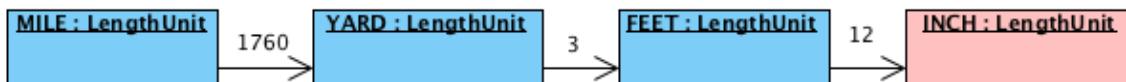
这种间接的等价对象有点让我不安，尽管代码中已经没有明显的重复，但它似乎依然违背了DRY原则，因为现在系统中有两个对象表现同一事物。这种数据上的重复，总是可能会造成潜在的问题。所以，我更希望LengthUnit::getBaseUnit()返回的就是INCH对象，而不是另外一个等价对象。

造成表示同一事物的两个对象的原因在于，我们把转换关系放在了LengthUnit的外部。

这种方式带来了一定的好处：不同的LengthUnit之间没有任何关系。它们的关系已经在外部通过常量定义表达过了。

但也正是由于各个LengthUnit之间是独立的，我们亦无法借助LengthUnit对象来自动推导出谁是“基准单位对象”，所以只能借助常量定义再重新生成一个。

如果我们把转换关系维持在LengthUnit内部，建立起如下图所示的关系，我们就可以通过任意一个LengthUnit对象获取到“基准单位对象”。



下面是相应的代码实现：

```
struct LengthUnit
{
private:
    LengthUnit()
        : m_conversionFactor(1)
        , m_baseUnit(0) {}

    LengthUnit( unsigned int conversionFactor
```

```

        , const LengthUnit& baseUnit)
: m_conversionFactor(conversionFactor)
, m_baseUnit(&baseUnit) {}

Amount toAmountInBaseUnit(const Amount& amount) const
{ return amount * getConversionFactor(); }

const LengthUnit& getBaseUnit() const
{
    if(m_baseUnit == 0) return *this;
    return m_baseUnit->getBaseUnit();
}

private:
    unsigned int getConversionFactor() const
{
    if(m_baseUnit == 0) return 1;
    return m_baseUnit->toAmountInBaseUnit(m_conversionFactor);
}

private:
    unsigned int m_conversionFactor;
    const LengthUnit* const m_baseUnit;
};


```

这个实现通过两个不同的构造函数来区分“基准对象”和非基准对象。而内部实现，则通过基准对象指针是否为空指针来判断一个实例是基准对象与否。

这样的改变对Length的实现也带来影响：为了能够通过一个LengthUnit实例得到“基准单位”，Length需要存储一个LengthUnit。

```

struct Length
{
    Length(const Amount& amount, const LengthUnit& unit);

    bool Length::operator==(const Length& rhs) const
    { return toAmountInBaseUnit() == rhs.toAmountInBaseUnit(); }

private:
    Amount Length::toAmountInBaseUnit() const
    { return m_unit.toAmountInBaseUnit(m_amount); }

private:
    Amount m_amount;
    const LengthUnit& m_unit;
    .....
};


```

而加法运算则相应的实现为：

```
Length operator+(const Length& lhs, const Length& rhs)
```

```

{
    unsigned int amountInBaseUnit =
        lhs.toAmountInBaseUnit() +
        rhs.toAmountInBaseUnit();

    return Length(amountInBaseUnit, lhs.m_unit.getBaseUnit());
}

```

引入空对象*

如果继续检视，会发现LengthUnit的实现中有两段模式重复的代码。

```

unsigned int LengthUnit::getConversionFactor() const
{
    if(m_baseUnit == 0) return 1;
    return m_baseUnit->toAmountInBaseUnit(m_conversionFactor);
}

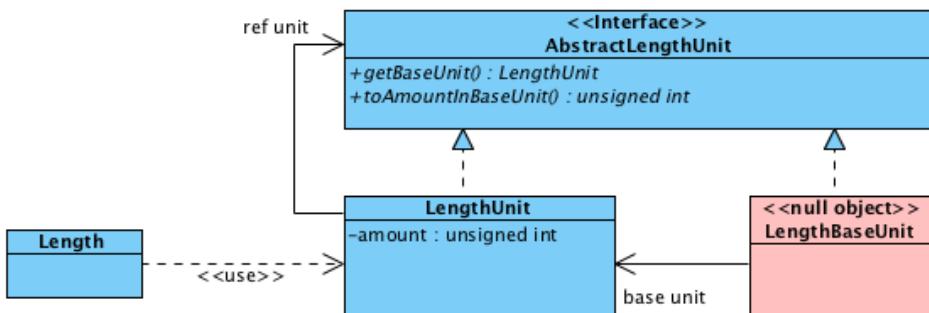
const LengthUnit& LengthUnit::getBaseUnit() const
{
    if(m_baseUnit == 0) return *this;
    return m_baseUnit->getBaseUnit();
}

```

这两段代码都是if-else结构，并且if的条件表达式是一样的，都是基于m_base_unit是否为空。

if-else已经是一种掩盖概念的坏味道，更何况还有两处重复。

我们可以通过“空对象模式”把if-else合二为一，两个问题就同时消失了。



```

///////////////////////////////
struct AbstractLengthUnit
{
    virtual Amount
    toAmountInBaseUnit(const Amount& amount) const = 0;
    virtual const LengthUnit& getBaseUnit() const = 0;

    virtual ~AbstractLengthUnit() {}
};

```

```

///////////////////////////////
struct LengthUnit : public AbstractLengthUnit
{
private:
    LengthUnit();
    LengthUnit( unsigned int conversionFactor
               , const LengthUnit& baseUnit);

    ~LengthUnit();

    .....

private:
    unsigned int m_conversionFactor;
    bool m_isBaseUnit;
    const AbstractLengthUnit* const m_baseUnit;

};

namespace
{
    struct LengthBaseUnit : public AbstractLengthUnit
    {
        LengthBaseUnit(const LengthUnit& baseUnit)
            : m_baseUnit(baseUnit) {}

        Amount toAmountInBaseUnit(const Amount& amount) const
        { return amount; }

        const LengthUnit& getBaseUnit() const
        { return m_baseUnit; }

private:
    const LengthUnit& m_baseUnit;
};

LengthUnit::LengthUnit()
    : m_conversionFactor(1)
    , m_isBaseUnit(true)
    , m_baseUnit(new LengthBaseUnit(*this)) {}

LengthUnit::LengthUnit
    ( unsigned int conversionFactor
    , const LengthUnit& baseUnit )
    : m_conversion_factor(conversionFactor)
    , m_isBaseUnit(false)
    , m_baseUnit(&baseUnit) {}

LengthUnit::~LengthUnit()
{ if(m_isBaseUnit) delete m_baseUnit; }

```

现在，那两个函数都变得更加简洁：

unsigned int

```
LengthUnit::getConversionFactor() const
{ return m_baseUnit->toAmountInBaseUnit(m_conversionFactor); }

const LengthUnit&
LengthUnit::getBaseUnit() const
{ return m_baseUnit->getBaseUnit(); }
```

这个例子并不是一个典型的“空对象模式”的使用场景。因为：

- Null Object一般定义为常量或者“Singleton”；
- 抽象类AbstractLengthUnit的客户就是它的实现类LengthUnit；
- 由于本来LengthUnit依赖的是具体类LengthUnit，在C++语言环境下，为了支持空对象模式，我们被迫提取了一个抽象类AbstractLengthUnit；
- AbstractLengthUnit对于LengthUnit的客户Length是完全无用的，但却由于它需要被LengthUnit实现，所以需要放在头文件中，最终让Length对其产生不必要的物理依赖；
- 由于LengthUnit的客户Length通过getBaseUnit()获取的必须是LengthUnit，所以，抽象类AbstractLengthUnit反过来依赖了它的子类LengthUnit，这在正常的设计下，应该是一种竭力避免的“坏味道”。

由于这些问题的存在，在这个例子中使用“空对象模式”，事实上有些overskill。而在典型的“空对象模式”的应用场景，这类问题都不应该存在。

NULL OBJECT

在编程中，最让人生厌的一项活动就是到处对异常参数进行检查。一是造成重复，因为到处充斥着类似的代码；二这样的代码量会占据一个系统相当的比重，让系统更加复杂和难以理解。

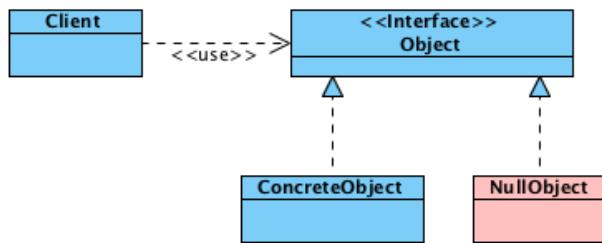
空指针，就是C/C++程序员必须应对的场景之一。尽管在C++中，程序员可以使用“引用”来代替不必要的指针。但并非所有的场合都可以使用引用：

- 初始引用对象未知（引用必须在定义时赋值）；
- 值可以修改（一个引用一旦被赋值，就不能再引用另外一个对象）
- 空对象是一种合理状态；

这种情况下，尤其在“防卫式编程”的价值观驱动下，程序员会在每个函数入口对传入的指针进行检查，否则，对空指针的直接访问会造成系统崩溃，这可是一个无法承受的重大事故。

为了让代码更加干净，在面向对象范式下，人们想起了用“多态”来屏蔽指针状态的差异。让一个空指针在某个边界位置指向一个“空对象”。随后的客户代码就无须再关心一个指针指向的是正常对象，还是“空对象”，两者已经通过更高层的抽象达到了概念上的一致。

一个典型的“空对象模式”如下：



这种做法，和通过“策略模式”或“状态模式”来消除switch-case背后的思想是一致的，都是通过发掘if-else，或switch-case背后的概念，来消除重复，让主体逻辑看起来更加简洁一致，从而增强软件的可理解和可维护性。“空对象模式”事实上是“策略模式”或“状态模式”的特例。

分离关注点

如果你仔细阅读了LengthUnit的新实现，你就会注意到代码中仍然有这样一段条件逻辑：

```
LengthUnit::~LengthUnit()
{ if(m_isBaseUnit) delete m_baseUnit; }
```

这是因为，两种对象的生命周期是不一样的：

- “空对象”由LengthUnit创建，它的生命周期和LengthUnit相同；
- 其它对象由构造时传入，生命周期和LengthUnit不同。

所以析构时必须对这两类对象进行区分处理。为此，我们还使用了一个bool型变量m_isBaseUnit来记录析构的处理模式。

经过分析，我们发现`m_isBaseUnit`的存在仅仅是为了标识`m_baseUnit`是否应该删除有关。而它对于其它业务逻辑的计算是没有帮助的。所以，它与`m_baseUnit`的紧密关系要大于与其它数据成员的紧密关系。

按照“高内聚”的原则，只有关联程度同样紧密的事物才应该被放到一起。否则，它们就是不同的“关注点”，现在我们将其分离。

```
template <typename T>
struct DelegatePointer
{
    DelegatePointer(const T& pointer, bool shouldDelete = false)
        : m_pointer(pointer)
        , m_shouldDelete(shouldDelete)
    {}

    T operator->() const
    { return m_pointer; }

    T operator*() const
    { return m_pointer; }

    ~DelegatePointer()
    { if(m_shouldDelete) delete m_pointer; }

private:
    DelegatePointer(const DelegatePointer& p);
    DelegatePointer& operator=(const DelegatePointer& p);

private:
    T m_pointer;
    const bool m_shouldDelete;
};
```

`LengthUnit`的实现可以修改为：

```
struct LengthUnit : public AbstractLengthUnit
{
private:
    .....
    ~LengthUnit();

private:
    bool m_isBaseUnit;
    const AbstractLengthUnit* const m_baseUnit;

    DelegatePointer<const AbstractLengthUnit* const> m_baseUnit;
    .....

};

LengthUnit::LengthUnit()
    : m_conversionFactor(1)
```

```

        , m_baseUnit(new LengthUnitBase(*this), true) {}

LengthUnit::LengthUnit
( unsigned int conversionFactor
, const LengthUnit& baseUnit )
: m_conversionFactor(conversionFactor)
, m_baseUnit(&baseUnit) {}

LengthUnit::~LengthUnit()
{ if(m_isBaseUnit) delete m_baseUnit; }

```

DELEGATE POINTER

在“依赖注入”思想大行其道的今天，对象的组合往往由Object，Factory和Client三个职责独立的角色来合作完成。其中Factory负责将Object注入Client来完成协作对象的组装。

在这种情况下，Object生命周期的管理就变得非常重要，我们不希望Client对其负责，因为这会增加Client的职责，增加Client对Object生命周期知识的耦合，并且会造成不同Client的代码重复。

既然Factory负责对象的创建，它必然知道对象的生命周期，所以，它理应将其管理起来，并且把这个知识隐藏起来。

一种方法是，Factory负责提供两个函数，一个函数负责创建对象，一个函数负责释放对象。正所谓“谁申请，谁释放”的原则。比如：

```

struct ObjectFactory
{
    Object* createObject();
    void destroyObject(Object* object);
};

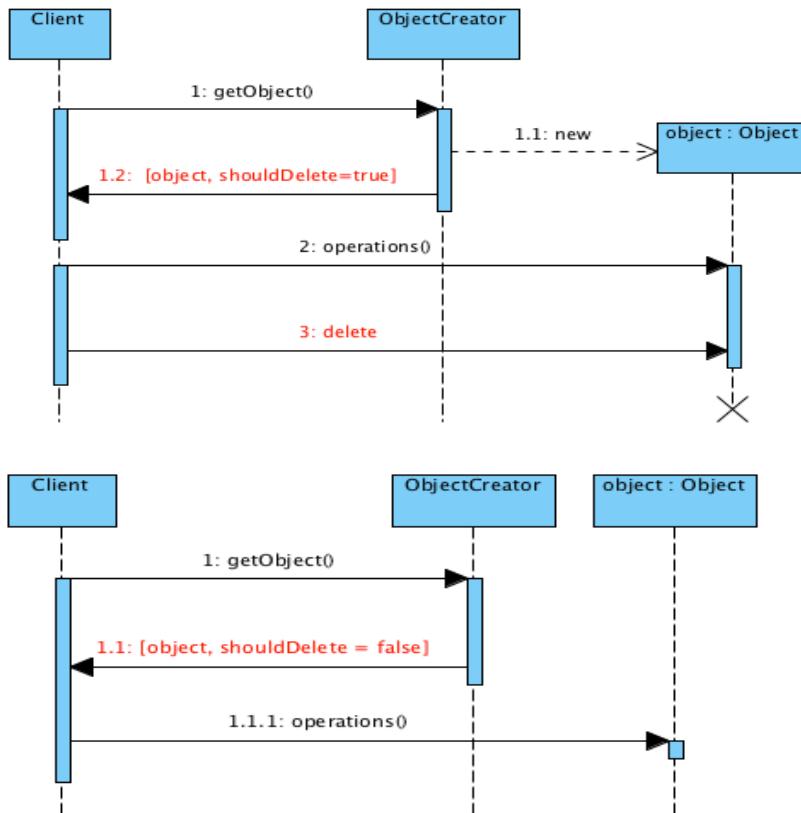
```

