REFACTORING

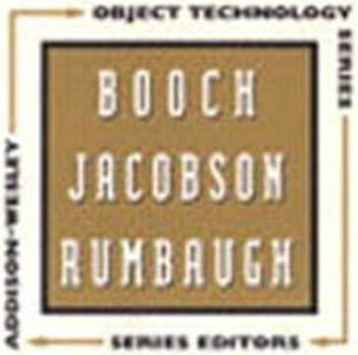
IMPROVING THE DESIGN OF EXISTING CODE

MARTIN FOWLER

With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts

Foreword by Erich Gamma

Object Technology International Inc.



Refactoring

The Addison-Wesley Object Technology Series

Grady Booch, Ivar Jacobson, and James Rumbaugh, Series Editors
For more information, check out the series web site at www.awprofessional.com/otseries.

Ahmed/Umrysh, *Developing Enterprise Java Applications with J2EE™ and UML*

Arlow/Neustadt, *Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML*

Arlow/Neustadt, *UML 2 and the Unified Process, Second Edition*

Armour/Miller, *Advanced Use Case Modeling: Software Systems*

Bellin/Simone, *The CRC Card Book*

Bergström/Råberg, *Adopting the Rational Unified Process: Success with the RUP*

Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*

Bittner/Spence, *Managing Iterative Software Development Projects*

Bittner/Spence, *Use Case Modeling*

Booch, *Object Solutions: Managing the Object-Oriented Project*

Booch, *Object-Oriented Analysis and Design with Applications, 3E*

Booch/Bryan, *Software Engineering with ADA, 3E*

Booch/Rumbaugh/Jacobson, *The Unified Modeling Language User Guide, Second Edition*

Box et al., *Effective COM: 50 Ways to Improve Your COM and MTS-based Applications*

Buckley/Pulsipher, *The Art of ClearCase® Deployment*

Carlson, *Modeling XML Applications with UML: Practical e-Business Applications*

Clarke/Baniassad, *Aspect-Oriented Analysis and Design*

Collins, *Designing Object-Oriented User Interfaces*

Conallen, *Building Web Applications with UML, 2E*

D'Souza/Wills, *Objects, Components, and Frameworks with UML: The Catalysis(SM) Approach*

Denney, *Succeeding with Use Cases*

Douglass, *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*

Douglass, *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*

Douglass, *Real Time UML, 3E: Advances in The UML for Real-Time Systems*

Eeles et al., *Building J2EE™ Applications with the Rational Unified Process*

Fowler, *Analysis Patterns: Reusable Object Models*

Fowler, *UML Distilled, 3E: A Brief Guide to the Standard Object Modeling Language*

Fowler et al., *Refactoring: Improving the Design of Existing Code*

Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*

Gomaa, *Designing Software Product Lines with UML*

Heinckiens, *Building Scalable Database Applications: Object-Oriented Design, Architectures, and Implementations*

Hofmeister/Nord/Dilip, *Applied Software Architecture*

Jacobson/Booch/Rumbaugh, *The Unified Software Development Process*

Jacobson/Ng, *Aspect-Oriented Software Development with Use Cases*

Jordan, *C++ Object Databases: Programming with the ODMG Standard*

Kleppe/Warmer/Bast, *MDA Explained: The Model Driven Architecture™: Practice and Promise*

Kroll/Kruchten, *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*

Kroll/MacIsaac, *Agility and Discipline Made Easy: Practices from OpenUP and RUP*

Kruchten, *The Rational Unified Process, 3E: An Introduction*

LaLonde, *Discovering Smalltalk*

Lau, *The Art of Objects: Object-Oriented Design and Architecture*

Leffingwell/Widrig, *Managing Software Requirements, 2E: A Use Case Approach*

Manassis, *Practical Software Engineering: Analysis and Design for the .NET Platform*

Marshall, *Enterprise Modeling with UML: Designing Successful Software through Business Analysis*

McGregor/Sykes, *A Practical Guide to Testing Object-Oriented Software*

Mellor/Balcer, *Executable UML: A Foundation for Model-Driven Architecture*

Mellor et al., *MDA Distilled: Principles of Model-Driven Architecture*

Naiburg/Maksimchuk, *UML for Database Design*

Oestereich, *Developing Software with UML, 2E: Object-Oriented Analysis and Design in Practice*

Page-Jones, *Fundamentals of Object-Oriented Design in UML*

Pohl, *Object-Oriented Programming Using C++, 2E*

Quatrani, *Visual Modeling with Rational Rose 2002 and UML*

Rector/Sells, *ATL Internals*

Reed, *Developing Applications with Visual Basic and UML*

Rosenberg/Scott, *Applying Use Case Driven Object Modeling with UML: An Annotated e-Commerce Example*

Rosenberg/Scott, *Use Case Driven Object Modeling with UML: A Practical Approach*

Royce, *Software Project Management: A Unified Framework*

Rumbaugh/Jacobson/Booch, *The Unified Modeling Language Reference Manual*

Schneider/Winters, *Applying Use Cases, 2E: A Practical Guide*

Smith, *IBM Smalltalk*

Smith/Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*

Tavares/Fertitta/Rector/Sells, *ATL Internals, Second Edition*

Tkach/Fang/So, *Visual Modeling Technique*

Unhelkar, *Process Quality Assurance for UML-Based Projects*

Warmer/Kleppe, *The Object Constraint Language, 2E: Getting Your Models Ready for MDA*

White, *Software Configuration Management Strategies and Rational ClearCase®: A Practical Introduction*

The Component Software Series

Clemens Szyperski, Series Editor
For more information, check out the series web site at www.awprofessional.com/csseries.

Cheesman/Daniels, *UML Components: A Simple Process for Specifying Component-Based Software*

Szyperski, *Component Software, 2E: Beyond Object-Oriented Programming*

Refactoring

Improving the Design of Existing Code

Martin Fowler

With contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts

ADDISON–WESLEY

An imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts • Harlow, England • Menlo Park, California
Berkeley, California • Don Mills, Ontario • Sydney
Bonn • Amsterdam • Tokyo • Mexico City

Library of Congress Cataloging-in-Publication Data

Fowler, Martin,
Refactoring : improving the design of existing code / Martin Fowler.
p. m. — (The Addison-Wesley object technology series)
Includes bibliographical references and index.
ISBN 0-201-48567-2
1. Software refactoring. 2. Object-oriented programming (Computer science) I. Title. II. Series.
QA76.76.R42F69 1999
005.1'4—dc21 99–20765
CIP

Executive Editor: J. Carter Shanklin
Project Editor: Krysia Bebick
Editorial Assistant: Kristin Erickson
Production Manager: John Fuller
Production Coordinator: Genevieve C. Rajewski
Copy Editor: Catherine Judge Allen
Composition: Kim Arney
Index: Irv Hershman
Proofreader: Amy Finch

ISBN 0-201-48567-2
Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
Twenty-ninth printing, November 2014

For Cindy

This page intentionally left blank

Contents

This page intentionally left blank

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.

- **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

This page intentionally left blank

This page intentionally left blank

Chapter 3

Bad Smells in Code

by Kent Beck and Martin Fowler

If it stinks, change it.
— Grandma Beck, discussing child-rearing philosophy

By now you have a good idea of how refactoring works. But just because you know how doesn't mean you know when. Deciding when to start refactoring, and when to stop, is just as important to refactoring as knowing how to operate the mechanics of a refactoring.

Now comes the dilemma. It is easy to explain to you how to delete an instance variable or create a hierarchy. These are simple matters. Trying to explain when you should do these things is not so cut-and-dried. Rather than appealing to some vague notion of programming aesthetics (which frankly is what we consultants usually do), I wanted something a bit more solid.

I was mulling over this tricky issue when I visited Kent Beck in Zurich. Perhaps he was under the influence of the odors of his newborn daughter at the time, but he had come up with the notion describing the "when" of refactoring in terms of smells. "Smells," you say, "and that is supposed to be better than vague aesthetics?" Well, yes. We look at lots of code, written for projects that span the gamut from wildly successful to nearly dead. In doing so, we have learned to look for certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring. (We are switching over to "we" in this chapter to reflect the fact that Kent and I wrote this chapter jointly. You can tell the difference because the funny jokes are mine and the others are his.)

One thing we won't try to do here is give you precise criteria for when a refactoring is overdue. In our experience no set of metrics rivals informed human intuition. What we will do is give you indications that there is trouble that can be solved by a refactoring. You will have to develop your own sense of how many instance variables are too many instance variables and how many lines of code in a method are too many lines.

