PRENTICE HALL

Robert C. Martin Series

Clean Code

A Handbook of Agile Software Craftsmanship

Foreword by James O. Coplien

Robert C. Martin

Clean Code

A Handbook of Agile Software Craftsmanship

The Object Mentors:

Robert C. Martin

Michael C. Feathers   Timothy R. Ottinger
Jeffrey J. Langr   Brett L. Schuchert
James W. Grenning   Kevin Dean Wampler
Object Mentor Inc.

Writing clean code is what you must do in order to call yourself a professional. There is no reasonable excuse for doing anything less than your best.

PRENTICE HALL

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Visit us on the Web: informit.com/ph

Library of Congress Cataloging-in-Publication Data

Martin, Robert C.
Clean code : a handbook of agile software craftsmanship / Robert C. Martin.
p. cm.
Includes bibliographical references and index.
ISBN 0-13-235088-2 (pbk. : alk. paper)
1. Agile software development. 2. Computer software—Reliability. I. Title.
QA76.76.D47M3652 2008
005.1—dc22 2008024750

ISBN-13: 978-0-13-235088-4
ISBN-10: 0-13-235088-2
Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
Fourteenth printing January 2015

Contents

This page intentionally left blank

Foreword

One of our favorite candies here in Denmark is Ga-Jol, whose strong licorice vapors are a perfect complement to our damp and often chilly weather. Part of the charm of Ga-Jol to us Danes is the wise or witty sayings printed on the flap of every box top. I bought a two-pack of the delicacy this morning and found that it bore this old Danish saw:

Ærlighed i små ting er ikke nogen lille ting.

"Honesty in small things is not a small thing." It was a good omen consistent with what I already wanted to say here. Small things matter. This is a book about humble concerns whose value is nonetheless far from small.

God is in the details, said the architect Ludwig mies van der Rohe. This quote recalls contemporary arguments about the role of architecture in software development, and particularly in the Agile world. Bob and I occasionally find ourselves passionately engaged in this dialogue. And yes, mies van der Rohe was attentive to utility and to the timeless forms of building that underlie great architecture. On the other hand, he also personally selected every doorknob for every house he designed. Why? Because small things matter.

In our ongoing "debate" on TDD, Bob and I have discovered that we agree that software architecture has an important place in development, though we likely have different visions of exactly what that means. Such quibbles are relatively unimportant, however, because we can accept for granted that responsible professionals give *some* time to thinking and planning at the outset of a project. The late-1990s notions of design driven *only* by the tests and the code are long gone. Yet attentiveness to detail is an even more critical foundation of professionalism than is any grand vision. First, it is through practice in the small that professionals gain proficiency and trust for practice in the large. Second, the smallest bit of sloppy construction, of the door that does not close tightly or the slightly crooked tile on the floor, or even the messy desk, completely dispels the charm of the larger whole. That is what clean code is about.

Still, architecture is just one metaphor for software development, and in particular for that part of software that delivers the initial *product* in the same sense that an architect delivers a pristine building. In these days of Scrum and Agile, the focus is on quickly bringing *product* to market. We want the factory running at top speed to produce software. These are human factories: thinking, feeling coders who are working from a product backlog or user story to create *product*. The manufacturing metaphor looms ever strong in such thinking. The production aspects of Japanese auto manufacturing, of an assembly-line world, inspire much of Scrum.

Yet even in the auto industry, the bulk of the work lies not in manufacturing but in maintenance—or its avoidance. In software, 80% or more of what we do is quaintly called "maintenance": the act of repair. Rather than embracing the typical Western focus on *producing* good software, we should be thinking more like home repairmen in the building industry, or auto mechanics in the automotive field. What does Japanese management have to say about *that*?

In about 1951, a quality approach called Total Productive Maintenance (TPM) came on the Japanese scene. Its focus is on maintenance rather than on production. One of the major pillars of TPM is the set of so-called 5S principles. 5S is a set of disciplines—and here I use the term "discipline" instructively. These 5S principles are in fact at the foundations of Lean—another buzzword on the Western scene, and an increasingly prominent buzzword in software circles. These principles are not an option. As Uncle Bob relates in his front matter, good software practice requires such discipline: focus, presence of mind, and thinking. It is not always just about doing, about pushing the factory equipment to produce at the optimal velocity. The 5S philosophy comprises these concepts:

- *Seiri*, or organization (think "sort" in English). Knowing where things are—using approaches such as suitable naming—is crucial. You think naming identifiers isn't important? Read on in the following chapters.
- *Seiton*, or tidiness (think "systematize" in English). There is an old American saying: *A place for everything, and everything in its place*. A piece of code should be where you expect to find it—and, if not, you should re-factor to get it there.
- *Seiso*, or cleaning (think "shine" in English): Keep the workplace free of hanging wires, grease, scraps, and waste. What do the authors here say about littering your code with comments and commented-out code lines that capture history or wishes for the future? Get rid of them.
- *Seiketsu*, or standardization: The group agrees about how to keep the workplace clean. Do you think this book says anything about having a consistent coding style and set of practices within the group? Where do those standards come from? Read on.
- *Shutsuke*, or discipline (*self*-discipline). This means having the discipline to follow the practices and to frequently reflect on one's work and be willing to change.

If you take up the challenge—yes, the challenge—of reading and applying this book, you'll come to understand and appreciate the last point. Here, we are finally driving to the roots of responsible professionalism in a profession that should be concerned with the life cycle of a product. As we maintain automobiles and other machines under TPM, breakdown maintenance—waiting for bugs to surface—is the exception. Instead, we go up a level: inspect the machines every day and fix wearing parts before they break, or do the equivalent of the proverbial 10,000-mile oil change to forestall wear and tear. In code, refactor mercilessly. You can improve yet one level further, as the TPM movement innovated over 50 years ago: build machines that are more maintainable in the first place. Making your code readable is as important as making it executable. The ultimate practice, introduced in TPM circles around 1960, is to focus on introducing entire new machines or replacing old ones. As Fred Brooks admonishes us, we should probably re-do major software chunks from scratch every seven years or so to sweep away creeping cruft. Perhaps we should update Brooks' time constant to an order of weeks, days or hours instead of years. That's where detail lies.

There is great power in detail, yet there is something humble and profound about this approach to life, as we might stereotypically expect from any approach that claims Japanese roots. But this is not only an Eastern outlook on life; English and American folk wisdom are full of such admonishments. The Seiton quote from above flowed from the pen of an Ohio minister who literally viewed neatness "as a remedy for every degree of evil." How about Seiso? *Cleanliness is next to godliness*. As beautiful as a house is, a messy desk robs it of its splendor. How about Shutsuke in these small matters? *He who is faithful in little is faithful in much*. How about being eager to re-factor at the responsible time, strengthening one's position for subsequent "big" decisions, rather than putting it off? *A stitch in time saves nine. The early bird catches the worm. Don't put off until tomorrow what you can do today.* (Such was the original sense of the phrase "the last responsible moment" in Lean until it fell into the hands of software consultants.) How about calibrating the place of small, individual efforts in a grand whole? *Mighty oaks from little acorns grow.* Or how about integrating simple preventive work into everyday life? *An ounce of prevention is worth a pound of cure. An apple a day keeps the doctor away.* Clean code honors the deep roots of wisdom beneath our broader culture, or our culture as it once was, or should be, and *can* be with attentiveness to detail.

Even in the grand architectural literature we find saws that hark back to these supposed details. Think of mies van der Rohe's doorknobs. That's *seiri*. That's being attentive to every variable name. You should name a variable using the same care with which you name a first-born child.

As every homeowner knows, such care and ongoing refinement never come to an end. The architect Christopher Alexander—father of patterns and pattern languages—views every act of design itself as a small, local act of repair. And he views the craftsmanship of fine structure to be the sole purview of the architect; the larger forms can be left to patterns and their application by the inhabitants. Design is ever ongoing not only as we add a new room to a house, but as we are attentive to repainting, replacing worn carpets, or upgrading the kitchen sink. Most arts echo analogous sentiments. In our search for others who ascribe God's home as being in the details, we find ourselves in the good company of the 19th century French author Gustav Flaubert. The French poet Paul Valery advises us that a poem is never done and bears continual rework, and to stop working on it is abandonment. Such preoccupation with detail is common to all endeavors of excellence. So maybe there is little new here, but in reading this book you will be challenged to take up good disciplines that you long ago surrendered to apathy or a desire for spontaneity and just "responding to change."

