软件工程技术丛书

HZ BOOKS

设计系列




# 软件再造

## 面向对象的软件再工程模式

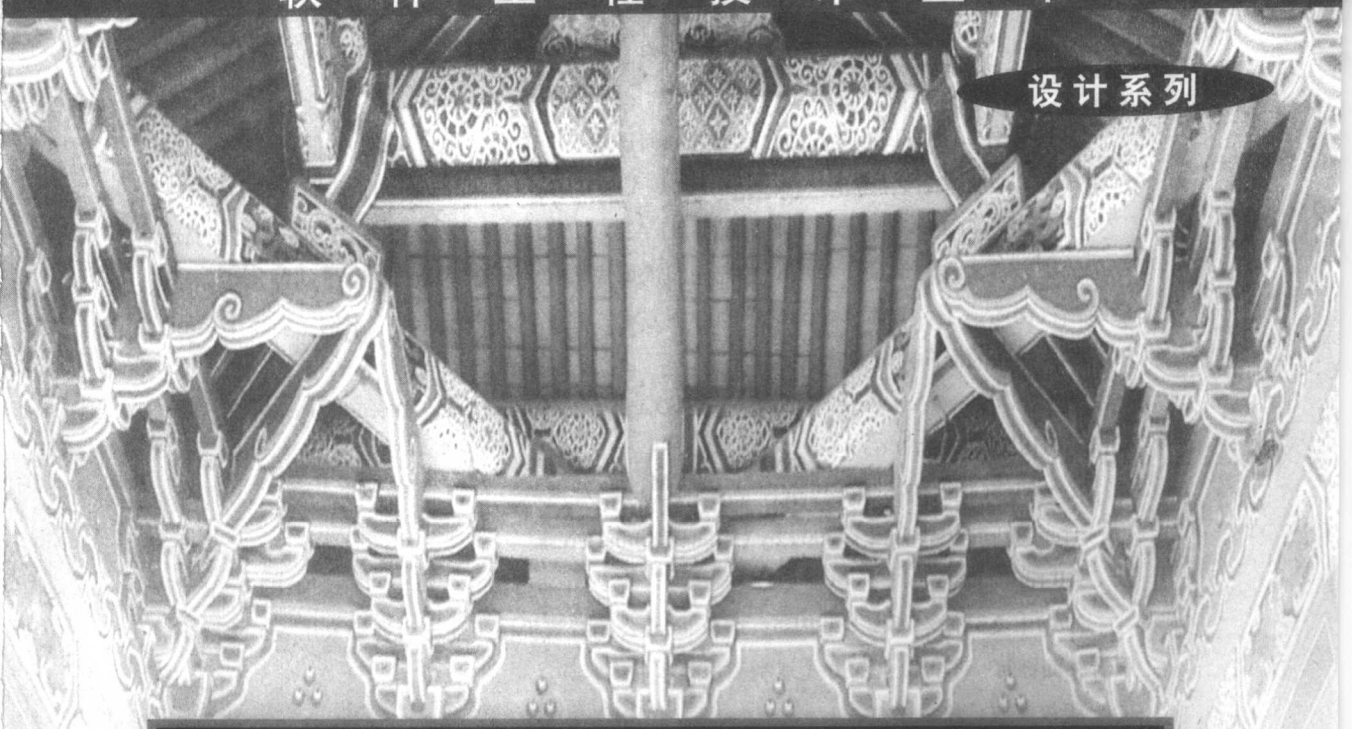
# Object-Oriented Reengineering Patterns

Serge Demeyer
Stéphane Ducasse 著
Oscar Nierstrasz
莫倩 王恺 译

机械工业出版社
China Machine Press

软 件 工 程 技 术 丛 书

设计系列



# 软件再造

## 面向对象的软件再工程模式

## Object-Oriented Reengineering Patterns

Serge Demeyer
Stéphane Ducasse 著
Oscar Nierstrasz

莫倩 王恺 译

机械工业出版社
China Machine Press

大部分有关软件工程的书讨论的都是如何进行正向的软件项目开发，而少有提及如何处理有价值的遗留系统。然而在现实中，许多人在工作中面临的都是处理现有的代码库。

本书作者从自身参与欧洲工业研究项目FAMOOS的实践出发，总结了对面向对象的软件系统进行再工程时的最佳实践，并把它们精炼成模式。每个模式都从分析问题入手，然后给出权衡考虑了各方面影响因素后的解决方案，并解释其合理性或给出例子。本书是一本实践性很强的面向对象软件再工程指南。

Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz: Object-Oriented Reengineering Patterns (ISBN 1-55860-639-4).



