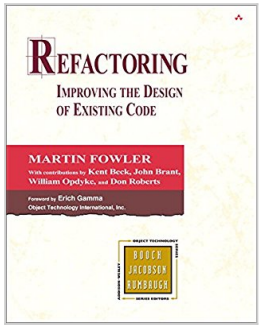

Refactoring: Improving the Design of Existing Code

By Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts

Refactoring: Improving the Design of Existing Code By Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts

As the application of object technology--particularly the Java programming language--has become commonplace, a new problem has emerged to confront the software development community. Significant numbers of poorly designed programs have been created by less-experienced developers, resulting in applications that are inefficient and hard to maintain and extend. Increasingly, software system professionals are discovering just how difficult it is to work with these inherited, non-optimal applications. For several years, expert-level object programmers have employed a growing collection of techniques to improve the structural integrity and performance of such existing software programs. Referred to as refactoring, these practices have remained in the domain of experts because no attempt has been made to transcribe the lore into a form that all developers could use. . .until now. In Refactoring: Improving the Design of Existing Software, renowned object technology mentor Martin Fowler breaks new ground, demystifying these master practices and demonstrating how software practitioners can realize the significant benefits of this new process. With proper training a skilled system designe

Refactoring: Improving the Design of Existing Code

By Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts

Refactoring: Improving the Design of Existing Code By Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts

As the application of object technology--particularly the Java programming language--has become commonplace, a new problem has emerged to confront the software development community. Significant numbers of poorly designed programs have been created by less-experienced developers, resulting in applications that are inefficient and hard to maintain and extend. Increasingly, software system professionals are discovering just how difficult it is to work with these inherited, non-optimal applications. For several years, expert-level object programmers have employed a growing collection of techniques to improve the structural integrity and performance of such existing software programs. Referred to as refactoring, these practices have remained in the domain of experts because no attempt has been made to transcribe the lore into a form that all developers could use. . .until now. In Refactoring: Improving the Design of Existing Software, renowned object technology mentor Martin Fowler breaks new ground, demystifying these master practices and demonstrating how software practitioners can realize the significant benefits of this new process. With proper training a skilled system designe

Refactoring: Improving the Design of Existing Code By Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts Bibliography

- Sales Rank: #12310 in Books
- Published on: 1999-07-08
- Ingredients: Example Ingredients
- Original language: English
- Number of items: 1
- Dimensions: 9.40" h x 1.10" w x 7.40" l, 2.10 pounds
- Binding: Hardcover
- 464 pages

Editorial Review

Amazon.com Review
Your class library works, but could it be better? *Refactoring: Improving the Design of Existing Code* shows how *refactoring* can make object-oriented code simpler and easier to maintain. Today refactoring requires considerable design know-how, but once tools become available, all programmers should be able to improve their code using refactoring techniques.

Besides an introduction to refactoring, this handbook provides a catalog of dozens of tips for improving code. The best thing about *Refactoring* is its remarkably clear presentation, along with excellent nuts-and-bolts advice, from object expert Martin Fowler. The author is also an authority on software patterns and UML, and this experience helps make this a better book, one that should be immediately accessible to any intermediate or advanced object-oriented developer. (Just like patterns, each refactoring tip is presented with a simple name, a "motivation," and examples using Java and UML.)

Early chapters stress the importance of testing in successful refactoring. (When you improve code, you have to test to verify that it still works.) After the discussion on how to detect the "smell" of bad code, readers get to the heart of the book, its catalog of over 70 "refactorings"--tips for better and simpler class design. Each tip is illustrated with "before" and "after" code, along with an explanation. Later chapters provide a quick look at refactoring research.

Like software patterns, refactoring may be an idea whose time has come. This groundbreaking title will surely help bring refactoring to the programming mainstream. With its clear advice on a hot new topic, *Refactoring* is sure to be essential reading for anyone who writes or maintains object-oriented software. -- *Richard Dragan*

Topics Covered: Refactoring, improving software code, redesign, design tips, patterns, unit testing, refactoring research, and tools.

From the Inside Flap

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, Javasoft John Brant, University of Illinois at Urbana-Champaign Scott Corley, High Voltage Software, Inc. Ward Cunningham, Cunningham & Cunningham, Inc. Stephane 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.