Unfortunately, we usually don't view such concerns as key cornerstones of the art of programming. We abandon our code early, not because it is done, but because our value system focuses more on outward appearance than on the substance of what we deliver.

This inattentiveness costs us in the end: *A bad penny always shows up*. Research, neither in industry nor in academia, humbles itself to the lowly station of keeping code clean. Back in my days working in the Bell Labs Software Production Research organization (*Production*, indeed!) we had some back-of-the-envelope findings that suggested that consistent indentation style was one of the most statistically significant indicators of low bug density. We want it to be that architecture or programming language or some other high notion should be the cause of quality; as people whose supposed professionalism owes to the mastery of tools and lofty design methods, we feel insulted by the value that those factory-floor machines, the coders, add through the simple consistent application of an indentation style. To quote my own book of 17 years ago, such style distinguishes excellence from mere competence. The Japanese worldview understands the crucial value of the everyday worker and, more so, of the systems of development that owe to the simple, everyday actions of those workers. Quality is the result of a million selfless acts of care—not just of any great method that descends from the heavens. That these acts are simple doesn't mean that they are simplistic, and it hardly means that they are easy. They are nonetheless the fabric of greatness and, more so, of beauty, in any human endeavor. To ignore them is not yet to be fully human.

Of course, I am still an advocate of thinking at broader scope, and particularly of the value of architectural approaches rooted in deep domain knowledge and software usability. The book isn't about that—or, at least, it isn't obviously about that. This book has a subtler message whose profoundness should not be underappreciated. It fits with the current saw of the really code-based people like Peter Sommerlad, Kevlin Henney and Giovanni Asproni. "The code is the design" and "Simple code" are their mantras. While we must take care to remember that the interface is the program, and that its structures have much to say about our program structure, it is crucial to continuously adopt the humble stance that the design lives in the code. And while rework in the manufacturing metaphor leads to cost, rework in design leads to value. We should view our code as the beautiful articulation of noble efforts of design—design as a process, not a static endpoint. It's in the code that the architectural metrics of coupling and cohesion play out. If you listen to Larry Constantine describe coupling and cohesion, he speaks in terms of code—not lofty abstract concepts that one might find in UML. Richard Gabriel advises us in his essay, "Abstraction Descant" that abstraction is evil. Code is anti-evil, and clean code is perhaps divine.

Going back to my little box of Ga-Jol, I think it's important to note that the Danish wisdom advises us not just to pay attention to small things, but also to be *honest* in small things. This means being honest to the code, honest to our colleagues about the state of our code and, most of all, being honest with ourselves about our code. Did we Do our Best to "leave the campground cleaner than we found it"? Did we re-factor our code before checking in? These are not peripheral concerns but concerns that lie squarely in the center of Agile values. It is a recommended practice in Scrum that re-factoring be part of the concept of "Done." Neither architecture nor clean code insist on perfection, only on honesty and doing the best we can. *To err is human; to forgive, divine.* In Scrum, we make everything visible. We air our dirty laundry. We are honest about the state of our code because code is never perfect. We become more fully human, more worthy of the divine, and closer to that greatness in the details.

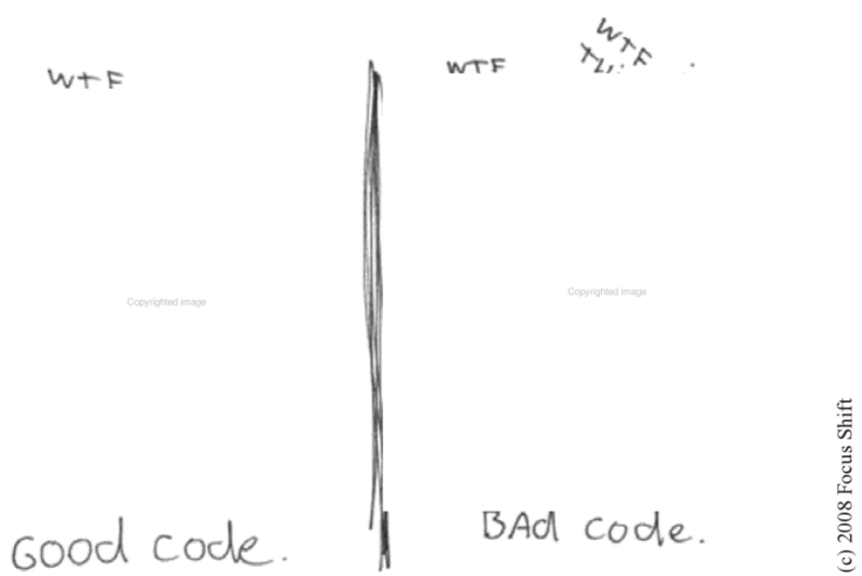
In our profession, we desperately need all the help we can get. If a clean shop floor reduces accidents, and well-organized shop tools increase productivity, then I'm all for them. As for this book, it is the best pragmatic application of Lean principles to software I have ever seen in print. I expected no less from this practical little group of thinking individuals that has been striving together for years not only to become better, but also to gift their knowledge to the industry in works such as you now find in your hands. It leaves the world a little better than I found it before Uncle Bob sent me the manuscript.

Having completed this exercise in lofty insights, I am off to clean my desk.

James O. Coplien
Mørdrup, Denmark

This page intentionally left blank

Introduction



Reproduced with the kind permission of Thom Holwerda.
http://www.osnews.com/story/19266/WTFs_m

Which door represents your code? Which door represents your team or your company? Why are we in that room? Is this just a normal code review or have we found a stream of horrible problems shortly after going live? Are we debugging in a panic, poring over code that we thought worked? Are customers leaving in droves and managers breathing down our necks? How can we make sure we wind up behind the *right* door when the going gets tough? The answer is: *craftsmanship*.

There are two parts to learning craftsmanship: knowledge and work. You must gain the knowledge of principles, patterns, practices, and heuristics that a craftsman knows, and you must also grind that knowledge into your fingers, eyes, and gut by working hard and practicing.

I can teach you the physics of riding a bicycle. Indeed, the classical mathematics is relatively straightforward. Gravity, friction, angular momentum, center of mass, and so forth, can be demonstrated with less than a page full of equations. Given those formulae I could prove to you that bicycle riding is practical and give you all the knowledge you needed to make it work. And you'd still fall down the first time you climbed on that bike.

Coding is no different. We could write down all the "feel good" principles of clean code and then trust you to do the work (in other words, let you fall down when you get on the bike), but then what kind of teachers would that make us, and what kind of student would that make you?

No. That's not the way this book is going to work.

Learning to write clean code is *hard work*. It requires more than just the knowledge of principles and patterns. You must *sweat* over it. You must practice it yourself, and watch yourself fail. You must watch others practice it and fail. You must see them stumble and retrace their steps. You must see them agonize over decisions and see the price they pay for making those decisions the wrong way.

Be prepared to work hard while reading this book. This is not a "feel good" book that you can read on an airplane and finish before you land. This book will make you work, *and work hard*. What kind of work will you be doing? You'll be reading code—lots of code. And you will be challenged to think about what's right about that code and what's wrong with it. You'll be asked to follow along as we take modules apart and put them back together again. This will take time and effort; but we think it will be worth it.

We have divided this book into three parts. The first several chapters describe the principles, patterns, and practices of writing clean code. There is quite a bit of code in these chapters, and they will be challenging to read. They'll prepare you for the second section to come. If you put the book down after reading the first section, good luck to you!

The second part of the book is the harder work. It consists of several case studies of ever-increasing complexity. Each case study is an exercise in cleaning up some code—of transforming code that has some problems into code that has fewer problems. The detail in this section is *intense*. You will have to flip back and forth between the narrative and the code listings. You will have to analyze and understand the code we are working with and walk through our reasoning for making each change we make. Set aside some time because *this should take you days*.

