PEARSON
Addison Wesley

模式的乐趣

The Joy of Patterns

Using Patterns for Enterprise Development

[美] Brandon Goldfedder 著
熊 节 译

清华大学出版社

模式的乐趣

The Joy of Patterns

Using Patterns for Enterprise Development

[美] Brandon Goldfedder 著

熊 节 译

清 华 大 学 出 版 社

北 京

EISBN: 0-201-65759-7

图书在版编目(CIP)数据

模式的乐趣/(美)高德菲德尔著；熊节译.—北京：清华大学出版社，2003.8
书名原文：The Joy of Patterns: Using Patterns for Enterprise Development
ISBN 7-302-06851-8
Ⅰ.软… Ⅱ.①高 .②熊. Ⅲ.软件设计 Ⅳ TP311 5

中国版本图书馆 CIP 数据核字(2003)第 053944 号

出 版 者：清华大学出版社　　**地　　址**：北京清华大学学研大厦
http://www.tup.com.cn　　**邮　　编**：100084
社总机：010-62770175　　**客户服务**：010-62776969
组稿编辑：汤斌浩
文稿编辑：郭福生
封面设计：立日新
印 刷 者：清华大学印刷厂
发 行 者：新华书店总店北京发行所\清华大学出版社出版发行
开　　本：185×230　**印张**：10.25　**字数**：221 千字
版　　次：2003 年 8 月第 1 版　2003 年 8 月第 1 次印刷
书　　号：ISBN 7-302-06851-8/TP • 5082
印　　数：1~4000
定　　价：19.80 元

致　谢

感谢所有的审阅者及其反馈意见，他们包括 Jeff Aikin、Steve Berczuk、Bob Hanmer、Neil Harrison 和 Linda Rising。感谢本书的责任编辑 Paul Becker，如果没有他不断地给我提出建议和指导，我简直不敢相信本书能够脱稿。另外，感谢本书的产品经理 Jacquelyn Doucette 和文字编辑 Bunny Ames，他们的工作使本书增色不少，并大大提高了本书的可读性。

最重要的，特别感谢我最好的朋友 Susan。我幸运地娶到了她，她花费了无数个夜晚帮我审阅书稿，检查拼写，帮助我保持清醒的头脑，鼓励我完成本书的写作。

前　言

为什么写这本书？

本书的内容将围绕着“如何正确使用模式来构建软件系统”这个主题展开。在本书中，我希望能通过讲解如何正确使用与具体的模式或系统无关的一般概念，让读者能够对现有模式的核心内容有一个完整的认识。在刚开始尝试使用模式的时候，开发人员和项目经理都常常无法做出正确的选择，因为他们完全忽视了模式的意义和相关的简单概念。相反，他们常常会把模式当作编码技巧，而不是一种描述系统设计的高级语言。本书就是要改变这种错误的观点，为读者提供确保成功所必需的方法。

在刚开始讲授软件设计的基础课时，我费了很大的力气来考虑一个问题：如何有效地“讲授”设计？我发现，最好的办法就是使用真实的范例，让学生参与其中，在利弊权衡中受到启发，并由此学到知识。我坚信这就是讲授设计技术的最根本的方法，所以，在我的培训课程中、在我的顾问工作中以及在本书中，我一直都在努力尝试使用这种教学方法。

本书所采用的方法是，通过一系列系统设计方案——从最初的想法到代码，让读者领会有关概念。在讲授设计课期间我很早就认识到，尽管设计与代码的关系并不大，但是让学生看到系统开发的真正最终产品——源代码——仍然是非常必要的。我提供了许多范例来说明：如何使用模式构建系统，这些系统将用目前最流行的三种编程语言——Visual Basic、C++和 Java——来实现。当深入探讨代码以说明一些关键概念时，我尽量不要求读者进行大量的编程工作或其他正规的编程练习。

为了将注意力集中在“如何使用模式”上，而不是去深入探究现有的各种模式[1]，本书将主要介绍几种核心的模式，即主要是现在已成为经典的《设计模式：可重用面向对象软件的元素》[Gam, 95]一书中提到的模式。另外，在适当的时候，本书还介绍另外几种独特的模式，其中有我自己发明的，也有其他作者发明的。

建议你按照章节顺序阅读本书，不过你也可以根据需要跳过某些章节。尽管如此，我还是建议你在阅读其他章节之前，首先阅读第 2 章“模式简介”。本书采用的一般叙述方法是，首先介绍模式和一些面向对象的概念与图示，这些内容都集中放在一起。然后，马上进入实际范例，向你展示用于解决不同问题的各种模式的应用，以及每种模式的优劣，作

