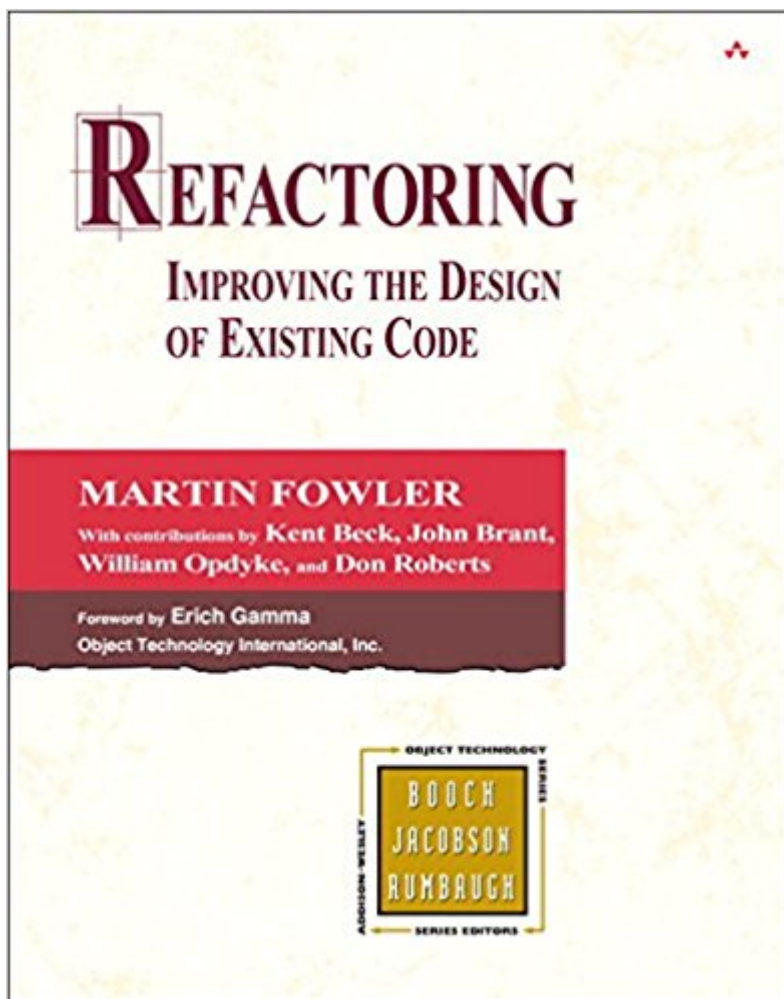The book was found

Refactoring: Improving The Design Of Existing Code

Synopsis

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

Book Information

Hardcover: 464 pages

Publisher: Addison-Wesley Professional; 1 edition (July 8, 1999)

Language: English

ISBN-10: 0201485672

ISBN-13: 978-0201485677

Product Dimensions: 7.4 x 1.2 x 9.2 inches

Shipping Weight: 2 pounds (View shipping rates and policies)

Average Customer Review: 4.4 out of 5 stars   212 customer reviews

Best Sellers Rank: #35,448 in Books (See Top 100 in Books)   #17 inÂ Books > Textbooks > Computer Science > Object-Oriented Software Design   #52 inÂ Books > Computers & Technology > Programming > Software Design, Testing & Engineering > Object-Oriented Design   #196 inÂ Books > Textbooks > Computer Science > Programming Languages

Customer Reviews

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

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 wereKen 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.

This book is a bit old. It is the first, or among the first, which addresses the refactoring issue. However, everything in it is relevant today.At the beginning and at the end you will find articles by various authors (Fowler, Beck, Opdyke, Roberts and Brant):* Refactoring , first example.* Principles of refactoring .* Bad smells in code .* Building Tests.* Toward a Catalog of Refactorings .* Big Refactorings .* Refactoring , Reuse , and Reality .* Refactoring Tools.* Putting It All Together.In the middle will find a great catalog of small transformations that define the steps to do the refactoring. This catalog, though simple is very important as explained in the first chapters.Fowler clearly explains why refactoring, some clues to identify the most important issues (code smells) to refactor in order to improves the design, and the catalog of transformations that are commonly used to solve each code smells.Dependending your experience and knowledge of software craftsmanship, you will surely perceive it more or less as a simple topic, but that makes it no less important.It is a fantastic book. It is one of the books that every software developer in the industry should read. The only reason I'm not giving 5 stars is that it is not a truly revealing book. The items inside are all very basic and simple. You should not expect anything astonishing and the first impression is that all of it is obvious, but it is explained in an exceptional way and the catalog created by Fowler is really great.

This book is replete with advanced knowledge of the situational logic of refactoring code in procedural and object oriented languages ... The point of view is that of a software engineer inclined to perfectionism ... Many typical kinds of bad code from a module or a function or a method or a