The third part of this book is the payoff. It is a single chapter containing a list of heuristics and smells gathered while creating the case studies. As we walked through and cleaned up the code in the case studies, we documented every reason for our actions as a heuristic or smell. We tried to understand our own reactions to the code we were reading and changing, and worked hard to capture why we felt what we felt and did what we did. The result is a knowledge base that desribes the way we think when we write, read, and clean code.

This knowledge base is of limited value if you don't do the work of carefully reading through the case studies in the second part of this book. In those case studies we have carefully annotated each change we made with forward references to the heuristics. These forward references appear in square brackets like this: [H22]. This lets you see the *context* in which those heuristics were applied and written! It is not the heuristics themselves that are so valuable, it is the *relationship between those heuristics and the discrete decisions we made while cleaning up the code in the case studies*.

To further help you with those relationships, we have placed a cross-reference at the end of the book that shows the page number for every forward reference. You can use it to look up each place where a certain heuristic was applied.

If you read the first and third sections and skip over the case studies, then you will have read yet another "feel good" book about writing good software. But if you take the time to work through the case studies, following every tiny step, every minute decision—if you put yourself in our place, and force yourself to think along the same paths that we thought, then you will gain a much richer understanding of those principles, patterns, practices, and heuristics. They won't be "feel good" knowledge any more. They'll have been ground into your gut, fingers, and heart. They'll have become part of you in the same way that a bicycle becomes an extension of your will when you have mastered how to ride it.

Acknowledgments

Thank you to my two artists, Jeniffer Kohnke and Angela Brooks. Jennifer is responsible for the stunning and creative pictures at the start of each chapter and also for the portraits of Kent Beck, Ward Cunningham, Bjarne Stroustrup, Ron Jeffries, Grady Booch, Dave Thomas, Michael Feathers, and myself.

Angela is responsible for the clever pictures that adorn the innards of each chapter. She has done quite a few pictures for me over the years, including many of the inside pictures in *Agile Software Develpment: Principles, Patterns, and Practices*. She is also my firstborn in whom I am well pleased.

A special thanks goes out to my reviewers Bob Bogetti, George Bullock, Jeffrey Overbey, and especially Matt Heusser. They were brutal. They were cruel. They were relentless. They pushed me hard to make necessary improvements.

Thanks to my publisher, Chris Guzikowski, for his support, encouragement, and jovial countenance. Thanks also to the editorial staff at Pearson, including Raina Chrobak for keeping me honest and punctual.

Thanks to Micah Martin, and all the guys at 8th Light (www.8thlight.com) for their reviews and encouragement.

Thanks to all the Object Mentors, past, present, and future, including: Bob Koss, Michael Feathers, Michael Hill, Erik Meade, Jeff Langr, Pascal Roy, David Farber, Brett Schuchert, Dean Wampler, Tim Ottinger, Dave Thomas, James Grenning, Brian Button, Ron Jeffries, Lowell Lindstrom, Angelique Martin, Cindy Sprague, Libby Ottinger, Joleen Craig, Janice Brown, Susan Rosso, et al.

Thanks to Jim Newkirk, my friend and business partner, who taught me more than I think he realizes. Thanks to Kent Beck, Martin Fowler, Ward Cunningham, Bjarne Stroustrup, Grady Booch, and all my other mentors, compatriots, and foils. Thanks to John Vlissides for being there when it counted. Thanks to the guys at Zebra for allowing me to rant on about how long a function should be.

And, finally, thank you for reading these thank yous.

On the Cover

The image on the cover is M104: The Sombrero Galaxy. M104 is located in Virgo and is just under 30 million light-years from us. At its core is a supermassive black hole weighing in at about a billion solar masses.

Does the image remind you of the explosion of the Klingon power moon *Praxis*? I vividly remember the scene in *Star Trek VI* that showed an equatorial ring of debris flying away from that explosion. Since that scene, the equatorial ring has been a common artifact in sci-fi movie explosions. It was even added to the explosion of Alderaan in later editions of the first *Star Wars* movie.

What caused this ring to form around M104? Why does it have such a huge central bulge and such a bright and tiny nucleus? It looks to me as though the central black hole lost its cool and blew a 30,000 light-year hole in the middle of the galaxy. Woe befell any civilizations that might have been in the path of that cosmic disruption.

Supermassive black holes swallow whole stars for lunch, converting a sizeable fraction of their mass to energy. $E = MC^2$ is leverage enough, but when M is a stellar mass: Look out! How many stars fell headlong into that maw before the monster was satiated? Could the size of the central void be a hint?

The image of M104 on the cover is a combination of the famous visible light photograph from Hubble (right), and the recent infrared image from the Spitzer orbiting observatory (below, right). It's the infrared image that clearly shows us the ring nature of the galaxy. In visible light we only see the front edge of the ring in silhouette. The central bulge obscures the rest of the ring.

But in the infrared, the hot particles in the ring shine through the central bulge. The two images combined give us a view we've not seen before and imply that long ago it was a raging inferno of activity.

Cover image: © Spitzer Space Telescope

1

Clean Code

You are reading this book for two reasons. First, you are a programmer. Second, you want to be a better programmer. Good. We need better programmers.

This is a book about good programming. It is filled with code. We are going to look at code from every different direction. We'll look down at it from the top, up at it from the bottom, and through it from the inside out. By the time we are done, we're going to know a lot about code. What's more, we'll be able to tell the difference between good code and bad code. We'll know how to write good code. And we'll know how to transform bad code into good code.

There Will Be Code

One might argue that a book about code is somehow behind the times—that code is no longer the issue; that we should be concerned about models and requirements instead. Indeed some have suggested that we are close to the end of code. That soon all code will be generated instead of written. That programmers simply won't be needed because business people will generate programs from specifications.

Nonsense! We will never be rid of code, because code represents the details of the requirements. At some level those details cannot be ignored or abstracted; they have to be specified. And specifying requirements in such detail that a machine can execute them *is programming*. Such a specification *is code*.

I expect that the level of abstraction of our languages will continue to increase. I also expect that the number of domain-specific languages will continue to grow. This will be a good thing. But it will not eliminate code. Indeed, all the specifications written in these higher level and domain-specific language will *be* code! It will still need to be rigorous, accurate, and so formal and detailed that a machine can understand and execute it.

The folks who think that code will one day disappear are like mathematicians who hope one day to discover a mathematics that does not have to be formal. They are hoping that one day we will discover a way to create machines that can do what we want rather than what we say. These machines will have to be able to understand us so well that they can translate vaguely specified needs into perfectly executing programs that precisely meet those needs.

This will never happen. Not even humans, with all their intuition and creativity, have been able to create successful systems from the vague feelings of their customers. Indeed, if the discipline of requirements specification has taught us anything, it is that well-specified requirements are as formal as code and can act as executable tests of that code!

Remember that code is really the language in which we ultimately express the requirements. We may create languages that are closer to the requirements. We may create tools that help us parse and assemble those requirements into formal structures. But we will never eliminate necessary precision—so there will always be code.

Bad Code

I was recently reading the preface to Kent Beck's book *Implementation Patterns.*[1] He says, ". . . this book is based on a rather fragile premise: that good code matters. . . ." A *fragile* premise? I disagree! I think that premise is one of the most robust, supported, and overloaded of all the premises in our craft (and I think Kent knows it). We know good code matters because we've had to deal for so long with its lack.

I know of one company that, in the late 80s, wrote a *killer* app. It was very popular, and lots of professionals bought and used it. But then the release cycles began to stretch. Bugs were not repaired from one release to the next. Load times grew and crashes increased. I remember the day I shut the product down in frustration and never used it again. The company went out of business a short time after that.

Two decades later I met one of the early employees of that company and asked him what had happened. The answer confirmed my fears. They had rushed the product to market and had made a huge mess in the code. As they added more and more features, the code got worse and worse until they simply could not manage it any longer. *It was the bad code that brought the company down.*

Have *you* ever been significantly impeded by bad code? If you are a programmer of any experience then you've felt this impediment many times. Indeed, we have a name for it. We call it *wading*. We wade through bad code. We slog through a morass of tangled brambles and hidden pitfalls. We struggle to find our way, hoping for some hint, some clue, of what is going on; but all we see is more and more senseless code.