**本书版权登记号：图字：01-2003-6716**

**图书在版编目（CIP）数据**

软件再造：面向对象的软件再工程模式 / 迪迈耶（Demeyer, S.）等著；莫倩等译. -北京：机械工业出版社，2004.10

（软件工程技术丛书　设计系列）

书名原文：Object-Oriented Reengineering Patterns

ISBN 7-111-15018-X

Ⅰ. 面…　Ⅱ. ①迪…　②莫…　Ⅲ. 面向对象语言-软件开发　Ⅳ. TP311.52

中国版本图书馆CIP数据核字（2004）第080609号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）

责任编辑：刘立卿

北京昌平奔腾印刷厂印刷·新华书店北京发行所发行

2004年10月第1版第1次印刷

787mm × 1092mm 1/16 · 12.25 印张

印数：0 001 - 4000 册

定价：29.00 元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

# 对本书的赞誉

“如何”实现软件重构已经在一些书籍中被很好地阐述，但是，“何时”与“为什么”要实现软件重构，就只能在实践中理解掌握。本书将带给你一个正确的开始，从中你将明白：什么时候应该开始重新设计一个系统，什么时候应该适时地停止，以及你为之付出的努力在未来预期能达到何种效果。

——Kent Beck, Three Rivers学院主任

本书通过一种易于理解和使用的方式，描述了大量实际的、内行的软件再工程知识和专业经验。书中的面向对象软件再工程模式，对每一位关心如何使用软件再工程技术来指导工作的人都会有所帮助。我真希望在我的书库中能更早有这本书！

——Frank Buschmann, 西门子AG高级首席工程师

本书不仅限于书名所描述的，它还包含更多的内容。有效的软件再工程是非常有意义的，它能够帮助你有目的地、高效地阅读别人的源代码，以进行可预期的改进。本书作者所重点强调的很有技巧的软件再工程行为模式，可以很容易地转换成你在构建具有可读性、可维护性软件系统时所需要的技巧。

——Adele Goldberg, Neometron公司

如果某个叫Dave（书中举例时用到的名字。——译者注）的家伙将一个大箱子交到我的办公室，箱子里装满程序文档和两张光盘——光盘是公司实现软件再工程时要用到的软件安装盘，这时候如果本书的作者在我身边，我将感到无比幸福！除此之外，如果手头有这本书的话，也是一件美妙的事！没有银弹，没有广告，也没有承诺软件再工程将会很简单——这是一本务实的、易读的、非常有用的书，可以为人们应对项目提供很有帮助的指南。赶快在那个叫Dave的家伙到你的办公室之前购买和阅读本书吧！它一定会消除你和你公司的许多痛苦！

——Linda Rising，独立顾问

# 序 1

在很长一段时间里，我一直对一件事情比较困惑——那就是在大部分讨论软件开发过程的书里，所讨论的都是从零开始编辑程序。我之所以对此感到困惑，是因为在人们开始编写代码时，这并不是他们最通常所面临的情境。很多人必须从改进既有代码库开始，即使这些代码库并不是他们自己编写的。理想情况下，这些代码经过了很好的设计和重构，但我们都知道理想情况在我们职业生涯中出现的几率。

本书很有意义的一个原因在于它采用的角度，它是从如何处理一些不完善但却有价值的代码库的角度撰写的。我还喜欢这样一个事实——那就是本书基于将学术理论与工业实践相结合的角度来写作。我曾经在FAMOOS团体成立之初在一个寒冷的初冬到伯尔尼拜访过他们。我非常喜欢他们的工作方式，他们在理论领域和实验室实践之间建立起了一个循环，尝试先将理论思想放入到真正的项目中检验，然后再从实验室实践中返回到理论总结。

以上的工作使得本书的实践性很强。本书为读者提供了构造模块，可以用来制定一个计划，以处理某个困难的源代码库。本书还阐述了使用各种技术（例如软件重构）时的语境。目前像本书这样的著作太少，这是一件让人遗憾的事情，因为目前进行软件再工程已经是一种非常普遍的工作了。但我至少可以很高兴地看到，虽然有关这个领域的书不多，但这些书都写得非常好，本书就是一个例子。

