Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters in all capitals.

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.
QA76.76.R42F54 2010
005.1’17--dc22
2009027577

Copyright © 2010 Pearson Education, Inc.
All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

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

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
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 all 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 Delegation 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

Symbols
||= operator, 257

A
Account class, 129
  Introduce Parameter Object refactoring, 322-324
  Move Field refactoring, 174-175
  Move Method refactoring, 170-172
  Remove Setting Method refactoring, 325-326
  Replace Error Code with Exception refactoring, 334-335
AccountNumberCapture module, 360-362
ActiveDeal class, 403
add_charge method, 322-323
add_course method, 223
add_customer method, 213
add_front_suspension method, 247
add_option method, 117
add_order method, 213

Add Parameter
  overview, 300
  step-by-step description, 301-302
  when to use, 300-301
add_rear_suspension method, 247
adjusted_capital method, 278-279
advantages of refactoring
  easier-to-understand software, 55-56
  faster programming, 56-57
  improved software design, 54-55
  why refactoring works, 60-61
algorithms, substituting
  goals, 132
  overview, 131-132
  step-by-step description, 132
alternative classes with difference interfaces, 83
Ambler, Scott, 65
a_method method, 125
amount calculation (video store program), moving, 12-18
amount_for method, moving, 12-18
APIs, disjointed, 86
apply method, 388
assert_equal method, 89
assertions, adding
  example, 294-295
  goals, 293
  overview, 292
  step-by-step description, 293-294
AssertValidKeys module, 146-147
attributes, 255-259

B
base_charge method, 309
base_price method, 111
Beck, Kent, 51, 54, 56, 69, 73
behavior, moving into classes, 223-224
benefits of refactoring
  easier-to-understand software, 55-56
  faster programming, 56-57
  improved software design, 54-55
  why refactoring works, 60-61
Bid class, Extract Module
  refactoring, 360-361
bidirectional association, changing
to unidirectional
  example, 215-217
  goals, 214
  overview, 213
bidirectional association, changing unidirectional association to
  example, 211-213
  goals, 210-211
  overview, 210
  step-by-step description, 211
Billing Scheme class, 414-416
Books class, 145-146
bugs
  finding by refactoring, 56
  refactoring when fixing bugs, 58

C
calculate_outstanding method, 107
case statement, 80
  replacing with polymorphism
    (video store program example), 32-49
chains, replacing temps with
  example, 115-117
  goals, 115
  overview, 114
  step-by-step description, 115
Change Bidirectional Association to
Unidirectional refactoring, 83
  example, 215-217
  overview, 213
  when to use, 214
Change Reference to
Value refactoring
  example, 199-201
  overview, 198
  step-by-step description, 199
  when to use, 198-199
Change Unidirectional Association
to Bidirectional refactoring
  example, 211-213
  overview, 210
  step-by-step description, 211
  when to use, 210-211
Check Security method, 271-273, 305

Chrysler Comprehensive
Compensation case study, 69-72
class annotations, adding
elements, 141-142
goals, 140
overview, 139-140
step-by-step description, 140-141

classes. See also specific classes
alternative classes with
difference interfaces, 83
data classes, 84
delegate classes
calling directly with Remove
Middle Man, 185-186
hiding with Hide Delegate,
181-184
extracting. See extracting
inappropriate intimacy, 83
incomplete library classes, 84
large classes, eliminating, 76
lazy classes, 81
merging hierarchy, 371-372
merging modules into, 362
moving behavior into, 223-224
moving into another class with
Inline Class
elements, 180-181
overview, 179
step-by-step
description, 180
when to use, 179
order, 128
replacing records with, 224
replacing with modules
elements, 393-395
overview, 392

Charge class, 322
charge method, 33-34, 45, 135
step-by-step
description, 393
when to use, 392
special case classes, 292
Code Complete: A Practical Handbook of Software Construction (McConnel), 71
code reviews, refactoring with, 58-59
code smells
alternative classes with difference interfaces, 83
case statements, 80
comments, 85
data classes, 84
data clumps, 79
disjointed APIs, 86
divergent change, 77
duplicated code, 74
feature envy, 78-79
inappropriate intimacy, 83
incomplete library classes, 84
large classes, 76
lazy classes, 81
long methods, 74-76
long parameter lists, 76-77
message chains, 82
metaprogramming, 86
middle man, 83
overview, 73-74
parallel inheritance
hierarchies, 81
primitives, 79-80
refused bequests, 84-85
repetitive boilerplate, 86
shotgun surgery, 78
speculative generality, 81
temporary fields, 82
Collapse Hierarchy, 81
overview, 371
step-by-step description, 372
when to use, 371
collection closure methods, replacing loops with
example, 133-135
goals, 133
step-by-step description, 133
collections, encapsulating
example, 220-223
goals, 219
overview, 219
step-by-step description, 219-220
CommandCenter class, 163
comments, 85
communication, telling managers about refactoring, 61
Company class
Expression Builder, adding, 348-352
Introduce Gateway refactoring, 343-346
compute method, 130-131
Concurrent Programming in Java (Lea), 297
conditional expressions
assertions
example, 294-295
goals, 293
overview, 292
step-by-step description, 293-294
consolidating
examples, 266-267
goals, 265-266
overview, 265
step-by-step
description, 266
consolidating duplicate conditional fragments
  example, 269
  goals, 268
  overview, 268
  step-by-step description, 268-269
control flags, removing
  examples, 271-274
  goals, 269-270
  overview, 269
  step-by-step description, 270-271
decomposing
  example, 263
  goals, 262
  overview, 261
  step-by-step description, 262
nested conditionals, replacing with guard clauses
  examples, 276-279
  goals, 275-276
  overview, 274-275
  step-by-step description, 276
null objects, adding
  examples, 288-291
  goals, 285-287
  overview, 284
  special cases, 292
  step-by-step description, 287-288
recomposing
  examples, 264-265
  goals, 264
  overview, 264
replacing with polymorphism
  example, 282-284
  goals, 280-281
overview, 279
step-by-step description, 281
conditional logic, removing, 225-226
Consolidate Conditional Expression refactoring
  examples, 266-267
  overview, 265
  step-by-step description, 266
  when to use, 265-266
Consolidate Duplicate Conditional Fragments refactoring
  example, 269
  overview, 268
  step-by-step description, 268-269
  when to use, 268
consolidating
  conditional expressions
    examples, 266-267
    goals, 265-266
    overview, 265
    step-by-step description, 266
duplicate conditional fragments
  example, 269
  goals, 268
  overview, 268
  step-by-step description, 268-269
constants, replacing magic numbers with
  goals, 218
  overview, 217
  step-by-step description, 218
constructors, replacing with factory methods
  example, 330-332
  goals, 329
Databases, problems with refactoring, 64-65
Data classes, 84
Data clumps, eliminating, 79
Data organization
- Changing bidirectional association to unidirectional
  - Example, 215-217
  - Goals, 214
  - Overview, 213
  - Step-by-step description, 214-215
- Changing reference objects into value objects
  - Example, 199-201
  - Goals, 198-199
  - Overview, 198
  - Step-by-step description, 199
- Changing unidirectional association to bidirectional
  - Example, 211-213
  - Goals, 210-211
  - Overview, 210
  - Step-by-step description, 211
- Changing value objects into reference objects
  - Example, 196-198
  - Goals, 195
  - Overview, 194
  - Step-by-step description, 195
Eagerly initialized attributes, 257-259
Encapsulating collections
- Example, 220-223
- Goals, 219
Index