Of course you have been impeded by bad code. So then—why did you write it?

Were you trying to go fast? Were you in a rush? Probably so. Perhaps you felt that you didn't have time to do a good job; that your boss would be angry with you if you took the time to clean up your code. Perhaps you were just tired of working on this program and wanted it to be over. Or maybe you looked at the backlog of other stuff that you had promised to get done and realized that you needed to slam this module together so you could move on to the next. We've all done it.

We've all looked at the mess we've just made and then have chosen to leave it for another day. We've all felt the relief of seeing our messy program work and deciding that a

1. [Beck07].

working mess is better than nothing. We've all said we'd go back and clean it up later. Of course, in those days we didn't know LeBlanc's law: *Later equals never.*

The Total Cost of Owning a Mess

If you have been a programmer for more than two or three years, you have probably been significantly slowed down by someone else's messy code. If you have been a programmer for longer than two or three years, you have probably been slowed down by messy code. The degree of the slowdown can be significant. Over the span of a year or two, teams that were moving very fast at the beginning of a project can find themselves moving at a snail's pace. Every change they make to the code breaks two or three other parts of the code. No change is trivial. Every addition or modification to the system requires that the tangles, twists, and knots be "understood" so that more tangles, twists, and knots can be added. Over time the mess becomes so big and so deep and so tall, they can not clean it up. There is no way at all.

As the mess builds, the productivity of the team continues to decrease, asymptotically approaching zero. As productivity decreases, management does the only thing they can; they add more staff to the project in hopes of increasing productivity. But that new staff is not versed in the design of the system. They don't know the difference between a change that matches the design intent and a change that thwarts the design intent. Furthermore, they, and everyone else on the team, are under horrific pressure to increase productivity. So they all make more and more messes, driving the productivity ever further toward zero. (See Figure 1-1.)

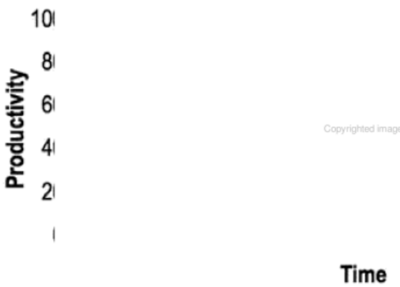


Figure 1-1
Productivity vs. time

The Grand Redesign in the Sky

Eventually the team rebels. They inform management that they cannot continue to develop in this odious code base. They demand a redesign. Management does not want to expend the resources on a whole new redesign of the project, but they cannot deny that productivity is terrible. Eventually they bend to the demands of the developers and authorize the grand redesign in the sky.

A new tiger team is selected. Everyone wants to be on this team because it's a greenfield project. They get to start over and create something truly beautiful. But only the best and brightest are chosen for the tiger team. Everyone else must continue to maintain the current system.

Now the two teams are in a race. The tiger team must build a new system that does everything that the old system does. Not only that, they have to keep up with the changes that are continuously being made to the old system. Management will not replace the old system until the new system can do everything that the old system does.

This race can go on for a very long time. I've seen it take 10 years. And by the time it's done, the original members of the tiger team are long gone, and the current members are demanding that the new system be redesigned because it's such a mess.

If you have experienced even one small part of the story I just told, then you already know that spending time keeping your code clean is not just cost effective; it's a matter of professional survival.

Attitude

Have you ever waded through a mess so grave that it took weeks to do what should have taken hours? Have you seen what should have been a one-line change, made instead in hundreds of different modules? These symptoms are all too common.

Why does this happen to code? Why does good code rot so quickly into bad code? We have lots of explanations for it. We complain that the requirements changed in ways that thwart the original design. We bemoan the schedules that were too tight to do things right. We blather about stupid managers and intolerant customers and useless marketing types and telephone sanitizers. But the fault, dear Dilbert, is not in our stars, but in ourselves. We are unprofessional.

This may be a bitter pill to swallow. How could this mess be *our* fault? What about the requirements? What about the schedule? What about the stupid managers and the useless marketing types? Don't they bear some of the blame?

No. The managers and marketers look to *us* for the information they need to make promises and commitments; and even when they don't look to us, we should not be shy about telling them what we think. The users look to us to validate the way the requirements will fit into the system. The project managers look to us to help work out the schedule. We are deeply complicit in the planning of the project and share a great deal of the responsibility for any failures; especially if those failures have to do with bad code!

"But wait!" you say. "If I don't do what my manager says, I'll be fired." Probably not. Most managers want the truth, even when they don't act like it. Most managers want good code, even when they are obsessing about the schedule. They may defend the schedule and requirements with passion; but that's their job. It's *your* job to defend the code with equal passion.

To drive this point home, what if you were a doctor and had a patient who demanded that you stop all the silly hand-washing in preparation for surgery because it was taking too much time?[2] Clearly the patient is the boss; and yet the doctor should absolutely refuse to comply. Why? Because the doctor knows more than the patient about the risks of disease and infection. It would be unprofessional (never mind criminal) for the doctor to comply with the patient.

So too it is unprofessional for programmers to bend to the will of managers who don't understand the risks of making messes.

The Primal Conundrum

Programmers face a conundrum of basic values. All developers with more than a few years experience know that previous messes slow them down. And yet all developers feel the pressure to make messes in order to meet deadlines. In short, they don't take the time to go fast!

True professionals know that the second part of the conundrum is wrong. You will *not* make the deadline by making the mess. Indeed, the mess will slow you down instantly, and will force you to miss the deadline. The *only* way to make the deadline—the only way to go fast—is to keep the code as clean as possible at all times.

The Art of Clean Code?

Let's say you believe that messy code is a significant impediment. Let's say that you accept that the only way to go fast is to keep your code clean. Then you must ask yourself: "How do I write clean code?" It's no good trying to write clean code if you don't know what it means for code to be clean!

The bad news is that writing clean code is a lot like painting a picture. Most of us know when a picture is painted well or badly. But being able to recognize good art from bad does not mean that we know how to paint. So too being able to recognize clean code from dirty code does not mean that we know how to write clean code!

2. When hand-washing was first recommended to physicians by Ignaz Semmelweis in 1847, it was rejected on the basis that doctors were too busy and wouldn't have time to wash their hands between patient visits.

Writing clean code requires the disciplined use of a myriad little techniques applied through a painstakingly acquired sense of "cleanliness." This "code-sense" is the key. Some of us are born with it. Some of us have to fight to acquire it. Not only does it let us see whether code is good or bad, but it also shows us the strategy for applying our discipline to transform bad code into clean code.

A programmer without "code-sense" can look at a messy module and recognize the mess but will have no idea what to do about it. A programmer *with* "code-sense" will look at a messy module and see options and variations. The "code-sense" will help that programmer choose the best variation and guide him or her to plot a sequence of behavior preserving transformations to get from here to there.

In short, a programmer who writes clean code is an artist who can take a blank screen through a series of transformations until it is an elegantly coded system.

What Is Clean Code?

There are probably as many definitions as there are programmers. So I asked some very well-known and deeply experienced programmers what they thought.

Bjarne Stroustrup, inventor of C++ and author of *The C++ Programming Language*

> *I like my code to be elegant and efficient. The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. Clean code does one thing well.*

Bjarne uses the word "elegant." That's quite a word! The dictionary in my MacBook® provides the following definitions: *pleasingly graceful and stylish in appearance or manner; pleasingly ingenious and simple.* Notice the emphasis on the word "pleasing." Apparently Bjarne thinks that clean code is *pleasing* to read. Reading it should make you smile the way a well-crafted music box or well-designed car would.

Bjarne also mentions efficiency—*twice*. Perhaps this should not surprise us coming from the inventor of C++; but I think there's more to it than the sheer desire for speed. Wasted cycles are inelegant, they are not pleasing. And now note the word that Bjarne uses to describe the consequence of that inelegance. He uses the word "tempt." There is a deep truth here. Bad code *tempts* the mess to grow! When others change bad code, they tend to make it worse.