这种方法可以解决对象生命周期管理责任分配的问题。但却会造成Client和Factory的耦合。尽管可以通过“依赖注入”，让Client无须调用Factory::createObject，但Client最后需要销毁Object的时候，必须调用Factory::destroyObject方法。

为了彻底解耦Factory和Client，我们只能让Factory告知Client是否一个对象应该被删除。这就是第二种方案：

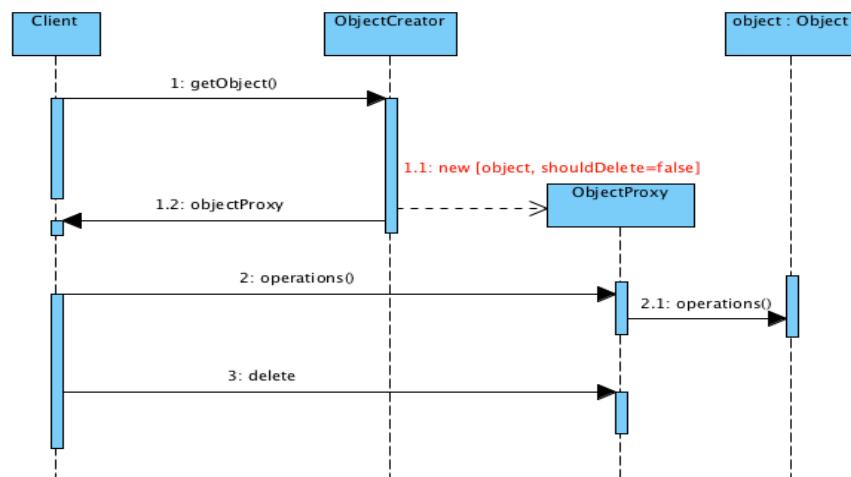
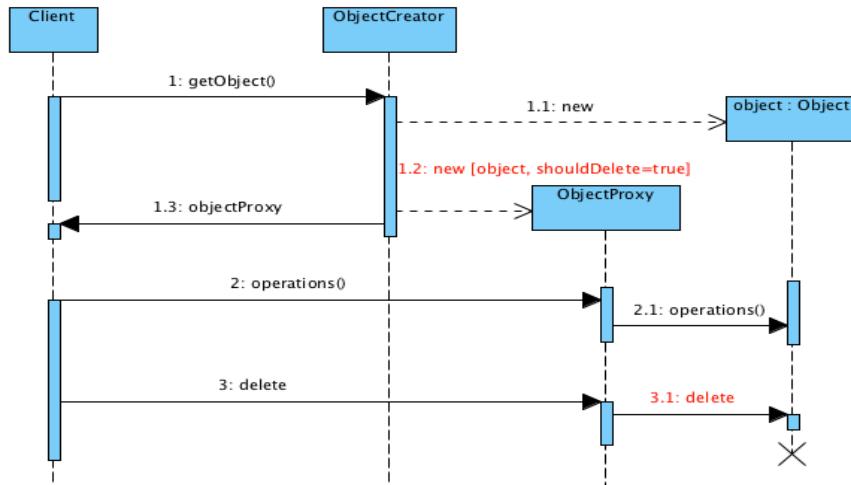
由于Factory负责创建Object，我们将其称为ObjectCreator。

在指针被赋值时，ObjectCreator就通过一个bool型变量标记了这个指针将来在无用时是否需要销毁。指针的“使用者”在不需要时，就根据ObjectCreator的指示来对指针进行销毁。下图表现了两种不同的场景。



但这种模式的缺点是，Client仍然需要关注ObjectCreator的指示。这就会造成所有的Client都要对此进行关注和管理，而其管理的模式一定都是重复的。

如果我们把ObjectCreator返回的信息组合 `[object, shouldDelete]` 作为一个对象，把bool型的指示封装在对象内部，由这个对象来管理，则Client将不再需要这样的关注。



从图中可以看出Client的所有交互在两种场景下完全一致。背后的不一致完全由ObjectCreator, 以及由它所创造的对象ObjectProxy管理起来。

这就是“代理指针”模式，其中的ObjectProxy扮演的就是代理指针的角色。下面就是在这种应用模式下的一种实现。

```

template <typename T>
struct DelegatePointer
{
    DelegatePointer(const T& pointer, bool isComposite = false)
        : m_pointer(pointer)
        , m_isComposite(isComposite)
    {}

    T operator->() const
    { return m_pointer; }

    T operator*() const
    { return m_pointer; }
}

```

```

void destroy()
{
    if(m_isComposite)
    {
        delete m_pointer;
        m_pointer = 0;
    }
}

private:
T m_pointer;
const bool m_isComposite;
};

```

对于指针的生命周期的管理，还有很多经典的模式。比如：Smart Pointer, Shared Pointer, Scoped Pointer等等。

不同的模式有着不同的应用场景。和Smart Pointer等模式不同的是，DelegatePointer的职责仅限于：按照Creator或者Factory的指示来管理对象的生命周期。Client在使用它的时候，其使用模式和一个普通指针没有任何不同。比如，当你把一个DelegatePointer实例赋值给另外一个DelegatePointer实例后，如果你对其中一个实例执行了destroy操作，那么另外一个实例的合法性则被破坏，但DelegatePointer并不负责帮助你管理这一点。

而它相对于Smart Pointer的优势则在于，

- Client可以明确的删除一个不再需要的对象，而无须考虑是否还有其它对于这个对象的引用；
 - 它所管理的对象并不要求必须通过new方法从堆中获取，这会给Factory更大的实现灵活度和简便性，而这种灵活度和简便性在某些约束下至关重要。
-

PIML模式*

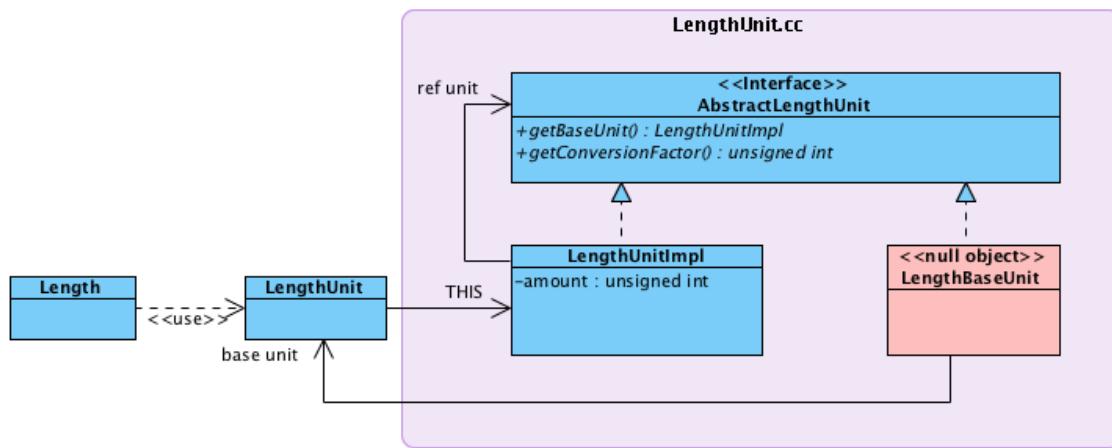
之前在谈到“空对象模式”的时候，指出了实现中的一些问题，其中一个是抽象类AbstractLengthUnit所带来的“物理耦合”问题。

AbstractLengthUnit是由“Null Object”引入的，作为一种具体实现方式，我们不希望它为外部可见，从而带来不必要的耦合。

基于PIMPL模式，LengthUnit仅仅应在头文件声明保留了两种信息：

- “slug模式” 约束（外部不能创建新的LengthUnit实例，以及允许用户使用的LengthUnit实例）
- Length需要调用的公开方法。

除此之外的所有代码元素都被移到LengthUnit模块内部。如下图所示。



需要特别指出的是，原本抽象接口AbstractLengthUnit需要被LengthUnit实现，但它现在变为由LengthUnitImpl实现。也就是说，它由一个外部可见的接口，彻彻底底的变成了一个内部实现细节。

所以，尽管PIMPL模式的初衷是为了帮助改善“物理耦合”，但某些情况下也会带来“逻辑耦合”的改善。

```
struct LengthUnit
{
    .....
private:
    friend struct LengthUnitImpl;
    LengthUnitImpl* THIS;
};

namespace {

    struct AbstractLengthUnit
    {
        virtual unsigned int getConversionFactor() const = 0;
        virtual const LengthUnit& getBaseUnit() const = 0;

        virtual ~AbstractLengthUnit() {}
    };
}
```

```

///////////
const unsigned int BASE_UNIT_CONV_FACTOR = 1;

/////////
struct LengthBaseUnit : public AbstractLengthUnit
{
    LengthBaseUnit(const LengthUnit& baseUnit)
        : m_baseUnit(baseUnit) {}

    unsigned int getConversionFactor() const
    { return BASE_UNIT_CONV_FACTOR; }

    const LengthUnit& getBaseUnit() const
    { return m_baseUnit; }

    const LengthUnit& m_baseUnit;
};

/////////
struct LengthUnitImpl : public AbstractLengthUnit
{
    LengthUnitImpl(const LengthUnit& baseUnit)
        : m_conversionFactor(BASE_UNIT_CONV_FACTOR)
        , m_baseUnit(new LengthBaseUnit(baseUnit), true) {}

    LengthUnitImpl( unsigned int conversionFactor
                  , const LengthUnit& refUnit)
        : m_conversionFactor(conversionFactor)
        , m_baseUnit(refUnit.THIS) {}

    unsigned int getConversionFactor() const
    { return m_refUnit->toAmountInBaseUnit(m_conversionFactor); }

    const LengthUnit& getBaseUnit() const
    { return m_refUnit->getBaseUnit(); }

    unsigned int m_conversionFactor;
    DelegatePointer<const AbstractLengthUnit* const> m_refUnit;
};

/////////
LengthUnit::LengthUnit()
    : THIS(new LengthUnitImpl(*this)) {}

LengthUnit::LengthUnit
( unsigned int conversionFactor
, const LengthUnit& refUnit )
    : THIS(new LengthUnitImpl(conversionFactor, refUnit)) {}

LengthUnit::~LengthUnit()
{ delete THIS; }

Amount
LengthUnit::toAmountInBaseUnit(const Amount& amount) const

```

```
{ return amount * THIS->getConversionFactor(); }

const LengthUnit&
LengthUnit::getBaseUnit() const
{ return THIS->getBaseUnit(); }
```

PIMPL

C++继承了C语言的编译模型，即通过头文件来放置一个模块的公开声明。其它模块通过include指令将自己所需模块的声明通过预处理展开到自己的编译单元。

理论上，头文件里只需要放置一个模块公开的内容，但由于C++的类无法像C函数那样分为定义部分和声明部分，所以，整个类，除了函数实现部分，都不得不放在头文件。

而头文件里的内容越多，客户对其所产生的物理耦合也越强。这种物理耦合会导致每次重新编译客户代码时都要消耗更多的时间，反过来，它的任何变更也都会引起客户代码的完全重新编译。

所以，我们希望仅仅在头文件里放入必要的内容。所以，我们通过分析类的构成和放置在头文件的必要性：

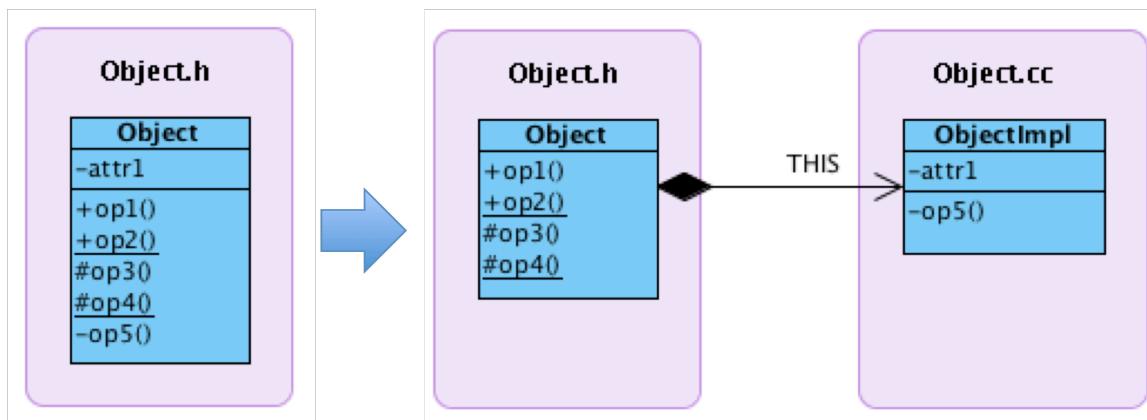
- 构造函数，无论它的可访问性如何，都应该放在头文件里；
- 类的公开函数（静态函数，成员函数）需要放在头文件里；
- 如果一个类允许继承，那么它就很可能存在受保护的函数（静态函数，成员函数）；而这部分必须为子类可见，所以需要放在头文件里；
- 类不应该存在受保护的数据，至少应该以Setter/Getter函数的方式提供；注意，即便提供Setter/Getter，也并非一定要成对提供Setter和Getter，而是根据父类希望子类以怎样的方式来访问数据；
- 私有的成员数据：对于外部客户和子类而言，从逻辑上都没有任何意义。只有物理空间分配方式上的意义，如果这一点不重要，则完全没有必要放在头文件里；
- 私有成员函数：对于子类和外部客户而言，没有任何意义，对于内存连续性也没有任何影响；
- 私有静态数据／函数：不仅仅放在头文件里没有意义，甚至放在类里都毫无意义，无论从哪个角度看。

- `inline`函数的代码：据说对于性能提高有帮助，但只有得到证明的情况下，它才应该存在于头文件里。

所以，在不考虑物理内存模型和性能的情况下，只有公开函数和受保护的函数（静态函数，成员函数）才是真正对客户（外部客户和子类）有意义的，其它的都是实现细节，都应该被“隐藏”起来，放入模块源代码文件。

PIMPL模式就是为此目的而生的解决方案。是C++社区为了提高模块内聚度，降低模块间耦合而常用的一种模式。

PIMPL是（Pointer to Implementation）的缩写。从它的名字就可以得知，它把所有与实现相关的元素全部隐藏到模块的源代码文件中的实现类中，原类则持有一个指向实现类的指针。如下图所示。



Object类被一分为二，Object只负责存放public和protected的元素；私有元素，全部由ObjectImpl负责承载。这样我们就可以把原本放在Object上，但事实上与客户无关的东西，统统放在ObjectImpl上来完成。