Martin Fowler
ThoughtWorks公司

# 序 2

好模式的标志之一是：当专家们读了它可能会说："当然如此，这是众所周知的!"但是初学者却可能会说："这很有趣，但它是否有效果呢？"模式应该容易被效仿，但是最有价值的模式却是一些不那么让人一目了然的模式。专家们已经从经验中明白了模式是有效的，但初学者在使用模式并获得自己的经验之前，就必须相信模式的作用。

在过去的几年里，我有机会将本书中的模式提供给许多人，并与之进行讨论。在我的这个模式讨论圈里，有一些人有数十年的咨询经验，他们能够很快与他人分享使用这些模式的经验。而讨论圈里年轻的成员们一旦信服了模式的价值之后，都开始喜爱这些经验了。

我让我的软件工程课上的学生学习了一些模式，将模式作为有关软件再工程的一个章节来学习。尽管一开始学生们都没有被模式的概念所打动，但这一章节的课程仍然上得很好。一开始学生们缺乏评估这些模式的实际经验；然而，还是有一个学生在做完暑假功课后跑来告诉我，在课程的所有内容中，反向工程中的模式最有用。在学生们取得这种经验之前，模式对他们而言似乎是可信的。而一旦他们获得经验之后，他们就会坚信模式的作用。

如果你在软件再工程方面有许多经验，你也许从本书中学不到太多的东西。但你仍然有必要阅读本书，因为你可以将本书送给同你一起工作的人们，在与他们讨论问题时可以使用本书的术语和概念。如果你是软件再工程的新手，那就更应该阅读本书，学习其中的模式，并且试用它们。你将学会许多有价值的东西；但是不要期望在使用模式之前完全理解它们，因为模式是实践中总结出的东西，而从实践得出的知识在没有实践之前是无法被人真正理解的。尽管如此，本书仍然会给你很大的帮助。当你在实践中有章可循时，你将很容易掌握需要学习的东西，而本书正为你提供了一个可靠的指南。

Ralph E. Johnson，伊利诺伊大学厄巴纳-尚佩恩分校

# 前 言

## 一个童话故事

从前有一位优秀的软件工程师，他的客户都确切地知道自己的需求。这位优秀的软件工程师努力地工作，并设计了一个完美的系统，这个系统能够解决客户当前和以后的所有问题。当这个完美的系统经过设计、实现和最终部署之后，客户都感到非常高兴。系统的维护工程师无需做任何工作就能让这个完美的系统很好地运行，客户和系统维护工程师从此以后一直快乐地生活着。

**为什么现实生活不能像这个童话故事一样呢？**

难道是因为没有优秀的工程师吗？难道是客户没有真正了解他们自己的需求吗？或者是因为根本就不存在完美的系统呢？

也许上面所说的都有一点道理，但真正的原因恐怕在于多年以前由 Manny Lehman和Les Belady所总结的软件发展的基本规律。这些基本规律中最重要的两点是：

**持续变化规律。**在一个真实的应用环境中使用的程序，一定是不断被改变的，否则那个应用环境本身就会变得越来越没有用处。

**复杂性不断增长的规律。**当一个程序不断地发展，它会变得越来越复杂，并需要消耗额外的资源来保持并简化它自己的结构。

换句话说，如果我们认为自己能够了解所有的需求，并能构建完美的系统，那么就是在自己骗自己。我们所能够做到的最好情况是构建一个有用的系统，并且能够运行足够长的时间，直到需要它做新的工作为止。

## 本书内容

本书的起因在于充分认识到这样一个事实——软件工程最有意义和最具挑战性的一面不是建立一个新的软件系统，而是重构既有系统。

从1996年11月到1999年12月，我们参加了一个名为FAMOOS的欧洲工业研究项目（ESPRIT项目 21975——掌握面向对象软件发展的基于框架的方法研究，Framework-based Approach for Mastering Object-Oriented Software Evolution）。合作伙伴包括Nokia（芬兰）、Daimler-Benz（德国）、Sema Group（西班牙）、Forschungszentrum Informatik Karlsruhe（德国）和伯尔尼大学（瑞士）。Nokia和Daimler-Benz都是面向对象技术的早期应用者，他们期望从这项研究中获益。目前，他们正面临遗留系统中的许多典型问题：有许多大型的和极有价值的面向对象软件系统，但这些系统很难适应不断变化的需求。FAMOOS项目的目标就是开发新的工具和技术，使这些面向对象的遗留系统恢复生机，从而能够持续发挥作用并更好地适应未来需求的变化。