You should use this chapter and the table on the inside back cover as a way to give you inspiration when you're not sure what refactorings to do. Read the chapter (or skim the table) to try to identify what it is you're smelling, then go to the refactorings we suggest to see whether they will help you. You may not find the exact smell you can detect, but hopefully it should point you in the right direction.

Duplicated Code

Number one in the stink parade is duplicated code. If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them.

The simplest duplicated code problem is when you have the same expression in two methods of the same class. Then all you have to do is *Extract Method (110)* and invoke the code from both places.

Another common duplication problem is when you have the same expression in two sibling subclasses. You can eliminate this duplication by using *Extract Method (110)* in both classes then *Pull Up Method (322)*. If the code is similar but not the same, you need to use *Extract Method (110)* to separate the similar bits from the different bits. You may then find you can use *Form Template Method (345)*. If the methods do the same thing with a different algorithm, you can choose the clearer of the two algorithms and use *Substitute Algorithm (139)*.

If you have duplicated code in two unrelated classes, consider using *Extract Class (149)* in one class and then use the new component in the other. Another possibility is that the method really belongs only in one of the classes and should be invoked by the other class or that the method belongs in a third class that should be referred to by both of the original classes. You have to decide where the method makes sense and ensure it is there and nowhere else.

Long Method

The object programs that live best and longest are those with short methods. Programmers new to objects often feel that no computation ever takes place, that object programs are endless sequences of delegation. When you have lived with such a program for a few years, however, you learn just how valuable all those little methods are. All of the payoffs of indirection—explanation, sharing,

and choosing—are supported by little methods (see Indirection and Refactoring on page 61).

Since the early days of programming people have realized that the longer a procedure is, the more difficult it is to understand. Older languages carried an overhead in subroutine calls, which deterred people from small methods. Modern OO languages have pretty much eliminated that overhead for in-process calls. There is still an overhead to the reader of the code because you have to switch context to see what the subprocedure does. Development environments that allow you to see two methods at once help to eliminate this step, but the real key to making it easy to understand small methods is good naming. If you have a good name for a method you don't need to look at the body.

The net effect is that you should be much more aggressive about decomposing methods. A heuristic we follow is that whenever we feel the need to comment something, we write a method instead. Such a method contains the code that was commented but is named after the intention of the code rather than how it does it. We may do this on a group of lines or on as little as a single line of code. We do this even if the method call is longer than the code it replaces, provided the method name explains the purpose of the code. The key here is not method length but the semantic distance between what the method does and how it does it.

Ninety-nine percent of the time, all you have to do to shorten a method is *Extract Method (110)*. Find parts of the method that seem to go nicely together and make a new method.

If you have a method with lots of parameters and temporary variables, these elements get in the way of extracting methods. If you try to use *Extract Method,* you end up passing so many of the parameters and temporary variables as parameters to the extracted method that the result is scarcely more readable than the original. You can often use *Replace Temp with Query (120)* to eliminate the temps. Long lists of parameters can be slimmed down with *Introduce Parameter Object (295)* and *Preserve Whole Object (288)*.

If you've tried that, and you still have too many temps and parameters, it's time to get out the heavy artillery: *Replace Method with Method Object (135)*.

How do you identify the clumps of code to extract? A good technique is to look for comments. They often signal this kind of semantic distance. A block of code with a comment that tells you what it is doing can be replaced by a method whose name is based on the comment. Even a single line is worth extracting if it needs explanation.

Conditionals and loops also give signs for extractions. Use *Decompose Conditional (238)* to deal with conditional expressions. With loops, extract the loop and the code within the loop into its own method.

Large Class

When a class is trying to do too much, it often shows up as too many instance variables. When a class has too many instance variables, duplicated code cannot be far behind.

You can *Extract Class (149)* to bundle a number of the variables. Choose variables to go together in the component that makes sense for each. For example, "depositAmount" and "depositCurrency" are likely to belong together in a component. More generally, common prefixes or suffixes for some subset of the variables in a class suggest the opportunity for a component. If the component makes sense as a subclass, you'll find *Extract Subclass (330)* often is easier.

Sometimes a class does not use all of its instance variables all of the time. If so, you may be able to *Extract Class (149)* or *Extract Subclass (330)* many times.

