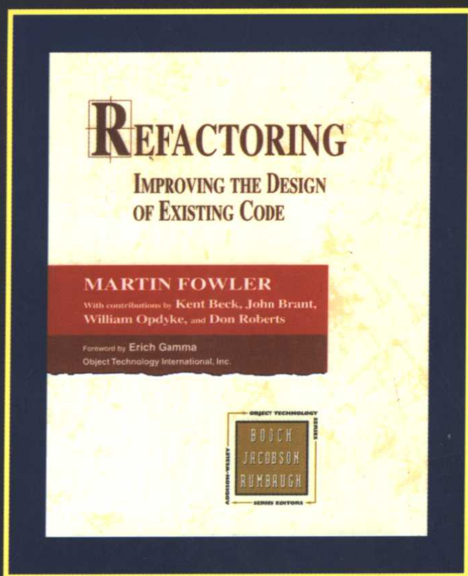
原版风暴·软件工程系列

Refactoring
Improving the Design of Existing Code

重构——改善既有代码的设计
（影印版）

[美] Martin Fowler 著



与《设计模式》齐名的经典巨著 ■

《设计模式》作者 Erich Gamma 为本书作序 ■

原汁原味，零距离领悟大师思想精髓 ■

中国电力出版社
www.infopower.com.cn

原版风暴·软件工程系列

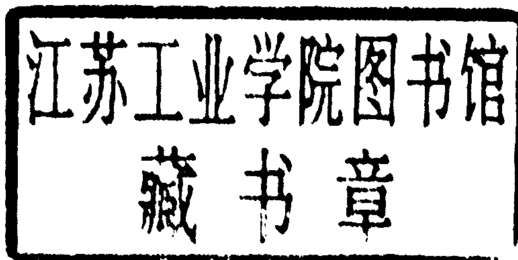
Refactoring

Improving the Design of Existing Code

重构——

改善既有代码的设计

（影印版）



［美］Martin Fowler 著

中国电力出版社

Refactoring: improving the design of existing code(ISBN 0-201-48567-2)
Martin Fowler
Copyright © 2000 Addison Wesley Longman, Inc.
Original English Language Edition Published by Addison Wesley Longman, Inc.
All rights reserved.
Reprinting edition published by PEARSON EDUCATION NORTH ASIA LTD and CHINA ELECTRIC POWER PRESS, Copyright © 2003.

本书影印版由 Pearson Education 授权中国电力出版社在中国境内（香港、澳门特别行政区和台湾地区除外）独家出版、发行。
未经出版者书面许可，不得以任何方式复制或抄袭本书的任何部分。

本书封面贴有 Pearson Education 防伪标签，无标签者不得销售。
北京市版权局著作合同登记号：图字：01-2003-1021
For sale and distribution in the People's Republic of China exclusively (except Taiwan, Hong Kong SAR and Macao SAR).
仅限于中华人民共和国境内（不包括中国香港、澳门特别行政区和中国台湾地区）销售发行。

图书在版编目（CIP）数据
重构——改善既有代码的设计 /（美）福勒　等著．—影印本．—北京：中国电力出版社，2003
（原版风暴・软件工程系列）
ISBN 7-5083-1501-4
Ⅰ.重…　Ⅱ.福…　Ⅲ.代码-程序设计-英文　Ⅳ.TP311.11
中国版本图书馆 CIP 数据核字（2003）第 027835 号

本书勘误及相关信息请访问：http://www.refactoring.com

丛 书 名：原版风暴・软件工程系列
书　　名：重构——改善既有代码的设计（影印版）
编　　著：（美）Martin Fowler
责任编辑：关敏
出版发行：中国电力出版社
　　　　地址：北京市三里河路6号　　邮政编码：100044
　　　　电话：（010）88515918　　传　　真：（010）88518169
印　　刷：北京地矿印刷厂
开　　本：787×1092　1/16　　印　　张：28.75
书　　号：ISBN 7-5083-1501-4
版　　次：2003 年 7 月北京第 1 版　　2004 年 1 月第 3 次印刷
定　　价：49.00 元
版权所有　翻印必究