overview, 219
step-by-step description, 219-220

lazily initialized attributes, 255-257

moving behavior into classes, 223-224
overview, 187-188

replacing arrays with objects
example, 202-206
goals, 201
overview, 201
step-by-step
description, 202

replacing data values with objects
example, 192-194
goals, 191
overview, 191
step-by-step
description, 192

replacing hashes with objects
example, 207-209
goals, 206
overview, 206
step-by-step
description, 207

replacing magic numbers with symbolic constants
goals, 218
overview, 217
step-by-step
description, 218

replacing records with data classes, 224

replacing subclasses with fields
example, 253-255
goals, 252
overview, 251

step-by-step description, 252-253

replacing type code with polymorphism
example, 227-232
goals, 225
overview, 225
removing conditional logic, 225-226
step-by-step
description, 233

replacing type code with state/strategy
example, 240-251
goals, 239
overview, 239
step-by-step
description, 239-240

self-encapsulating fields
example, 189-191
goals, 188-189
overview, 188
step-by-step
description, 189

data values, replacing with objects
example, 192-194
goals, 191
overview, 191
step-by-step description, 192

@days_overdrawn instance
variable, 170
Deal class, Tease Apart Inheritance refactoring, 401-404
Decompose Conditional refactoring
element, 263
overview, 261
step-by-step description, 262
when to use, 262
decomposing conditional expressions
element, 263
goals, 262
overview, 261
step-by-step description, 262
Decorator class, 159
def_each method, 154-155
definition of refactoring, 52-54
delegate classes
calling directly with Remove Middle Man
element, 186
overview, 185
step-by-step description, 185
when to use, 185
hiding with Hide Delegate
element, 183-184
overview, 181
step-by-step description, 183
when to use, 182-183
deliaction
replacing inheritance with
element, 387-389
goals of, 386
overview, 386
step-by-step description, 386-387
replacing with hierarchy
element, 390-392
goals, 390
overview, 389
step-by-step description, 390
deprecation, refactoring
with, 205-206
design
design changes that are difficult to refactor, 65-66
improving with refactoring, 54-55
relationship with refactoring, 67-68
developer tests, 91-92
development of refactoring, 51-52
disability_amount method, 265-266
discount_factor method, 114
discount method, 124-125
disjointed APIs, 86
distance_traveled method, 123
divergent change, 77
domain logic, separating from view
element, 408-412
goal of, 406-407
overview, 406
step-by-step description, 407-408
duplicated code, 74
duplicate methods, eliminating with
Extract Surrounding Method
element, 137-139
overview, 135-136
step-by-step description, 136-137
when to use, 136
Dynamic Method
Definition refactoring
elements, 153-157
overview, 152
step-by-step description, 153
when to use, 153
dynamic method definitions
- examples, 153-157
- goals, 153
- overview, 152
- replacing dynamic receptors with
  - examples, 158-160
  - goals, 158
  - overview, 158
  - step-by-step description, 158
- step-by-step description, 153

dynamic receptors
- isolating, 160
  - example, 162-165
  - goals, 161
  - step-by-step description, 162
- replacing with dynamic method definitions, 158-160

E

Eagerly Initialized Attribute
- refactoring, 257-259

Employee class
- assertions, 294-295
  Replace Delegation with Hierarchy refactoring, 390-391

Encapsulate Collection
- refactoring, 84
  - example, 220-223
  - overview, 219
  - step-by-step description, 219-220
  - when to use, 219

encapsulating collections
- example, 220-223
- goals, 219

overview, 219
step-by-step description, 219-220
eql? method, 200
derror codes, replacing
  with exceptions
    - examples, 334-337
    - goals, 333
    - overview, 332
    - step-by-step description, 333-334
eval, moving from runtime to parse time, 165-166

exceptions
- replacing error codes with
  - examples, 334-337
  - goals, 333
  - overview, 332
  - step-by-step description, 333-334
- replacing with tests
  - example, 338-341
  - goals, 338
  - overview, 337
  - step-by-step description, 338

expense_limit method, 292

explaining variables, adding
- examples, 119-121
- goals, 118
- overview, 117-118
- step-by-step description, 119