As with a class with too many instance variables, a class with too much code is prime breeding ground for duplicated code, chaos, and death. The simplest solution (have we mentioned that we like simple solutions?) is to eliminate redundancy in the class itself. If you have five hundred-line methods with lots of code in common, you may be able to turn them into five ten-line methods with another ten two-line methods extracted from the original.

As with a class with a huge wad of variables, the usual solution for a class with too much code is either to *Extract Class (149)* or *Extract Subclass (330)*. A useful trick is to determine how clients use the class and to use *Extract Interface (341)* for each of these uses. That may give you ideas on how you can further break up the class.

If your large class is a GUI class, you may need to move data and behavior to a separate domain object. This may require keeping some duplicate data in both places and keeping the data in sync. *Duplicate Observed Data (189)* suggests how to do this. In this case, especially if you are using older Abstract Windows Toolkit (AWT) components, you might follow this by removing the GUI class and replacing it with Swing components.

Long Parameter List

In our early programming days we were taught to pass in as parameters everything needed by a routine. This was understandable because the alternative was global data, and global data is evil and usually painful. Objects change this situation because if you don't have something you need, you can always ask

another object to get it for you. Thus with objects you don't pass in everything the method needs; instead you pass enough so that the method can get to everything it needs. A lot of what a method needs is available on the method's host class. In object-oriented programs parameter lists tend to be much smaller than in traditional programs.

This is good because long parameter lists are hard to understand, because they become inconsistent and difficult to use, and because you are forever changing them as you need more data. Most changes are removed by passing objects because you are much more likely to need to make only a couple of requests to get at a new piece of data.

Use *Replace Parameter with Method (292)* when you can get the data in one parameter by making a request of an object you already know about. This object might be a field or it might be another parameter. Use *Preserve Whole Object (288)* to take a bunch of data gleaned from an object and replace it with the object itself. If you have several data items with no logical object, use *Introduce Parameter Object (295)*.

There is one important exception to making these changes. This is when you explicitly do not want to create a dependency from the called object to the larger object. In those cases unpacking data and sending it along as parameters is reasonable, but pay attention to the pain involved. If the parameter list is too long or changes too often, you need to rethink your dependency structure.

Divergent Change

We structure our software to make change easier; after all, software is meant to be soft. When we make a change we want to be able to jump to a single clear point in the system and make the change. When you can't do this you are smelling one of two closely related pungencies.

Divergent change occurs when one class is commonly changed in different ways for different reasons. If you look at a class and say, "Well, I will have to change these three methods every time I get a new database; I have to change these four methods every time there is a new financial instrument," you likely have a situation in which two objects are better than one. That way each object is changed only as a result of one kind of change. Of course, you often discover this only after you've added a few databases or financial instruments. Any change to handle a variation should change a single class, and all the typing in the new class should express the variation. To clean this up you identify everything that changes for a particular cause and use *Extract Class (149)* to put them all together.

Shotgun Surgery

Shotgun surgery is similar to divergent change but is the opposite. You whiff this when every time you make a kind of change, you have to make a lot of little changes to a lot of different classes. When the changes are all over the place, they are hard to find, and it's easy to miss an important change.

In this case you want to use *Move Method (142)* and *Move Field (146)* to put all the changes into a single class. If no current class looks like a good candidate, create one. Often you can use *Inline Class (154)* to bring a whole bunch of behavior together. You get a small dose of divergent change, but you can easily deal with that.

Divergent change is one class that suffers many kinds of changes, and shotgun surgery is one change that alters many classes. Either way you want to arrange things so that, ideally, there is a one-to-one link between common changes and classes.

Feature Envy

The whole point of objects is that they are a technique to package data with the processes used on that data. A classic smell is a method that seems more interested in a class other than the one it actually is in. The most common focus of the envy is the data. We've lost count of the times we've seen a method that invokes half-a-dozen getting methods on another object to calculate some value. Fortunately the cure is obvious, the method clearly wants to be elsewhere, so you use *Move Method (142)* to get it there. Sometimes only part of the method suffers from envy; in that case use *Extract Method (110)* on the jealous bit and *Move Method (142)* to give it a dream home.

Of course not all cases are cut-and-dried. Often a method uses features of several classes, so which one should it live with? The heuristic we use is to determine which class has most of the data and put the method with that data. This step is often made easier if *Extract Method (110)* is used to break the method into pieces that go into different places.