Smell	Common Refactorings
Alternative Classes with Different Interfaces, p. 85	*Rename Method (273), Move Method (142)*
Comments, p. 87	*Extract Method (110), Introduce Assertion (267)*
Data Class, p. 86	*Move Method (142), Encapsulate Field (206), Encapsulate Collection (208)*
Data Clumps, p. 81	*Extract Class (149), Introduce Parameter Object (295), Preserve Whole Object (288)*
Divergent Change, p. 79	*Extract Class (149)*
Duplicated Code, p. 76	*Extract Method (110), Extract Class (149), Pull Up Method (322), Form Template Method (345)*
Feature Envy, p. 80	*Move Method (142), Move Field (146), Extract Method (110)*
Inappropriate Intimacy, p. 85	*Move Method (142), Move Field (146), Change Bidirectional Association to Unidirectional (200), Replace Inheritance with Delegation (352), Hide Delegate (157)*
Incomplete Library Class, p. 86	*Introduce Foreign Method (162), Introduce Local Extension (164)*
Large Class, p. 78	*Extract Class (149), Extract Subclass (330), Extract Interface (341), Replace Data Value with Object (175)*
Lazy Class, p. 83	*Inline Class (154), Collapse Hierarchy (344)*
Long Method, p. 76	*Extract Method (110), Replace Temp with Query (120), Replace Method with Method Object (135), Decompose Conditional (238)*

Smell	Common Refactorings
Long Parameter List, p. 78	*Replace Parameter with Method (292), Introduce Parameter Object (295), Preserve Whole Object (288)*
Message Chains, p. 84	*Hide Delegate (157)*
Middle Man, p. 85	*Remove Middle Man (160), Inline Method (117), Replace Delegation with Inheritance (355)*
Parallel Inheritance Hierarchies, p. 83	*Move Method (142), Move Field (146)*
Primitive Obsession, p. 81	*Replace Data Value with Object (175), Extract Class (149), Introduce Parameter Object (295), Replace Array with Object (186), Replace Type Code with Class (218), Replace Type Code with Subclasses (223), Replace Type Code with State/ Strategy (227)*
Refused Bequest, p. 87	*Replace Inheritance with Delegation (352)*
Shotgun Surgery, p. 80	*Move Method (142), Move Field (146), Inline Class (154)*
Speculative Generality, p. 83	*Collapse Hierarchy (344), Inline Class (154), Remove Parameter (277), Rename Method (273)*
Switch Statements, p. 82	*Replace Conditional with Polymorphism (255), Replace Type Code with Subclasses (223), Replace Type Code with State/Strategy (227), Replace Parameter with Explicit Methods (285), Introduce Null Object (260)*
Temporary Field, p. 84	*Extract Class (149), Introduce Null Object (260)*

此为试读，需要完整PDF请访问：www.ertongbook.com

Foreword

"Refactoring" was conceived in Smalltalk circles, but it wasn't long before it found its way into other programming language camps. Because refactoring is integral to framework development, the term comes up quickly when "frameworkers" talk about their craft. It comes up when they refine their class hierarchies and when they rave about how many lines of code they were able to delete. Frameworkers know that a framework won't be right the first time around—it must evolve as they gain experience. They also know that the code will be read and modified more frequently than it will be written. The key to keeping code readable and modifiable is refactoring—for frameworks, in particular, but also for software in general.

So, what's the problem? Simply this: Refactoring is risky. It requires changes to working code that can introduce subtle bugs. Refactoring, if not done properly, can set you back days, even weeks. And refactoring becomes riskier when practiced informally or ad hoc. You start digging in the code. Soon you discover new opportunities for change, and you dig deeper. The more you dig, the more stuff you turn up. . .and the more changes you make. Eventually you dig yourself into a hole you can't get out of. To avoid digging your own grave, refactoring must be done systematically. When my coauthors and I wrote *Design Patterns,* we mentioned that design patterns provide targets for refactorings. However, identifying the target is only one part of the problem; transforming your code so that you get there is another challenge.