Expression Builders, adding
- example, 348-352
- goals, 347
- overview, 346
- step-by-step description, 347

expressions. See conditional expressions
Extract Class refactoring
methods with no local variables, 104-105
overview, 102
reassigning local variables, 106-108
step-by-step description, 103
modules
example, 360-362
goals, 358-359
overview, 357-358
step-by-step description, 359
subclasses
example, 364-367
goals, 363
overview, 363
step-by-step description, 363-364
surrounding methods
example, 137-139
overview, 135-136
step-by-step description, 136-137
when to use, 136

Extract Hierarchy refactoring
example, 414-416
overview, 412
step-by-step description, 413-414
when to use, 413

Extract Method refactoring
eliminating duplicated code, 74
methods with local variables, 105-106
methods with no local variables, 104-105
overview, 102
reassigning local variables, 106-108
step-by-step description, 103
video store program
example, 10
when to use, 102

extracting
classes
changing data clumps into objects, 79
eliminating large classes, 76
example, 177-179
organizing orphan variables, 82
overview, 175
step-by-step description, 176-177
when to use, 175-176

frequent renter points (video store program), 18-21
hierarchy of classes
example, 414-416
goals, 413
overview, 412
step-by-step description, 413-414

methods
goals, 102
methods with local variables, 105-106
Index

Extract Module refactoring
example, 360-362
overview, 357-358
step-by-step description, 359
when to use, 358-359
Extract Subclass refactoring
example, 364-367
overview, 363
step-by-step description, 363-364
when to use, 363
Extract Surrounding
Method refactoring
eliminating duplicated code, 74
example, 137-139
overview, 135-136
step-by-step description, 136-137
when to use, 136

*eXtreme Programming eXplained*
(Beck), 51

F

failure method, 152-153
feature envy, 78-79
fields
encapsulating fields
example, 189-191
goals, 188-189
overview, 188
step-by-step description, 189
moving with Move Field
dexample, 174-175
overview, 172
step-by-step description, 173
when to use, 173

replacing subclasses with
goals, 252
overview, 251
step-by-step description, 252-255
temporary fields, 82
File class, writing tests for, 89-91
FileTest class, 89
finding references, 99
Foo class, 205
format of refactorings, 97-98
Form Template Method refactoring
eliminating duplicated code, 74
overview, 372
step-by-step description, 372
template method with
extension of modules
(example), 380-385
template method with
inheritance (example), 374-379
when to use, 373
found_friends method, 131-132
found_miscreant method, 273-274, 305-306
found_person method, 305
Fowler, Martin, 69
frequent_renter_points method, 47
frequent_renter_points temporary variable, 19, 22
frequent renter points (video store program), extracting, 18-21
FrontSuspensionMountainBike class, 228-231, 236-237, 245, 369-371
FullSuspensionMountainBike class, 228-231, 237, 284
Index

G
Gamma class, 130
Gateways, adding
example, 342-346
goals, 341-342
overview, 341
step-by-step description, 342
GemStone, 71
generalization refactorings
  Collapse Hierarchy
    overview, 371
    step-by-step
description, 372
    when to use, 371
  Extract Module
    example, 360-362
    overview, 357-358
    step-by-step
description, 359
    when to use, 358-359
  Extract Subclass
    example, 364-367
    overview, 363
    step-by-step description,
    363-364
    when to use, 363
Form Template Method
  overview, 372
  step-by-step
description, 374
  template method with
extension of modules
(example), 380-385
  template method with
inheritance (example),
374-379
  when to use, 373
Inline Module, 362
Introduce Inheritance
  example, 369-371
  overview, 368
  step-by-step
description, 369
  when to use, 368
Pull Down Method
  overview, 356
  step-by-step
description, 357
  when to use, 357
Pull Up Method
  example, 355-356
  overview, 353
  step-by-step
description, 355
  when to use, 354
Replace Abstract Superclass
  with Module
    example, 393-395
    overview, 392
    step-by-step
description, 393
    when to use, 392
Replace Delegation
  with Hierarchy
    example, 390-392
    overview, 389
    step-by-step
description, 390
    when to use, 390
Replace Inheritance
  with Delegation
    example, 387-389
    overview, 386
    step-by-step description,
    386-387
    when to use, 386
  goals, setting, 418
H

Hash class, 157
Haungs, Jim, 71
HeatingPlan class, 315-317
Hide Delegate refactoring, 82
  example, 183-184
  overview, 181
  step-by-step description, 183
  when to use, 182-183
Hide Method refactoring
  overview, 327
  step-by-step description, 328
  when to use, 327
hiding
delegates
  example, 183-184
  overview, 181
  step-by-step description, 183
  when to use, 182-183
methods
  goals, 327
  overview, 327
  step-by-step description, 328
hierarchy
  hierarchy of classes, creating
    example, 414-416
    goals, 413
    overview, 412
    step-by-step description, 413-414
  replacing delegation with
    example, 390-392
    goals, 390
    overview, 389
    step-by-step description, 390
history of refactoring, 51-52

HtmlStatement class, 376-378, 381-382
html_statement method, 30, 375

I

immutable objects, 199
importance of big refactorings, 398
ImportedItem class, 190
inappropriate intimacy, 83
incomplete library classes, 84
indirection and refactoring, 61-63
inheritance, 374-379
  adding to code
    example, 369-371
    goals, 368
    overview, 368
    step-by-step description, 369
  refactoring
    examples, 401-404
    goal of, 400
    overview, 399
    step-by-step description, 400-401
parallel inheritance
  hierarchies, 81
  replacing with delegation
    example, 387-389
    goals of, 386
    overview, 386
    step-by-step description, 386-387
video store program
  example, 36-49
initialize_courses method, 222
initialize method, 130, 190, 254
initializing attributes, 255-259
Index

**Inline Class refactoring, 81**
example, 180-181
organizing changes
into single class, 78
overview, 179
removing unnecessary
debation, 81
step-by-step description, 180
when to use, 179

**Inline Method refactoring**
overview, 108-109
step-by-step description, 109-110
when to use, 109

**inline methods**
goals, 109
overview, 108-109
step-by-step description, 109-110
when to use, 109

**Inline Module refactoring, 81, 362**

**Inline Temp refactoring, 110**
inline temps, 110

**instance_variable_defined?**
method, 257

@interest_rate field, moving, 174

**interfaces, changing, 63-64**

**Introduce Assertion refactoring, 85**
example, 294-295
overview, 292
step-by-step description, 293-294
when to use, 293

**Introduce Class Annotation**
refactoring, 86
examples, 141-142
overview, 139-140
step-by-step description, 140-141
when to use, 140

**Introduce Explaining**
Variable refactoring
eamples, 119-121
overview, 117-118
step-by-step description, 119
when to use, 118

**Introduce Expression Builder**
refactoring, 86
example, 348-352
overview, 346
step-by-step description, 347
when to use, 347

**Introduce Gateway refactoring, 86**
example, 342-346
overview, 341
step-by-step description, 342
when to use, 341-342

**Introduce Inheritance refactoring**
example, 369-371
overview, 368
step-by-step description, 369
when to use, 368

**Introduce Named**
Parameter refactoring
eamples, 143-147
overview, 142
shortening long parameter
lists, 77
step-by-step description, 143
when to use, 142-143

**Introduce Null Object**
refactoring, 82
examples, 288-291
overview, 284
special cases, 292
step-by-step description, 287-288
when to use, 285-287
libraries, incomplete library classes, 84
local variables
  example, 105-106
  reassigning, 106-108
long methods, shortening, 74-76
long parameter lists, shortening, 76-77
loops, replacing with collection closure methods, 133-135

M
magic numbers, replacing with symbolic constants
  goals, 218
  overview, 217
  step-by-step description, 218
managers, telling about refactoring, 61
measuring performance, Chrysler Comprehensive Compensation case study, 69-72
merging
  class hierarchy, 371-372
  modules into including class, 362
message chains, 82
MessageCollector class, 164
metaprogramming, 86
method_missing method, 156, 160
methods
  add_charge, 322-323
  add_course, 223
  add_customer, 213
  add_front_suspension, 247
  add_option, 117
  add_order, 213
  add_rear_suspension, 247

J-K
Jeffries, Ron, 69, 285
JobItem class, extracting LaborItem subclass from, 364-367
Johnson, Ralph, 51
Join class, 394
joins_for_table method, 394
Knuth, Donald, 166

L
LaborItem class, extracting from JobItem class, 364-367
Laptop class, 346-352
large classes, eliminating, 76
large refactorings, tips for, 420
Lazily Initialized Attributes, 255-257
lazy classes, 81
Lea, Doug, 297
learning refactoring, 418-419
Ledger class, 126
adjusted_capital, 278-279
def_each, 154-155
defining dynamically
    examples, 153-157
goals, 153
overview, 152
replacing dynamic
    receptors with dynamic
method definition,
    158-160
step-by-step
description, 153
disability_amount, 265-266
discount, 124-125
discount_factor, 114
distance_traveled, 123
duplicate methods, eliminating,
    135-139
duplicate methods, eliminating,
    135-139
eql?, 200
disability_amount, 265-266
discount, 124-125
discount_factor, 114
distance_traveled, 123
duplicate methods, eliminating,
    135-139
eql?, 200
expense_limit, 292
disability_amount, 265-266
discount, 124-125
discount_factor, 114
distance_traveled, 123
duplicate methods, eliminating,
    135-139
eql?, 200
Expression Builders, adding
    example, 348-352
goals, 347
overview, 346
step-by-step
description, 347
extracting
    goals, 102
methods with local
    variables, 105-106
methods with no local
    variables, 104-105
overview, 102
reassigning local variables,
    106-108
step-by-step
description, 103
failure, 152-153
found_friends, 131-132
a_method, 125
amount_for, 12-18
apply, 388
assert_equal, 89
base_charge, 309
base_price, 111
calculate_outstanding, 107
charge, 33-34, 45, 135
check_security, 271-273, 305
collection closure methods,
    replacing loops with
        example, 133-135
goals, 133
step-by-step
description, 133
collection closure methods,
    replacing loops with
        example, 133-135
goals, 133
step-by-step
description, 133
compute, 130-131
constructors, replacing with
    factory methods, 328-332
count_descendants_matching, 138-139
create_bill, 355, 415
creating
    goals, 373
overview, 372
step-by-step
description, 374
template method with
    extension of modules
        (example), 380-385
template method with
    inheritance (example), 374-379
found_miscreant, 273-274, 305-306
found_person, 305
frequent_renter_points, 47
Gateways, adding
every, 342-346
goals, 341-342
overview, 341
step-by-step description, 342
hiding
goals, 327
overview, 327
step-by-step description, 328
html_statement, 30, 375
initialize, 130, 190, 254
initialize_courses, 222
inline methods
goals, 109
overview, 108-109
step-by-step description, 109-110
instance_variable_defined?, 257
isolating dynamic receptors, 160
every, 162-165
goals, 161
step-by-step description, 162
joins_for_table, 394
long methods, shortening, 74-76
method_missing, 156, 160
MountainBike, 250
moving to subclasses
goals of, 357
overview, 356
step-by-step description, 357
moving up to superclass
every, 355-356
goals, 354
overview, 353
step-by-step description, 355
moving with Move Method
every, 170-172
overview, 167
step-by-step description,
168-170
when to use, 168
not_summer, 263
number_of_descendants_named,
138-139
number_of_living_descendants,
138-139
office_telephone_number, 300
off_road_ability, 228, 244
overdraft_charge, 170-171
parameters
adding, 300-302	named parameters, 142-147	named parameters, removing, 147-150
parameterized methods,
creating, 307-310
parameter objects, creating,
320-324
removing, 302-303
removing assignments to,
124-127
replacing with explicit
methods, 310-313
replacing with methods,
317-320
unused default parameters,
removing, 150-152
pay_amount, 275-277
price, 112-121, 228, 247,
319-320
price_code, 48
print_owing, 104-108
product_count_items, 150-151
remove_customer, 213
remove_order, 213
removing
  example, 325-327
  goals, 325
  overview, 324
  step-by-step
description, 325
renaming, 298
  example, 299-300
  goals, 298-299
  step-by-step
description, 299
replacing with method objects
  example, 129-131
  goals, 128
  overview, 127
  step-by-step
description, 129
resource, 339-341
reward_points, 264
RigidMountainBike, 250
send_alert_if_miscreant_in, 306
separating query from modifier
  concurrency issues, 307
  example, 305-307
  goals, 304
  overview, 303
  step-by-step
description, 304-305
statement, 374, 380
  code listing after
  refactoring, 9
  code listing before
  refactoring, 7-8
  extracting frequent renter
  points, 18-21