1 如果你希望搜集大型设计模式目录，建议你参考 Linda Rising 编写的 *The Patterns Almanac*[Ris, 00]。

为系统体系结构设计的指导。

在学习设计模式时，你需要完成一次思维上的转变，从一个新的角度来看待软件开发。可是在我讲授设计模式课程期间，我发现人们一开始都还没有完成这个转变。作为一名教师，我可以从学生呆滞的眼神中明白这一点(尽管这常常是由于早起的疲惫还没散去)。当课程进行到大约三分之一的时候，学生们开始将支离破碎的概念拼凑到一起，并且认识到：模式不仅仅是编码技巧或语言技术。突然之间，学生们的整个视野都变宽了，他们很快就领会到了这些技术对于整个软件开发过程的重要性。

作为本书的读者，我希望你能认真阅读本书及其中的例子，并获得同样的启发。一旦认识到了软件设计工作的内在规律，S 很快你就能得到这种启发。然后，你就能从美学的角度来看待模式，而不仅仅是把模式当作编码技巧。你会把模式看成设计的关键工具，并能够正确使用模式。同样重要的是，你将知道如何使用模式，以及什么情况下不使用模式。根据工作中的约束和场景看待问题的能力，会让你在未来获得极大的成就。到了这个时候，我们的目标就完全达到了。

最后，希望你喜欢本书。

目　录

第 1 章　最初的思考：个人笔记

1.1　背　　景

我从 12 岁就开始编程了，这要感谢我们家的一位密友，是他给了我一张 Radio Shack 计算机课程的听课证。幸运的是，当地的一家商店很乐意让一个 12 岁的小孩来为他们的计算机编程，以表明使用计算机是多么的容易。自那时起，我的每一天(除了少数假日之外，那些日子我被禁止携带计算机)都用来编程或进行某一层面上的软件开发。我想这就是外界常常对我有些偏见的原因所在。

20 多年过去了，我也写了数十万行的代码。现在，我得到一个机会与人们分享自己觉得重要的东西。我的运气好得出奇，可以向很多聪明人学习，包括项目中的同事和各个培训班中的学生。在此期间，我学到了许多技术和方法——其中一些对我很有帮助，而另一些则是我必须抛弃的。某些关键技术，一旦被我真正学到，就完全改变了我对于软件开发的整个看法。

1994 年，我在开发一个嵌入式笔式输入系统，用于一种专用的笔记本大小的格式输入(form entry)设备(更像是大号的 Palm Pilot)。这属于那样一种技术性项目：能满足目标，但是永远无法真正获得商业性成功所需的市场空间。那时，我开始注意到自己曾经使用过的并在其他的一些系统中采用的一些特殊的或通用的技巧，包括我现在工作的公司和我曾经工作过的其他公司的系统。我们花了一些时间试图找到某种方法，以便记录和提炼这些技术。

正是在那个时候，一位朋友指引我去研究汇编成文的模式，然后我就一头扎进这个领域，开始研究 Christopher Alexander 和其他人的成果。我发现，这些模式不但解决了我们当时面临的问题，而且还让我大开眼界，使我以一种全新的方式来看待问题，并把系统体系结构作为一个整体来理解。我还发现了自己从前的许多错误，并且最终明白我们以前的某些努力为什么会失败。我下定决心，决不再重犯那些错误，并寻找把这种知识与其他人分享的途径。

好了，让我们开始吧。首先，为大家介绍我最初所写的一种模式，它代表了一条非常重要的原则，在设计任何软件系统时，我都会尽量遵循这条原则。在本章，只对它做一简单的介绍。本书后面会详细解释这种模式的关键部分，并在全书中展示它的应用。然而，不管你是否同意，重要的是你应该明白：模式在你的脑海中形成一幅图画，它以一种通俗

易懂的方式传达模式的重要概念和应用。

模式名称：High Road Development(捷径开发)[1]

问题

面对不断变化的和未来的需求，如何有效地处理现有的需求？

场景

开发人员创建或扩展一个系统。

约束

- 通常无法完全理解未来的需求，并且未来的需求非常可能发生变化。
- 在重用方面的努力常常导致设计出一些精巧的组件，而这些组件却无法重用，甚至常常无法完成。
- 与当前的需求相比，人们常常不清楚且容易忽视什么是“未来的需求”。
- 开发人员倾向于把任何问题都仅仅看作是编码问题；在解决这类问题时，他们常常不考虑设计的作用。
- 通过虚方法和模板等机制，C++和其他面向对象语言提供了提高软件灵活性的途径。不幸的是，许多开发人员却没有正确地利用这些工具。