Martin Fowler and the contributing authors make an invaluable contribution to object-oriented software development by shedding light on the refactoring process. This book explains the principles and best practices of refactoring, and points out when and where you should start digging in your code to improve it. At the book's core is a comprehensive catalog of refactorings. Each refactoring describes the motivation and mechanics of a proven code transformation. Some of the refactorings, such as Extract Method or Move Field, may seem obvious.

But don't be fooled. Understanding the mechanics of such refactorings is the key to refactoring in a disciplined way. The refactorings in this book will help you change your code one small step at a time, thus reducing the risks of evolving your design. You will quickly add these refactorings and their names to your development vocabulary.

My first experience with disciplined, "one step at a time" refactoring was when I was pair-programming at 30,000 feet with Kent Beck. He made sure that we applied refactorings from this book's catalog one step at a time. I was amazed at how well this practice worked. Not only did my confidence in the resulting code increase, I also felt less stressed. I highly recommend you try these refactorings: You and your code will feel much better for it.

—Erich Gamma
Object Technology International, Inc.

Preface

Once upon a time, a consultant made a visit to a development project. The consultant looked at some of the code that had been written; there was a class hierarchy at the center of the system. As he wandered through the hierarchy, the consultant saw that it was rather messy. The higher-level classes made certain assumptions about how the classes would work, assumptions that were embodied in inherited code. That code didn't suit all the subclasses, however, and was overridden quite heavily. If the superclass had been modified a little, then much less overriding would have been necessary. In other places some of the intention of the superclass had not been properly understood, and behavior present in the superclass was duplicated. In yet other places several subclasses did the same thing with code that could clearly be moved up the hierarchy.

The consultant recommended to the project management that the code be looked at and cleaned up, but the project management didn't seem enthusiastic. The code seemed to work and there were considerable schedule pressures. The managers said they would get around to it at some later point.

The consultant had also shown the programmers who had worked on the hierarchy what was going on. The programmers were keen and saw the problem. They knew that it wasn't really their fault; sometimes a new pair of eyes are needed to spot the problem. So the programmers spent a day or two cleaning up the hierarchy. When they were finished, the programmers had removed half the code in the hierarchy without reducing its functionality. They were pleased with the result and found that it became quicker and easier both to add new classes to the hierarchy and to use the classes in the rest of the system.

The project management was not pleased. Schedules were tight and there was a lot of work to do. These two programmers had spent two days doing work that had done nothing to add the many features the system had to deliver in a few months time. The old code had worked just fine. So the design was a bit more "pure" a bit more "clean." The project had to ship code that worked,

not code that would please an academic. The consultant suggested that this cleaning up be done on other central parts of the system. Such an activity might halt the project for a week or two. All this activity was devoted to making the code look better, not to making it do anything that it didn't already do.

How do you feel about this story? Do you think the consultant was right to suggest further clean up? Or do you follow that old engineering adage, "if it works, don't fix it"?

I must admit to some bias here. I was that consultant. Six months later the project failed, in large part because the code was too complex to debug or to tune to acceptable performance.

The consultant Kent Beck was brought in to restart the project, an exercise that involved rewriting almost the whole system from scratch. He did several things differently, but one of the most important was to insist on continuous cleaning up of the code using refactoring. The success of this project, and role refactoring played in this success, is what inspired me to write this book, so that I could pass on the knowledge that Kent and others have learned in using refactoring to improve the quality of software.

What Is Refactoring?

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written.

"Improving the design after it has been written." That's an odd turn of phrase. In our current understanding of software development we believe that we design and then we code. A good design comes first, and the coding comes second. Over time the code will be modified, and the integrity of the system, its structure according to that design, gradually fades. The code slowly sinks from engineering to hacking.

Refactoring is the opposite of this practice. With refactoring you can take a bad design, chaos even, and rework it into well-designed code. Each step is simple, even simplistic. You move a field from one class to another, pull some code out of a method to make into its own method, and push some code up or down a hierarchy. Yet the cumulative effect of these small changes can radically improve the design. It is the exact reverse of the normal notion of software decay.

With refactoring you find the balance of work changes. You find that design, rather than occurring all up front, occurs continuously during development. You learn from building the system how to improve the design. The resulting interaction leads to a program with a design that stays good as development continues.