作为一种降低“物理耦合”的手段，PIMPL是一种非常有效的模式。它从物理上彻底分离了“接口”和“实现”。其效用等价于单一实现的“抽象接口”。

PIMPL的可以有如下几个好处：

- 物理耦合的降低，降低重新编译的概率，加快编译速度；
- 头文件里类的定义更加干净，容易理解；
- 在不增加外部接口的情况下，程序员在实现时无须在头文件和源代码文件之间切换，所有内部实现都在头文件里完成；

- 对于模块内函数，无须分声明和实现两部分，直接以inline的方式实现ObjectImpl的所有函数；尤其对于私有函数，这样的消除重复是非常有价值的；
- 某些实现所造成的继承关系，无须由Object来继承，而是让ObjectImpl来继承，隐藏了无须客户关心的继承关系。

尽管PIMPL模式威力强大，但它也存在一定的副作用：

- 每个对象都分配了两块内存区域，分别用来保存Object和ObjectImpl。在某些系统上，对于需要大量分配的对象可能会有“内存碎片”方面的顾虑；
- 对于public的函数，有可能会造成多一次实现（头文件声明一次，源文件实现一次，ObjectImpl实现一次）；但大多数情况下，public函数都可以通过直接访问ObjectImpl的变量或函数来完成；
- ObjectImpl依靠new操作产生，这就导致了，无论Object在哪里分配（静态数据区，栈，堆），ObjectImpl必然在堆中分配。如果一个Object对象被定义为全局／静态变量，当其被构造时，系统的内存管理器有可能还没有初始化完成；在某些嵌入式系统上，这会导致风险。

简化实现*

最终我们发现，引入“空对象模式”的原因在于“基准单位”和“非基准单位”处理上的不同。当我们找到这个本质问题之后，就可以将这两个概念进行一致化，得到抽象类型AbstractLengthUnit，从而得出下面的简化实现。

```
struct LengthUnit
{
private:
    //用以构造基准单位
    LengthUnit();

    //用以构造非基准单位
    LengthUnit( unsigned int conversionFactor
               , const LengthUnit& baseUnit);

    ~LengthUnit();

public:
    Amount toAmountInBaseUnit(const Amount& amount) const;
    const LengthUnit& getBaseUnit() const;
}
```

```

private:
    struct AbstractLengthUnit;
    const AbstractLengthUnit* const m_unit;
};

struct AbstractLengthUnit
{
    virtual unsigned int getConversionFactor() const = 0;
    virtual const LengthUnit& getBaseUnit() const = 0;

    virtual ~AbstractLengthUnit() {}
};

namespace {

    ///////////////////////////////////////////////////
    const unsigned int BASE_UNIT_CONV_FACTOR = 1;

    ///////////////////////////////////////////////////
    struct BaseUnit : public AbstractLengthUnit
    {
        BaseUnit(const LengthUnit& baseUnit) : m_baseUnit(baseUnit) {}

        unsigned int getConversionFactor() const
        { return BASE_UNIT_CONV_FACTOR; }

        const LengthUnit& getBaseUnit() const
        { return m_baseUnit; }

        const LengthUnit& m_baseUnit;
    };

    ///////////////////////////////////////////////////
    struct NonBaseUnit : public AbstractLengthUnit
    {
        NonBaseUnit( unsigned int conversionFactor
            , const LengthUnit& refUnit)
            : m_conversionFactor(conversionFactor)
            , m_refUnit(refUnit) {}

        unsigned int getConversionFactor() const
        { return m_refUnit.toAmountInBaseUnit(m_conversionFactor); }

        const LengthUnit& getBaseUnit() const
        { return m_refUnit.getBaseUnit(); }

        unsigned int m_conversionFactor;
        const LengthUnit& m_refUnit;
    };
}

///////////////////////////////////////////////////
LengthUnit::LengthUnit()
    : THIS(new BaseUnit(*this)) {}

LengthUnit::LengthUnit
( unsigned int conversionFactor, const LengthUnit& refUnit )

```

```

    : THIS(new NonBaseUnit(conversionFactor, refUnit)) {}

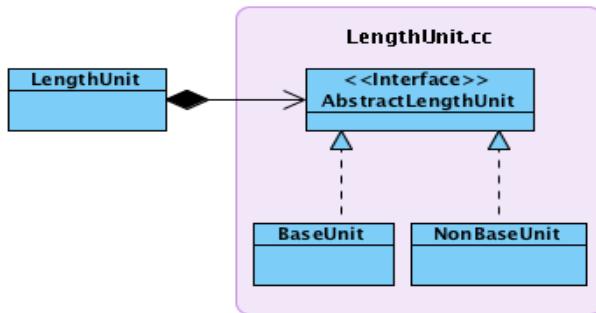
LengthUnit::~LengthUnit()
{ delete THIS; }

Amount
LengthUnit::toAmountInBaseUnit(const Amount& amount) const
{ return amount * THIS->getConversionFactor(); }

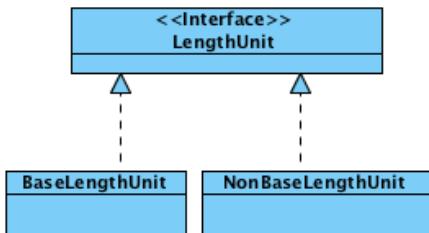
const LengthUnit&
LengthUnit::getBaseUnit() const
{ return THIS->getBaseUnit(); }

```

注意在这个实现里，AbstractLengthUnit是由LengthUnit内部实现机制而创造出来的概念。所以，我们应该将其封装在LengthUnit模块内部。



由于Slug的约束，我们希望把所有的Slug都定义和实现在一个类中。如果没有这方面的考虑，上述实现也可以转化为更为简单的形式：



需求六

问题描述

实现一个容量 (Volume) 库。通过这个库，用户可以使用TSP (茶匙), TBSP (汤匙), OZ (盎司)为单位来表示一个容量。其中，

- $1 \text{ TBSP} = 3 \text{ TSP}$
- $1 \text{ OZ} = 2 \text{ TBSP}$
- 当以TBSP为单位来表示一个容量时，精度为 1 TBSP
- 当以TSP为单位来表示一个容量时，精度为 1 TSP
- 当以OZ为单位来表示一个容量时，精度为 1 OZ
- 可以对比任意两个容量的相等性
- 只允许用户使用现有的三个容量单位来表示容量
- 两个容量可以相加

编写用例

不难发现，当前对于Volume的需求和之前对于Length的需求几乎是一致的。所以，设计可以完全照搬过来。这里就不再列出Volume的相关用例。

满足功能

和Length的设计一样，Volume的实现代码如下：

```
struct Volume
{
    Volume( const Amount& amount
            , const VolumeUnit& unit);

    bool operator==(const Volume& rhs) const;
    bool operator!=(const Volume& rhs) const;

private:
    Amount m_amount;
    const VolumeUnit& m_unit;

private:
    friend Volume operator+(const Volume& lhs, const Volume& rhs);
};
```

而VolumeUnit也和LengthUnit类似。同样，我们使用“slug模式”来对VolumeUnit的创建进行约束。

```

struct VolumeUnit
{
private:
    VolumeUnit();
    VolumeUnit( unsigned int conversionFactor
                , const VolumeUnit& refUnit);

    ~VolumeUnit();

public:
    Amount toAmountInBaseUnit(const Amount& amount) const;
    const VolumeUnit& getBaseUnit() const;

public:
    __UNIT_SLUG_DECL(Volume, TSP);
    __UNIT_SLUG_DECL(Volume, TBSP);
    __UNIT_SLUG_DECL(Volume, OZ);

private:
    struct AbstractVolumeUnit;
    const AbstractVolumeUnit* const m_unit;
};

///////////////////////////////
#define TSP __UNIT_CONST_DECL(Volume, TSP)
#define TBSP __UNIT_CONST_DECL(Volume, TBSP)
#define OZ __UNIT_CONST_DECL(Volume, OZ)

```

消除重复

由于我们对Volume的所有设计都照搬了Length。毫无意外，两者的代码几乎完全一样。

Volume和Length的高度相似性，是否意味着背后有着更加本质的概念？如果有，它应该是什么？

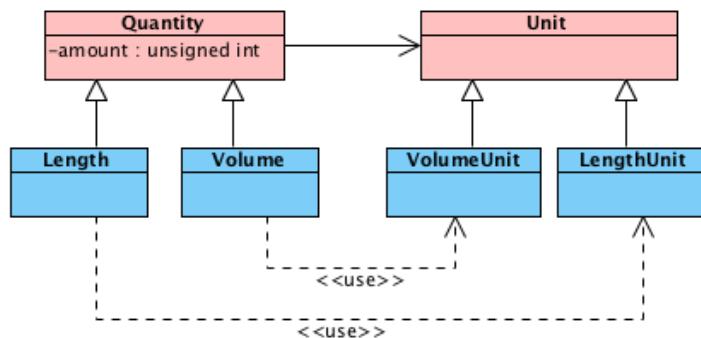
我们来仔细审视一下Volume和Length的构成，发现它们都有两个内在属性：

- 数量 (Amount)
- 单位 (Unit)

这样的一个组合构成的概念应该是一个Quantity。Volume和Length只不过是两种更具体的Quantity。



而LengthUnit和VolumeUnit也不过是一个更抽象概念Unit的泛化。



所以，我们可以把Volume和Length的代码合一，得到一个新的类：Quantity。

```

struct Quantity
{
    Quantity( const Amount& amount
              , const Unit& unit);

    bool operator==(const Quantity& rhs) const;
    bool operator!=(const Quantity& rhs) const;

private:
    Amount m_amount;
    const Unit& m_unit;

private:
    friend
    Quantity operator+(const Quantity& lhs, const Quantity& rhs);
};

```

类似的，我们把LengthUnit和VolumeUnit的实现合一，得到Unit的代码：

```

struct Unit
{
private:
    Unit();
    Unit( unsigned int conversionFactor
          , const Unit& refUnit);

```

```

~Unit();

public:
    Amount toAmountInBaseUnit(const Amount& amount) const;
    const Unit& getBaseUnit() const;

public:
    __UNIT_SLUG_DECL(MILE);
    __UNIT_SLUG_DECL(YARD);
    __UNIT_SLUG_DECL(FEET);
    __UNIT_SLUG_DECL(INCH);

    __UNIT_SLUG_DECL(TSP);
    __UNIT_SLUG_DECL(TBSP);
    __UNIT_SLUG_DECL(OZ);

private:
    struct AbstractUnit;
    const AbstractUnit* const m_unit;
};

#define TSP __UNIT_CONST_DECL(TSP)
#define TBSP __UNIT_CONST_DECL(TBSP)
#define OZ __UNIT_CONST_DECL(OZ)

#define INCH __UNIT_CONST_DECL(INCH)
#define FEET __UNIT_CONST_DECL(FEET)
#define YARD __UNIT_CONST_DECL(YARD)
#define MILE __UNIT_CONST_DECL(MILE)

```

提高表达力

当我们将两套计量体系合二为一之后，用户创建一个长度对象的UI则变为：

Length(12, MILE) → **Quantity(12, MILE)**

但是，原来的表现方式似乎更直观一些。

另外，作为更早的需求，已经有用户在使用**Length**，让这些用户进行修改，已经是一件困难的事情，更何况还要修改为更加缺乏表现力的方式。所以，最好的选择是依然提供原有的UI。

对于 C++语言而言，这并非一个艰难的任务。通过**typedef**给个别名即可。

```

typedef Quantity Length;

typedef Quantity Volume;

```

降低耦合度

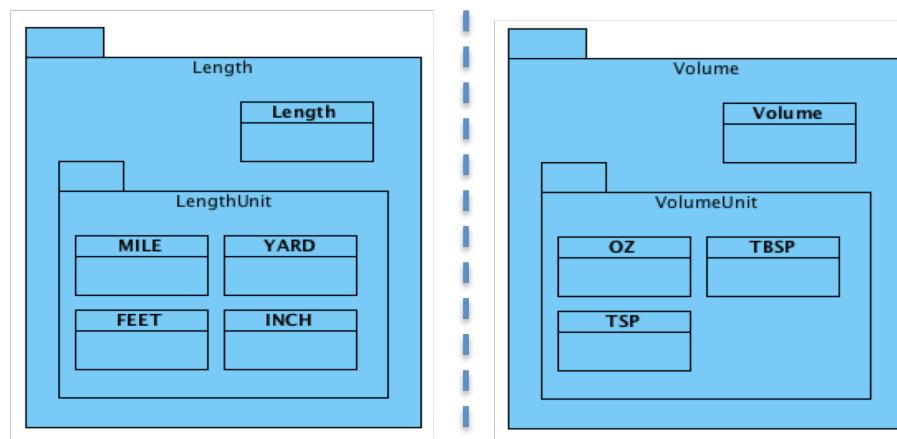
看起来表达力的问题已经得到了很好的解决。但它引发了我们进一步的思考——

Length和Volume究竟是什么关系？

- 它们尽管可以在实现上存在完全的共性。但从用户的观点看，Length和Volume应该是完全不相干的两套体系。一个用户可能仅仅用这套库来表现Length，那么他就只需要Length相关的Unit实例；那么为何Unit模块还提供了Volume的实例？
- 如果随后再增加一套新的计量体系，比如面积呢？是否也需要把面积相关的所有Unit实例都定义在Unit模块？

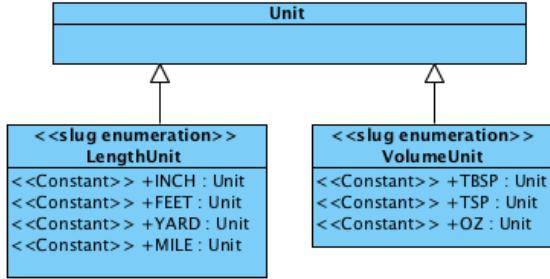
这样的设计，让一个仅仅需要使用Length的客户也被迫从物理上（编译时和链接时）耦合所有的其它Unit实例。这违背了原则：“不要强迫客户依赖它不需要依赖的东西”。另外，这也会让Unit模块更加的不稳定：任何一套体系的Unit实例变化，都会造成Unit模块的修改。

所以，两套体系的设计从逻辑上应该是两个独立的包，它们应该可以独立的发布，独立的重用。



从实现上，两套计量体系在Unit的计算行为上是一致的，所以，所有的Unit相关的行为逻辑都应该放在Unit类里。

为了分离不同的计量体系，同时要用“slug模式”来约束Unit实例的创建，所以应该将原来全部放置在Unit里的属于不同计量体系的Unit实例，放到不同的类里。如下图所示：