class are outlined in Chapter 3 item by item 22 different situations are described and critiqued ... Much of the remainder of the book outlines what remedies the principal author and the other authors suggest as to how to refactor the code to meet this book's standards ...One doubt immediately comes to mind: what about the economics in industry of refactoring? Whilst I agree, say, that a car self parking system needs to be well nigh perfect in that this is the only way to assure safety for car drivers and nearby pedestrians, many an employer will say no just say no to refactoring exercises for perfection's sake! For example, many financial systems have the odd unimportant user interface bug that there's simply not the funds to get fixed.And every source code change involves risks: the risk that bugs are introduced; the risk that the unit and system testing will be inadequate; the risk of feature drift, the application semantics being misunderstood; and, the risk of loss of source code though related hardware failure. Source code control systems and people employed as change managers need to question whether the risk of some minor change should be countenanced, or rejected if too risky!It is possible that too many refactoring exercises might take a large system that works well and lead to it degenerating into an unholy mess, with added introduced bugs, the destruction of the original architect's vision, and changes that erode the system's integrity and purpose. This is a management problem. Quality code is not free. If a large system really needs extentive refactoring the real question is as to whether over the expected lifetime of the code it would be cheaper to refactor, or to throw the system away and start from scratch!It is overall more advisable to get the system right in the first place. A better design with a better architecture if put together for a hypothetical version 1 of a large system will reduce maintenance cost down the track ...Nevertheless an excellent book of practical advice on refactoring and code quality!An advanced reference work for third year university software engineering students, practising software engineers, and software engineering managers. Managers whose background is other than a career in programming would particularly benefit from this work.

Refactoring: Improving the Design of Existing Code is one of those amazing books that every professional developer should have on their book shelf. The bulk of this book is a catalog of refactorings, but there is more to it as I will explain below.In case you aren't aware of what refactoring is, I'll give you Fowlers definition."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." For the most part this means cleaning up your existing - yet working - code. It involves anything from renaming a method to be more concise with the purpose of that method, to breaking up switch statements into a polymorphic structure. There are many different techniques used to

refactor your code, which is what you learn in this book.Right off the bat Fowler throws you into a small sample application that is poorly designed. He then takes you through a few different refactoring techniques that improve the design of this simple application. Right from the start you see how effective refactoring can be. From there he goes into topics such as how to detect "bad smells" in code. This chapter is particularly informative and entertaining. You also learn a little bit about testing. After the introductory chapters you begin to dig into a deep catalog of refactorings. Each one is named. Like design patterns - naming the refactoring and building a vocabulary really helps in communicating thoughts and ideas.The catalog of refactorings is extremely useful. They are structured so that each refactoring has a name, a motivation, the mechanics and a simple example. This is very effective. As I said earlier, the name is useful because it helps build your programming vocabulary and it helps in communicating thoughts and ideas. The motivation explains why the refactoring should be done and when it should/shouldn't be used. The mechanics provide a step-by-step description of how to carry out the refactoring and the example shows a small example of the refactoring in use. All examples are written in Java 1.1.Although the examples are written in Java the book is still very good for any developer. Developers that have never written a line of code in Java, C++, C#, or anything similar may have a little bit of a tougher time working through this book. Luckily most examples are very small and simple so even if you fall into this category you shouldn't have too much of a learning curve. Some of the code is a bit outdated and can be done a bit better now-a-days but what do you expect? This book was written 8+ years ago! Times have changed. The ideas are still very relevant though, which is what makes this book so timeless.Martin Fowler books are always a joy to read. His writing style is humorous, yet often very blunt and to the point. Just like UML Distilled, he is able to communicate a lot of ideas into a very short amount of space - the book is a bit dense in other words, which is very good in my opinion. Martin Fowler does not beat around the bush and he has very strong opinions on certain topics. Unlike a lot of books you read, he actually writes with personality. I have a hard time putting his books down. Here is an example of the type of verbiage he uses...On how comments can be a "bad smell":"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." - Martin Fowler. Here he is talking about how people use comments to hide bad code, or "bad smells".I highly recommend this book. If you are a professional developer or plan on becoming one then click the "Buy Now" button without second thought. This is one of those rare books worth its weight in gold - I would spend $100.00 on a book like this if I had to.

This is a great read to improve legacy code and to follow for new projects. It is for work related use not recreational reading.

Refactoring: Improving the Design of Existing Code 2012 International Existing Building Code (International Code Council Series) 2015 International Existing Building Code Commentary (International Building Code Commentary) AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis Refactoring to Patterns 2015 International Existing Building Code 2012 International Plumbing Code (Includes International Private Sewage Disposal Code) (International Code Council Series) Building Code Basics: Commercial; Based on the International Building Code (International Code Council Series) Improving Inter-professional Collaborations: Multi-Agency Working for Children's Wellbeing (Improving Learning) Graphic Design Success: Over 100 Tips for Beginners in Graphic Design: Graphic Design Basics for Beginners, Save Time and Jump Start Your Success (graphic ... graphic design beginner, design skills) Abundance by Design: Discover Your Unique Code for Health, Wealth and Happiness with Human Design (Life by Human Design Book 1) How to Code in 10 Easy Lessons: Learn how to design and code your very own computer game (Super Skills) Simplifying Innovation: Doubling Speed to Market and New Product Profits with Your Existing Resources: Guided Innovation Get the Deed!: Take over any property "Subject To" the Existing Financing No Cash No Credit No Banks The Onion Book of Known Knowledge: A Definitive Encyclopaedia of Existing Information Building Screened Rooms: Creating Backyard Retreats, Screening in Existing Structures, A Complete How-to Guide Residential Energy: Cost Savings and Comfort for Existing Buildings (6th Edition) All About Dog Daycare... A Blueprint for Success: For New and Existing Dog Daycare Owners The Truth of Revelation, Demonstrated by an Appeal to Existing Monuments, Sculptures, Gems, Coins, and Medals Seismic Evaluation and Retrofit of Existing Buildings: ASCE/SEI 41-13 (Standard) (Asce Standard)