Of course there are several sophisticated patterns that break this rule. From the Gang of Four [Gang of Four] Strategy and Visitor immediately leap to mind. Kent Beck's Self Delegation [Beck] is another. You use these to combat the divergent change smell. The fundamental rule of thumb is to put things together that change together. Data and the behavior that references that data usually change together, but there are exceptions. When the exceptions occur,

we move the behavior to keep changes in one place. Strategy and Visitor allow you to change behavior easily, because they isolate the small amount of behavior that needs to be overridden, at the cost of further indirection.

Data Clumps

Data items tend to be like children; they enjoy hanging around in groups together. Often you'll see the same three or four data items together in lots of places: fields in a couple of classes, parameters in many method signatures. Bunches of data that hang around together really ought to be made into their own object. The first step is to look for where the clumps appear as fields. Use *Extract Class (149)* on the fields to turn the clumps into an object. Then turn your attention to method signatures using *Introduce Parameter Object (295)* or *Preserve Whole Object (288)* to slim them down. The immediate benefit is that you can shrink a lot of parameter lists and simplify method calling. Don't worry about data clumps that use only some of the fields of the new object. As long as you are replacing two or more fields with the new object, you'll come out ahead.

A good test is to consider deleting one of the data values: if you did this, would the others make any sense? If they don't, it's a sure sign that you have an object that's dying to be born.

Reducing field lists and parameter lists will certainly remove a few bad smells, but once you have the objects, you get the opportunity to make a nice perfume. You can now look for cases of feature envy, which will suggest behavior that can be moved into your new classes. Before long these classes will be productive members of society.

Primitive Obsession

Most programming environments have two kinds of data. Record types allow you to structure data into meaningful groups. Primitive types are your building blocks. Records always carry a certain amount of overhead. They may mean tables in a database, or they may be awkward to create when you want them for only one or two things.

One of the valuable things about objects is that they blur or even break the line between primitive and larger classes. You can easily write little classes that are indistinguishable from the built-in types of the language. Java does have

primitives for numbers, but strings and dates, which are primitives in many other environments, are classes.

People new to objects usually are reluctant to use small objects for small tasks, such as money classes that combine number and currency, ranges with an upper and a lower, and special strings such as telephone numbers and ZIP codes. You can move out of the cave into the centrally heated world of objects by using *Replace Data Value with Object (175)* on individual data values. If the data value is a type code, use *Replace Type Code with Class (218)* if the value does not affect behavior. If you have conditionals that depend on the type code, use *Replace Type Code with Subclasses (223)* or *Replace Type Code with State/Strategy (227)*.

If you have a group of fields that should go together, use *Extract Class (149)*. If you see these primitives in parameter lists, try a civilizing dose of *Introduce Parameter Object (295)*. If you find yourself picking apart an array, use *Replace Array with Object (186)*.

Switch Statements

One of the most obvious symptoms of object-oriented code is its comparative lack of switch (or case) statements. The problem with switch statements is essentially that of duplication. Often you find the same switch statement scattered about a program in different places. If you add a new clause to the switch, you have to find all these switch statements and change them. The object-oriented notion of polymorphism gives you an elegant way to deal with this problem.

Most times you see a switch statement you should consider polymorphism. The issue is where the polymorphism should occur. Often the switch statement switches on a type code. You want the method or class that hosts the type code value. So use *Extract Method (110)* to extract the switch statement and then *Move Method (142)* to get it onto the class where the polymorphism is needed. At that point you have to decide whether to *Replace Type Code with Subclasses (223)* or *Replace Type Code with State/Strategy (227)*. When you have set up the inheritance structure, you can use *Replace Conditional with Polymorphism (255)*.

If you only have a few cases that affect a single method, and you don't expect them to change, then polymorphism is overkill. In this case *Replace Parameter with Explicit Methods (285)* is a good option. If one of your conditional cases is a null, try *Introduce Null Object (260)*.

Parallel Inheritance Hierarchies

Parallel inheritance hierarchies is really a special case of shotgun surgery. In this case, every time you make a subclass of one class, you also have to make a subclass of another. You can recognize this smell because the prefixes of the class names in one hierarchy are the same as the prefixes in another hierarchy.