Pragmatic Dave Thomas and Andy Hunt said this a different way. They used the metaphor of broken windows.[3] A building with broken windows looks like nobody cares about it. So other people stop caring. They allow more windows to become broken. Eventually they actively break them. They despoil the facade with graffiti and allow garbage to collect. One broken window starts the process toward decay.

Bjarne also mentions that error handing should be complete. This goes to the discipline of paying attention to details. Abbreviated error handling is just one way that programmers gloss over details. Memory leaks are another, race conditions still another. Inconsistent naming yet another. The upshot is that clean code exhibits close attention to detail.

Bjarne closes with the assertion that clean code does one thing well. It is no accident that there are so many principles of software design that can be boiled down to this simple admonition. Writer after writer has tried to communicate this thought. Bad code tries to do too much, it has muddled intent and ambiguity of purpose. Clean code is *focused*. Each function, each class, each module exposes a single-minded attitude that remains entirely undistracted, and unpolluted, by the surrounding details.

Grady Booch, author of *Object Oriented Analysis and Design with Applications*

> *Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designer's intent but rather is full of crisp abstractions and straightforward lines of control.*

Grady makes some of the same points as Bjarne, but he takes a *readability* perspective. I especially like his view that clean code should read like well-written prose. Think back on a really good book that you've read. Remember how the words disappeared to be replaced by images! It was like watching a movie, wasn't it? Better! You saw the characters, you heard the sounds, you experienced the pathos and the humor.

Reading clean code will never be quite like reading *Lord of the Rings*. Still, the literary metaphor is not a bad one. Like a good novel, clean code should clearly expose the tensions in the problem to be solved. It should build those tensions to a climax and then give

3. http://www.pragmaticprogrammer.com/booksellers/2004-12.html

the reader that "Aha! Of course!" as the issues and tensions are resolved in the revelation of an obvious solution.

I find Grady's use of the phrase "crisp abstraction" to be a fascinating oxymoron! After all the word "crisp" is nearly a synonym for "concrete." My MacBook's dictionary holds the following definition of "crisp": *briskly decisive and matter-of-fact, without hesitation or unnecessary detail.* Despite this seeming juxtaposition of meaning, the words carry a powerful message. Our code should be matter-of-fact as opposed to speculative. It should contain only what is necessary. Our readers should perceive us to have been decisive.

"Big" Dave Thomas, founder of OTI, godfather of the Eclipse strategy

> *Clean code can be read, and enhanced by a developer other than its original author. It has unit and acceptance tests. It has meaningful names. It provides one way rather than many ways for doing one thing. It has minimal dependencies, which are explicitly defined, and provides a clear and minimal API. Code should be literate since depending on the language, not all necessary information can be expressed clearly in code alone.*

Big Dave shares Grady's desire for readability, but with an important twist. Dave asserts that clean code makes it easy for *other* people to enhance it. This may seem obvious, but it cannot be overemphasized. There is, after all, a difference between code that is easy to read and code that is easy to change.

Dave ties cleanliness to tests! Ten years ago this would have raised a lot of eyebrows. But the discipline of Test Driven Development has made a profound impact upon our industry and has become one of our most fundamental disciplines. Dave is right. Code, without tests, is not clean. No matter how elegant it is, no matter how readable and accessible, if it hath not tests, it be unclean.

Dave uses the word *minimal* twice. Apparently he values code that is small, rather than code that is large. Indeed, this has been a common refrain throughout software literature since its inception. Smaller is better.

Dave also says that code should be *literate*. This is a soft reference to Knuth's *literate programming*.[4] The upshot is that the code should be composed in such a form as to make it readable by humans.

4. [Knuth92].

Michael Feathers, author of *Working Effectively with Legacy Code*

> *I could list all of the qualities that I notice in clean code, but there is one overarching quality that leads to all of them. Clean code always looks like it was written by someone who cares. There is nothing obvious that you can do to make it better. All of those things were thought about by the code's author, and if you try to imagine improvements, you're led back to where you are, sitting in appreciation of the code someone left for you—code left by someone who cares deeply about the craft.*

One word: care. That's really the topic of this book. Perhaps an appropriate subtitle would be *How to Care for Code*.

Michael hit it on the head. Clean code is code that has been taken care of. Someone has taken the time to keep it simple and orderly. They have paid appropriate attention to details. They have cared.

Ron Jeffries, author of *Extreme Programming Installed* and *Extreme Programming Adventures in C#*

Ron began his career programming in Fortran at the Strategic Air Command and has written code in almost every language and on almost every machine. It pays to consider his words carefully.

> *In recent years I begin, and nearly end, with Beck's rules of simple code. In priority order, simple code:*
>
> - *Runs all the tests;*
> - *Contains no duplication;*
> - *Expresses all the design ideas that are in the system;*
> - *Minimizes the number of entities such as classes, methods, functions, and the like.*
>
> *Of these, I focus mostly on duplication. When the same thing is done over and over, it's a sign that there is an idea in our mind that is not well represented in the code. I try to figure out what it is. Then I try to express that idea more clearly.*
>
> *Expressiveness to me includes meaningful names, and I am likely to change the names of things several times before I settle in. With modern coding tools such as Eclipse, renaming is quite inexpensive, so it doesn't trouble me to change. Expressiveness goes*

> *beyond names, however. I also look at whether an object or method is doing more than one thing. If it's an object, it probably needs to be broken into two or more objects. If it's a method, I will always use the Extract Method refactoring on it, resulting in one method that says more clearly what it does, and some submethods saying how it is done.*
>
> *Duplication and expressiveness take me a very long way into what I consider clean code, and improving dirty code with just these two things in mind can make a huge difference. There is, however, one other thing that I'm aware of doing, which is a bit harder to explain.*
>
> *After years of doing this work, it seems to me that all programs are made up of very similar elements. One example is "find things in a collection." Whether we have a database of employee records, or a hash map of keys and values, or an array of items of some kind, we often find ourselves wanting a particular item from that collection. When I find that happening, I will often wrap the particular implementation in a more abstract method or class. That gives me a couple of interesting advantages.*
>
> *I can implement the functionality now with something simple, say a hash map, but since now all the references to that search are covered by my little abstraction, I can change the implementation any time I want. I can go forward quickly while preserving my ability to change later.*
>
> *In addition, the collection abstraction often calls my attention to what's "really" going on, and keeps me from running down the path of implementing arbitrary collection behavior when all I really need is a few fairly simple ways of finding what I want.*
>
> *Reduced duplication, high expressiveness, and early building of simple abstractions. That's what makes clean code for me.*

Here, in a few short paragraphs, Ron has summarized the contents of this book. No duplication, one thing, expressiveness, tiny abstractions. Everything is there.

Ward Cunningham, inventor of Wiki, inventor of Fit, coinventor of eXtreme Programming. Motive force behind Design Patterns. Smalltalk and OO thought leader. The godfather of all those who care about code.

> *You know you are working on clean code when each routine you read turns out to be pretty much what you expected. You can call it beautiful code when the code also makes it look like the language was made for the problem.*

Statements like this are characteristic of Ward. You read it, nod your head, and then go on to the next topic. It sounds so reasonable, so obvious, that it barely registers as something profound. You might think it was pretty much what you expected. But let's take a closer look.

"... pretty much what you expected." When was the last time you saw a module that was pretty much what you expected? Isn't it more likely that the modules you look at will be puzzling, complicated, tangled? Isn't misdirection the rule? Aren't you used to flailing about trying to grab and hold the threads of reasoning that spew forth from the whole system and weave their way through the module you are reading? When was the last time you read through some code and nodded your head the way you might have nodded your head at Ward's statement?

Ward expects that when you read clean code you won't be surprised at all. Indeed, you won't even expend much effort. You will read it, and it will be pretty much what you expected. It will be obvious, simple, and compelling. Each module will set the stage for the next. Each tells you how the next will be written. Programs that are *that* clean are so profoundly well written that you don't even notice it. The designer makes it look ridiculously simple like all exceptional designs.