在项目一开始，我们的思路是将这些大型的、面向对象的应用转换成框架——也就是通用的应用，从而能够使用大量不同的编程技术进行简单的重新配置。但我们很快发现，这件事说起来容易做起来难。尽管这个基本的研究思路听起来不错，但我们却很难决定应该转换遗留系统中的哪一部分以及到底如何转换。实际上，一开始光是理解遗留系统就不是一件很容易的事，更不用说发现其中的错误了（如果有的话）。

我们从这个项目中学到了许多东西。我们首先学到的是，遗留系统的代码并非都是不好的，遗留系统之所以有问题只是因为在原有系统设计和部署之后需求已发生了许多变化。软件系统经常被多次修改以适应变化的需求，系统忍受着*设计漂移*——初始的体系结构和设计不可能了解未来的所有需求变化，这就导致了不可能对系统进行更多的改进。这一点就像Lehman 和Belady所总结的软件发展规律所描述的那样。

研究中最使我们吃惊的是，尽管我们研究的每一个案例都是因为不同原因需要进行软件再工程，例如不再与原系统匹配、需求扩大、移植到新的环境等原因，但所有这些系统所出现的实际问题却都惊人的相似。这就启发我们，可能采用一些简单的技术就能够很好地解决许多相似的问题。

我们发现几乎所有的软件再工程活动都必须从某种反向工程开始，因为你无法信任原有文档（如果你碰巧有一些文档的话）。基本上，你会分析源代码，运行系统，采访用户和开发者，以建立遗留系统的模型。那么你必须确定，要进一步开发并修正系统，存在的障碍在哪里。这是*再工程*的关键。如果你有足够的先见之明，并且了解所有你今天所知道的新需求，这将帮助你将遗留系统转化为你即将建立的新系统。因为你不可能重建所有的系统，所以必须忽略细枝末节，仅仅再工程最重要的部分。

在FAMOOS项目之后，我们又参加了许多其他软件再工程项目，并将FAMOOS的研究成果进一步加以验证和优化。

本书中总结了我们所掌握的研究成果，希望它能够帮助那些需要对面向对象系统进行软件再工程的人们。我们并不认为我们能够给出解决所有问题的答案，但是我们确实阐明了一系列简单有效的技术，这些技术必将给大家带来很好的帮助。

## 为什么使用模式

模式是一个重复的基调，一个不断重复出现的事件或者程序结构。*设计模式*是一种通用的解决方案，用来解决重复的设计问题[Gamm95]。由于这些设计问题从来都不是一模一样的——仅仅是非常类似，因此相应的解决方案就不是一段程序，而是*能够传达最佳实践的文档*。

近年来模式不断在文献中出现，它被用来归档那些解决许多不同问题的最佳实践。尽管许多问题和解决方案都可以通过模式来表述，但如果只应用于简单问题就有点大材小用了。模式作为一种文档描述形式是非常有用和有意义的，它要解决的问题常常是设计上的许多必然的矛盾*影响因素*，而相应的解决方案则描述了这些必然因素间的权衡。举例来说，很多著名的设计模式都是以增加设计的复杂性为代价来提高系统运行的灵活性。

本书阐述了一系列用来对遗留系统进行软件再工程和反向工程的模式。这些模式都不能盲

目地使用。每一个模式解决一些矛盾影响因素，也涉及到某些权衡。理解这些权衡考虑是成功使用模式的基础。因此，模式形式看起来是记录我们最佳实践的最自然的方式，这些最佳实践是我们通过再工程项目所认识到的。

一种模式语言是一组相关模式，这些模式可以结合使用以解决一系列复杂问题。我们发现将一些模式组合起来形成模式簇很有用。因此我们是按照模式簇（一个较小的模式语言集合）来组织本书章节的。