The general strategy for eliminating the duplication is to make sure that instances of one hierarchy refer to instances of the other. If you use *Move Method (142)* and *Move Field (146)*, the hierarchy on the referring class disappears.

Lazy Class

Each class you create costs money to maintain and understand. A class that isn't doing enough to pay for itself should be eliminated. Often this might be a class that used to pay its way but has been downsized with refactoring. Or it might be a class that was added because of changes that were planned but not made. Either way, you let the class die with dignity. If you have subclasses that aren't doing enough, try to use *Collapse Hierarchy (344)*. Nearly useless components should be subjected to *Inline Class (154)*.

Speculative Generality

Brian Foote suggested this name for a smell to which we are very sensitive. You get it when people say, "Oh, I think we need the ability to do this kind of thing someday" and thus want all sorts of hooks and special cases to handle things that aren't required. The result often is harder to understand and maintain. If all this machinery were being used, it would be worth it. But if it isn't, it isn't. The machinery just gets in the way, so get rid of it.

If you have abstract classes that aren't doing much, use *Collapse Hierarchy (344)*. Unnecessary delegation can be removed with *Inline Class (154)*. Methods with unused parameters should be subject to *Remove Parameter (277)*. Methods named with odd abstract names should be brought down to earth with *Rename Method (273)*.

Speculative generality can be spotted when the only users of a method or class are test cases. If you find such a method or class, delete it and the test case

that exercises it. If you have a method or class that is a helper for a test case that exercises legitimate functionality, you have to leave it in, of course.

Temporary Field

Sometimes you see an object in which an instance variable is set only in certain circumstances. Such code is difficult to understand, because you expect an object to need all of its variables. Trying to understand why a variable is there when it doesn't seem to be used can drive you nuts.

Use *Extract Class (149)* to create a home for the poor orphan variables. Put all the code that concerns the variables into the component. You may also be able to eliminate conditional code by using *Introduce Null Object (260)* to create an alternative component for when the variables aren't valid.

A common case of temporary field occurs when a complicated algorithm needs several variables. Because the implementer didn't want to pass around a huge parameter list (who does?), he put them in fields. But the fields are valid only during the algorithm; in other contexts they are just plain confusing. In this case you can use *Extract Class* with these variables and the methods that require them. The new object is a method object [Beck].

Message Chains

You see message chains when a client asks one object for another object, which the client then asks for yet another object, which the client then asks for yet another another object, and so on. You may see these as a long line of getThis methods, or as a sequence of temps. Navigating this way means the client is coupled to the structure of the navigation. Any change to the intermediate relationships causes the client to have to change.

The move to use here is *Hide Delegate (157)*. You can do this at various points in the chain. In principle you can do this to every object in the chain, but doing this often turns every intermediate object into a middle man. Often a better alternative is to see what the resulting object is used for. See whether you can use *Extract Method (110)* to take a piece of the code that uses it and then *Move Method (142)* to push it down the chain. If several clients of one of the objects in the chain want to navigate the rest of the way, add a method to do that.

Some people consider any method chain to be a terrible thing. We are known for our calm, reasoned moderation. Well, at least in this case we are.

Middle Man

One of the prime features of objects is encapsulation—hiding internal details from the rest of the world. Encapsulation often comes with delegation. You ask a director whether she is free for a meeting; she delegates the message to her diary and gives you an answer. All well and good. There is no need to know whether the director uses a diary, an electronic gizmo, or a secretary to keep track of her appointments.

However, this can go too far. You look at a class's interface and find half the methods are delegating to this other class. After a while it is time to use *Remove Middle Man (160)* and talk to the object that really knows what's going on. If only a few methods aren't doing much, use *Inline Method (117)* to inline them into the caller. If there is additional behavior, you can use *Replace Delegation with Inheritance (355)* to turn the middle man into a subclass of the real object. That allows you to extend behavior without chasing all that delegation.

Inappropriate Intimacy

Sometimes classes become far too intimate and spend too much time delving in each others' private parts. We may not be prudes when it comes to people, but we think our classes should follow strict, puritan rules.

Overintimate classes need to be broken up as lovers were in ancient days. Use *Move Method (142)* and *Move Field (146)* to separate the pieces to reduce the intimacy. See whether you can arrange a *Change Bidirectional Association to Unidirectional (200)*. If the classes do have common interests, use *Extract Class (149)* to put the commonality in a safe place and make honest classes of them. Or use *Hide Delegate (157)* to let another class act as go-between.