And what about Ward's notion of beauty? We've all railed against the fact that our languages weren't designed for our problems. But Ward's statement puts the onus back on us. He says that beautiful code *makes the language look like it was made for the problem*! So it's *our* responsibility to make the language look simple! Language bigots everywhere, beware! It is not the language that makes programs appear simple. It is the programmer that make the language appear simple!

Schools of Thought

What about me (Uncle Bob)? What do I think clean code is? This book will tell you, in hideous detail, what I and my compatriots think about clean code. We will tell you what we think makes a clean variable name, a clean function, a clean class, etc. We will present these opinions as absolutes, and we will not apologize for our stridence. To us, at this point in our careers, they *are* absolutes. They are *our school of thought* about clean code.

Martial artists do not all agree about the best martial art, or the best technique within a martial art. Often master martial artists will form their own schools of thought and gather students to learn from them. So we see *Gracie Jiu Jistu*, founded and taught by the Gracie family in Brazil. We see *Hakkoryu Jiu Jistu*, founded and taught by Okuyama Ryuho in Tokyo. We see *Jeet Kune Do*, founded and taught by Bruce Lee in the United States.

Students of these approaches immerse themselves in the teachings of the founder. They dedicate themselves to learn what that particular master teaches, often to the exclusion of any other master's teaching. Later, as the students grow in their art, they may become the student of a different master so they can broaden their knowledge and practice. Some eventually go on to refine their skills, discovering new techniques and founding their own schools.

None of these different schools is absolutely *right*. Yet within a particular school we *act* as though the teachings and techniques *are* right. After all, there is a right way to practice Hakkoryu Jiu Jitsu, or Jeet Kune Do. But this rightness within a school does not invalidate the teachings of a different school.

Consider this book a description of the *Object Mentor School of Clean Code*. The techniques and teachings within are the way that *we* practice *our* art. We are willing to claim that if you follow these teachings, you will enjoy the benefits that we have enjoyed, and you will learn to write code that is clean and professional. But don't make the mistake of thinking that we are somehow "right" in any absolute sense. There are other schools and other masters that have just as much claim to professionalism as we. It would behoove you to learn from them as well.

Indeed, many of the recommendations in this book are controversial. You will probably not agree with all of them. You might violently disagree with some of them. That's fine. We can't claim final authority. On the other hand, the recommendations in this book are things that we have thought long and hard about. We have learned them through decades of experience and repeated trial and error. So whether you agree or disagree, it would be a shame if you did not see, and respect, our point of view.

We Are Authors

The `@author` field of a Javadoc tells us who we are. We are authors. And one thing about authors is that they have readers. Indeed, authors are *responsible* for communicating well with their readers. The next time you write a line of code, remember you are an author, writing for readers who will judge your effort.

You might ask: How much is code really read? Doesn't most of the effort go into writing it?

Have you ever played back an edit session? In the 80s and 90s we had editors like Emacs that kept track of every keystroke. You could work for an hour and then play back your whole edit session like a high-speed movie. When I did this, the results were fascinating.

The vast majority of the playback was scrolling and navigating to other modules!

Bob enters the module.
He scrolls down to the function needing change.
He pauses, considering his options.
Oh, he's scrolling up to the top of the module to check the initialization of a variable.
Now he scrolls back down and begins to type.

Ooops, he's erasing what he typed!
He types it again.
He erases it again!
He types half of something else but then erases that!
He scrolls down to another function that calls the function he's changing to see how it is called.
He scrolls back up and types the same code he just erased.
He pauses.
He erases that code again!
He pops up another window and looks at a subclass. Is that function overridden?

. . .

You get the drift. Indeed, the ratio of time spent reading vs. writing is well over 10:1. We are *constantly* reading old code as part of the effort to write new code.

Because this ratio is so high, we want the reading of code to be easy, even if it makes the writing harder. Of course there's no way to write code without reading it, so *making it easy to read actually makes it easier to write.*

There is no escape from this logic. You cannot write code if you cannot read the surrounding code. The code you are trying to write today will be hard or easy to write depending on how hard or easy the surrounding code is to read. So if you want to go fast, if you want to get done quickly, if you want your code to be easy to write, make it easy to read.

The Boy Scout Rule

It's not enough to write the code well. The code has to be *kept clean* over time. We've all seen code rot and degrade as time passes. So we must take an active role in preventing this degradation.

The Boy Scouts of America have a simple rule that we can apply to our profession.

Leave the campground cleaner than you found it.[5]

If we all checked-in our code a little cleaner than when we checked it out, the code simply could not rot. The cleanup doesn't have to be something big. Change one variable name for the better, break up one function that's a little too large, eliminate one small bit of duplication, clean up one composite `if` statement.

Can you imagine working on a project where the code *simply got better* as time passed? Do you believe that any other option is professional? Indeed, isn't continuous improvement an intrinsic part of professionalism?

5. This was adapted from Robert Stephenson Smyth Baden-Powell's farewell message to the Scouts: "Try and leave this world a little better than you found it . . ."

Prequel and Principles

In many ways this book is a "prequel" to a book I wrote in 2002 entitled *Agile Software Development: Principles, Patterns, and Practices* (PPP). The PPP book concerns itself with the principles of object-oriented design, and many of the practices used by professional developers. If you have not read PPP, then you may find that it continues the story told by this book. If you have already read it, then you'll find many of the sentiments of that book echoed in this one at the level of code.

In this book you will find sporadic references to various principles of design. These include the Single Responsibility Principle (SRP), the Open Closed Principle (OCP), and the Dependency Inversion Principle (DIP) among others. These principles are described in depth in PPP.

Conclusion

Books on art don't promise to make you an artist. All they can do is give you some of the tools, techniques, and thought processes that other artists have used. So too this book cannot promise to make you a good programmer. It cannot promise to give you "code-sense." All it can do is show you the thought processes of good programmers and the tricks, techniques, and tools that they use.

Just like a book on art, this book will be full of details. There will be lots of code. You'll see good code and you'll see bad code. You'll see bad code transformed into good code. You'll see lists of heuristics, disciplines, and techniques. You'll see example after example. After that, it's up to you.

Remember the old joke about the concert violinist who got lost on his way to a performance? He stopped an old man on the corner and asked him how to get to Carnegie Hall. The old man looked at the violinist and the violin tucked under his arm, and said: "Practice, son. Practice!"

Bibliography

[Beck07]: *Implementation Patterns*, Kent Beck, Addison-Wesley, 2007.

[Knuth92]: *Literate Programming*, Donald E. Knuth, Center for the Study of Language and Information, Leland Stanford Junior University, 1992.

This page intentionally left blank

2

Meaningful Names

by Tim Ottinger

Introduction

Names are everywhere in software. We name our variables, our functions, our arguments, classes, and packages. We name our source files and the directories that contain them. We name our jar files and war files and ear files. We name and name and name. Because we do so much of it, we'd better do it well. What follows are some simple rules for creating good names.

Use Intention-Revealing Names

It is easy to say that names should reveal intent. What we want to impress upon you is that we are *serious* about this. Choosing good names takes time but saves more than it takes. So take care with your names and change them when you find better ones. Everyone who reads your code (including you) will be happier if you do.

The name of a variable, function, or class, should answer all the big questions. It should tell you why it exists, what it does, and how it is used. If a name requires a comment, then the name does not reveal its intent.

```
int d; // elapsed time in days
```

The name `d` reveals nothing. It does not evoke a sense of elapsed time, nor of days. We should choose a name that specifies what is being measured and the unit of that measurement:

```
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

Choosing names that reveal intent can make it much easier to understand and change code. What is the purpose of this code?

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
    if (x[0] == 4)
      list1.add(x);
  return list1;
}
```

Why is it hard to tell what this code is doing? There are no complex expressions. Spacing and indentation are reasonable. There are only three variables and two constants mentioned. There aren't even any fancy classes or polymorphic methods, just a list of arrays (or so it seems).

The problem isn't the simplicity of the code but the *implicity* of the code (to coin a phrase): the degree to which the context is not explicit in the code itself. The code implicitly requires that we know the answers to questions such as:

1. What kinds of things are in `theList`?
2. What is the significance of the zeroth subscript of an item in `theList`?
3. What is the significance of the value `4`?
4. How would I use the list being returned?