我们并不认为本书的模式簇在任何情况下都是“完备的”，我们也不认为本书的模式覆盖了软件再工程的所有方面，我们当然也不会认为本书阐述了面向对象软件再工程的系统化方法。我们在本书中所要表达的，仅仅是我们曾遇到和曾采用过的一些最佳实践，这些实践展示出非常有意思的协同作用。这种很强的协同作用不仅包括在每一种模式簇中，而且还包括各个模式簇间以各种重要方式相互关联。因此，书中的每一章都不但包含一个模式图谱——模式图谱阐述了如何将模式作为一种“语言”来实现某种功能，而且每个模式还列出并解释了怎样与其他模式结合或组合，无论这些模式是否在同一个模式簇里。

## 本书的读者对象

本书主要是献给那些需要实现面向对象系统软件再工程的实践者们。如果从一个极端的观点来看，可以说每一个软件项目都是一个软件再工程项目。因此本书面向的读者群是非常广大的。

我们认为，对任何哪怕只有少许面向对象开发经验的读者来说，本书中的大部分模式都应该是比较熟悉的，本书的目的是阐述实践它们的细节。

# 目 录

# 第 1 章
# 软件再工程模式

## 为什么我们要实施软件再工程

所谓遗产就是继承的有价值的东西。与此类似，遗留软件就是所继承的有价值的软件。继承这个事实本身意味着继承的东西已经有些过时了。继承的遗留软件有可能采用的是过时的编程语言，或者是用陈旧的开发方法开发的。更有可能的是，它已经在程序员手中转手了数次，并且到处有修改和改写的痕迹。

你的遗留软件也可能并没有那么老。但由于快速开发工具和快速人事变动，软件系统会以你难以想象的速度沦为遗留系统。如果这个软件它确实有价值，那么无论如何，你也不可能将它废弃。

有一些遗留软件对你的业务来说会非常重要，这同时也就是所有问题产生的根源——为了获取业务上的成功，就必须不断地准备改进软件以适应变化的业务环境。因此那些帮助你维持业务运行的软件就必须是可改进的。幸运的是，许多软件系统可以被改进，或者当它们不再服务于原有目标时，可以被简单地废弃或替换。但是，一个遗留系统除非花费很大代价，否则往往是既不能被替换，也无法被升级。软件再工程的目标，就是充分减少遗留系统的复杂性，使它能够持续被使用或者能够以一种可以接受的代价被持续改进。

对一个软件系统实施软件再工程的确切原因可以是各不相同的，例如：

- 可能希望将一个组合的系统拆分成独立的部分，以便各个部分能够方便地面向不同的市场，或者方便地以不同的方法组合起来。
- 可能希望提高性能。（经验表明，正确的做事顺序是“首先是做，然后是做对，再然后才是做快。”因此，在考虑性能之前，也可能想为了清理代码而实施软件再工程。）
- 可能想将系统移植到一个新的平台。在移植之前，可能需要重新构建体系结构，以将与平台无关的代码清楚地分离出来。
- 可能想将设计抽取出来，并将此作为一个新实现的第一步。
- 可能想试验新技术，比如新出现的标准或函数库，作为消减维护费用的第一步。
- 可能想通过将有关系统的知识用文档记录下来，使它更易维护，从而减少系统对人员的依赖。

正如我们将看到的，对系统实施软件再工程尽管存在许多不同的原因，但遗留软件面临的实际技术问题却经常是非常相似的。因此，我们能够使用一些非常通用的技术，至少是能完成软件再工程中的一部分工作。

## 理解软件再工程的需求

如何知道何时会遇上遗留软件的问题呢？

有这么一句名言，“如果它并没有坏，就不要去修补它。”对于任何正在运行某种重要功能的程序代码，如果它们表面上看起来运行得还不错，人们一般都会采取此种态度作为不去碰它的借口。但问题在于，人们没有认识到：某种东西坏了，可能会以多种方式表现出来。从功能性观点来看，某种东西只有在它不再提供当初设计执行的功能时才是坏了。但从软件系统维护的观点来看，一段程序当它不再能被维护的时候就一定是坏了。