initial code listing, 3-4
moving amount
calculation, 12-18
removing temporary
variables, 22-31
renaming variables, 10-11
substituting algorithms
goals, 132
overview, 131-132
step-by-step
description, 132
summer_charge, 263
telephone_number,
renaming, 299
temporary variables
  explaining variables,
  117-121
  inline temps, 110
  replacing with chains,
  114-117
  replacing with queries,
  111-114
  splitting, 121-124
total_amount_for_order_lines,
  411
total_charge, 23-25
triple, 126
usage_in_range, 309
value, 379, 383
winter_charge, 263
withdraw, 336
middle man classes, removing,
  83, 185-186
MissingCustomer class, 289
Module class, 205
module extensions, replacing type
code with
  example, 234-238
  goals, 233
overview, 232
step-by-step description, 233
modules
AccountNumberCapture, 360-362
AssertValidKeys, 146-147
CustomInitializers, 141-144
extracting
  example, 360-362
  goals, 358-359
  overview, 357-358
  step-by-step
description, 359
merging into including class, 362
Person, 391
replacing superclasses with
  example, 393-395
  overview, 392
  step-by-step
description, 393
when to use, 392
MonthlyStatement class, 385
MountainBike class, 227-229, 234-236, 240-248, 280-283, 369
MountainBike method, 250
Move Eval from Runtime to Parse Time refactoring, 165-166
Move Field refactoring
  example, 174-175
  organizing changes into single class, 78
  overview, 172
  reducing inappropriate intimacy, 83
  step-by-step description, 173
  when to use, 173
Move Method refactoring, 13
  example, 170-172
moving behaviors into data classes, 84
organizing changes into single class, 78
overview, 167
reducing inappropriate intimacy, 83
step-by-step description, 168-170
when to use, 168
Movie class, 2
moving amount calculation (video store program), 12-18
behavior into classes, 223-224
fields. See Move Field refactoring
methods. See Move Method refactoring
methods to subclasses
goals of, 357
overview, 356
step-by-step
description, 357
methods up to superclass
example, 355-356
goals, 354
overview, 353
step-by-step
description, 355
N
named parameters
adding
  examples, 143-147
  goals, 142-143
  overview, 142
  step-by-step
description, 143
removing

example, 148-150
goals, 148
overview, 147
step-by-step
description, 148

special cases, 292

step-by-step description, 287-288

parameter objects, creating, 320-324

preserving whole objects

example, 315-317
goals, 313-314
overview, 313
step-by-step description, 314-315

reference objects

changing into value objects, 198-201

changing value objects into, 194-198

replacing arrays with

example, 202-205
goals, 201
overview, 201
step-by-step
description, 202

replacing data values with

example, 192-194
goals, 191
overview, 191
step-by-step
description, 192

replacing hashes with

example, 207-209
goals, 206
overview, 206
step-by-step
description, 207

replacing with arrays, 206

value objects

changing reference objects into, 198-201
changing to reference objects, 194-198

nature of refactoring, 397-398

nested conditionals, replacing with guard clauses

examples, 276-279
goals, 275-276
overview, 274-275
step-by-step description, 276

NetworkResult class, 208-210

not_summer method, 263

NullCustomer class, 290-291

null objects, adding

examples, 288-291
goals, 285-288
overview, 284
special cases, 292

number_of_descendants_named method, 138-139

number_of_living_descendants method, 138-139

O

objects

converting procedural design to

example, 406
goal of, 405
overview, 405
step-by-step
description, 406

method objects, replacing

methods with, 127-131

null objects, adding

examples, 288-291
goals, 285-287
overview, 284

value objects
office_telephone_number method, 300
off_road_ability method, 228, 244
Opdyke, Bill, 52
Order class, 128, 192-193, 196, 212-216, 411
OrdersController class, 409-410
organizing data. See data organization
orphan variables, organizing, 82
overdraft_charge method, 170-171

P
parallel inheritance hierarchies, 81
parameterized methods, creating
  example, 308-310
  goals, 308
  overview, 307
  step-by-step description, 308
Parameterize Method refactoring
  example, 308-310
  overview, 307
  step-by-step description, 308
  when to use, 308
parameter lists, shortening, 76-77
parameters
  adding to methods
    goals, 300-301
    overview, 300
    step-by-step description, 301-302
  named parameters
    adding, 142-147
    removing, 147-150
parameter objects, creating, 320-324
removing assignments to
  example, 125-127
  goals, 124-125

  step-by-step
description, 125
removing from methods
  goals, 302
  overview, 302
  step-by-step description, 302-303
replacing with explicit methods
  example, 311-313
  goals, 310-311
  overview, 310
  step-by-step
description, 311
replacing with methods
  example, 318-320
  goals, 318
  overview, 317
  step-by-step
description, 318
unused default parameters, removing
  example, 151-152
  goals, 151
  overview, 150
  step-by-step
description, 151
partners, value of, 419
pay_amount method, 275-277
performance
effect of refactoring on, 70-71
measuring, Chrysler
  Comprehensive Compensation case study, 69-72
Performance class, 203-204
Person class, 165, 220-222, 253-254
  Expression Builder, adding, 348-352
  Extract Class refactoring, 177-179
Index