The answers to these questions are not present in the code sample, *but they could have been*. Say that we're working in a mine sweeper game. We find that the board is a list of cells called `theList`. Let's rename that to `gameBoard`.

Each cell on the board is represented by a simple array. We further find that the zeroth subscript is the location of a status value and that a status value of 4 means "flagged." Just by giving these concepts names we can improve the code considerably:

```
public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<int[]>();
  for (int[] cell : gameBoard)
    if (cell[STATUS_VALUE] == FLAGGED)
      flaggedCells.add(cell);
  return flaggedCells;
}
```

Notice that the simplicity of the code has not changed. It still has exactly the same number of operators and constants, with exactly the same number of nesting levels. But the code has become much more explicit.

We can go further and write a simple class for cells instead of using an array of `int`s. It can include an intention-revealing function (call it `isFlagged`) to hide the magic numbers. It results in a new version of the function:

```
public List<Cell> getFlaggedCells() {
  List<Cell> flaggedCells = new ArrayList<Cell>();
  for (Cell cell : gameBoard)
    if (cell.isFlagged())
      flaggedCells.add(cell);
  return flaggedCells;
}
```

With these simple name changes, it's not difficult to understand what's going on. This is the power of choosing good names.

Avoid Disinformation

Programmers must avoid leaving false clues that obscure the meaning of code. We should avoid words whose entrenched meanings vary from our intended meaning. For example, `hp`, `aix`, and `sco` would be poor variable names because they are the names of Unix platforms or variants. Even if you are coding a hypotenuse and `hp` looks like a good abbreviation, it could be disinformative.

Do not refer to a grouping of accounts as an `accountList` unless it's actually a `List`. The word list means something specific to programmers. If the container holding the accounts is not actually a `List`, it may lead to false conclusions.[1] So `accountGroup` or `bunchOfAccounts` or just plain `accounts` would be better.

1. As we'll see later on, even if the container *is* a `List`, it's probably better not to encode the container type into the name.

Beware of using names which vary in small ways. How long does it take to spot the subtle difference between a `XYZControllerForEfficientHandlingOfStrings` in one module and, somewhere a little more distant, `XYZControllerForEfficientStorageOfStrings`? The words have frightfully similar shapes.

Spelling similar concepts similarly is *information*. Using inconsistent spellings is *disinformation*. With modern Java environments we enjoy automatic code completion. We write a few characters of a name and press some hotkey combination (if that) and are rewarded with a list of possible completions for that name. It is very helpful if names for very similar things sort together alphabetically and if the differences are very obvious, because the developer is likely to pick an object by name without seeing your copious comments or even the list of methods supplied by that class.

A truly awful example of disinformative names would be the use of lower-case `L` or uppercase `O` as variable names, especially in combination. The problem, of course, is that they look almost entirely like the constants one and zero, respectively.

```
int a = l;
if ( O == l )
  a = O1;
else
  l = 01;
```

The reader may think this a contrivance, but we have examined code where such things were abundant. In one case the author of the code suggested using a different font so that the differences were more obvious, a solution that would have to be passed down to all future developers as oral tradition or in a written document. The problem is conquered with finality and without creating new work products by a simple renaming.

Make Meaningful Distinctions

Programmers create problems for themselves when they write code solely to satisfy a compiler or interpreter. For example, because you can't use the same name to refer to two different things in the same scope, you might be tempted to change one name in an arbitrary way. Sometimes this is done by misspelling one, leading to the surprising situation where correcting spelling errors leads to an inability to compile.[2]

It is not sufficient to add number series or noise words, even though the compiler is satisfied. If names must be different, then they should also mean something different.

2. Consider, for example, the truly hideous practice of creating a variable named `klass` just because the name `class` was used for something else.

Number-series naming (a1, a2, .. aN) is the opposite of intentional naming. Such names are not disinformative—they are noninformative; they provide no clue to the author's intention. Consider:

```
public static void copyChars(char a1[], char a2[]) {
  for (int i = 0; i < a1.length; i++) {
    a2[i] = a1[i];
  }
}
```

This function reads much better when source and destination are used for the argument names.

Noise words are another meaningless distinction. Imagine that you have a Product class. If you have another called ProductInfo or ProductData, you have made the names different without making them mean anything different. Info and Data are indistinct noise words like a, an, and the.

Note that there is nothing wrong with using prefix conventions like a and the so long as they make a meaningful distinction. For example you might use a for all local variables and the for all function arguments.[3] The problem comes in when you decide to call a variable theZork because you already have another variable named zork.

Noise words are redundant. The word variable should never appear in a variable name. The word table should never appear in a table name. How is NameString better than Name? Would a Name ever be a floating point number? If so, it breaks an earlier rule about disinformation. Imagine finding one class named Customer and another named CustomerObject. What should you understand as the distinction? Which one will represent the best path to a customer's payment history?

There is an application we know of where this is illustrated. we've changed the names to protect the guilty, but here's the exact form of the error:

```
getActiveAccount();
getActiveAccounts();
getActiveAccountInfo();
```

How are the programmers in this project supposed to know which of these functions to call?

In the absence of specific conventions, the variable moneyAmount is indistinguishable from money, customerInfo is indistinguishable from customer, accountData is indistinguishable from account, and theMessage is indistinguishable from message. Distinguish names in such a way that the reader knows what the differences offer.

Use Pronounceable Names

Humans are good at words. A significant part of our brains is dedicated to the concept of words. And words are, by definition, pronounceable. It would be a shame not to take

3. Uncle Bob used to do this in C++ but has given up the practice because modern IDEs make it unnecessary.

advantage of that huge portion of our brains that has evolved to deal with spoken language. So make your names pronounceable.

If you can't pronounce it, you can't discuss it without sounding like an idiot. "Well, over here on the bee cee arr three cee enn tee we have a pee ess zee kyew int, see?" This matters because programming is a social activity.

A company I know has `genymdhms` (generation date, year, month, day, hour, minute, and second) so they walked around saying "gen why emm dee aich emm ess". I have an annoying habit of pronouncing everything as written, so I started saying "gen-yah-mudda-hims." It later was being called this by a host of designers and analysts, and we still sounded silly. But we were in on the joke, so it was fun. Fun or not, we were tolerating poor naming. New developers had to have the variables explained to them, and then they spoke about it in silly made-up words instead of using proper English terms. Compare

```
class DtaRcrd102 {
  private Date genymdhms;
  private Date modymdhms;
  private final String pszqint = "102";
  /* ... */
};
```

to

```
class Customer {
  private Date generationTimestamp;
  private Date modificationTimestamp;;
  private final String recordId = "102";
  /* ... */
};
```

Intelligent conversation is now possible: "Hey, Mikey, take a look at this record! The generation timestamp is set to tomorrow's date! How can that be?"

Use Searchable Names

Single-letter names and numeric constants have a particular problem in that they are not easy to locate across a body of text.

One might easily grep for `MAX_CLASSES_PER_STUDENT`, but the number 7 could be more troublesome. Searches may turn up the digit as part of file names, other constant definitions, and in various expressions where the value is used with different intent. It is even worse when a constant is a long number and someone might have transposed digits, thereby creating a bug while simultaneously evading the programmer's search.

Likewise, the name `e` is a poor choice for any variable for which a programmer might need to search. It is the most common letter in the English language and likely to show up in every passage of text in every program. In this regard, longer names trump shorter names, and any searchable name trumps a constant in code.