因此，怎样才能分辨出软件很快会完蛋呢？幸运的是，有许多警告信号能够告知你即将面临的困难。下面就是这样一些征兆，它们通常不会独立地发生，而往往是同时发生的。

- 文档陈旧或者缺乏文档。文档陈旧对软件遗留系统来说是一个非常清晰的信号，表明该系统已经经历了多次更改。缺乏文档则是一个警告信号，一旦原来的开发者离开项目，问题立刻会出现。
- 没有经过测试。比最新文档更重要的是，对所有系统组件进行完整的单元测试，以及覆盖所有重要用例和场景的系统测试。缺乏这些测试，表明系统再继续发展将会面临极大的风险和代价。
- 原有开发者和用户的离开。除非有一个结构清晰、文档规范、经过良好测试的软件系统，否则一旦原有开发者和用户离开，系统将迅速恶化成一个结构混乱、缺乏文档的系统。
- 有关系统内部知识的缺失。这也是一个坏信号。文档会与既有的代码库不同步，没人能够知道系统是如何工作的。
- 对整个系统缺乏完整的理解。不但没有人理解漂亮的打印文档，而且也没有人能对整个系统有一个良好的整体把握。
- 花费太长时间使系统成为一个生产系统。在系统的某些地方，可能无法正常工作。修改程序可能要花费太长时间，系统可能缺乏自动回归测试，又或者系统修改后难以部署这些修改。除非你明白并着手处理这些问题，否则系统将变得更糟。
- 花费太长的时间处理简单的改进。这是Lehman和Belady总结的复杂性增长规律中的一个明确信号：系统当前已经如此复杂，以至于即使是非常简单的改进，实现起来也很困难。如果对你的系统而言，实现简单的改进就需要花费很长的时间，那么毫无疑问无法实现更复杂的改进。如果有许多简单的改进等着你实现，你当然没有机会去处理复杂的问题。
- 需要不断修改错误。错误好像永远都不会消失。每当修改了一个错误，另一个错误又冒了出来。这种情况告诉你——应用的某些部分已经变得如此复杂，以至于你不再能准确地估计小修改所造成的影响。更进一步说，应用程序的体系结构已不再与需求相匹配，因此即使是微小的改变也会导致不可预期的结果。
- 可维护性的依赖关系。当在一处修改完一个错误，另一个错误又会在其他地方冒出来，这种情况通常表明——系统的体系结构已经恶化到某种地步，以至于系统逻辑上分离的模块不再相互独立。

- 构建时间很长。很长的重编译时间会降低你进行系统更改的能力。很长的构建时间还意味着系统组织太复杂，以至于编译工具无法高效工作。
- 拆分产品非常困难。如果产品有很多客户，而每次为每位客户做产品模块功能的裁减时都很困难，这就意味着系统体系结构不再与工作相适应。
- 重复的代码。当系统不断发展时，重复代码问题会自然出现。当实现几乎相同的代码或者合并软件系统的不同版本时，都会出现重复代码问题。如果不采用重构的方法将通用的部分进行合适的抽象，从而消除重复代码，系统维护工作就会成为一个噩梦，因为同样的代码需要在许多地方同时被修改。
- 代码的味道。重复代码就是一个代码味道很坏而且需要改进的例子。程序代码中定义太长的方法、太大的类、太长的参数列表、太多的选择语句和数据类等，都是一些Kent Beck等人在著作[Fowl99]中所记载的代码味道很坏的例子。代码味道通常表明一个系统没有经过软件再工程，就已经在被不断地扩展和修改。

## 对象技术有什么特殊

尽管本书中所讨论的许多技术可以运用到任何软件系统中，我们还是选择面向对象的遗留系统作为重点。这样选择有很多原因，但主要的原因是我们认为现在正是时机——早期采用面向对象技术的人们开始发现，他们曾期望通过切换到对象技术获得好处，这些其实很难实现。

目前，有一些重要的遗留系统甚至是采用Java实现的。一部分软件变成遗留系统并不是由于时间的原因，而是由于没有以再工程方式开发和改写系统的速度造成的。

从这些经验中得出的错误结论是："对象技术不好，我们需要其他东西"。我们已经看到，已涌现出许多新的技术以拯救现状：模式、组件、UML、XMI等等。其中的任何一项技术的发展都是好事，但某种意义上它们都偏离了重点。

从本书中可以得到一个结论，即对象技术相当好，但是我们得小心对待它们。要理解这个观点，考虑一下，既然面向对象系统被认为具有灵活性、可维护性和重用性等众多好处，为什么遗留系统的问题还是会出现在面向对象系统中呢？