What's in This Book?

This book is a guide to refactoring; it is written for a professional programmer. My aim is to show you how to do refactoring in a controlled and efficient manner. You will learn to refactor in such a way that you don't introduce bugs into the code but instead methodically improve the structure.

It's traditional to start books with an introduction. Although I agree with that principle, I don't find it easy to introduce refactoring with a generalized discussion or definitions. So I start with an example. Chapter 1 takes a small program with some common design flaws and refactors it into a more acceptable object-oriented program. Along the way we see both the process of refactoring and the application of several useful refactorings. This is the key chapter to read if you want to understand what refactoring really is about.

In Chapter 2 I cover more of the general principles of refactoring, some definitions, and the reasons for doing refactoring. I outline some of the problems with refactoring. In Chapter 3 Kent Beck helps me describe how to find bad smells in code and how to clean them up with refactorings. Testing plays a very important role in refactoring, so Chapter 4 describes how to build tests into code with a simple open-source Java testing framework.

The heart of the book, the catalog of refactorings, stretches from Chapter 5 through Chapter 12. This is by no means a comprehensive catalog. It is the beginning of such a catalog. It includes the refactorings that I have written down so far in my work in this field. When I want to do something, such as *Replace Conditional with Polymorphism (255)*, the catalog reminds me how to do it in a safe, step-by-step manner. I hope this is the section of the book you'll come back to often.

In this book I describe the fruit of a lot of research done by others. The last chapters are guest chapters by some of these people. Chapter 13 is by Bill Opdyke, who describes the issues he has come across in adopting refactoring in commercial development. Chapter 14 is by Don Roberts and John Brant, who describe the true future of refactoring, automated tools. I've left the final word, Chapter 15, to the master of the art, Kent Beck.

Refactoring in Java

For all of this book I use examples in Java. Refactoring can, of course, be done with other languages, and I hope this book will be useful to those working with other languages. However, I felt it would be best to focus this book on Java because it is the language I know best. I have added occasional notes for refactoring in other languages, but I hope other people will build on this foundation with books aimed at specific languages.

To help communicate the ideas best, I have not used particularly complex areas of the Java language. So I've shied away from using inner classes, reflection, threads, and many other of Java's more powerful features. This is because I want to focus on the core refactorings as clearly as I can.

I should emphasize that these refactorings are not done with concurrent or distributed programming in mind. Those topics introduce additional concerns that are beyond the scope of this book.

Who Should Read This Book?

This book is aimed at a professional programmer, someone who writes software for a living. The examples and discussion include a lot of code to read and understand. The examples are all in Java. I chose Java because it is an increasingly well-known language that can be easily understood by anyone with a background in C. It is also an object-oriented language, and object-oriented mechanisms are a great help in refactoring.

Although it is focused on the code, refactoring has a large impact on the design of system. It is vital for senior designers and architects to understand the principles of refactoring and to use them in their projects. Refactoring is best introduced by a respected and experienced developer. Such a developer can best understand the principles behind refactoring and adapt those principles to the specific workplace. This is particularly true when you are using a language other than Java, because you have to adapt the examples I've given to other languages.

Here's how to get the most from this book without reading all of it.

- **If you want to understand what refactoring is,** read Chapter 1; the example should make the process clear.
- **If you want to understand why you should refactor,** read the first two chapters. They will tell you what refactoring is and why you should do it.

此为试读，需要完整PDF请访问：www.ertongbook.com

- **If you want to find where you should refactor,** read Chapter 3. It tells you the signs that suggest the need for refactoring.
- **If you want to actually do refactoring,** read the first four chapters completely. Then skip-read the catalog. Read enough of the catalog to know roughly what is in there. You don't have to understand all the details. When you actually need to carry out a refactoring, read the refactoring in detail and use it to help you. The catalog is a reference section, so you probably won't want to read it in one go. You should also read the guest chapters, especially Chapter 15.

Building on the Foundations Laid by Others

I need to say right now, at the beginning, that I owe a big debt with this book, a debt to those whose work over the last decade has developed the field of refactoring. Ideally one of them should have written this book, but I ended up being the one with the time and energy.