解决方案

在实现时永远不要写任何不需要的代码；相反，应该把精力放在设计上，确保系统体系结构可以处理所有可能的情景(实际上，只能是尽可能多的情景)。这是一个相对软性的目标，但是向这个方向努力是必要的。与刚性的硬编码不同，设计模式和其他的一些机制使系统可以灵活扩展，而且这种扩展是通过添加新的成分完成的，而不是改变现有的代码。另外，如果增加一层间接层或者使用设计模式，可以得到更简单的解决方案，就不要使用任何占位代码或其他占位符。

结果场景

结果是这样一个系统：它能够满足现有的需求，并且在必要的时候能够应付未来的需求。在设计层面添加这些新能力的成本，通常要低于在代码层面提前做准备的成本。特别是当我们无法预知确切的需求(不然我们在一开始时就把它搞定了)时，更是如此。所以，我

1 以前我把这种模式称作“Build for Today, Design for Tomorrow(为今天编码，为明天设计)”。不过 Steve Berczuk 认为名词短语通常比动词短语更适合作模式的名称，所以他建议我将这种模式改名为“High Road Development(捷径开发)”。

们在代码层面提前所做的努力通常会导致并非最佳的设计，甚至完全走向错误的方向。另一方面，应用这种模式时必须注意一点：它很有可能增加设计的复杂度(尽管并不一定增加编码的复杂度)。

原理

由于把关注的焦点从实现转移到了设计上，所以我们可以更有效地处理更高层面的问题，而这正是常常被完全忽略了的问题。常常会发生这样的事：最初预想的系统特性永远得不到实现，而另一些特性却因为市场的压力而实现了。一般来说，这些新特性都可以轻松实现，因为我们把软件的体系结构设计得具有扩展性。

别名

Build for Today, Design for Tomorrow(为今天编码，为明天设计)。

已知应用

在我开发的几个大型框架结构中，我都应用了这种模式。其中一个例子就是 Total Commissions Systems(总佣金系统，TCS)，这是一个企业级的佣金计算系统。在这个系统中，由于应用了 High Road Development 模式，我们可以在版本 1 到版本 2 的升级过程中快速完成核心计算部分的重新设计，以消除硬编码对插入式(plug-in)体系结构的影响。这次修改只花了几天时间，并且没有对系统的其他部分造成重大影响。

重构和极限编程

重构(refactoring)[Fow, 99]提供了另一种使用模式的方法。而且，如果有一个足够强大的测试框架，就可以成功地利用重构方法。重构的基本思想就是集中精力使设计简化，并且在新的需求出现时提供一个持续发展(而非扩展)的环境。在使用基于重构的方法时，应该考虑使用诸如极限编程(Extreme Programming，XP)[Bec, 00]之类的快速开发方法。

从较高的层面上来看，XP 是一种基于团队的方法，其中不同开发周期的区分已经变得模糊，因为它把每个阶段的时间都减到了最低限度。尽管类似的方法(例如 RAD 和即时编程)已经被成功运用了许多年，但 XP 方法才刚刚开始引起人们的注意。一般而言，对高风险、多变化环境下的内部项目，使用标准过程很可能会失败，而 XP 方法则大有用武之地。不过我认为，当开发小组由比较多的初级程序员组成时，XP 也会是最好的方法。

XP 是一种备受争议的方法。对于它，我也只是处在研究阶段。尽管发现其中有许多东西让我不敢苟同(这主要是由于我所观察的开发环境的问题)，但是我总是欣赏任何能让我产生疑问、让我感觉新鲜的东西。不管怎么说，我还是强烈建议你了解这种方法。

你最好是以非常快的速度阅读 Beck 的书[Bec, 00]，它会向你提出几个关于软件开发过程的问题。但是不管过去使用的是什么过程或方法，在构建软件系统的时候，你都应该以本书中讨论的方法为基础。

1.2 小　　结

在本书中以及在我帮助开发的所有系统中，我都努力把 High Road Development 模式作为构建系统的主要原则来运用。这使我不会去做无谓的设计，也不会限制当新的需求出现时扩展系统的能力。另一方面，它也为其他模式的应用提供了一个准则。在本书的后面部分，我们将探索确保你的设计也可以获得同样灵活性和功能的方法。