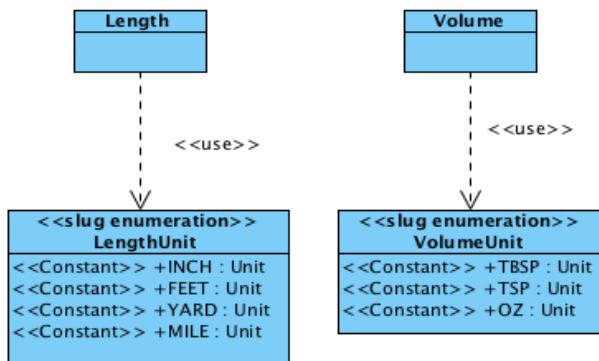


但现在我们发现一个新的问题：在原来的设计中，为了不让用户有能力创建Unit实例，我们把Unit的构造函数设为private的。现在为了把Length和Volume两套计量体系分开，我们创建了Unit的两个子类，LengthUnit和VolumeUnit，它们的职责是代表各自体系的Unit，并控制本体系Unit实例的创建。

为了能够创建实例，LengthUnit和VolumeUnit必须能够访问Unit的构造函数。所以需要Unit的构造函数由private的变为protected的。

这似乎为系统留下一个漏洞：用户可以创造出新的Unit子类，像LengthUnit和VolumeUnit那样，然后通过子类创造出新的Unit实例。

但如果我们能够把两套体系彻底分离，让Length只能使用LengthUnit的实例，Volume只能使用VolumeUnit的实例，同时我们禁止用户创造出LengthUnit类型和VolumeUnit类型的实例，我们就不用担心用户通过继承的方式来创造出新的Unit实例。因为它所创造出来的实例，无法应用于Length和Volume体系的计算。如下图所示：



这种情况下，Unit允许继承式创建的“缺陷”，反而变成了一种扩展机制。当用户需要构建一个新的计量体系时，就可以像Length和Volume一样，通过它构造出自己的Unit实例以及相关约束。

但到目前为止，我们仅仅区分了不同的Unit类型。而Unit的客户是Quantity，只有Quantity的计算规则能够区分不同的Unit类型，我们之前的设想才能实现。

隐含需求

不难发现，在现有的需求背后，事实上存在一个隐含约束：一个表示“长度”的Quantity对象和一个表示“容量”的Quantity对象应该不能进行互操作：即两类Quantity对象之间不能进行相等性对比，也不能相加。

现在我们的实现是通过“别名机制”来区分Length和Volume两种类型的Quanity。但仅仅依靠“别名机制”并不能实现这个约束。

区分不同的Unit

一种方案是将Unit进行“类别”区分。不同“类别”的Unit所表示的Quantity对象不能对比相等性。代表“类别”的值可以在Unit构造时传入。那么，我们应该用什么来区分Unit的类别？

马上浮进C++程序员脑海的应该是枚举。比如：

```
struct Unit
{
protected:
    enum UnitType
    {
        LENGTH_UNIT_TYPE,
        VOLUME_UNIT_TYPE
    };

protected:
    Unit(UnitType unitType);
    .....
private:
    UnitType m_unitType;
};
```

但这种方式同样会带来Unit不稳定的问题；同时，会造成Length和Volume的耦合，从而无法独立的提供Length和Volume计量体系。

一个解决办法是，我们不定义枚举，而是人为的分配不同的值给各个计量体系，比如：

```
typedef unsigned int UnitType;

struct Unit
```

```

{
protected:
    Unit(UnitType unitType);
    .....
private:
    UnitType m_unitType;
};

///////////////////////////////
const UnitType LENGTH_UNIT_TYPE = 0;

LengthUnit::LengthUnit()
    : Unit(LENGTH_UNIT_TYPE) {}

///////////////////////////////
const UnitType VOLUME_UNIT_TYPE = 1;

VolumeUnit::VolumeUnit()
    : Unit(VOLUME_UNIT_TYPE) {}

```

不幸的是，尽管这种办法让Unit稳定了下来，但各个计量体系在这个值的分配上产生了耦合关系：一个计量体系在使用某个值的时候，必须确保它没有被其它体系所使用。

为了打破这种耦合关系，我们需要一种方法，让UnitType的值可以随时扩展，同时又不会引起Volume和Length的耦合。

方案一

我们观察到VolumeUnit和LengthUnit是两个不同的类，所以它们的typeid也肯定不等。我们可以利用这一点来区分不同的Unit “类别” 。

```

struct Unit
{
protected:
    Unit(const Unit& baseUnit);

public:
    bool hasSameTypeWith(const Unit& rhs) const
    { return typeid(*this) == typeid(rhs); }

    .....
};

```

不幸的是，这样的实现方式并不工作。因为C++的typeid得到结果是变量的类型，而不是变量所持的对象类型。所以，在方法hasSameTypeWith()的实现里，无论是表达式typeid(*this)还是typeid(rhs)，返回的类型都是const Unit&。

为了让每个子类都能够准确的返回自己的类型信息，一个解决办法是：让每个子类在构造时计算自己的类型信息，并把它保存下来。

```
struct Unit
{
protected:
    Unit(const std::type_info& unitType);

public:
    bool hasSameTypeWith(const Unit& rhs) const
    { return m_unitType == rhs.m_unitType; }

private:
    const std::type_info& m_unitType;
    .....
};

LengthUnit::LengthUnit(unsigned int conversionFactor)
: Unit(conversionFactor, typeid(this)) {}

VolumeUnit::VolumeUnit(unsigned int conversionFactor)
: Unit(conversionFactor, typeid(this)) {}
```

我们发现LengthUnit和VolumeUnit的构造函数的代码完全是一样的（但其背后的求值是不同的，它们分别等价于typeid(LengthUnit*)和typeid(VolumeUnit*)）。

如果你不喜欢这种重复，则可以借助多态来消除它：

```
struct Unit
{
    bool hasSameTypeWith(const Unit& rhs) const
    { return getTypeInfo() == rhs.getTypeInfo(); }

private:
    virtual const std::type_info& getTypeInfo() const
    { return typeid(*this); }

    .....
};
```

方案二

上述方案使用了typeid，而typeid的使用需要你的系统支持RTTI。但在很多嵌入式设备上，RTTI是被禁止的。

这种情况下，可以利用一个机制来模拟RTTI：在同一地址空间，两个变量的地址不等。

```

struct UnitType
{
    bool operator==(const UnitType& rhs) const
    { return this == &rhs; }

    bool operator!=(const UnitType& rhs) const
    { return !operator==(rhs); }
};

///////////////////////////////
struct Unit
{
protected:
    Unit(const UnitType& unitType);
    .....

public:
    bool hasSameTypeWith(const Unit& rhs) const
    { return m_unitType == rhs.m_unitType; }
    .....

private:
    const UnitType& m_unitType;
};

/////////////////////////////
const UnitType LENGTH_UNIT_TYPE = UnitType();

LengthUnit::LengthUnit() : Unit(LENGTH_UNIT_TYPE) {}

/////////////////////////////
const UnitType VOLUME_UNIT_TYPE = UnitType();

VolumeUnit::VolumeUnit() : Unit(VOLUME_UNIT_TYPE) {}

```

这两种方案，无论使用哪一种，都在Unit层面对Length和Volume进行了良好的解耦。让Unit变成了一个与具体计量体系完全无关的可复用类。

区分不同的Quantity

下面需要考虑的一个问题是：当把一个Length对象和一个Volume对象进行相等性比较时，应该怎么处理？

至少有两种方式可供选择：

- 抛出异常
- 认为二者不相等

第一种方式，应该是更合理的方式。因为用户试图对比两个根本无法比较的东西，应该属于一种异常行为。

但问题是，很多嵌入式系统是不允许使用异常的。

在不使用异常的情况下，第二种方式就是当前唯一的选择。支持我们这样做的另外一个重要理由是：即便是支持异常的Java，其对象equals方法的实现模式，也把两个无法比较的对象认为是不相等的。

```
bool Quantity::operator==(const Quantity& rhs) const
{
    return m_unit.hasSameTypeWith(rhs.m_unit) &&
           getAmountInBaseUnit() == rhs.getAmountInBaseUnit();
}
```

相等性的问题解决了，下一个需要解决的是加法问题。

加法操作不像相等性判断那样是一个“Yes or No”操作，而是要返回一个新的Quantity对象。在不允许使用异常的情况下，我们必须构造一个表示非法的Quantity对象作为返回值。

这种情况下，用户使用我们库的方式可能如下：

```
TEST(两个不是同一计量体系的Quantity对象相加应该返回一个非法的Quantity对象)
{
    Quantity result = Quantity(1, OZ) + Quantity(1, MILE);
    ASSERT_FALSE(result.isValid());
}
```

这样的UI需要给Quantity对象加入一个新的public方法isValid()。

如果我们不希望Quantity增加额外的方法，则可以提供这样的UI：

```
TEST(两个不是同一计量体系的Quantity对象相加应该返回一个非法的Quantity对象)
{
    Quantity result = Quantity(1, OZ) + Quantity(1, MILE);
    ASSERT_TRUE(result == INVALID_QUANTITY);
}
```

为了满足上述约束，有多种实现方法。比如，在构造Quantity对象时，传入一个表示“非法”的Unit实例。如下：

```
Quantity operator+(const Quantity& lhs, const Quantity& rhs)
{
    if(not m_unit.hasSameTypeWith(rhs.m_unit))
    {
        return Quantity(1, INVALID_UNIT);
    }
    ....
}
```

而这个代表“非法”的Unit实例不应该归属于LengthUnit或者VolumeUnit，而应该是一个独立于所有计量体系的值。

一种方法是将其定义直接放在Unit里：

```
const UnitType INVALID_UNIT_TYPE = UnitType();

struct Unit
{
    .....

public:
    static const Unit& getInvalidUnit()
    {
        static Unit invalid(1, INVALID_UNIT_TYPE);
        return invalid;
    }
};
```

还有一种方法无须修改Unit，而是来扩展它：

```
const UnitType INVALID_UNIT_TYPE = UnitType();

struct InvalidUnit : public Unit
{
private:
    InvalidUnit(): Unit(INVALID_UNIT_TYPE) {}

public:
    static const InvalidUnit& getInstance()
    {
        static InvalidUnit invalid;
        return invalid;
    }
};
```

提高表现力

这些方法都可以解决返回一个非法Quantity对象的问题。设计手法干净漂亮，这让我们再次为自己的熟练的设计技巧感到满足和骄傲。

但是... 我们真的想让用户以这样的方式使用我们的库吗？

```
Quantity result = quantity1 + quantity2;

if(result == INVALID_QUANTITY) {...}
```

用户每进行一次加法运算，都要先判断结果的合法性。这会增加用户代码不必要的复杂性，让用户代码更加晦涩；还会让客户代码到处充斥着重复的错误处理模式。

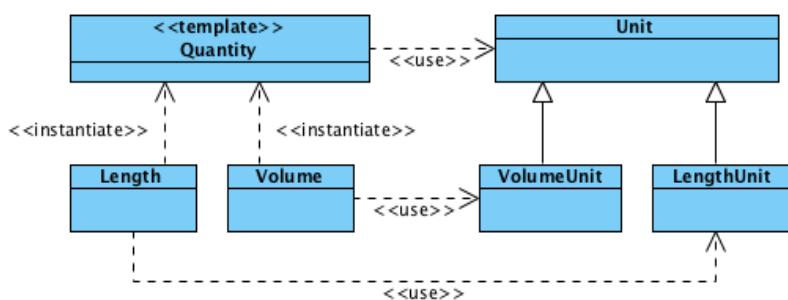
如果我们让这种情况根本不可能发生，无论是这套库的设计，还是用户的代码都可以避免处理这种lousy的情况。

C++是一种强类型的语言，编译器可以在编译阶段发现各种类型错误，这可以让程序员无须编写任何类型防错代码，从而让实现代码更加简洁、专注，避免一切原本不必要的工作。

为了利用这一点，我们就必须让Length和Volume变成不同的类型，而且要确保Length只能使用LengthUnit实例，Volume只能使用VolumeUnit实例。

将Quantity实现为模版，即可以实现代码的完全复用，又可以做到约束只有同一类型的Quantity才可以进行互操作。

所有公开类的关系如下图所示：



两棵继承树，在一棵树上面的每个类都在另一棵上有一个对应的类。这样的结构让人生厌，并且往往都潜藏着严重的设计问题。我们在设计中应该竭力避免这样的设计。

但对于这个设计，我们却是有意而为之。我们通过这种方式彻底的分离两种本质上不应该在一起的两套体系。但由于它们在概念上的一致性，为了消除逻辑上的重复，我们将其提取出以复用为目的的基类。在叶子节点，事实上并没有什么行为，它们只是一个作为不同的身份标识罢了。

```
template <typename UNIT>
struct Quantity
{
private:
    Quantity(const Amount& amount, const UNIT& unit) {...}

public:
    Quantity(const Amount& amount, const UNIT& unit) {...}
```

```

    bool operator==(const Quantity& rhs) const {...}
    bool operator!=(const Quantity& rhs) const {...}

private:
    Amount toAmountInBaseUnit() const {...}

private:
    friend
    Quantity<UNIT> operator+( const Quantity<UNIT>& lhs
                                , const Quantity<UNIT>& rhs) const
    {
        Amount amountInBaseUnit =
            lhs.toAmountInBaseUnit() + rhs.toAmountInBaseUnit();

        return Quantity<UNIT>( amountInBaseUnit
                                , lhs.m_unit.getBaseUnit());
    }

private:
    Amount m_amount;
    const Unit& m_unit;
};

///////////////////////////////////////////////////
typedef Quantity<LengthUnit> Length;
///////////////////////////////////////////////////
typedef Quantity<VolumeUnit> Volume;

```

测试与重用

可以独立测试的单位，就是可重用的单位。

TBW

预防胜于治疗

错误／异常处理，一直是程序员痛恨，却无法摆脱的梦魇。如果一个系统中仅仅包含Happy Path的实现，那么这个系统的代码规模会大大缩小，而逻辑清晰度则大大增加。

C++以及更加现代的面向对象语言，都提供了概念——“异常”，以及异常处理机制，这就大大的简化了程序员的工作，并且让实现代码能够更加关注真正的业务逻辑。

但异常机制并不是免费的，尤其是C++的异常机制，对于空间和时间都有较大的消耗。所以，大多数实时性嵌入式系统都会禁止使用异常。程序员只好重新回到“处处防卫，处处判断返回值”的编程模式下。

即便允许使用异常，只要一个类可能出现异常的状况，那么这个类的所有客户都必须编写相应的异常处理代码，这仍然是一个让人不悦的工作。

错误／异常就像强盗，一旦让其闯入，你就需要不断与其周旋——编写防备代码，DEBUG——耗去你的时间和精力，还把你的房间搞的一团糟。

所以，如果能够在事先就杜绝一些错误发生的可能，那么程序员就无须为之付出相应的精力，在保持系统简洁的同时，也会让系统更加健壮（你能确保程序员的错误处理代码没有错吗？）。

C++作为一门强类型的静态语言，我们要充分利用它的优势。让它的严格类型检查作为我们忠实的看门狗，在编译时就可以辨认出任何可能溜进来的BUG，从而不让用户存在犯错的可能（恶意用户刻意的hack除外）。

这样的思想可以延伸到所有的方面。比如对于复杂的函数，你可以看到不同团队有着各种各样的应对策略，比如，“单点退出”，“if和else必须成对”等等，而这些策略又会让一个函数更加复杂（更多的临时变量，更多的没有必要的else块）。而如果根本不允许存在复杂的函数，那么所有的这些问题也就根本不会存在。

需求七

问题描述

基于调试等目的需要，可以将一个Length对象输出到屏幕上。输入格式的规则如下：

- 如果一个对象的“基准单位数量”在一个更大的单位上的倍数非0，则显示此对象在此单位上的倍数，以及此单位的名字；
- 如果一个对象的“基准单位数量”在一个更大的单位上的倍数为0，则无须显示此对象在此单位上的倍数，以及此单位的名字；
- 如果余数在一个较小的单位上倍数非0，则显示此对象在此单位上的倍数，以及此单位的名字；
- 如果余数在一个较小的单位上倍数为0，则无须显示此对象在此单位上的倍数，以及此单位的名字；
- 如果一个对象的“基准单位数量”为0，则显示0，以及基准单位的名字；
- 如果存在多个“数量+单位”组合，则按照单位大小，从左向右排列；
- 数量和单位之间，由一个空格分开；
- “数量+单位”之间由一个空格分开。

例如：

- 对象Length(14, INCH) 显示为：1 FEET 2 INCH
- 对象Length(24, INCH) 显示为：2 FEET
- 对象Length(39, INCH) 显示为：1 YARD 3 INCH
- 对象Length(1762, YARD) 显示为：1 MILE 2 INCH

编写用例

需求是要把信息输出到屏幕上，基于测使用例应该是“Self-Checking”的原则，我们的自动化测试用例必须能够捕捉到输出到屏幕上的信息是否正确。而从屏幕上获取信息是件相对困难的事情。

但“将对象输出到屏幕”这个需求事实上包含了两个关注点：

- 输出的内容
- 输出的目的地（设备）

我们只需要验证“输出的内容”是否正确即可。

由于输出的内容是一个字符串，所以，我们可以让Length对象有一个toString()方法。这样用户就可以这样使用我们的软件。

```
Length length(25, INCH);  
std::cout << length.toString() << std::endl;
```

不过，如果上述用法可以改为：

```
Length length(25, INCH);  
std::cout << length << std::endl;
```

就更加的完美。这是C++社区在进行对象序列化时的常用方式。

既然Length对象可以输出到“标准输出”std::cout，而std::cout是std::ostream的一个实例。它自然也应该可以输出到任何其它std::ostream上。而std::ostringstream是std::ostream的一个子类，这样我们就可以把Length对象输出到一个字符流，从而进行自动化测试。

```
TEST(能够将一个非0的长度对象按照格式要求输出到流中: 14 INCH => 1 FEET 2 INCH)  
{  
    std::ostringstream oss;  
    oss << Length(14, INCH);  
    ASSERT_EQ("1 FEET 2 INCH", oss.str());  
}  
  
TEST(能够将一个非0的长度对象按照格式要求输出到流中: 24 INCH => 2 FEET)  
{  
    std::ostringstream oss;  
    oss << Length(24, INCH);  
    ASSERT_EQ("2 FEET", oss.str());  
}  
  
TEST(能够将一个数量为0的长度对象按照格式要求输出到流中: 0 MILE => 0 INCH)  
{  
    std::ostringstream oss;  
    oss << Length(0, MILE);  
    ASSERT_EQ("0 INCH", oss.str());  
}
```

可以看到这些测试用例的模式是完全重复的，所以需要消除它们：

```
void assert_output(const Length& length, const std::string& output)  
{  
    std::ostringstream oss;  
    oss << length;  
    ASSERT_EQ(output, oss.str());  
}
```

```

TEST(能够将一个非0的长度对象按照格式要求输出到流中: 14 INCH => 1 FEET 2 INCH)
{
    assert_output(Length(14, INCH), "1 FEET 2 INCH");
}

TEST(能够将一个非0的长度对象按照格式要求输出到流中: 24 INCH => 2 FEET)
{
    assert_output(Length(24, INCH), "2 FEET");
}

TEST(能够将一个数量为0的长度对象按照格式要求输出到流中: 0 MILE => 0 INCH)
{
    assert_output(Length(0, MILE), "0 INCH");
}

```

由于Length的实现放在模版类Quantity，而当前的需求仅仅针对Length，所以我们不应该在Quantity中修改，而是应该把其放入Length类。

Length现在是模版类Quantity的一个实例化。作为一个新的扩展功能，我们应该让Length变成Quantity<LengthUnit>的子类。

```

///////////////////////////////
struct Length : public Quantity<LengthUnit>
{
    .....
private:
    void format(std::ostream& os) const {}

    friend std::ostream&
    operator<<(std::ostream& os, const Length& length) const;
};

std::ostream&
operator<<(std::ostream& os, const Length& length)
{
    format(os);
    return os;
}

```

现在我们已经搭好了整个架子，这套代码是用来处理流问题的常见实现模式。

可测试性设计，抽象&测试替身

无论是是否在使用TDD，你都应该在设计时考虑一个SUT (System Under Test) 的可测试性问题。

比如，当前需求需要向屏幕上输出一个对象。如果你愿意，可以通过一些非常手段去屏幕上抓取输出信息，然后对其进行验证。

但这样的方案可能会导致测试方案的复杂度上升。同时还可能导致测试方案的不稳定，因为不同的平台环境下，抓取屏幕信息的方式可能是不同的。

类似的情况还有很多，比如时钟。如果你的SUT依赖了某个具体的时钟值，想让测试可以恰巧运行在那一精准时刻，其概率应该比买彩票中奖高不了多少。

还比如，一个用例关注于“内存耗尽”时SUT的处理。而构造“内存耗尽”的场景，则需要相应的技巧，同样它也有平台相关的特点。

TBW

实现功能

这个需求牵扯的问题似乎比较复杂，所以我们先尝试着给出一个快速的实现：

```
void Length::format(std::ostream& os) const
{
    if(m_amount == 0)
    {
        os << 0 << " " << "INCH";
        return;
    }

    Amount restAmount = getAmountInBaseUnit();
    bool firstUnitFound = false;

    ///////////////////////////////////////////////////
    unsigned int amountInMile = restAmount / MILE.toAmountInBaseUnit(1);
    if(amountInMile > 0)
    {
        os << amountInMile << " " << "MILE";
        firstUnitFound = true;
    }
    restAmount %= MILE.toAmountInBaseUnit(1);

    ///////////////////////////////////////////////////
    unsigned int amountInYard = restAmount / YARD.toAmountInBaseUnit(1);
    if(amountInYard > 0)
    {
        if(firstUnitFound) os << " ";
        os << amountInYard << " " << "YARD";
        firstUnitFound = true;
    }
}
```

```

}
restAmount %= YARD.toAmountInBaseUnit(1);

///////////////////////////////
unsigned int amountInFeet = restAmount / FEET.toAmountInBaseUnit(1);
if(amountInFeet > 0)
{
    if(firstUnitFound) os << " ";
    os << amountInFeet << " " << "FEET";
    firstUnitFound = true;
}
restAmount %= FEET.toAmountInBaseUnit(1);

///////////////////////////////
unsigned int amountInInch = restAmount;
if(amountInInch > 0)
{
    if(firstUnitFound) os << " ";
    os << amountInInch << " " << "INCH";
}
}

```

是的，这个实现臭气熏天，你很可能已经产生了头晕耳鸣目眩，想要呕吐的感觉，再多看一眼都会是一种折磨。

然后，你屏住呼吸，咬紧牙关，在半昏半醒中拿起键盘，在命令行敲出四个字母：**m-a-k-e**，然后虚弱的按下回车...快速翻滚的屏幕上最后定格在“绿色”的“OK”：它...竟然...是成功的！

消除重复

既然这份代码的功能实现是正确的，我们就应该以这份代码为基础，进行整理和演化。毕竟看到这样的代码，才是一个程序员现实中的常态。

好吧，让我们做几次深呼吸，开始这次重构之旅。

首先，我们发现在不同单位上的处理代码非常相似，只有对MILE和INCH的处理略有差别。经过分析，我们发现，它们事实上可以写成完全一样的模式。之所以它们有些不一样，是因为某些代码没有必要；但即便写上，也不会造成错误的结果。

现在我们把它们的模式改为精确一致的方式：

```

void Length::format(std::ostream& os) const
{
    .....
///////////////////////////////
unsigned int amountInMile = restAmount / MILE.toAmountInBaseUnit(1);

```

```

if(amountInMile > 0)
{
    if(firstUnitFound) os << " ";
    os << amountInMile << " " << "MILE";
    firstUnitFound = true;
}
restAmount %= MILE.toAmountInBaseUnit(1);

///////////////////////////////
unsigned int amountInYard = restAmount / YARD.toAmountInBaseUnit(1);
if(amountInYard > 0)
{
    if(firstUnitFound) os << " ";
    os << amountInYard << " " << "YARD";
    firstUnitFound = true;
}
restAmount %= YARD.toAmountInBaseUnit(1);

/////////////////////////////
unsigned int amountInFeet = restAmount / FEET.toAmountInBaseUnit(1);
if(amountInFeet > 0)
{
    if(firstUnitFound) os << " ";
    os << amountInFeet << " " << "FEET";
    firstUnitFound = true;
}
restAmount %= FEET.toAmountInBaseUnit(1);

/////////////////////////////
unsigned int amountInInch = restAmount / INCH.toAmountInBaseUnit(1);
if(amountInInch > 0)
{
    if(firstUnitFound) os << " ";
    os << amountInInch << " " << "INCH";
    firstUnitFound = true;
}
restAmount %= INCH.toAmountInBaseUnit(1);

}

```

提取方法

```

namespace
{
    void formatOnUnit
    (
        const Unit& unit
        , const std::string& name
        , std::ostream& os
        , Amount& restAmount
        , bool& firstUnitFound )
    {
        unsigned int amount = restAmount / unit.toAmountInBaseUnit(1);
        if(amount > 0)
        {
            if(firstUnitFound) os << " ";

```

```

        os << amount << " " << name;
        firstUnitFound = true;
    }
    restAmount = restAmount % unit.toAmountInBaseUnit(1);
}
}

///////////////////////////////
void Length::format(std::ostream& os) const
{
    .....
    formatOnUnit(MILE, "MILE", os, restAmount, firstUnitFound);
    formatOnUnit(YARD, "YARD", os, restAmount, firstUnitFound);
    formatOnUnit(FEET, "FEET", os, restAmount, firstUnitFound);
    formatOnUnit(INCH, "INCH", os, restAmount, firstUnitFound);
}
}

```

表格驱动

```

void Length::format(std::ostream& os) const
{
    .....
    struct UnitInfo
    {
        const Unit& unit;
        const std::string name;
    }
    allUnitsInOrder[] =
    { {MILE, "MILE"}
    , {YARD, "YARD"}
    , {FEET, "FEET"}
    , {INCH, "INCH"}
    };
    Amount restAmount = getAmountInBaseUnit();
    bool firstUnitFound = false;
    for(size_t i=0; i<SIZEOF(allUnitsInOrder); i++)
    {
        formatOnUnit
        (
            allUnitsInOrder[i].unit
            , allUnitsInOrder[i].name
            , os
            , restAmount
            , firstUnitFound );
    }
}
}

```

我们注意到，通过数组allUnitsInOrder，每个Unit实例都和一个名字建立了一一映射关系；这表明“名字”应该是Unit的一个属性。

为了更好的封装，Unit最好不要提供getName()这样的Getter方法。既然我们调用getName()的目的是为了将名字输出到std::ostream中，那么我们就让Unit来完成这项职责。代码如下：

```

struct Unit
{
protected:
    Unit( const std::string& name);
    Unit( unsigned int conversionFactor
        , const Unit& refUnit
        , const std::string& name);

.....
private:
    const std::string m_name;

private:
    friend
    std::ostream& operator<<(std::ostream& os, const Unit& unit)
    {
        os << unit.m_name;
        return os;
    }
};

```

这样，Length的format代码就可以重构为：

```

namespace
{
    void formatOnUnit
    ( const Unit& unit
    , std::ostream& os
    , Amount& restAmount
    , bool& firstUnitFound)
    {
        unsigned int amount = restAmount / unit.toAmountInBaseUnit(1);
        if(amount > 0)
        {
            if(firstUnitFound) os << " ";
            os << amount << " " << unit;
            firstUnitFound = true;
        }
        restAmount %= unit.toAmountInBaseUnit(1);
    }
}

void Length::format(std::ostream& os) const
{
.....
    const Unit& allUnitsInOrder[] = {MILE, YARD, FEET, INCH};

    for(size_t i=0; i<SIZEOF(allUnitsInOrder); i++)
    {
        formatOnUnit(allUnitsInOrder[i], os, restAmount, firstUnitFound);
    }
}

```

以函数对象取代函数

这时，我们发现函数formatOnUnit的后三个参数都是“InOut”类型的，它们的作用只是在不同的调用中维持状态。

对于这种类型的函数，Kent Beck给出的建议是：把它转换成类，让函数调用过程中不要出现这些语义模糊，仅为内部实现机制而存在的参数。

```
namespace
{
    struct Formatter
    {
        Formatter(std::ostream& os, const Amount& amountInBaseUnit)
            : m_os(os)
            , m_firstUnitFound(false)
            , m_restAmount(amountInBaseUnit) {}

        void operator()(const Unit& unit)
        {
            unsigned int amount =
                m_restAmount / unit.toAmountInBaseUnit(1);
            if(amount > 0)
            {
                if(m_firstUnitFound) m_os << " ";
                m_os << amount << " " << unit;
                m_firstUnitFound = true;
            }
            m_restAmount %= unit.toAmountInBaseUnit(1);
        }

        private:
            const Unit& m_os;
            bool m_firstUnitFound;
            Amount m_restAmount;
    };
}

void Length::format(std::ostream& os) const
{
    ...
    const Unit& allUnitsInOrder[] = {MILE, YARD, FEET, INCH};

    Formatter formatter(os, getAmountInBaseUnit());
    for(size_t i=0; i<SIZEOF(allUnitsInOrder); i++)
    {
        formatter(allUnitsInOrder[i]);
    }
}
```

分离关注点

把肮脏的重复代码消除之后，MILE, YARD等名字就开始显得很刺眼。

由于format算法确实需要如下两个知识：

- 系统定义了哪些LengthUnit实例
- 现有LengthUnit实例之间的大小顺序

而这两个知识由LengthUnit来定义和维护，如果让format再次明确的列出它们，则会造成重复。同时，还会造成Length对具体的LengthUnit实例的耦合。

所以，我们应该通过LengthUnit来获取这些知识，而不是重复性的说明。

将这两个知识合在一起，则是一个新的概念：“LengthUnit实例有序集合”。

```
void Length::format(std::ostream& os) const
{
    .....
    Formatter formatter(os, getAmountInBaseUnit());

    std::vector<const UNIT&>& allUnitsInOrder = \
        LengthUnit::getAllUnitsInOrder();

    std::for_each( allUnitsInOrder.begin()
                  , allUnitsInOrder.end()
                  , formatter);
}
```

ITERATOR vs. TDA

如果一个系统需要某种具体的集合，比如Students。有很大的概率，你可以看到这样的实现方式：

```
const std::vector<Student>& Course::getAllAttendents() const;
```

然后，Course的客户代码可能如下：

```
const std::vector<Students>& attendents = course.getAllAttendents();

std::vector<Students>::const_iterator i = attendents.begin();
for(; i != attendents.end(); i++) {...}
```

这样的设计，让客户直接耦合了一个具体的容器类型。如果随后发现用std::list是更好的选择，这个变更将会波及Course的所有客户。

C++迭代器为各种容器类的访问提供了统一的界面，让容器的用户可以针对不同的容器写出相同的访问代码，这就让用户代码较容易的与具体的容器类型解耦。所以，程序员可以通过C++的“别名机制”对此进行解耦：

```
typedef std::vector<Student> Students;
```

```
const Students& Course::getAllAttendents() const;
```

然后，Course的客户代码改变为：

```
const Students& attendents = course.getAllAttendents();  
Students::const_iterator i=attendents.begin();  
for(; i != attendents.end(); i++) {...}
```

但故事并没有就此结束。此设计仍然存在一个问题：它隐含着一个重要的契约，即抽象数据类型Students必须提供迭代器作为遍历机制。

如果由于内存，性能或算法的原因，在STL库中找不到合适的容器类型，就需要自己来定义新容器。但由于这个容器是为当下的特殊需要而定义的，当下看不到任何重用的机会，所以无须定义一个通用容器；那么最简单的选择就是定义一个具体的Students类。

为了不让客户代码受到影响，就必须为Students提供迭代器的访问方式。

```
struct Students  
{  
    struct iterator {...};  
  
    const_iterator begin() const;  
    const_iterator end() const;  
    ....  
private:  
    // 为了内存，性能或算法原因而自定义的数据结构  
};
```

C++迭代器的功能非常灵活和强大，用户可以以任意的方式遍历容器，可以对容器的任意一个区段应用排序和查找算法，等等。

站在“容器”这个通用概念提供者的角度，提供这样的灵活性，从而让容器具备更好的可重用性，是非常重要的。

但Students是一个为了具体需要而存在的类，与STL所提供的通用容器不同，其目标不在于广泛的可重用性，而是为了在一个具体系统中满足具体的功能要求。

在这种情况下，迭代器的灵活性是我们并不需要的，为了定义迭代器而花费的成本也就没有太大价值。

不仅价值不大，很多时候反而会让用户付出不必要的代价。比如，在这个例子中，如果所有用户对Students访问都是“全面遍历”方式，那么所有用户都需要重复的编写如下模式的代码：

```
Students::const_iterator i = attendents.begin();
for(; i != attendents.end(); i++) {...}
```

即便是写成下面这样，也没有让情况好多少。

```
std::for_each(attendents.begin(), attendents.end(), doSth);
```

这种情况下，Tell, Don't Ask原则往往是更好的选择。

如果用户对于Students的访问需求是明确的，比如，是为了计算总成绩，那么就让Students直接提供相应的方法即可。

```
struct Students
{
    Score getTotalScores() const;
    .....
private:
    // 为了内存，性能或算法原因而自定义的数据结构
};
```

如果不同用户遍历Students有着不同的目的，并且那些目的不属于Students的职责，则可以通过策略模式来应对。

```
struct StudentVisitor
{
    virtual void visitStudent(const Student& student) const = 0;
    virtual ~StudentVisitor() {}
};

struct Students
{
    Score visitAllStudents(StudentVisitor* visitor) const;
    .....
private:
    // 为了内存，性能或算法原因而自定义的数据结构
};
```

或使用模版：

```
struct Students
{
    template <typename StudentVisitor>
    Score visitAllStudents(StudentVisitor visitor) const;
```

```
.....  
private:  
    // 为了内存，性能或算法原因而自定义的数据结构  
};
```

无论哪种方案，我们都把对于内部数据结构的遍历控制在了Student内部。这样设计，即消除了用户代码的重复，又无须提供复杂的迭代器机制。

由于这样的设计有着更好的封装性，所以，即便STL容器能够满足需求，也不应该直接使用它，而是把其当作Students的一个内部实现细节。如下：

```
struct Students  
{  
    Score getTotalScores() const;  
    .....  
private:  
    typedef std::vector<Student> StudentList;  
    StudentList m_allStudents;  
};
```

上述的讨论，仍然是基于如何对数据进行访问的问题。更本质的问题是，任何一个通用容器都只是数据，而没有行为（容器本身的行为，只是为了操作容器本身）。一个具备集合概念的类不应该只有数据而没有自己的行为。所以，通用容器类只应该作为具体集合概念类的内部数据，而不应该被当作集合概念类的直接表示。

提取类

回到我们的问题：LengthUnit::getAllUnitsInOrder()方法的返回值类型为std::vector。而Length真正需要的是一个“LengthUnit实例有序集合”。根据之前讨论的结论，我们让LengthUnit返回一个更加抽象的数据类型：OrderedUnits。

由于访问OrderedUnits的方式非常明确——将所有Unit实例从大到小遍历一遍，以格式化一个Length对象。因此，可以把“遍历”职责移到OrderedUnits。

```
struct UnitsVisitor  
{  
    virtual void visit(const Unit& unit) = 0;  
  
    virtual ~UnitsVisitor() {}  
};
```

```

///////////
struct OrderedUnits
{
    void addUnit(const Unit& unit)
    {
        m_unitsInOrder.push_back(unit);
        std::sort( m_unitsInOrder.begin()
                  , m_unitsInOrder.end());
    }

    void accept(UnitsVisitor* visitor) const
    {
        for( Units::const_reverse_iterator i = m_unitsInOrder.begin();
              i != m_unitsInOrder.end(); i++)
        {
            visitor->visit(*i);
        }
    }

private:
    typedef std::vector<const Unit&> Units;
    Units m_unitsInOrder;
};

```

为了能够和OrderedUnits配合工作，我们需要让Formatter实现UnitsVisitor接口。对于Formatter的现有实现，我们只需要把operator()改为visit即可。

```

struct Formatter : public UnitsVisitor
{
    .....
    void operator()(const Unit& unit)
    void visit(const Unit& unit){...}
};

///////////
void Length::format(std::ostream& os) const
{
    if(m_amount == 0)
    {
        os << 0 << " " << "INCH";
        return;
    }

    Formatter fomatter(os, getAmountInBaseUnit());

    OrderedUnits* orderedUnits = UNIT::getAllUnitsInOrder();
    orderedUnits->accept(&fomatter);
}

```

提高表达力

现在format的代码已经相当干净，除了那个if结构。

那条语句事实上也是在做format的工作，只不过针对的是数量为0的Length对象。所以，我们先尝试将它移入Formatter。

```
void Formatter::visit(const Unit& unit)
{
    if(m_restAmount == 0)
    {
        os << 0 << " " << unit.getBaseUnit();
        return;
    }

    unsigned int amount = m_restAmount / unit.toAmountInBaseUnit(1);
    if(amount > 0)
    {
        if(m_firstUnitFound) m_os << " ";
        m_os << amount << " " << unit;
        m_firstUnitFound = true;
    }

    m_restAmount %= unit.toAmountInBaseUnit(1);
}

///////////////////////////////
void Length::format(std::ostream& os) const
{
    Formatter formatter(os, getAmountInBaseUnit());

    OrderedUnits* orderedUnits = LengthUnit::getAllUnitsInOrder();
    orderedUnits->accept(&formatter);
}
```

不幸的是，这是一次错误的重构。对于Length(0, MILE)这样的对象，它格式化的结果是：0 INCH 0 INCH 0 INCH 0 INCH；而对于Length(12, INCH)，则输出为：1 FEET 0 INCH。

之所以产生这样的结果，是因为OrderedUnits没有提供终止遍历的机制。它总是会遍历所有的Units，并依次调用UnitsVisitor。

现在，我们给OrderedUnits加上这样的机制：通过Visitor的返回值来告诉OrderedUnits是否需要继续遍历。代码如下：

```
struct UnitsVisitor
{
    virtual bool visit(const Unit& unit) = 0;

    virtual ~UnitsVisitor() {}
};

void OrderedUnits::accept(UnitsVisitor* visitor) const
```

```

{
    for( Units::const_iterator i = m_unitsInOrder.begin();
        i != m_unitsInOrder.end(); i++)
    {
        if(!visitor->visit(*i)) break;
    }
}

```

现在Formatter可以重构为：

```

bool Formatter::visit(const Unit& unit)
{
    if(m_restAmount == 0)
    {
        os << 0 << " " << unit.getBaseUnit();
        return false;
    }

    unsigned int amount = m_restAmount / unit.toAmountInBaseUnit(1);
    if(amount > 0)
    {
        if(m_firstUnitFound) m_os << " ";
        m_os << amount << " " << unit;
        m_firstUnitFound = true;
    }

    m_restAmount %= unit.toAmountInBaseUnit(1);

    return (m_restAmount != 0);
}

```

提取函数

现在visit函数工作正确，但其内部却包含两个职责：一个是零对象的序列化，一个是非零对象序列化。为了提高代码的清晰度，我们需要将两个职责分开——分到两个不同的函数里。

```

bool Formatter::formatZeroLength(const Unit& unit)
{
    os << 0 << " " << unit.getBaseUnit();
    return false;
}

///////////
bool Formatter::formatNonZeroLength(const Unit& unit)
{
    unsigned int amount = m_restAmount / unit.toAmountInBaseUnit(1);
    if(amount > 0)
    {
        if(m_firstUnitFound) m_os << " ";
        m_os << amount << " " << unit;
        m_firstUnitFound = true;
    }

    m_restAmount %= unit.toAmountInBaseUnit(1);
}

```

```

    return (m_restAmount != 0);
}

///////////
bool Formatter::visit(const Unit& unit)
{
    return (m_restAmount == 0) ?
        formatZeroLength(unit) :
        formatNonZeroLength(unit);
}

```

消除魔数

我们发现代码中由多个地方都直接使用了空格。这事实上是一种字符串类型的魔数。

并且，虽然表现都是空格，但它们所代表的语义是不同的。一类代表的是“数量和单位之间的分隔符”，另外一类代表的是“单位之间分隔符”。所以，我们要分别给它们定义清晰的语义，以提高代码的表现力。

```

const std::string SINGLE_SPACE = " ";
const std::string DELIMITER = SINGLE_SPACE;
const std::string UNIT_DELIMITER = SINGLE_SPACE;

///////////
void Formatter::formatOnUnit(const Amount& amount, const Unit& unit)
{
    os << amount << DELIMITER << unit;
}

///////////
bool Formatter::formatNonZeroLength(const Unit& unit)
{
    .....
    if(m_firstUnitFound) m_os << UNIT_DELIMITER;
    .....
}

```

分离关注点

现在我们注意到下面代码中存在嵌套的if语句。这让代码的逻辑更加难以理解。尽管它的复杂度还没有发展到难以容忍的地步，但让代码更加清晰总是应该鼓励的行为。

```

bool Formatter::formatNonZeroLength(const Unit& unit)
{
    unsigned int amount = m_restAmount / unit.toAmountInBaseUnit(1);
    if(amount > 0)
    {
        if(m_firstUnitFound) m_os << UNIT_DELIMITER;
    }
}

```

```

        formatOnUnit(amount, unit);
        m_firstUnitFound = true;
    }

    m_restAmount %= unit.toAmountInBaseUnit(1);
    return (m_restAmount != 0);
}

```

然后我们发现，与m_firstUnitFound有关的逻辑，是在控制“单位之间的分隔符”的输出。所以，我们现把它抽取成函数。

```

void Formatter::formatUnitDelimiter()
{
    if(m_firstUnitFound) m_os << UNIT_DELIMITER;
    m_firstUnitFound = true;
}

///////////
bool Formatter::formatNonZeroLength(const Unit& unit)
{
    .....
    if(amount > 0)
    {
        formatUnitDelimiter();
        formatOnUnit(amount, unit);
    }
    .....
}

```

然后我们发现如果把对其的调用放到formatUnit里会让现由if结构更清晰。

```

void Formatter::formatOnUnit(const Amount& amount, const Unit& unit)
{
    formatUnitDelimiter();
    os << amount << DELIMITER << unit;
}

///////////
bool Formatter::formatNonZeroLength(const Unit& unit)
{
    .....
    if(amount > 0) formatOnUnit(amount, unit);
    .....
}

```

更进一步，如果能够把实现写成下列的形式，formatUnit的表达方式就更加简洁。

```

void Formatter::formatUnit(const Amount& amount, const Unit& unit)
{ os << UNIT_DELIMITER << amount << DELIMITER << unit; }

```

这种情况下，UNIT_DELIMETER背后存在逻辑控制，所以，我们需要把它的类型定义为一个类。并且这个类应该能够向一个std::ostream实例进行输出。所以它被实现为下面的方式：

```
namespace
{
    struct Delimeter
    {
        Delimeter(const std::string& delimiter)
            : m_isFirstTime(false) {}

        void toStream(std::ostream& os) const
        {
            if(not m_isFirstTime) os << UNIT_DELIMETER;
            m_isFirstTime = true;
        }

    private:
        mutable bool m_isFirstTime;

        friend std::ostream& operator<<
            ( std::ostream& os
            , const UnitDelimeter& delimeter)
    {
        delimeter.toStream(os);
        return os;
    }
};

}
```

由于它的状态变迁与单个Length对象的序列化过程有关，所以它应该是Length的实例变量。

```
///////////
struct Formatter : public UnitsVisitor
{
    Formatter(std::ostream& os, const Amount& amountInBaseUnit)
        : m_os(os)
        , m_unitDelimeter(UNIT_DELIMETER)
        , m_restAmount(amountInBaseUnit) {}

    ...

private:
    void formatUnit(const Amount& amount, const Unit& unit)
    { os << m_unitDelimeter << amount << DELIMITER << unit; }

    ...

private:
    std::ostream& m_os;
    Delimeter m_unitDelimeter;
    Amount m_restAmount;
};
```