Hide Delegate refactoring, 183-184
Remove Middle Man refactoring, 186
Person module, 391
Policy class, 387-389
polymorphism
  replacing conditional logic with
    example, 282-284
goals, 280-281
overview, 279
step-by-step
description, 281
video store program
example, 32-49
replacing type code with
example, 227-232
goals, 225
overview, 225
removing conditional logic,
225-226
step-by-step description,
226-227
PostData class, 156
The Pragmatic Programmer
(Thomas), 87
Preserve Whole Object refactoring
example, 315-317
overview, 313
reducing parameter lists, 79
shortening long parameter lists, 77
step-by-step description,
314-315
when to use, 313-314
price_code method, 48
price method, 112-121, 228, 247,
319-320
primitives, 79-80
print_owing method, 104-108
problems with refactoring
  databases, 64-65
design changes, 65-66
interface changes, 63-64
when not to refactor, 66-67
procedural design, converting to
objects
example, 406
goal of, 405
overview, 405
step-by-step description, 406
Product class, 127
ProductController class, 330-332
product_count_items method,
150-151
programs. See video store program
Pull Down Method refactoring
overview, 356
step-by-step description, 357
when to use, 357
Pull Up Method refactoring
example, 355-356
overview, 353
step-by-step description, 355
when to use, 354
Push Down Method refactoring, 85
Q
QA (quality assurance) tests, 91-92
queries
  replacing temps with
    example, 112-114
goals, 111-112
    step-by-step
description, 112
separating from modifiers
concurrency issues, 307
example, 305-307
goals, 304
overview, 303
step-by-step description, 304-305

R
reassigning local variables, 106-108
Recompose Conditional refactoring
examples, 264-265
overview, 264
when to use, 264
recomposing conditional expressions
examples, 264-265
goals, 264
overview, 264
Recorder class, 162-164
records, replacing with data classes, 224
Red/Green/Refactor movement, 87-88
Refactoring Databases (Sadalage and Ambler), 65
refactorings. See specific refactorings
refactoring tips
learning refactoring, 418-419
overview, 417-418
tips for large refactorings, 420
working with a partner, 419
reference objects
changing into value objects
example, 199-201
goals, 198-199
overview, 198
step-by-step description, 199
changing value objects into
example, 196-198
goals, 195