Two of the leading proponents of refactoring are **Ward Cunningham** and **Kent Beck.** They used it as a central part of their development process in the early days and have adapted their development processes to take advantage of it. In particular it was my collaboration with Kent that really showed me the importance of refactoring, an inspiration that led directly to this book.

Ralph Johnson leads a group at the University of Illinois at Urbana-Champaign that is notable for its practical contributions to object technology. Ralph has long been a champion of refactoring, and several of his students have worked on the topic. **Bill Opdyke** developed the first detailed written work on refactoring in his doctoral thesis. **John Brant** and **Don Roberts** have gone beyond writing words into writing a tool, the Refactoring Browser, for refactoring Smalltalk programs.

Acknowledgments

Even with all that research to draw on, I still needed a lot of help to write this book. First and foremost, Kent Beck was a huge help. The first seeds were planted in a bar in Detroit when Kent told me about a paper he was writing for the *Smalltalk Report* [Beck, hanoi]. It not only provided many ideas for me to steal for Chapter 1 but also started me off in taking notes of refactorings. Kent helped in other places too. He came up with the idea of code smells, encouraged

me at various sticky points, and generally worked with me to make this book work. I can't help thinking he could have written this book much better himself, but I had the time and can only hope I did the subject justice.

As I've written this, I wanted to share much of this expertise directly with you, so I'm very grateful that many of these people have spent some time adding some material to this book. Kent Beck, John Brant, William Opdyke, and Don Roberts have all written or co-written chapters. In addition, Rich Garzaniti and Ron Jeffries have added useful sidebars.

Any author will tell you that technical reviewers do a great deal to help in a book like this. As usual, Carter Shanklin and his team at Addison-Wesley put together a great panel of hard-nosed reviewers. These were

- Ken Auer, Rolemodel Software, Inc.
- Joshua Bloch, Sun Microsystems, Java Software
- John Brant, University of Illinois at Urbana-Champaign
- Scott Corley, High Voltage Software, Inc.
- Ward Cunningham, Cunningham & Cunningham, Inc.
- Stéphane Ducasse
- Erich Gamma, Object Technology International, Inc.
- Ron Jeffries
- Ralph Johnson, University of Illinois
- Joshua Kerievsky, Industrial Logic, Inc.
- Doug Lea, SUNY Oswego
- Sander Tichelaar

They all added a great deal to the readability and accuracy of this book, and removed at least some of the errors that can lurk in any manuscript. I'd like to highlight a couple of very visible suggestions that made a difference to the look of the book. Ward and Ron got me to do Chapter 1 in the side-by-side style. Joshua suggested the idea of the code sketches in the catalog.

In addition to the official review panel there were many unofficial reviewers. These people looked at the manuscript or the work in progress on my Web pages and made helpful comments. They include Leif Bennett, Michael Feathers, Michael Finney, Neil Galarneau, Hisham Ghazouli, Tony Gould, John Isner, Brian Marick, Ralf Reissing, John Salt, Mark Swanson, Dave Thomas,

and Don Wells. I'm sure there are others who I've forgotton; I apologize and offer my thanks.

A particularly entertaining review group is the infamous reading group at the University of Illinois at Urbana-Champaign. Because this book reflects so much of their work, I'm particularly grateful for their efforts captured in real audio. This group includes Fredrico "Fred" Balaguer, John Brant, Ian Chai, Brian Foote, Alejandra Garrido, Zhijiang "John" Han, Peter Hatch, Ralph Johnson, Songyu "Raymond" Lu, Dragos-Anton Manolescu, Hiroaki Nakamura, James Overturf, Don Roberts, Chieko Shirai, Les Tyrell, and Joe Yoder.

Any good idea needs to be tested in a serious production system. I saw refactoring have a huge effect on the Chrysler Comprehensive Compensation system (C3). I want to thank all the members of that team: Ann Anderson, Ed Anderi, Ralph Beattie, Kent Beck, David Bryant, Bob Coe, Marie DeArment, Margaret Fronczak, Rich Garzaniti, Dennis Gore, Brian Hacker, Chet Hendrickson, Ron Jeffries, Doug Joppie, David Kim, Paul Kowalsky, Debbie Mueller, Tom Murasky, Richard Nutter, Adrian Pantea, Matt Saigeon, Don Thomas, and Don Wells. Working with them cemented the principles and benefits of refactoring into me on a firsthand basis. Watching their progress as they use refactoring heavily helps me see what refactoring can do when applied to a large project over many years.

Again I had the help of J. Carter Shanklin at Addison-Wesley and his team: Krysia Bebick, Susan Cestone, Chuck Dutton, Kristin Erickson, John Fuller, Christopher Guzikowski, Simone Payment, and Genevieve Rajewski. Working with a good publisher is a pleasure; they provided a lot of support and help.

Talking of support, the biggest sufferer from a book is always the closest to the author, in this case my (now) wife Cindy. Thanks for loving me even when I was hidden in the study. As much time as I put into this book, I never stopped being distracted by thinking of you.

Martin Fowler
Melrose, Massachusetts
fowler@acm.org
http://ourworld.compuserve.com/homepages/martin_fowler

Contents

Foreword xiii

Preface xv

What Is Refactoring? xvi

What's in This Book? xvii

Who Should Read This Book? xviii

Building on the Foundations Laid by Others xix

Acknowledgments xix

Chapter 1: Refactoring, a First Example 1

The Starting Point 1

The First Step in Refactoring 7

Decomposing and Redistributing the Statement Method 8

Replacing the Conditional Logic on Price Code
with Polymorphism 34

Final Thoughts 52

Chapter 2: Principles in Refactoring 53

Defining Refactoring 53

Why Should You Refactor? 55

When Should You Refactor? 57

What Do I Tell My Manager? 60

Problems with Refactoring 62

Refactoring and Design 66

Refactoring and Performance 69

Where Did Refactoring Come From? 71

Chapter 3: Bad Smells in Code (*by Kent Beck and Martin Fowler*) .. 75
Duplicated Code 76
Long Method 76
Large Class 78
Long Parameter List 78
Divergent Change 79
Shotgun Surgery 80
Feature Envy 80
Data Clumps 81
Primitive Obsession 81
Switch Statements 82
Parallel Inheritance Hierarchies 83
Lazy Class 83
Speculative Generality 83
Temporary Field 84
Message Chains 84
Middle Man 85
Inappropriate Intimacy 85
Alternative Classes with Different Interfaces 85
Incomplete Library Class 86
Data Class 86
Refused Bequest 87
Comments 87

Chapter 4: Building Tests 89
The Value of Self-testing Code 89
The JUnit Testing Framework 91
Adding More Tests 97

Chapter 5: Toward a Catalog of Refactorings 103
Format of the Refactorings 103
Finding References 105
How Mature Are These Refactorings? 106

Chapter 6: Composing Methods 109
Extract Method 110
Inline Method 117

Inline Temp 119
Replace Temp with Query 120
Introduce Explaining Variable 124
Split Temporary Variable 128
Remove Assignments to Parameters 131
Replace Method with Method Object 135
Substitute Algorithm 139

Chapter 7: Moving Features Between Objects **141**

Move Method 142
Move Field 146
Extract Class 149
Inline Class 154
Hide Delegate 157
Remove Middle Man 160
Introduce Foreign Method 162
Introduce Local Extension 164

Chapter 8: Organizing Data **169**

Self Encapsulate Field 171
Replace Data Value with Object 175
Change Value to Reference 179
Change Reference to Value 183
Replace Array with Object 186
Duplicate Observed Data 189
Change Unidirectional Association to Bidirectional 197
Change Bidirectional Association to Unidirectional 200
Replace Magic Number with Symbolic Constant 204
Encapsulate Field 206
Encapsulate Collection 208
Replace Record with Data Class 217
Replace Type Code with Class 218
Replace Type Code with Subclasses 223
Replace Type Code with State/Strategy 227
Replace Subclass with Fields 232

此为试读，需要完整PDF请访问：www.ertongbook.com