Inheritance often can lead to overintimacy. Subclasses are always going to know more about their parents than their parents would like them to know. If it's time to leave home, apply *Replace Inheritance with Delegation (352)*.

Alternative Classes with Different Interfaces

Use *Rename Method (273)* on any methods that do the same thing but have different signatures for what they do. Often this doesn't go far enough. In these cases the classes aren't yet doing enough. Keep using *Move Method (142)* to

move behavior to the classes until the protocols are the same. If you have to redundantly move code to accomplish this, you may be able to use *Extract Superclass (336)* to atone.

Incomplete Library Class

Reuse is often touted as the purpose of objects. We think reuse is overrated (we just use). However, we can't deny that much of our programming skill is based on library classes so that nobody can tell whether we've forgotten our sort algorithms.

Builders of library classes are rarely omniscient. We don't blame them for that; after all, we can rarely figure out a design until we've mostly built it, so library builders have a really tough job. The trouble is that it is often bad form, and usually impossible, to modify a library class to do something you'd like it to do. This means that tried-and-true tactics such as *Move Method (142)* lie useless.

We have a couple of special-purpose tools for this job. If there are just a couple of methods that you wish the library class had, use *Introduce Foreign Method (162)*. If there is a whole load of extra behavior, you need *Introduce Local Extension (164)*.

Data Class

These are classes that have fields, getting and setting methods for the fields, and nothing else. Such classes are dumb data holders and are almost certainly being manipulated in far too much detail by other classes. In early stages these classes may have public fields. If so, you should immediately apply *Encapsulate Field (206)* before anyone notices. If you have collection fields, check to see whether they are properly encapsulated and apply *Encapsulate Collection (208)* if they aren't. Use *Remove Setting Method (300)* on any field that should not be changed.

Look for where these getting and setting methods are used by other classes. Try to use *Move Method (142)* to move behavior into the data class. If you can't move a whole method, use *Extract Method (110)* to create a method that can be moved. After a while you can start using *Hide Method (303)* on the getters and setters.

Data classes are like children. They are okay as a starting point, but to participate as a grownup object, they need to take some responsibility.

Refused Bequest

Subclasses get to inherit the methods and data of their parents. But what if they don't want or need what they are given? They are given all these great gifts and pick just a few to play with.

The traditional story is that this means the hierarchy is wrong. You need to create a new sibling class and use *Push Down Method (328)* and *Push Down Field (329)* to push all the unused methods to the sibling. That way the parent holds only what is common. Often you'll hear advice that all superclasses should be abstract.

You'll guess from our snide use of *traditional* that we aren't going to advise this, at least not all the time. We do subclassing to reuse a bit of behavior all the time, and we find it a perfectly good way of doing business. There is a smell, we can't deny it, but usually it isn't a strong smell. So we say that if the refused bequest is causing confusion and problems, follow the traditional advice. However, don't feel you have to do it all the time. Nine times out of ten this smell is too faint to be worth cleaning.

The smell of refused bequest is much stronger if the subclass is reusing behavior but does not want to support the interface of the superclass. We don't mind refusing implementations, but refusing interface gets us on our high horses. In this case, however, don't fiddle with the hierarchy; you want to gut it by applying *Replace Inheritance with Delegation (352)*.

Comments

Don't worry, we aren't saying that people shouldn't write comments. In our olfactory analogy, comments aren't a bad smell; indeed they are a sweet smell. The reason we mention comments here is that comments often are used as a deodorant. It's surprising how often you look at thickly commented code and notice that the comments are there because the code is bad.

Comments lead us to bad code that has all the rotten whiffs we've discussed in the rest of this chapter. Our first action is to remove the bad smells by refactoring. When we're finished, we often find that the comments are superfluous.

If you need a comment to explain what a block of code does, try *Extract Method (110)*. If the method is already extracted but you still need a comment to explain what it does, use *Rename Method (273)*. If you need to state some rules about the required state of the system, use *Introduce Assertion (267)*.

When you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous.

A good time to use a comment is when you don't know what to do. In addition to describing what is going on, comments can indicate areas in which you aren't sure. A comment is a good place to say *why* you did something. This kind of information helps future modifiers, especially forgetful ones.

Index