overview, 194
step-by-step description, 195
references, finding, 99
refused bequests, 84-85
Remove Assignments to Parameters refactoring
example, 125-127
overview, 124
step-by-step description, 125
when to use, 124-125
Remove Control Flag refactoring
examples, 271-274
overview, 269
step-by-step description, 270-271
when to use, 269-270
remove_customer method, 213
Remove Middle Man refactoring, 83
example, 186
overview, 185
step-by-step description, 185
when to use, 185
Remove Named Parameter refactoring
example, 148-150
overview, 147
step-by-step description, 148
when to use, 148
remove_order method, 213
Remove Parameter refactoring
overview, 302
step-by-step description, 302-303
when to use, 302
Remove Setting Method refactoring, 84
example, 325-327
overview, 324
step-by-step description, 325
when to use, 325
Remove Unused Default Parameter refactoring
example, 151-152
overview, 150
step-by-step description, 151
when to use, 151
removing
assignments to parameters
example, 125-127
goals, 124-125
step-by-step description, 125
code smells. See code smells
control flags
examples, 271-274
goals, 269-270
overview, 269
step-by-step description, 270-271
middle man, 83
named parameters
example, 148-150
goals, 148
overview, 147
step-by-step description, 148
repetitive boilerplate, 86
setting methods
example, 325-327
goals, 325
overview, 324
step-by-step description, 325
temporary variables, 22-31
unused default parameters
example, 151-152
goals, 151
overview, 150
step-by-step description, 151
Rename Method refactoring
example, 299-300
step-by-step description, 299
when to use, 298-299
renaming
methods
example, 299-300
goals, 298-299
step-by-step description, 299
variables, 10-11
Rental class, 2-3
repetitive boilerplate, removing, 86
Replace Abstract Superclass with
Module refactoring
example, 393-395
overview, 392
step-by-step description, 393
when to use, 392
Replace Array with
Object refactoring, 80
example, 202-206
overview, 201
step-by-step description, 202
when to use, 201
Replace Conditional with
Polymorphism refactoring
example, 282-284
overview, 279
step-by-step description, 281
when to use, 280-281
Replace Constructor with Factory
Method refactoring
example, 330-332
overview, 328-329
<table>
<thead>
<tr>
<th>Refactoring</th>
<th>Example</th>
<th>Overview</th>
<th>Step-by-Step Description</th>
<th>When to Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replace Inheritance with Delegation</td>
<td>387-389</td>
<td>386</td>
<td>386-387</td>
<td>386</td>
</tr>
<tr>
<td>Replace Loop with Collection</td>
<td>133-135</td>
<td>133</td>
<td>133</td>
<td>133</td>
</tr>
<tr>
<td>Replace Magic Number with Symbolic Constant</td>
<td>158-160</td>
<td>158</td>
<td>158</td>
<td>158</td>
</tr>
<tr>
<td>Replace Dynamic Receptor with Dynamic Method Definition</td>
<td>129-131</td>
<td>127</td>
<td>129</td>
<td>128</td>
</tr>
<tr>
<td>Replace Nested Conditional with Guard Clauses</td>
<td>276-279</td>
<td>274-275</td>
<td>276</td>
<td>275-276</td>
</tr>
<tr>
<td>Replace Exception with Test</td>
<td>338-341</td>
<td>337</td>
<td>338</td>
<td>338</td>
</tr>
<tr>
<td>Replace Hash with Object</td>
<td>207-209</td>
<td>206</td>
<td>207</td>
<td>206</td>
</tr>
<tr>
<td>Replace Data Value with Object</td>
<td>192-194</td>
<td>191</td>
<td>192</td>
<td>191</td>
</tr>
<tr>
<td>Replace Delegation with Hierarchy</td>
<td>390-392</td>
<td>389</td>
<td>390</td>
<td>390</td>
</tr>
<tr>
<td>Replace Error Code with Exception</td>
<td>334-337</td>
<td>332</td>
<td>333-334</td>
<td>333</td>
</tr>
<tr>
<td>Replace Exception with Explicit Methods</td>
<td>311-313</td>
<td>310</td>
<td>311</td>
<td>310-311</td>
</tr>
<tr>
<td>Replace Hash with Object</td>
<td>207-209</td>
<td>206</td>
<td>207</td>
<td>206</td>
</tr>
<tr>
<td>Replace Delegation with Hierarchy</td>
<td>387-389</td>
<td>386</td>
<td>386-387</td>
<td>386</td>
</tr>
<tr>
<td>Replace Loop with Collection</td>
<td>133-135</td>
<td>133</td>
<td>133</td>
<td>133</td>
</tr>
<tr>
<td>Replace Magic Number with Symbolic Constant</td>
<td>158-160</td>
<td>158</td>
<td>158</td>
<td>158</td>
</tr>
<tr>
<td>Replace Dynamic Receptor with Dynamic Method Definition</td>
<td>129-131</td>
<td>127</td>
<td>129</td>
<td>128</td>
</tr>
<tr>
<td>Replace Nested Conditional with Guard Clauses</td>
<td>276-279</td>
<td>274-275</td>
<td>276</td>
<td>275-276</td>
</tr>
<tr>
<td>Replace Exception with Test</td>
<td>338-341</td>
<td>337</td>
<td>338</td>
<td>338</td>
</tr>
<tr>
<td>Replace Hash with Object</td>
<td>207-209</td>
<td>206</td>
<td>207</td>
<td>206</td>
</tr>
</tbody>
</table>
Replace Parameter with Method refactoring
  example, 318-320
  overview, 317
  shortening long parameter lists, 77
  step-by-step description, 318
  when to use, 318
Replace Record with Data Class refactoring, 224
Replace Subclass with Fields refactoring
  example, 253-255
  overview, 251
  step-by-step description, 252-253
  when to use, 252
Replace Temp with Chain refactoring
  example, 115-117
  overview, 114
  when to use, 115
Replace Temp with Query refactoring
  example, 112-114
  overview, 111
  step-by-step description, 112
  when to use, 111-112
Replace Type Code with Module Extension refactoring, 80
  example, 234-238
  overview, 232
  step-by-step description, 233
  when to use, 233
Replace Type Code with Polymorphism refactoring, 80
  example, 227-232
  overview, 225
  removing conditional logic, 225-226
  step-by-step description, 226-227
  when to use, 225
Replace Type Code with State/Strategy refactoring, 80
  example, 240-251
  overview, 239
  step-by-step description, 239-240
  video game program example, 38-45
  when to use, 239
replacing
  algorithms
    goals, 132
    overview, 131-132
    step-by-step description, 132
  arrays with objects
    example, 202-206
    goals, 201
    overview, 201
    step-by-step description, 202
  conditional logic
    with polymorphism
      example, 282-284
      goals, 280-281
      overview, 279
      step-by-step description, 281
      video store program example, 32-49
    constructors with
      factory methods
      example, 330-332
goals, 329
overview, 328-329
step-by-step
description, 329
data values with objects
eexample, 192-194
goals, 191
overview, 191
step-by-step
description, 192
delegation with hierarchy
eexample, 390-392
goals, 390
overview, 389
step-by-step
description, 192
dynamic receptors with dynamic method definitions
eexample, 158-160
goals, 158
overview, 158
step-by-step
description, 158
hashes with objects
eexample, 207-209
goals, 206
overview, 206
step-by-step
description, 207
inheritance with delegation
eexample, 387-389
goals of, 386
overview, 386
step-by-step
description, 386-387
loops with collection
closure methods
eexample, 133-135
goals, 133
step-by-step
description, 133
magic numbers with symbolic constants
goals, 218
overview, 217
step-by-step
description, 218
methods with method objects
eexample, 129-131
goals, 128
overview, 127
step-by-step
description, 129
nested conditionals with guard clauses
eexample, 276-279
goals, 275-276
overview, 274-275
step-by-step
description, 276
parameters with methods
eexample, 318-320
goals, 318
overview, 317
step-by-step
description, 318
records with data classes, 224
subclasses with fields
eexample, 253-255
goals, 252
overview, 251
step-by-step
description, 252-253
temps with chains
eexample, 115-117
goals, 115
overview, 114
step-by-step
description, 115
temps with queries
  example, 112-114
goals, 111-112
step-by-step
description, 112
type code with
module extensions
  example, 234-238
goals, 233
overview, 232
step-by-step
description, 233
type code with polymorphism
  example, 227-232
goals, 225
overview, 225
removing conditional logic,
  225-226
step-by-step description,
  226-227
type code with state/strategy
  example, 240-251
goals, 239
overview, 239
step-by-step description,
  239-240
resource method, 339-341
ResourceStack class, 338-339
reward_points method, 264
RigidMountainBike class, 228-231,
  243-244, 247, 282-283
RigidMountainBike method, 250
Roberts, Don, 51, 57
Room class, 315
Rule of Three, 57