From the Back Cover

As the application of object technology--particularly the Java programming language--has become commonplace, a new problem has emerged to confront the software development community. Significant numbers of poorly designed programs have been created by less-experienced developers, resulting in applications that are inefficient and hard to maintain and extend. Increasingly, software system professionals are discovering just how difficult it is to work with these inherited, "non-optimal" applications. For several years, expert-level object programmers have employed a growing collection of techniques to improve the structural integrity and performance of such existing software programs. Referred to as "refactoring," these practices have remained in the domain of experts because no attempt has been made to transcribe the lore into a form that all developers could use. . .until now. In ***Refactoring: Improving the Design of Existing Code,*** renowned object technology mentor **Martin Fowler** breaks new ground, demystifying these master practices and demonstrating how software practitioners can realize the significant benefits of this new process.

With proper training a skilled system designer can take a bad design and rework it into well-designed, robust code. In this book, **Martin Fowler** shows you where opportunities for refactoring typically can be found, and how to go about reworking a bad design into a good one. Each refactoring step is simple--seemingly too simple to be worth doing. Refactoring may involve moving a field from one class to another, or pulling some code out of a method to turn it into its own method, or even pushing some code up or down a hierarchy. While these individual steps may seem elementary, the cumulative effect of such small changes can radically improve the design. Refactoring is a proven way to prevent software decay.

In addition to discussing the various techniques of refactoring, the author provides a detailed catalog of more than seventy proven refactorings with helpful pointers that teach you when to apply them; step-by-step instructions for applying each refactoring; and an example illustrating how the refactoring works. The illustrative examples are written in Java, but the ideas are applicable to any object-oriented programming language.

Users Review

From reader reviews:

William Boehme:

What do you ponder on book? It is just for students as they are still students or this for all people in the world, what the best subject for that? Only you can be answered for that issue above. Every person has various personality and hobby for each and every other. Don't to be forced someone or something that they don't wish do that. You must know how great and important the book Refactoring: Improving the Design of Existing Code. All type of book could you see on many resources. You can look for the internet methods or other social media.

Joseph Singleton:

As people who live in the actual modest era should be change about what going on or data even knowledge to make all of them keep up with the era which can be always change and make progress. Some of you maybe will update themselves by examining books. It is a good choice for yourself but the problems coming to you actually is you don't know what type you should start with. This Refactoring: Improving the Design of Existing Code is our recommendation to help you keep up with the world. Why, as this book serves what you want and need in this era.

Wilda Alexander:

Playing with family in a park, coming to see the water world or hanging out with good friends is thing that usually you might have done when you have spare time, and then why you don't try thing that really opposite from that. One particular activity that make you not experience tired but still relaxing, trilling like on roller coaster you are ride on and with addition info. Even you love Refactoring: Improving the Design of Existing Code, you can enjoy both. It is very good combination right, you still want to miss it? What kind of hang-out type is it? Oh come on its mind hangout men. What? Still don't obtain it, oh come on its named reading friends.

Ashley Gibson:

E-book is one of source of know-how. We can add our expertise from it. Not only for students but additionally native or citizen want book to know the upgrade information of year for you to year. As we know those textbooks have many advantages. Beside many of us add our knowledge, may also bring us to around the world. By book Refactoring: Improving the Design of Existing Code we can consider more advantage. Don't one to be creative people? For being creative person must choose to read a book. Only choose the best book that suitable with your aim. Don't possibly be doubt to change your life with that book Refactoring: Improving the Design of Existing Code. You can more pleasing than now.

Read Refactoring: Improving the Design of Existing Code By Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts for online ebook

Refactoring: Improving the Design of Existing Code By Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts Free PDF d0wnl0ad, audio books, books to read, good books to read, cheap books, good books, online books, books online, book reviews epub, read books online, books to read online, online library, greatbooks to read, PDF best books to read, top books to read Refactoring: Improving the Design of Existing Code By Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts books to read online.

Online Refactoring: Improving the Design of Existing Code By Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts ebook PDF download

Refactoring: Improving the Design of Existing Code By Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts Doc

Refactoring: Improving the Design of Existing Code By Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts Mobipocket

Refactoring: Improving the Design of Existing Code By Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts EPub

IMLY47G2TF6: Refactoring: Improving the Design of Existing Code By Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts