Addison-Wesley Professional Ruby Series

REFACTORING

RUBY EDITION

JAY FIELDS ▪ SHANE HARVIE ▪ MARTIN FOWLER
with KENT BECK

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data:

Fields, Jay, 1979-

Refactoring / Jay Fields, Shane Harvie, and Martin Fowler. -- Ruby ed.

p. cm.

ISBN-13: 978-0-321-60350-0 (hardback : alk. paper)

ISBN-10: 0-321-60350-8 (hardback : alk. paper) 1. Software refactoring. 2. Ruby (Computer program

language) I. Harvie, Shane, 1980- II. Fowler, Martin, 1963- III. Title.

QA76.76.R42F54 2010

005.1'17--dc22

2009027577

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671 3447

ISBN-13: 978-0-321-60350-0

ISBN-10: 0-321-60350-8

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.

First printing October 2009

Associate Publisher
Mark Taub

Acquisitions Editor
Greg Doench

Managing Editor
Kristy Hart

Project Editor
Andy Beaster

Copy Editor
Geneil Breeze

Indexer
Erika Millen

Proofreader
Jennifer Gallant

Technical Reviewers
Chad Fowler
Clinton Begin
Justin Gehtland

Publishing Coordinator
Michelle Housley

Cover Designer
Chuti Prasertsith

Compositor
Jake McFarland

Foreword

I remember what it was like to learn object-oriented (OO) programming; As I learned OO, I was left with a low-grade tension—a feeling that I was missing something. Some new concepts felt simple and familiar in a way that told you there was a depth underlying them waiting to be discovered. That can be an unsettling feeling.

I read the literature on design patterns with great interest but, disappointingly, derived little enlightenment. I talked to other developers, browsed the Web, read books, and perused source code but remained convinced that there was something important that wasn't coming through. I understood how the tools of object orientation worked, but I was unable to apply them in a way that *felt* right to me.

Then I picked up the first edition of this book.

Software is not created in one inspired moment. The usual focus on the *artifacts* of the development process obscures the fact that software development is in fact a process. More specifically, as *Refactoring* taught me, it is a series of small decisions and actions all made through the filter of a set of values and the desire to create something excellent.

Understanding that software development is a constant activity and not a static event helps us to remember that code can and should be organic. Good code is easy to change. Bad code can incrementally be made easier to change. Code that's easy to change is fun to work with. Code that's hard to change is stressful to work with. And the more changes you make, without refactoring it, the more stressful working with it becomes.

So becoming a software developer is less about what good code is than about how to *make* good code. Software doesn't just spring into being. It's created by humans, one keystroke at a time. Refactoring is the book from which I learned how to do that process well. It taught me how to sit down and write great code, one tiny piece at a time.

When I initially read *Refactoring*, I was on a small team whose responsibility was to help larger groups write better software. At meetings and code reviews, I would carry the hard-covered book around with me, wielding it as both a weapon and a shield. I was passionate about my job and (more strongly) the craft of software development, and I'm sure that the developers we worked with often dreaded the sight of me and this book heading toward their cubicles. I didn't so much refer to the book's contents in these meetings as just have it with me as a reminder of what it represented for me: Our work can be great if we always remember that it should be great and we take the simple steps to *make* it great.

Looking back on that time with the advantage of hindsight, I realize that the languages and tools we were using were working against us. The techniques in this book were born out of Smalltalk development. In a dynamic environment, refactoring flourishes. So it's only fitting that they should be reborn here in Ruby. As a longtime Rubyist it is thrilling to see the book that made such a profound difference for me become available to developers who speak Ruby as their primary programming language.

Refactoring: Ruby Edition will serve as a guiding light for a new generation of Rubyists who will learn to create better, more flexible software and (I hope) to love the craft of software development as much as I have.

—Chad Fowler
Co-Director, Ruby Central, Inc.
CTO, InfoEther, Inc.

Preface

Just over a decade ago I (Martin) worked on a project with Kent Beck. This project, called C3, became rather known as the project that marked the birth of extreme programming and helped fuel the visibility of what we now know as the agile software movement.

We learned a lot of things on that project, but one thing that particularly struck me was Kent's methodical way of continually reworking and improving the design of the system. I had always been a fan of writing clear code, and felt it was worthwhile to spend time cleaning up problematic code to allow a team to develop features swiftly. Kent introduced me to a technique, used by a number of leading Smalltalkers, that did this far more effectively than I had done it before. It's a technique they called *refactoring*, and soon I wanted to talk about it wherever I went. However, there was no book or similar resource I could point people to so that they could learn about this technique themselves. Kent and the other Smalltalkers weren't inclined to write one, so I took on the project.

My *Refactoring* book was popular and appears to have played a significant role in making refactoring a mainstream technique. With the growth of Ruby in the past few years, it made sense to put together a Ruby version of the book, this is where Jay and Shane stepped in.

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.

Many people find the phrase *improving the design after it has been written* rather odd. For many years most people believed that design comes first, and the coding comes second. Over time the code gets 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 an instance variable 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 Ruby programmer. Our aim is to show you how to do refactoring in a controlled and efficient manner. You 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 we 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 we cover more of the general principles of refactoring, some definitions, and the reasons for doing refactoring. We outline some of the problems with refactoring. In Chapter 3 Kent Beck helps us describe how to find bad smells in code and how to clean them up with refactorings. Testing plays an important role in refactoring, so Chapter 4 describes how to build tests into code with a simple 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 we have written down so far in our work in this field. When we want to do something, such as Replace Conditional with Polymorphism, the catalog reminds us how to do it in a safe, step-by-step manner. We hope this is the section of the book you come back to often.

Refactoring in Ruby

When I wrote the original *Refactoring* book, I used Java to illustrate the techniques, mainly because Java was a widely read language. Most of the refactoring techniques apply whatever the language, so many people have used the original book to help them in their refactoring outside Java.

But obviously it helps you to learn refactoring in the language that you mostly program in. With many people learning the Ruby language, and with refactoring being a core part of the Ruby culture, we felt it was particularly important to provide a way for Rubyists to learn about refactoring—particularly if they don't have a background in curly-brace languages.

So Jay and Shane took on the task of going through my original book, and reworking it for Ruby. They started with the original text and meticulously went through it to remove all the Javaisms and rework the text to make sense in a Ruby context. They are experienced Ruby programmers who also have a good background in Java and C#, so they have the right background to do this well.

They also added some new refactorings that are particular to Ruby. Truth be told most of the refactorings are the same as those you need in any other object-oriented language, but there are a few new ones that come into play.

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.

Although it is focused on the code, refactoring has a large impact on the design of a 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.

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.

We wrote this book assuming you haven't come across refactoring before and haven't read the original book, so you can treat this as a fully blown introduction to the subject. You start with either this book or the original, depending on which language you prefer as your focus.

I Have the Original Book—Should I Get This?

Probably not. If you're familiar with the original book you won't find a lot of new material here. You'll need to adjust the original refactorings to the Ruby language, but if you're like us you shouldn't find that an inordinate challenge.

There are a couple of reasons where we think an owner of the original book might consider getting a copy of the Ruby edition. The first reason is if you're not too familiar with Java and found the original book hard to follow because of that unfamiliarity. If so we hope you find a Ruby-focused book easier to work with. The second reason is if you're leading a Ruby team that has people who would struggle with the original book's Java focus. In that case a Ruby book would be a better tool to help pass on your understanding of refactoring.

Building on the Foundations Laid by Others

Occasionally people referred to me (Martin) as something like, "The Father of Refactoring." I always cringe when they do this because, although my book has helped to popularize refactoring, it certainly isn't my creation. In particular I built my work on the foundations laid by some leading people in the Smalltalk community

Two of the leading developers of refactoring were Ward Cunningham and Kent Beck. They used it as a central part of their development process in the early days and 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 long series of 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 developed the world's first automated refactoring tool: the Smalltalk Refactoring Browser.

Many people have developed ideas in refactoring since my book. In particular, tool development has exploded. Any serious IDE now needs a "refactoring" menu, and many people now treat refactoring as an essential part of their development tools. It's important to point out that you can refactor effectively without a tool—but it sure makes it easier!

Making the Ruby Edition

People often wonder about how a book gets made, particularly when there's several people involved.

Martin began the original *Refactoring* book in early 1997. He did it by making notes of refactorings he did while programming, so these notes could remind him how to do certain refactorings efficiently. (These turned into the mechanics section of the book.) The book was published in 1999 and has sold steadily—around 15,000 copies a year.

Jay approached Martin in 2006 about doing a Ruby version. Jay looked around for people to help, and Shane was soon contributing enough to be a full author. Martin hasn't done much on this edition as his writing attention has been on other projects, but we left his name on the cover since he essentially provided the first draft, much of which is still there.

Chapter 3

Bad Smells in Code

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 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, and 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 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 in both classes and then Pull Up Method. If the code is similar but not the same, you need to use Extract Method to separate the similar bits from the different bits. You may then find you can use Form Template Method. If the methods do the same thing with a different algorithm, you can choose the clearer of the two algorithms and use Substitute Algorithm. If the duplication is in the middle of the method, use Extract Surrounding Method.

If you have duplicated code in two unrelated classes, consider using Extract Class or Extract Module 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 the section "Indirection and Refactoring" in Chapter 2, "Principles in Refactoring.")

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 Object Oriented 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. Find parts of the method that seem to go nicely together and make a new method.

If you have a method with many 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 or Replace Temp with Chain to eliminate the temps. Long lists of parameters can be slimmed down with Introduce Parameter Object and Preserve Whole Object.

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.

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 to deal with conditional expressions. Replace loops with Collection

Closure Methods and consider using Extract Method on the call to the closure method and the closure itself.

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 to bundle a number of the variables. Choose variables to go together in the component that makes sense for each. For example, `deposit_amount` and `deposit_currency` 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 often is easier. Another option if the component doesn't make sense as a delegate is Extract Module.

Sometimes a class does not use all of its instance variables all of the time. If so, you may be able to Extract Class, Extract Module, or Extract Subclass 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 a lot of duplicate code, 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, Extract Module, or Extract Subclass. A useful trick is to determine how clients use the class and to use Extract Module for each of these uses. That may give you ideas on how you can further break up the class.

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 when you can get the data in one parameter by making a request of an object you already know about. This object might be an instance variable or it might be another parameter. Use Preserve Whole Object 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 to clump them together, or Introduce Named Parameter to improve the fluency.

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 or module, and all the typing in the new class/module should express the variation. To clean this up you identify everything that changes for a particular cause and use Extract Class 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 and Move Field 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 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 to get it there. Sometimes only part of the method suffers from envy; in that case use Extract Method on the jealous bit and Move Method 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 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 pattern from his Smalltalk Best Practices book [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 many places: instance variables in a couple of classes, and 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 instance variables. Use Extract Class on the instance variables to turn the clumps into an object. Then turn your attention to method signatures using Introduce Parameter Object or Preserve Whole Object 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 attributes of the new object. As long as you are replacing two or more instance variables 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 instance variable 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 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. Ruby makes everything an object, but for the sake of this discussion, we're designating built-in types such as Fixnum and String as primitives.

People new to objects are usually reluctant to use small objects for small tasks, such as money classes that combine number and currency, 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 on individual data values. If you have conditionals that depend on a type code, use Replace Type Code with Polymorphism, Replace Type Code with Module Extension, or Replace Type Code with State/Strategy.

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

Case Statements

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

Most times when you see a case statement you should consider polymorphism. The issue is where the polymorphism should occur. Often the case statement matches on a type code. You want the method or class that hosts the type code value. So use Extract Method to extract the case statement and then Move Method to get it onto the class where the polymorphism is needed. At that point you have to decide whether to Replace Type Code with Polymorphism, Replace Type Code with Module Extension, or Replace Type Code with State/Strategy.

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 is a good option. If one of your conditional cases is a null, try Introduce Null Object.

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 and Move Field, 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 or modules that aren't doing enough, try to use Collapse Hierarchy. Nearly useless components should be subjected to Inline Class or Inline Module.

Speculative Generality

Speculative generality is 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 classes or modules that aren't doing much, use Collapse Hierarchy. Unnecessary delegation can be removed with Inline Class. Methods with unused parameters should be subject to Remove Parameter. Methods named with odd names should be brought down to earth with Rename Method.

Speculative generality can be spotted when the only users of a method, a code branch, or an entire class are test cases. If you find this type of code, 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 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 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 instance variables. But the instance variables 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 object, and so on. You may see these as a long line of get_this 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. In principle you can apply Hide Delegate to potentially 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 to take a piece of the code that uses it and then Move Method 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 and talk to the object that really knows what's going on. If only a few methods aren't doing much, use Inline Method to inline them into the caller. If there is additional behavior, you can use Replace Delegation with Hierarchy to turn the real object into a module and include it in the middle man. 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 into each other's private parts. We may not be prudes when it comes to people, but we think our classes should follow strict, puritan rules.

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

Inheritance often can lead to over-intimacy. 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.

Alternative Classes with Different Interfaces

Use Rename Method 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 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 Module or Introduce Inheritance 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.

In other languages extending an existing library class can be impossible or messy. However, Ruby's open classes make this easy to fix using Move Method to move the behavior needed directly to the library class.

Data Class

These are classes that have attributes, and nothing else. Such classes are dumb data holders and are almost certainly being manipulated in far too much detail by other classes. Use Remove Setting Method on any instance variable that should not be changed. If you have collection instance variables, check to see whether they are properly encapsulated and apply Encapsulate Collection if they aren't.

Look for where these getting and setting methods are used by other classes. Try to use Move Method to move behavior into the data class. If you can't move a whole method, use Extract Method to create a method that can be moved. After a while you can start using Hide Method 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 to push all the unused methods to the sibling. That way the parent holds only what is common.

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 public methods of the superclass. We don't mind refusing implementations, but refusing public methods 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.

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. If the method is already extracted but you still need a comment to explain what it does, use Rename Method. If you need to state some rules about the required state of the system, use Introduce Assertion.

Tip 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.

Metaprogramming Madness

While in most cases Ruby's dynamic nature provides great benefits, it can be misused. Some metaprogramming techniques can result in obfuscated code. The `method_missing` hook, for example, often results in code that is difficult to understand. It can be a powerful tool if an object's interface cannot be determined at coding time, but unless it's absolutely necessary I use Replace Dynamic Receptor with Dynamic Method Definition or even a simple Extract Method to remove the `method_missing` definition. If the `method_missing` definition is truly needed, I might use Isolate Dynamic Receptor to separate concerns.

Disjointed API

Libraries are often written with flexibility as the number one priority. The author needs to build in this flexibility so that her library can be used by many different people in many different ways. This flexibility often presents itself as a relatively fine-grained, disjointed API, with many configuration options.

More often than not, an individual project will not take advantage of all the configuration options. The same configuration options will be used over and over. If this is the case, use Introduce Gateway to interact with the API in a simplified way.

Introduce Expression Builder can be applied to both internal and external APIs to interact with the public interface in a more fluent manner.

Repetitive Boilerplate

One of the easiest ways to remove duplication is Extract Method. Extract the method and call it from multiple places. Some kinds of methods become so commonplace that we can go even further. Take for example `attr_reader` in Ruby. Implementing attribute readers is so common in object-oriented languages that the author of Ruby decided to provide a succinct way to declare them. Introduce Class Annotation involves annotating a class by calling a class method from the class definition in the same way that `attr_reader` is called. Most code isn't simple enough to declare in this way, but when the purpose of the code can be captured clearly in a declarative statement, Introduce Class Annotation can clarify the intention of your code.

Index