提高内聚性 *

我们现在再回到Formatter的实现里，发现一些表达式都是围绕unit.toAmountInBaseUnit(1)展开的，这究竟意味着什么？

```
bool Formatter::formatNonZeroLength(const Unit& unit)
{
    unsigned int amount = m_restAmount / unit.toAmountInBaseUnit(1);
    if(amount > 0) formatOnUnit(amount, unit);
    m_restAmount %= unit.toAmountInBaseUnit(1);

    return (m_restAmount != 0);
}
```

稍加分析，我们就能得知，表达式 amountInBase/unit.toAmountInBaseUnit(1)事实上是toAmountInBaseUnit的逆操作——把“基准单位数量”转化为“当前单位的数量”。所以我们将其移动到Unit。

```
struct Unit
{
    .....

public:
    Amount byAmountInBaseUnit(const Amount& amountInBaseUnit) const
    { return amountInBaseUnit / toAmountInBaseUnit(1); }

    bool Formatter::formatNonZeroLength(const Unit& unit)
    {
        unsigned int amount = unit.byAmountInBaseUnit(m_restAmount);
        if(amount > 0) formatOnUnit(amount, unit);
        m_restAmount -= unit.toAmountInBaseUnit(amount);

        return (m_restAmount != 0);
    }
}
```

如果从更加本质的角度分析，Unit应该提供的服务应该为“任意两个单位之间的数量转化”：

$$(AmountInUnit1, Unit1) \Leftrightarrow (AmountInUnit2, Unit2)$$

因此，Unit应该提供的通用接口应该为：

```
struct Unit
{
    .....

    Amount toAmountInUnit(const Amount& amount, const Unit& unit);
    Amount byAmountInUnit(const Amount& amount, const Unit& unit);
```

```
};
```

从函数所表达的语义来看，Amount必须是浮点数。这其实也是Unit的本质抽象的内在性质。

如果对于浮点数的处理没有引入任何复杂度的话，那么无疑应该建立这样的本质抽象。但我们不希望引入当前没有需要的复杂度，我们希望仍然把所有的计算建立在更为简单的整数基础上。

在整数计算的约束下，Unit的现有函数：

- toAmountInBaseUnit
- byAmountInBaseUnit

在Unit之间的转换系数都是整数的前提下，前者可以保证整数演算；但后者仍然需要浮点数的支持。为了避免这样的尴尬局面，我们给后者一个语义更加明确的名字。

```
struct Unit
{
    .....
public:
    unsigned int toMultiple(const Amount& amountInBaseUnit) const
    { return (unsigned int)(amountInBaseUnit/toAmountInBaseUnit(1)); }

    bool Formatter::formatNonZeroLength(const Unit& unit)
    {
        unsigned int multiple = unit.toMultiple(m_restAmount);
        if(multiple > 0) formatOnUnit(multiple, unit);
        m_restAmount -= unit.toAmountInBaseUnit(multiple);

        return (m_restAmount != 0);
    }
}
```

缩小依赖范围

下面的代码违反了“笛米特法则”：

```
void Length::format(std::ostream& os) const
{
    Formatter formatter(os, getAmountInBaseUnit());

    OrderedUnits* orderedUnits = LengthUnit::getAllUnitsInOrder();
    orderedUnits->accept(&formatter);
}
```

这样的设计，让Length不必要的耦合了OrderedUnits。所以，我们将其重构为：

```
void Length::format(std::ostream& os) const
{
    Formatter formatter(os, getAmountInBaseUnit());
    LengthUnit::visitAllUnitsInOrder(&formatter);
}
```

而LengthUnit的实现则变为：

```
struct LengthUnit : public Unit
{
    .....
public:
    static OrderedUnits* getAllUnitsInOrder();
    static void visitAllUnitsInOrder(UnitsVisitor* visitor);
};

namespace
{
    OrderedUnits& getOrderedUnits()
    {
        static OrderedUnits orderedUnits;
        return orderedUnits;
    }
}

///////////////////////////////
void LengthUnit::visitAllUnitsInOrder(UnitsVisitor* visitor)
{ getOrderedUnits().accept(visitor); }
```

笛米特法则 & TDA

笛米特法则的定义是：只与你的直接朋友交谈 (talk only to your immediate friends)。

直观的说，就是不应该出现类似下面形式的代码。

```
struct Foo
{
    void getBar() const { return bar; }

private:
    Bar bar;
};

struct Object
{
    .....
```

```
void doSomething() { foo.getBar().doSomething(); }

private:
    Foo foo;
};
```

所谓“直接朋友”，就是你直接持有的，或者通过参数传入的对象。比如，在此例中，Foo是Object的“直接朋友”，Bar是Foo的“直接朋友”。

但Object并没有直接持有Bar，所以Bar不是Object的“直接朋友”。因此，Object不应该与Bar交谈，即不应该调用Bar的方法——事实上，Object压根就不应该知道Bar的存在。

违反“笛米特法则”所造成的问题，从getBar()这个Getter方法就能看出，它破坏了Foo的封装，了解到了Foo的内部实现细节，从而增强了Object和Foo之间的耦合。另外，Object耦合了它本不该知道的类型Bar。

按照“最少知识原则”，这些知识不应该为Object所见。所以，上述代码应重构为：

```
struct Foo
{
    void doSomething() { bar.doSomething(); }

private:
    Bar bar;
};

struct Object
{
    void doSomething() { foo.doSomething(); }

private:
    Foo foo;
};
```

自动注册

我们创建了OrderedUnits，但问题是把所有的LengthUnit实例放入OrderedUnits？

一种方案是写一个初始化函数：

```
namespace
{
```

```

////////// OrderedUnits& getOrderedUnits()
{
    static OrderedUnits orderedUnits;
    return orderedUnits;
}

void addUnit(const Unit* const unit)
{
    getOrderedUnits().addUnit(&unit);
}
}

void LengthUnit::initOrderedUnits()
{
    addUnit(&MILE);
    addUnit(&YARD);
    addUnit(&FEET);
    addUnit(&INCH);
}

```

这种方案的缺点是：重复！每次你创建一个新的Unit，都要记得来这里添加一条addUnit语句。并且要求用户必须在系统初始化阶段一定要预先调用LengthUnit::initOrderedUnits()。

但如果addUnit可以被自动完成，则上述问题都将不复存在。问题是如何自动完成？

一种方法是在每个LengthUnit的构造里完成。这样每个LengthUnit被创建时，将会自动将自己注册过去。

但这种方法有两个问题：1. 让LengthUnit对OrderedUnits产生了耦合；2. 最重要的是，由于LengthUnit实例的创建都是Lazy的，所以，如果没有人引用某个实例，那么此实例将无法得到创建和注册。

所以，我们必须建立一种强制性的机制，让每个实例都可以在系统初始化的阶段自动完成。而全局变量，或者非局部静态变量会在这个阶段自动得到构造。对，是的，我们要利用非局部静态变量的构造函数来完成自动注册。

```

namespace
{
    .....
    struct UnitAutoRegistry
    {
        UnitAutoRegistry(const Unit& unit)
        { getOrderedUnits().addUnit(&unit); }
    };
}

```

```

///////////
const LengthUnit& LengthUnit::getINCH()
{
    static LengthUnit unit;
    return unit;
}
namespace
{ UnitAutoRegistry INCH_REGISTRY(INCH); }

///////////
const LengthUnit& LengthUnit::getFEET()
{
    static LengthUnit unit(12, getINCH());
    return unit;
}
namespace
{ UnitAutoRegistry FEET_REGISTRY(FEET); }

```

看到这样的解决方案，你一定会质疑，现在确实自动注册了，但每定义一个单位，你都要手工的添加一个Registry。这重复不是仍然在吗？

没错，但我们有强大的工具——宏。它一直都是解决这类重复代码模式的良药。

```

#define DEFINE_UNIT(unit, eq, conversionFactor, refUnit) \
const LengthUnit& LengthUnit::get##unit() \
{ \
    static LengthUnit instance(conversionFactor, get##refUnit)); \
    return instance; \
} \
namespace \
{ UnitAutoRegistry unit##_REGISTRY(unit); }

```

需要特别注意的是，能够使用自动注册的前提是，要保证没有其它全局变量或非局部静态变量在构造时直接或间接的访问AllLengthUnits。否则，由于模块间全局变量或者非局部静态变量的初始化顺序并不确定，当AllLengthUnits被访问时，LengthUnit实例可能尚未注册，从而导致错误的系统行为。

当然，定义非基本类型的全局变量或非局部静态变量本来就是一种不安全的行为。任何C++团队都应该禁止这样的行为。在这种情况下，我们就无须担心这样的问题。

但我们的自动注册机制却恰恰是利用了“全局”变量的性质来完成的，这就又违背了这种限制。但它没有数据，没有跨模块访问其它数据或函数，没有任何公开服务——它本身是安全的。

如果不能保证这一点，在没有得到更好的解决方案之前，宁愿忍受一定的重复，也要保证功能的正确性。

依赖注入

TBW

需求八

问题描述

增加一种新的Length对象输出格式，

- 以Inch为单位输出任何Length对象
- 数量和单位之间以一个空格分隔

例如：

- Length(2, FEET) => 24 INCH
- Length(2, YARD) => 72 INCH

当输出一个Length对象时，用户可以在两种输出格式中自由选择任何一种。

SPIKE

这是一种新的格式化。为了寻找解决问题的思路，我们有必要参考一下已经存在的类似需求的设计。

之前，我们将格式输出实现成了流模型，代码如下：

```
void Length::format(std::ostream& os) const {...}

///////////////////////////////
std::ostream&
operator<<(std::ostream& os, const Length& length)
{
    format(os);
    return os;
}
```

由于要加入一种新的格式化方式，我们似乎可以这么做：

```
enum Format
{
    FORMAT1,
    FORMAT2
};

/////////////////////////////
void Length::format1(std::ostream& os) const {...}

/////////////////////////////
void Length::format2(std::ostream& os) const {...}
```

```

///////////
std::ostream&
operator<<(std::ostream& os, const Length& length, Format format)
{
    if(format == FOMRAT1) length.format1(os);
    else length.format2(os);

    return os;
}

```

不幸的是，函数**operator<<**有其格式要求，我们无法传入三个参数。所以，我们必须想其它办法让**format**的信息能够在函数内得到访问。

UI设计

一种办法是通过**Length**构造函数传入，并将其保存起来：

```

struct Length : public Quantity<LengthUnit>
{
    Length(const Amount& amount
           , const LengthUnit& unit
           , Format format);
    .....
private:
    Format m_format;
};

```

对于这个需求而言，这可以工作。但我们希望用户以这样的方式在创建一个**Length**对象吗？

```
Length(2, MILE, FORMAT1);
```

为了让**Length**的构造仍然保持它应该的样子，聪明的我们马上会想起**Setter**方法。

```

struct Length : public Quantity<LengthUnit>
{
    void setFormat(Format format);
    .....
private:
    Format m_format;
};

```

但更根本的问题是，`format`并不是`Length`的内在属性，它仅为格式化输出而存在。其生命周期和变化率⁵与`Length`的内在属性是不一致的。我们没有理由让`Length`在无须进行格式化的时候，却要始终背负着这样的属性。

如果不应该把它保存在`Length`里，则只能通过其它方式传入。由于`operator<<`要求只能有两个参数，所以，我们可以把后面两个参数合成一个新的类型。

```
///////////////////////////////
struct Length : public Quantity<LengthUnit>
{
    .....
public:
    void format1(std::ostream& os);
    void format2(std::ostream& os);
};

///////////////////////////////
struct FormatLength
{
    FormatLength(const Length& length, Format format);

private:
    void toStream(std::ostream& os) const
    {
        if(format == FOMRAT1) m_length.format1(os);
        else m_length.format2(os);
    }

private:
    Format m_format;
    const Length& m_length;

private:
    friend std::ostream&
    operator<<(std::ostream& os, const FormatLength& format)
    {
        formatLength.toStream(os);
        return os;
    }
};
```

这种情况下，用户的使用方式就变为：

```
std::cout << FormatLength(Length(1, MILE), FORMAT1) << std::endl;
std::cout << FormatLength(Length(2, INCH), FORMAT2) << std::endl;
```

Not bad，这样的UI已经可以接受：它的语义某种程度上看起来像C语言的`printf`。在没有更好的解决方案之前，我们先使用它。

⁵ 见Kent Beck《实现模式》3.2.6

SPIKE

TBW

开闭问题

审视一下Length的实现，就会再次看见令人生厌的有枚举伴随的if-else，并且很有潜力发展成switch-case。

```
void
FormatLength::toStream(std::ostream& os) const
{
    if(format == FOMRAT1) m_length.format1(os);
    else m_length.format2(os);
}
```

这样的设计方式意味着：任何时候只要增加或删除一种新的输出格式，这段代码都必须修改。很明显，它违背了“开放封闭原则”。

根据“开放封闭原则”，我们需要让Length对于输出格式的变化是封闭的，然后提供一个抽象，让其对于输出格式的扩展是开放的。

由于不断扩展的是“输出格式”，所以我们将抽象接口定义为Formatter。其职责是“将一个长度对象按照指定格式进行格式化，并输出到指定的流对象”。按照这个描述，我们可以将其定义为：

```
struct Formatter
{
    virtual
    void format(std::ostream& os, const Length& length) const = 0;

    virtual ~Formatter() {}
};
```

根据Formatter的定义，我们就可以将Length相关代码修改为：

```
///////////////////////////////
struct Length : public Quantity<LengthUnit>
{
    .....
public:
    void format(std::ostream& os, const Formatter& formatter) const;
```

```

};

///////////////
struct FormatLength
{
    FormatLength(const Length& length, const Formatter& formatter);

private:
    void toStream(std::ostream& os) const
    { formatter.format(os, m_length); }

private:
    const Formatter& m_formatter;
    const Length& m_length;

private:
    friend std::ostream&
    operator<<(std::ostream& os, const FormatLength& format);
}

```

这样的实现已经让Length与具体的Formatter彻底解开耦合。

重构原有Formatter

在实现新的Formatter之前，我们需要先把原来的格式化相关代码重构为一个Formatter。既然它是将所有LengthUnit从大到小级联式格式化，我们就把它称做CascadeFormatter。然后我们写下它的结构性代码：

```

struct CascadeFormatter : public Formatter
{
    void format(std::ostream& os, const Length& length) const;
};

```

原来的格式化是通过Length的format方法完成的，如下：

```

void Length::format(std::ostream& os) const
{
    Formatter formatter(os, getAmountInBaseUnit());
    LengthUnit::visitAllUnitsInOrder(&formatter);
}