第2章 模式简介

简而言之，模式也是世界上出现的一种事物，其规则告诉我们如何创建这种事物，以及何时必须创建它。它既是过程，也是事物；既是对存在的事物的描述，也是对创建这种事物的过程的描述。

——Alexander，1979

2.1 体系结构的构件

与每一片都独一无二的雪花不同，软件更像是“垒高”(LEGO)积木玩具。我们可以按照各种方式把积木组合起来，得到我们需要的形状，但主要的构件是相同的。不过，对于软件来说，这种“构件”并不仅仅是算法和数据结构。同样，模式也不仅仅是构件，它们更像是用于制造不同积木的模具，它们提供了一种途径——确保从该模具中制造出的积木都可以组合在一起，并且还提供了其他的支持和定义，以便使更多的积木可以连接起来。

更具体地说，在软件体系结构中，不同部分所解决的问题并非是惟一的。在任何一个领域里，不同的系统中总会出现同样的问题。让我们看一个具体的例子。我们经常需要在软件中处理多个对象，而这些对象依赖于其他对象的状态。比如说，在一个记账系统中，你可能会提供多种数据视图，其中可能包括多个电子表格视图和图形视图。当你在某个电子表格中修改数据时，用户完全有理由希望其他的视图也会随之改变(自动完成，无须选择刷新选项)。

但是，模式已经超越了具体问题的范畴。比如说，在一个嵌入式系统中，你也可能遇到一个类似的问题：多个对象(例如监视器)监视某一硬件设备(通过软件封装或设备驱动程序)，当发生中断(例如电力不足)时，该硬件设备必须迅速做出反应。一种模式能够描述这种情况，并说明如何以一种容易移植的方式来解决这个问题。例如，这种监控方法可以被描述为 Observer 模式[Gam, 95]，摘要如下。

Observer 模式

目的

定义对象间的“一对多”相关性，使得当一个对象的状态发生改变时，可以自动通知并更新所有相关对象。

变化

彼此相关的对象的数量会发生变化，相关对象的更新方式也会发生变化。

结构

Observer 模式的结构如图 2.1 所示。

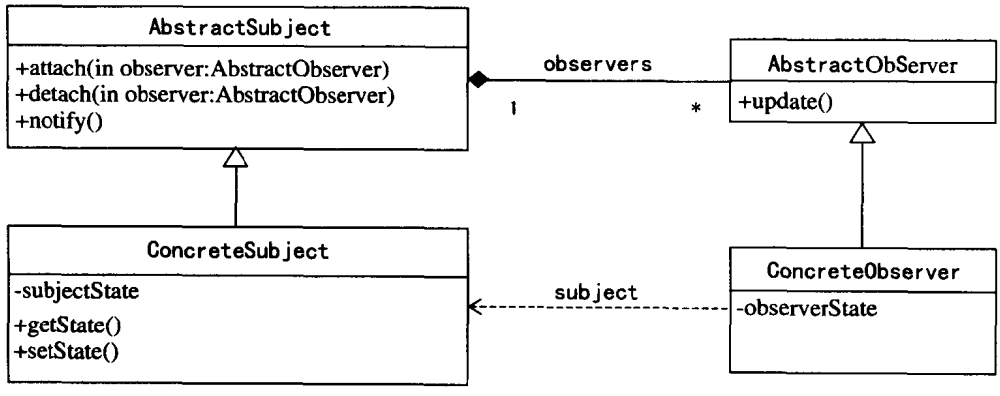

图 2.1 Observer模式的结构

注释

Observer 模式也叫做 Publisher-Subscriber(出版者/订户)模式[Bus, 96]，是文档/视图概念中的一个主要特征。Observer 模式的复杂性通常体现在实现决策方面，例如，是使用推模(将相关主题的数据发送给观察器)还是使用拉模型(由观察器自主获取所需信息)。Observer 模式还经常与 Proxy 模式[Gam, 95]一起使用，以允许使用远程观察器。如果系统有一些远程客户，就需要考虑是使用推通知模型，还是拉通知模型，因为这涉及到远程通知对性能的影响和网络开销问题。

2.2 模式的形式

模式的一个主要目标就是，以一种别人容易接受的方式，捕捉那些重复出现的问题的解决方案(以及可以使用这些解决方案的约束和场景)。

在捕捉这些信息的时候，我们要尽量理解这种解决方案何以奏效。此时，我们通常可以总结出模式的不同特征，并更深入地理解其工作原理。在“收获”信息的过程中，我们经常会发现，一些相关的模式可能具有相同的甚至更高的价值。另外，由于对问题的约束和其他要素进行了细致的观察，我们可以更加深入地了解软件开发过程的本质。