S
Sadalage, Pramod, 65
SearchCriteria class, 141-144

Select class, 115-117
Self-Delegation pattern, 78
Self Encapsulate Field
  refactoring, 174
    example, 189-191
    overview, 188
    step-by-step description, 189
    when to use, 188-189
self-testing code, 87-88
send_alert_if_miscreant_in
  method, 306
Separate Domain from Presentation
  refactoring
    example, 408-412
    overview, 406
    step-by-step description,
      407-408
    when to use, 406-407
Separate Query from
  Modifier refactoring
    concurrency issues, 307
    example, 305-307
    overview, 303
    step-by-step description,
      304-305
    when to use, 304
separating domain logic from view
  example, 408-412
  goal of, 406-407
  overview, 406
  step-by-step description,
    407-408
shortening
  methods, 74-76
  parameter lists, 76-77
shotgun surgery, 78
Smalltalk, 51
smells. See code smells
software design, improving with
  refactoring, 54-55
special case class, 292
speculative generality, 81
Split Temporary
Variable refactoring
element, 122-124
everview, 121-122
step-by-step description, 122
when to use, 122
splitting temporary variables
element, 122-124
goals, 122
neverview, 121-122
step-by-step description, 122
Statement class, 379
statement method, 374, 380
code listing after refactoring, 9
code listing before
refactoring, 7-8
extracting frequent renter
points, 18-21
initial code listing, 3-4
moving amount calculation,
12-18
removing temporary variables,
22-31
renaming variables, 10-11
state/strategy, replacing type
code with
element, 240-251
goals, 239
neverview, 239
step-by-step description,
239-240
subclasses
extracting
element, 364-367
goals, 363
neverview, 363
step-by-step description,
363-364
moving methods into
goals of, 357
neverview, 356
step-by-step
description, 357
replacing with fields
element, 253-255
goals, 252
neverview, 251
step-by-step description,
252-253
Substitute Algorithm refactoring
neverview, 131-132
step-by-step description, 132
when to use, 132
substituting algorithms
goals, 132
neverview, 131-132
step-by-step description, 132
summer_charge method, 263
superclasses, replacing with modules
element, 393-395
neverview, 392
step-by-step description, 393
when to use, 392
symbolic constants, replacing magic
numbers with
goals, 218
neverview, 217
step-by-step description, 218
T
tangled inheritance, refactoring
elements, 401-404
goal of, 400
neverview, 399
step-by-step description,
400-401
Tease Apart Inheritance refactoring
  examples, 401-404
  overview, 399
  step-by-step description, 400-401
  when to use, 400
TelephoneNumber class
  defining, 177-178
  Inline Class refactoring
    example, 180-181
telephone_number method, renaming, 299
telling managers about refactoring, 61
template methods, creating
  goals, 373
  overview, 372
  step-by-step description, 374
  template method with extension of modules
    (example), 380-385
  template method with inheritance (example), 374-379
temporary fields, 82
temporary variables
  explaining variables, adding
    examples, 119-121
    goals, 118
    overview, 117-118
    step-by-step
      description, 119
  frequent_renter_points, 19, 22
  inline temps, 110
  removing, 22-23, 26-31
  replacing with chains
    example, 115-117
    goals, 115
    overview, 114
    step-by-step
      description, 115
replacing with queries
  example, 112-114
  goals, 111-112
  overview, 111
  step-by-step
    description, 112
splitting
  example, 122-124
  goals, 122
  overview, 121-122
  step-by-step
    description, 122
total_amount, 22-23
total_amount_for_order_lines method, 411
Test::Unit testing framework, 88-91
testing
  developer tests, 91-92
  importance of, 9-10
  QA (quality assurance) tests, 91-92
  self-testing code, 87-88
  Test::Unit testing framework, 88-91
  video store program, 6-7
  writing tests, 92-95
tests, replacing exceptions with
  example, 338-341
  goals, 338
  overview, 337
  step-by-step description, 338
TextStatement class, 376-377, 381
Thomas, Dave, 87
tips for refactoring
  learning refactoring, 418-419
  overview, 417-418
  tips for large refactorings, 420
  working with a partner, 419
  total_amount_for_order_lines
  method, 411
Index

goals, 214
overview, 213
changing to bidirectional
description, 211
example, 211-213
goals, 210-211
overview, 210
step-by-step
description, 211
Unified Modeling Language (UML)
diagrams, 20-21
unused default
parameters, removing
description, 151
example, 151-152
goals, 151
overview, 150
step-by-step description, 151
usage_in_range method, 309

V

value method, 379, 383
value objects
changing reference objects into,
removing conditional logic,
state/strategy
example, 240-251
goals, 239
overview, 239
step-by-step description, 239-240

U

UML (Unified Modeling Language)
diagrams, 20-21
unidirectional association
changing bidirectional to
description, 215-217
inline temps, 110
removing, 22-31
replacing with chains, 114-117
replacing with queries, 111-114
splitting, 121-124
total_amount, 22-23

video store program
charge method, 33-34, 45
Customer class, 3
design issues, 5-6
frequent_renter_points method, 47
html_statement method, 30
inheritance, 36-49
Movie class, 2
overview, 1-2
price_code method, 48
Rental class, 3
replacing conditional logic with polymorphism, 32-49
statement method
code listing after refactoring, 9
code listing before refactoring, 7-8
extracting frequent renter points, 18-21
initial code listing, 3-4
moving amount calculation, 12-18
removing temporary variables, 22-31
renaming variables, 10-11
testing, 6-7
total_charge method, 23-25
Unified Modeling Language (UML) diagrams, 20-21
views, separating domain logic from example, 408-412
goal of, 406-407
overview, 406
step-by-step description, 407-408

W-X-Y-Z
when not to refactor, 66-67
when to refactor
for greater understanding, 59-60
overview, 57
Rule of Three, 57
when adding function, 57-58
when fixing bugs, 58
with code reviews, 58-59
why refactoring works, 60-61
winter_charge method, 263
withdraw method, 336
writing tests, 92-95