```

我们先将这段代码移过来：

```

void CascadeFormatter::format
( std::ostream& os
, const Length& length) const
{
    Formatter formatter(os, getAmountInBaseUnit());
    LengthUnit::visitAllUnitsInOrder(&formatter);
}

```

然后我们发现原有实现中也有一个类叫Formatter。这个不难，先将其改成其它名字。由于它是一个模块内部的类，外部不可见，在没有想到更好的名字之前，我们先将其命名为FormatterImpl。然后把它从Length模块移过来。

```
namespace
{
    struct FormatterImpl : public UnitsVisitor {...};

void CascadeFormatter::format
( std::ostream& os
, const Length& length) const
{
    FormatterImpl formatter(os, getAmountInBaseUnit());
    LengthUnit::visitAllUnitsInOrder(&formatter);
}
```

然后我们发现FormatterImpl需要通过Length的私有方法getAmountInBaseUnit来获取“基准单位数量”。一个简单的解决办法是把私有方法公开化，这样代码就变为：

```
void CascadeFormatter::format
( std::ostream& os
, const Length& length) const
{
    FormatterImpl formatter(os, length.getAmountInBaseUnit());
    LengthUnit::visitAllUnitsInOrder(&formatter);
}
```

但这样就造成了Length暴露了客户不需要的接口。

保护封装

Tell, Don't Ask原则是解决封装问题的有力武器。如果Formatter的format方法传入的不是length对象，而是格式化算法所需要的数据，我们就不必破坏Length的封装。

```
void CascadeFormatter::format
( std::ostream& os
, const Length& length
, const Amount& amountInBaseUnit) const
{
    FormatterImpl formatter(os, amountInBaseUnit);
    LengthUnit::visitAllUnitsInOrder(&formatter);
}
```

而Length相关的代码则变为：

```
void
Length::format( std::ostream& os
                , const Formatter& formatter) const
{ formatter.format(os, getAmountInBaseUnit()); }

void
FormatLength::toStream(std::ostream& os) const
{
    formatter.format(os, m_length);
    m_length.format(os, formatter);
}
```

确定接口

我们的重构改变了CascadeFormatter的format参数。所以，我们也需要把接口Formatter进行同步修改：

```
struct Formatter
{
    virtual void format
    ( std::ostream& os
    , const Length& length
    , const Amount& amountInBaseUnit) const = 0;

    virtual ~Formatter() {}
};
```

等一下！由于接口的定义需要高度抽象，以满足各种实现类的需要，所以当我们修改接口的时候，一定要非常慎重。

让我们暂时停下来，仔细想想：对于任意一个Formatter，为了格式化一个length对象，它究竟需要从Length获取的信息是什么？

作为Length对象，它能够提供的信息无非两种组合：

1. 数量 + 单位
2. 基准单位数量

两者对于表现一个Length而言是等价的。

但后者却可以通过前者计算得到，所以我们似乎应该提供第一种组合。这种组合要求Length必须以（数量，单位）二元组的方式保存数据。由此产生了如下问题：

- Length的“单位”信息对于Length的格式化是否是必要的？

- 假设是必要的，那么Length应该保存什么单位？
 - 假如用户通过Length(1, FEET)的方式创建了一个对象。它内部究竟应该保存1 FEET，还是12 INCH？
 - 如果必须保存用户指定的单位，那么Length(6, INCH) + Length(6, INCH)得到的Length对象应该保存什么单位？
-

对象的哲学

当一个对象被创建出来之后，对象自身就代表了它所表现的事物。

比如，我们拿出一根棍子，你测量它，或不测量它，棍子的长度就在那里，不增不减。

而Length(12, INCH)和Length(1, FEET)只是构造这个对象的方式，而不是结果。作为结果的对象都毫无差异的代表了那根棍子的长度——与测量，与数量，与单位都无关的长度。

一个更直观的例子是：你准备创建一个1000元的账户。而你存入1000个1元硬币，还是存入10张100元纸币，所创建的账户都是完全相同的。你对账户进行的任何操作，其结果都不会因为你最初存入形式的不同而有任何差异。

所以，任意两个对象，无论构造的方式有多大的差异，只要它们表现的是同一事物，它们的外在行为就必须是完全一致的，只要这一点能够保证，其“内部表现”是否一致则无关紧要。

反过来，对于同一类型的两个对象，只要所有外在行为完全一致，我们就认为它们描述的是同一事物，无论其“内部表现”有多么大的差异。

这就是为什么我们应该依赖一个对象的外部接口，而不要依赖其“内部表示”的关键哲学。任何试图获取对象“内部表示”的企图，都是违反逻辑的。它造成的不稳定和系统僵化都只是这种本质错误所表现出来的“症状”罢了。

既然Length(12, INCH)和Length(1, FEET)的内部表示并不确定，那么它的（数量，单位）组合，就会有多种可能。比如，一个Length(12, INCH)对象既可以表现为

(12, INCH), 又可以表现为(1, FEET)。既然任何一种组合都可以，用户则无须关心一个Length是以何种方式组合的。

至于必要性，既然(基准单位数量, 基准单位)也是一种合理的组合，那么理论上如果“基准单位”是一种人人皆可自由获取的知识，那么就没有再传递的必要。但在我们当前的设计里，“基准单位”必须通过某个Unit实例来调用getBaseUnit才能获取，所以，我们仍然需要将其传递出去。毕竟(数量, 单位)作为Quantity的本质属性，要比“基准单位数量”这个概念要稳定的多。

所以，我们将Formatter接口定义如下：

```
struct Formatter
{
    virtual void format
        ( std::ostream& os
        , const Amount& amount
        , const Unit& unit) const = 0;

    virtual ~Formatter() {}
};
```

而Length的相关代码则重构为：

```
void
Length::format( std::ostream& os
                , const Formatter& formatter) const
{ formatter.format(os, m_amount, m_unit); }
```

CascadeFormatter的代码则重构为：

```
void CascadeFormatter::format
( std::ostream& os
, const Amount& amount
, const Unit& unit) const
{
    FormatterImpl formatter(os, unit.toAmountInBaseUnit(amount));
    LengthUnit::visitAllUnitsInOrder(&formatter);
}
```

实现需求

现在我们就可以通过实现一个新的Formatter来实现我们的需求。

由于它的格式化是以“基准单位”进行格式化输出，所以我们将其命名为BaseUnitFormatter。

```

struct BaseUnitFormatter : public Formatter
{
    void format
        ( std::ostream& os
        , const Amount& amount
        , const Unit& unit) const
    {
        os << unit.toAmountInBaseUnit(amount)
        << DELIMITER
        << unit.getBaseUnit();
    }
};

```

定义常量

现在，我们已经有了两个Formatter。但我们之前定义的UI需要通过常量来指定输出格式：

```

std::cout << FormatLength(Length(1, MILE), CASCADE_FORMAT);
std::cout << FormatLength(Length(2, INCH), BASEUNIT_FORMAT);

```

为了适应这样的UI，我们需要定义这两个常量：

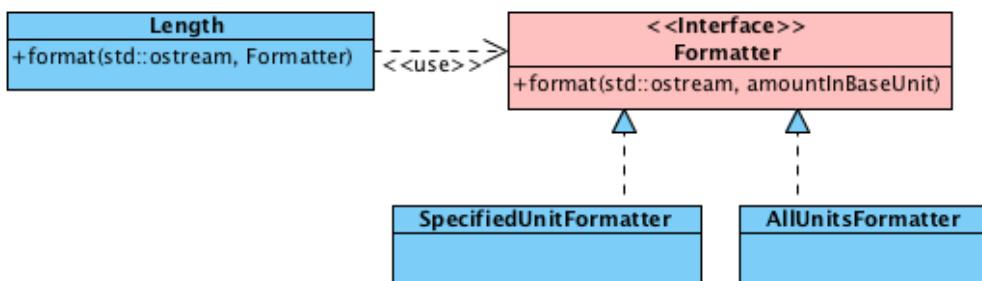
```

#define CASCADE_FORMAT (CascadeFormatter::getInstance())
#define BASEUNIT_FORMAT (BaseUnitFormatter::getInstance())

```

总结

最后来看看我们的实现结构：

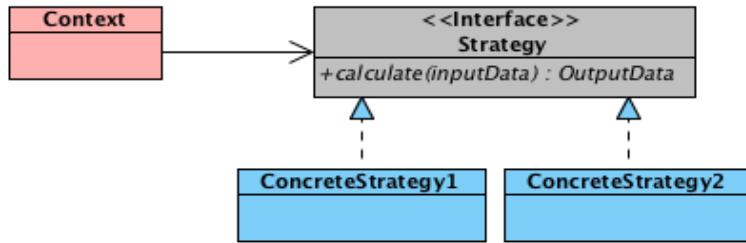


而这个结构，就是“策略模式”的一种具体体现。

STRATEGY & TDA

当算法的责任方需要在多个算法之间进行选择时，通过将“算法责任方”和“具体算法”进行解耦而得到。

一个标准的策略模式结构如下图所示。在图中，Context属于“算法责任方”，而Strategy则是“算法”的抽象。ConcreteStrategy1和ConcreteStrategy2则是“具体算法”。



如果将算法都实现在Context，则Context面临着随着算法变化而变化的风险，这就违背了“开放封闭原则”。通过“分离关注点”，将算法从Context分离出来，并将不同算法分离，各自成为抽象概念“策略”的不同实现。就实现了“开闭”性。

关键的部分在于，用户如何将context和strategy结合在一起？

第一种方法：用户可以持有策略对象，然后将Context实例传递给策略对象，如下：

```
OutputData ConcreteStrategy1::calc(Context* context)
{
    InputData1 data1 = context->getData1();
    InputData2 data2 = context->getData2();
    // 使用data进行策略计算，得到outputData
    return outputData;
}
```

这种方式的问题如下：

- 首先造成了Strategy对于Context的耦合。而Strategy真正需要的是InputData1和InputData2，这就增加了Strategy不必要的耦合（多耦合了一种类型）。
- 所有的Strategy都必须调用一次context::getData1()和context::getData2()才能得到自己真正关心的InputData1和InputData2，造成不要的重复；

- Context所提供的接口，不仅仅可以被Strategy所使用，也可以被其它Context的客户所使用，更多的依赖者会造成Context更加难以变化。如果某一天，Context无法再提供InputData，不仅影响它和Strategy之间的契约，还会影响到更多的客户。

既然第一种方案有这些问题，我们可以将其改进，得到第二种方案：

```
OutputData ConcreteStrategy1::calc(InputData1 data1, InputData2 data2)
{
    // 使用data1和data2进行策略计算，得到outputData
    return outputData;
}
```

然后，客户直接持有Context，和Strategy的某个实例，自己进行传递：

```
strategy.calc(context.getData1(), context.getData2());
```

而这样的方案并没有得到太多改进，虽然现在Strategy不再依赖Context，但对于重复问题没有任何改进，反而进一步的恶化。如果用户进行100次策略运算，就需要100次进行这样的重复：

```
strategy.calc(context.getData1(), context.getData2());
```

第一种方案只是会造成策略代码的重复，但仍然是可控的，因为有多少个策略，就只会重复多少次。但现在，你无法预期客户会调用多少次，而客户每调用一次都会重复一次。尤其是当Strategy需要的参数更多时，这更是一种负担。比如，

```
strategy.calc(
    context.getData1()
    , context.getData2()
    , context.getData3());
```

当然我们有办法消除这样的重复，比如提供一个函数：

```
OutputData calc(Strategy* strategy, Context* context)
{ return strategy->calc(context->getData1(), context->getData2()); }
```

但站在“高内聚”的角度，这样的一个孤魂野鬼，应该归属于某个类。通过它的参数我们可以得知，它要么应该属于Strategy，要么应该属于Context。

如果属于Strategy，那就是第一种方案，我们已经知道它的问题。那剩下的只有Context：

```
OutputData Context::calc(Strategy* strategy)
```

```
{ return strategy->calc(getData1(), getData2()); }
```

一旦移入Context，就没有必要再暴露getData()方法，而将其转化为内部私有实现。比如：

```
OutputData Context::calc(Strategy* strategy)
{ return strategy->calc(m_data1, m_data2); }
```

这样就同时解决了暴露Getter的问题。这就是“Tell, Don't Ask”原则的第三种形式。

采用Tell, Don't Ask原则的另外两个更重要的好处是：

1. 用户代码的稳定：无论Context和Strategy之间的契约如何变化，用户的调用方式始终是：

```
context.calc(strategy);
```

比如，Strategy::calc的参数由两个变为三个，但用户的代码却不会发生改变。

2. Context::calc的实现可以非常灵活，比如：

```
OutputData Context::calc(Strategy* strategy)
{
    //在strategy调用之前的代码，比如：outputData = m_data0 * 5;
    outputData += strategy->calc(m_data1, m_data2);
    //在strategy调用之后的代码，比如：outputData += m_data3;
    return outputData;
}
```

如果采用前两个方案，很难在没有重复，没有增加耦合的情况下做到这一点。所以，策略模式是在“开放封闭”和“Tell, Don't Ask”原则指导下的产物。

“策略模式”应该是GoF经典设计模式中应用最为广泛的一个（很多人应该会对此结论表示强烈质疑，因为在很多C++项目中，“单例模式”才是应用最广泛的）。

这样的结果并不偶然。它是两个重要原则的产物，由此具备的“可扩展性”和“保护封装”，让相关结构具备非常良好的稳定性和弹性。

在CascadeFormatter的实现里，UnitsVisitor和OrderedUnits之间的关系事实上也已经应用了“策略模式”。而Quantity和Unit之间的关系亦是“策略模式”的一种体现。

价值观，原则，模式与实践

事实上，我们是在不知不觉中应用的“策略模式”。这个过程准确的反映了原则和模式的关系。

原则是我们有意识的用来指导我们实践的准则。而模式则是在原则指导下，用来在实践中解决某类问题的常用方法。

之所以要了解模式：是为了让我们在碰到同类问题时，有成熟的经验可以借鉴。但借鉴的时候，仍然需要原则的指导，否则就无法区分一个模式是否适用于当前问题，就会造成滥用模式。

所以，我们需要优先理解和掌握原则。有了原则的指导，不仅可以帮助我们正确的应用现有模式，还可以让我们在实践中总结出新的模式。

而价值观，则更上一层，原则是在价值观的指导下产生的。不同的价值观，会产生截然不同的原则。

我们经常会发现，在针对一个细节非常明确的问题上，双方仍然在争论不休。其根本原因往往在于双方价值观的不同。而经验和背景的差异则往往是导致价值观不同的重要因素。这种情况下，如果不首先统一价值观，则辩论本身已经失去意义。除非想通过辩论来统一价值观。

价值观->原则->模式->实践（图）