也许最实际的而且与传统的文档方法不同的是，模式可以从更高的抽象层面描述系统，并让我们可以捕捉静态的和动态的行为。这种形式的文档对于知识传播的价值，怎么说都不会言过其实。

编写模式的形式有好几种。尽管具体使用哪种形式并不重要，但如果能遵循某一种固定的形式，编写者就可以不必考虑形式的问题，从而集中精力说明真正的问题。但不管使用哪种形式，模式的文档中都应该以某种方式包SS下列要素。

- **名称/别名**：每种模式都必须有一个独一无二的名称，这样它才能深入人心。名称必须是便于记忆的，并且必须能够从名称引申出与它所描述的系统。但是，一种模式可能有好几个名称，有时候这也会引起混乱。
- **问题**：模式文档的形式应该包含该模式所说明的特定问题。如果我们把“问题”部分看作是对我们试图达到的目标的抽象描述，那么“解决方案”就是我们用来达到目标的方法。一个问题可以有许多种解决方案，那么我们怎样判断哪种方案是最好的或者是“正确的”呢？所以，我们首先必须考虑另一个因素——约束。
- **约束**：约束就是必须考虑的任何事项，就是让问题变得困难、让显而易见的解决方案失效的那些东西。我们可以认为约束就是模式必须服从的外界因素。在一个问题中会有无数的约束，所以我们按照场景来分清约束的主次。约束包括环境问题、语言问题、组织问题和平台问题。例如，一个高度熟练的开发小组在使用Java在UNIX平台上开发电信软件时，他们所面对的约束与使用COBOL进行金融大型机开发的小组必定是不同的。选择了一种模式之后，这种模式可以消除许多约束，但会留下一些或强或弱的约束，通过其他的模式来消除。

  如果选择了恰当的模式，应当可以消除尚未消除的约束。这种模式可以改造系统中其他的模式，使它们更加强壮。这很像图2.2所示的拼图，随着越来越多的图片的加入，整个拼图变得越来越稳定。而且，如果把一片错误的图片强行塞入拼图，整个拼图就会变形、散架。把整个拼图看成软件系统，把图片看成模式，这样你

就可以从一个全新的视角来观察系统的动态了。

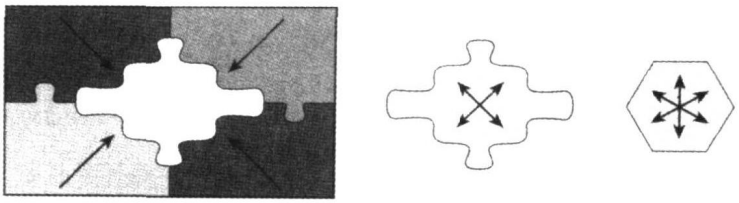

图 2.2 约束会造成压力，也会加固整体，就像拼图一样

但是，必须承认，系统中的约束可能是极其复杂的。这与国际象棋存在着许多相似之处(如图 2.3 所示)。如果从战略的角度来看待国际象棋，它全部的意义就是加强某种约束以削弱另外的约束。当然，目标就是向对方“王”施加足够的压力，并在己方“王”周围的区域布下足够的保护，以确保制服对手。