首先，任何人如果曾使用过一套有价值的、已有的面向对象代码库，会发现很难找到对象。更真实的感觉是，面向对象应用系统的结构一般是隐藏的。所看到的是一大堆类和类继承的层次关系。但这些都无法告诉你，在运行时刻存在哪些对象，以及这些对象之间是如何合作完成所期望的结果。理解一个面向对象系统是一个反向工程的过程，本书描述的技术能够帮助读者解决这一问题。更进一步，通过再工程代码，你能获得一个结构更透明、更易理解的系统。

其次，任何人如果想要扩展已有的面向对象系统，必须认识到"重用并不是无代价的"。事实上，重用任何一段代码都是很困难的，除非在设计时就考虑到重用性。更重要的是，在重用上的投入需要管理部门的承诺，需要将合适的组织机构基础设施放置在正确的位置上，并且在头脑中有清晰的、可衡量的目标[Gold95]。

我们还不擅长以一种方式管理面向对象的软件项目，即将重用正确地纳入考虑之中。典型的重用技术到来得太迟了。我们使用面向对象的建模技术来建立丰富复杂的对象模型，并

期望当实现软件时能重用某些东西。但到那时会发现，将这些丰富的模型映射到任何标准的组件库上，不做出巨大的努力是不可能的。我们提出的一些再工程技术将解决如何在事后利用这些组件。

然而，关键的问题在于——“正确”的设计和对象的组织结构，并不是（或不简单是）来自初始的需求，而是理解了这些需求如何演变的结果。世界是不断变化的这个事实，不应该被单纯地看成一个问题，而应该作为解决问题的关键。

任何一个成功的软件系统都会受到遗留系统某些症状的影响。面向对象的遗留系统，只要它的结构和设计无需再响应变化的需求，它就是一个成功的面向对象系统。形成“连续不断地进行再工程的文化”是获得灵活的、可维护的面向对象系统的首要条件。

## 再工程生命周期

再工程和反向工程经常在同样的上下文中被提及。这两个词常常被混淆，因此我们有必要在这里澄清它们。Chikofsky和Cross[Chik92]是这样定义这两个词的：

**反向工程**是分析一个主题系统的过程，用来识别系统组件及其相互关系，产生系统的另一种表示形式或更高层的系统抽象。

这就是说，反向工程本质上关注的是如何理解一个系统以及它是如何工作的。

**再工程**……是检查和“改变主题系统”，用一种新的形式和这种新形式的实现来重建它。

另外，再工程关注的是重建一个系统，通常是修改一些已有的和将要发生的问题，更重要的是对进一步开发和扩展做准备。

对“反向工程”的介绍是为了更清楚地定义“前向工程”，因此我们给出如下定义：

前向工程是这样一个传统过程，即从高层抽象和逻辑的、与实现无关的设计，到系统的物理实现的转变过程。

尽管许多人接受了这样一个观点，前向工程的过程是迭代的，并且遵循Barry Boehm的软件开发螺旋模型[Boeh88]，但是前向工程的过程到底怎样，以及如何工作，这些仍是一个很有争议的问题。在这个模型中，软件系统通过反复地收集需求、评估风险、设计出新的版本并评价结果，来开发出连续的版本。这样一个通用的框架能适用于许多不同的特定的过程模型，这些模型在实践中已被应用。

如果说前向工程是从高层的需求和模型向具体实现转换，反向工程就是从具体实现向更抽象模型的回归，再工程就是从一种具体实现向另一种具体实现转换。

图1-1说明了这一观点。前向工程可以理解为从高层、抽象模型和制品到不断具体的模型和制品的转换过程。反向工程从代码重建出高层模型和制品。再工程是从一种低层表示向另一种低层表示的转换过程，同时由此重新产生高层制品。

要关注的重点是，再工程不仅仅是转换源码，而是从各个层次转换一个系统。正因为这个原因，同时谈论反向工程和再工程很有意义。在典型的遗留系统中，可以发现不仅是源码，所

有的文档和描述都不一致。由于人们不可能转换所不理解的东西，反向工程就成为再工程的一个前提条件。

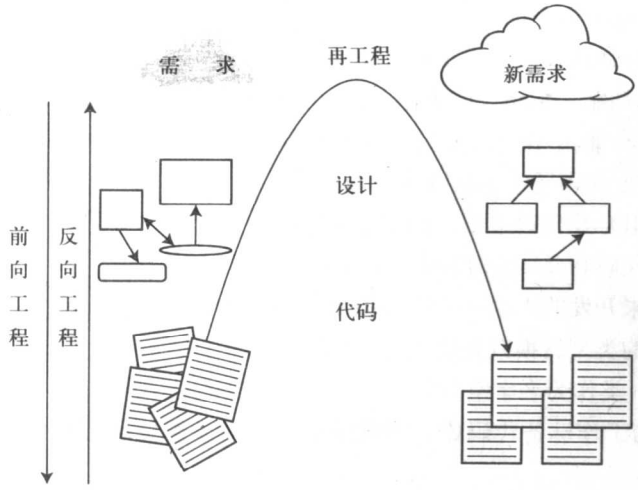



图1-1　前向工程、反向工程和再工程

### 反向工程

当试图理解系统是如何运行时，就开始了反向工程。一般来说，只需要对需要修改、扩展和替换的那一部分软件进行反向工程。(有时仅仅是为了理解系统如何使用而进行反向工程。这也是要求再工程的一个标志)。因此，反向工程的工作主要集中在重建软件文档和识别潜在问题，为再工程做准备。

在反向工程时，可以使用许多不同的信息源。例如，可以:

- 阅读现有文档。
- 阅读源代码。
- 运行软件。
- 采访使用者和开发者。
- 编写并执行测试用例。
- 产生并分析追踪结果。
- 使用不同的工具，产生源代码的高层视图和追踪结果。
- 分析版本历史。

当完成这些工作后，将逐渐建立软件的细化模型，跟踪不同的问题和答案，清理技术文档。同时还要当心需要修改的问题。

### 再工程

尽管再工程一个系统会有各种不同的原因，但实际的技术问题却非常相似。一般既有粗粒

度的结构问题，又有细粒度的设计问题。典型的粗粒度问题包括以下方面：

- 文档不完整。文档不存在或文档与实际系统不一致。
- 不恰当的分层。丢失或不恰当的分层往往会妨碍系统的灵活性和适应性。
- 缺乏模块化。模块间的强耦合性妨碍了系统发展。
- 重复的代码。“拷贝、粘贴，编辑”的方法既快又简单，但却会带来系统维护的噩梦。
- 功能性重复。类似的功能由不同的项目团队重复实现，导致代码的膨胀。

面向对象软件中的细粒度问题通常包括以下方面：

- 错误使用继承。使用组合及代码重用，而不是多态继承。
- 不用继承。使用重复代码和case语句来选择对象行为。
- 不恰当的操作位置。未开发的内聚——外部操作代替内部类的操作。
- 封装冲突。使用显式的类型转换以及C++友元方法。
- 类滥用。缺乏内聚——类作为名字空间。

最后，需要为再工程的工作准备代码库，为系统中准备修改或替换的部分开发全面的测试用例。

再工程需要做大量类似的相关工作。当然，其中最重要的一点是评估系统中哪一部分需要修改，哪一部分需要替换。

实际执行的代码转换工作会分为很多类别。正如Chikofsky和Cross所说：

**重新构建**（restructuring）是在保持系统的外部行为的情况下，在相同的相对抽象层上从一种表示向另一种表示的转化。

**重新构建**一般是指源代码的转换（例如从非结构化的“意大利面条式”的代码转化成结构化的或者“无goto语句”的代码），但它也可能涉及到设计层。

**重构**（refactoring）是在面向对象的上下文环境中重新构建。Martin Fowler对其定义如下[Fowl99]：

**重构**是一种转换软件系统的过程，它不改变代码的外部行为，而是改善它的内部结构。

软件“再工程”和软件“维护”的区别可能很难区分。IEEE组织曾多次试图给出软件维护的定义，其中一个定义如下：

……在软件交付使用后对软件产品的修改，用来改正错误，改善性能或其他属性，或者改进现有产品来适应变化的环境。

许多人可能会认为，“维护”是一项日常事务，而“再工程”则是大幅度的、付出大量努力的对系统的彻底重写，如图1-1所示。而另一些人可能认为，再工程是生活工作的一种方式。先开发一些，然后再工程一些，接着开发更多，如此反复下去[Beck00]。事实上，有一种很好的依据支持这样一种观点，即一种不断地再工程的文化，对获得健康、可维护的软件系统是非常必要的。

然而，不断地再工程目前还没有成为一种常见的实践行为。由于这个原因，我们在本书中提供了大量的进行再工程工作时使用的模式。但是请记住，我们所提出的大部分技术在进行小