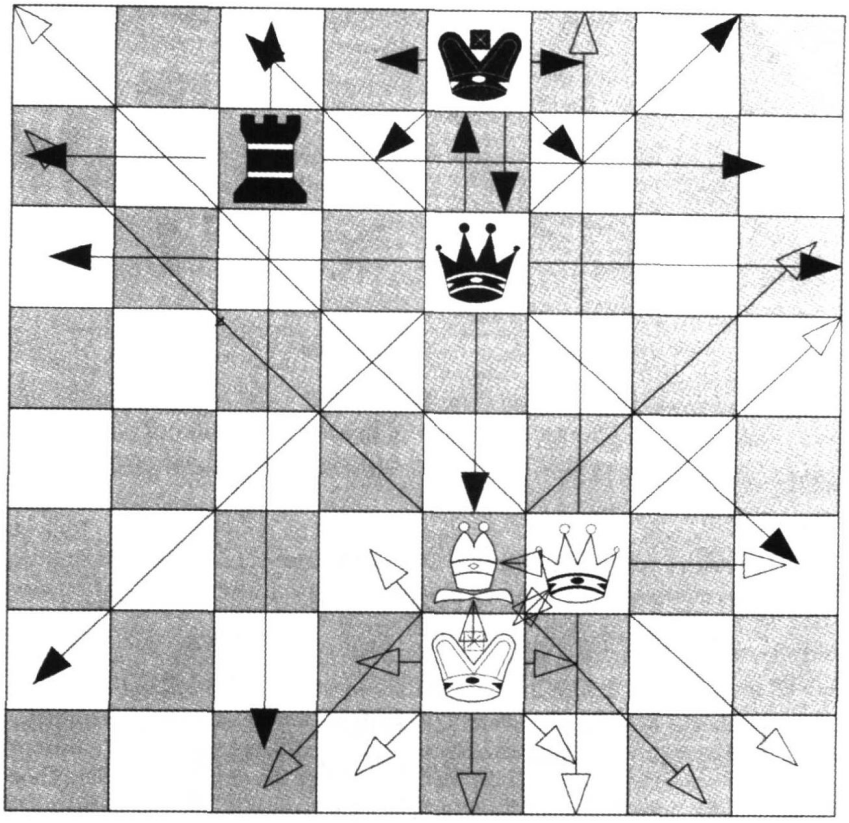

图 2.3 国际象棋中的约束

(来源：Dennis K. Fitzgerald的Visio模板，Web/Visio组建立的CIS 72627, 1442 模板)

如果把模式恰当地组合在一起，它们就会形成一个融洽的复杂系统。每一种模式都应该是对其他模式的补充，每增加一种模式都应该让系统更加强壮。这样的系统可以不断发展，且不会增加维护的压力，也不会降低整个系统的性能——系统增大后通常会导致这种结果。我们希望创建一个各部分互为补充的框架(如图 2.4 所示)，希望每一个新增的部分都能让其他部分更加强壮。

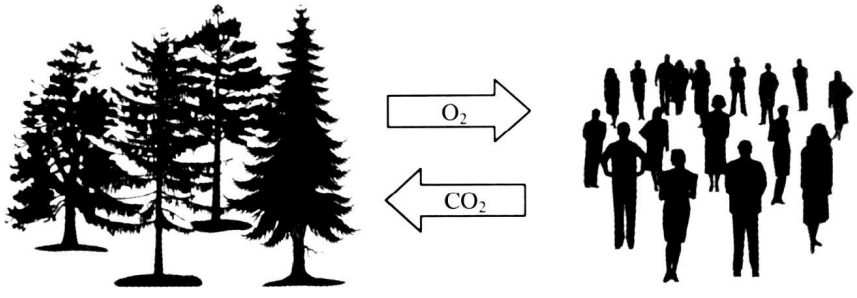

图 2.4　一个各部分互为补充的框架

模式不会存在于真空之中。实际上，模式不仅依赖其自身，而且依赖于整个体系结构中所有其他模式。每种模式本身都由相互依赖的其他成分组成。正像我们可以用少量的词来表达许多复杂的思想一样，我们也可以用少数几种模式组成许多复杂的系统。如果正确使用了模式，就可以创建出有序的、完整的、仍然可扩展的系统。在所有案例中，都应该存在一个协调的系统。

随着你对模式的进一步了解，从现在开始，你可以把要解决的问题看成一张约束图，而且你应该明白：在其中加入的任何东西都将对整个系统造成某种影响。当然，你可以希望这种影响有助于协调你要消除的约束。

不幸的是，软件开发人员常常贸然着手解决问题，而不去考虑问题的约束，不去理解问题的本质。结果，没有经验的开发人员经常冒冒失失地开始编码，而根本不知道如何编码，甚至不知道所编写的解决方案是否合适。别那么着急，回过头来考虑一下问题的本质，然后选择一种合适的工具，这可以让你节省下数千行的代码，可以让你避免做无用功。

真正的问题和约束通常都是未知的，但是不管怎样，我们还是必须开发系统。在这种时候，模式的使用就更显重要了。未知的东西会变成强大的约束，为了消除这些约束，并降低犯错误的风险，我们必须使用模式。随着对未知领域的逐渐理解，当我们最终理解整个问题时，可以通过模式来提供变化的能力——这就是设计中的真正挑战。如果所有的需求都可以预先知道了，而且不可能发生变化，那么设计就完全没有意义了。可惜，没有人能生活在这么舒服的世界里。相反，我们所面临的惟一不变的挑战就是变化。