My personal preference is that single-letter names can ONLY be used as local variables inside short methods. *The length of a name should correspond to the size of its scope*

[N5]. If a variable or constant might be seen or used in multiple places in a body of code, it is imperative to give it a search-friendly name. Once again compare

```
for (int j=0; j<34; j++) {
  s += (t[j]*4)/5;
}
```

to

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
  int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
  int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
  sum += realTaskWeeks;
}
```

Note that `sum`, above, is not a particularly useful name but at least is searchable. The intentionally named code makes for a longer function, but consider how much easier it will be to find `WORK_DAYS_PER_WEEK` than to find all the places where 5 was used and filter the list down to just the instances with the intended meaning.

Avoid Encodings

We have enough encodings to deal with without adding more to our burden. Encoding type or scope information into names simply adds an extra burden of deciphering. It hardly seems reasonable to require each new employee to learn yet another encoding "language" in addition to learning the (usually considerable) body of code that they'll be working in. It is an unnecessary mental burden when trying to solve a problem. Encoded names are seldom pronounceable and are easy to mis-type.

Hungarian Notation

In days of old, when we worked in name-length-challenged languages, we violated this rule out of necessity, and with regret. Fortran forced encodings by making the first letter a code for the type. Early versions of BASIC allowed only a letter plus one digit. Hungarian Notation (HN) took this to a whole new level.

HN was considered to be pretty important back in the Windows C API, when everything was an integer handle or a long pointer or a `void` pointer, or one of several implementations of "string" (with different uses and attributes). The compiler did not check types in those days, so the programmers needed a crutch to help them remember the types.

In modern languages we have much richer type systems, and the compilers remember and enforce the types. What's more, there is a trend toward smaller classes and shorter functions so that people can usually see the point of declaration of each variable they're using.

Java programmers don't need type encoding. Objects are strongly typed, and editing environments have advanced such that they detect a type error long before you can run a compile! So nowadays HN and other forms of type encoding are simply impediments. They make it harder to change the name or type of a variable, function, or class. They make it harder to read the code. And they create the possibility that the encoding system will mislead the reader.

```
PhoneNumber phoneString;
// name not changed when type changed!
```

Member Prefixes

You also don't need to prefix member variables with `m_` anymore. Your classes and functions should be small enough that you don't need them. And you should be using an editing environment that highlights or colorizes members to make them distinct.

```
public class Part {
  private String m_dsc; // The textual description
  void setName(String name) {
    m_dsc = name;
  }
}
_________________________________________________

public class Part {
  String description;
  void setDescription(String description) {
    this.description = description;
  }
}
```

Besides, people quickly learn to ignore the prefix (or suffix) to see the meaningful part of the name. The more we read the code, the less we see the prefixes. Eventually the prefixes become unseen clutter and a marker of older code.

Interfaces and Implementations

These are sometimes a special case for encodings. For example, say you are building an ABSTRACT FACTORY for the creation of shapes. This factory will be an interface and will be implemented by a concrete class. What should you name them? `IShapeFactory` and `ShapeFactory`? I prefer to leave interfaces unadorned. The preceding `I`, so common in today's legacy wads, is a distraction at best and too much information at worst. I don't want my users knowing that I'm handing them an interface. I just want them to know that it's a `ShapeFactory`. So if I must encode either the interface or the implementation, I choose the implementation. Calling it `ShapeFactoryImp`, or even the hideous `CShapeFactory`, is preferable to encoding the interface.

Avoid Mental Mapping

Readers shouldn't have to mentally translate your names into other names they already know. This problem generally arises from a choice to use neither problem domain terms nor solution domain terms.

This is a problem with single-letter variable names. Certainly a loop counter may be named `i` or `j` or `k` (though never `l`!) if its scope is very small and no other names can conflict with it. This is because those single-letter names for loop counters are traditional. However, in most other contexts a single-letter name is a poor choice; it's just a place holder that the reader must mentally map to the actual concept. There can be no worse reason for using the name `c` than because `a` and `b` were already taken.

In general programmers are pretty smart people. Smart people sometimes like to show off their smarts by demonstrating their mental juggling abilities. After all, if you can reliably remember that `r` is the lower-cased version of the url with the host and scheme removed, then you must clearly be very smart.

One difference between a smart programmer and a professional programmer is that the professional understands that *clarity is king*. Professionals use their powers for good and write code that others can understand.

Class Names

Classes and objects should have noun or noun phrase names like `Customer`, `WikiPage`, `Account`, and `AddressParser`. Avoid words like `Manager`, `Processor`, `Data`, or `Info` in the name of a class. A class name should not be a verb.

Method Names

Methods should have verb or verb phrase names like `postPayment`, `deletePage`, or `save`. Accessors, mutators, and predicates should be named for their value and prefixed with `get`, `set`, and `is` according to the javabean standard.[4]

```
string name = employee.getName();
customer.setName("mike");
if (paycheck.isPosted())...
```

When constructors are overloaded, use static factory methods with names that describe the arguments. For example,

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);
```

is generally better than

```
Complex fulcrumPoint = new Complex(23.0);
```

Consider enforcing their use by making the corresponding constructors private.

4. http://java.sun.com/products/javabeans/docs/spec.html

Don't Be Cute

If names are too clever, they will be memorable only to people who share the author's sense of humor, and only as long as these people remember the joke. Will they know what the function named `HolyHandGrenade` is supposed to do? Sure, it's cute, but maybe in this case `DeleteItems` might be a better name. Choose clarity over entertainment value.

Cuteness in code often appears in the form of colloquialisms or slang. For example, don't use the name `whack()` to mean `kill()`. Don't tell little culture-dependent jokes like `eatMyShorts()` to mean `abort()`.

Say what you mean. Mean what you say.

Pick One Word per Concept

Pick one word for one abstract concept and stick with it. For instance, it's confusing to have `fetch`, `retrieve`, and `get` as equivalent methods of different classes. How do you remember which method name goes with which class? Sadly, you often have to remember which company, group, or individual wrote the library or class in order to remember which term was used. Otherwise, you spend an awful lot of time browsing through headers and previous code samples.

Modern editing environments like Eclipse and IntelliJ provide context-sensitive clues, such as the list of methods you can call on a given object. But note that the list doesn't usually give you the comments you wrote around your function names and parameter lists. You are lucky if it gives the parameter *names* from function declarations. The function names have to stand alone, and they have to be consistent in order for you to pick the correct method without any additional exploration.

Likewise, it's confusing to have a `controller` and a `manager` and a `driver` in the same code base. What is the essential difference between a `DeviceManager` and a `ProtocolController`? Why are both not `controllers` or both not `managers`? Are they both Drivers really? The name leads you to expect two objects that have very different type as well as having different classes.

A consistent lexicon is a great boon to the programmers who must use your code.

Don't Pun

Avoid using the same word for two purposes. Using the same term for two different ideas is essentially a pun.

If you follow the "one word per concept" rule, you could end up with many classes that have, for example, an `add` method. As long as the parameter lists and return values of the various `add` methods are semantically equivalent, all is well.

However one might decide to use the word `add` for "consistency" when he or she is not in fact adding in the same sense. Let's say we have many classes where `add` will create a new value by adding or concatenating two existing values. Now let's say we are writing a new class that has a method that puts its single parameter into a collection. Should we call this method `add`? It might seem consistent because we have so many other `add` methods, but in this case, the semantics are different, so we should use a name like `insert` or `append` instead. To call the new method `add` would be a pun.

Our goal, as authors, is to make our code as easy as possible to understand. We want our code to be a quick skim, not an intense study. We want to use the popular paperback model whereby the author is responsible for making himself clear and not the academic model where it is the scholar's job to dig the meaning out of the paper.

Use Solution Domain Names

Remember that the people who read your code will be programmers. So go ahead and use computer science (CS) terms, algorithm names, pattern names, math terms, and so forth. It is not wise to draw every name from the problem domain because we don't want our coworkers to have to run back and forth to the customer asking what every name means when they already know the concept by a different name.

The name `AccountVisitor` means a great deal to a programmer who is familiar with the VISITOR pattern. What programmer would not know what a `JobQueue` was? There are lots of very technical things that programmers have to do. Choosing technical names for those things is usually the most appropriate course.

Use Problem Domain Names

When there is no "programmer-eese" for what you're doing, use the name from the problem domain. At least the programmer who maintains your code can ask a domain expert what it means.

Separating solution and problem domain concepts is part of the job of a good programmer and designer. The code that has more to do with problem domain concepts should have names drawn from the problem domain.

Add Meaningful Context

There are a few names which are meaningful in and of themselves—most are not. Instead, you need to place names in context for your reader by enclosing them in well-named classes, functions, or namespaces. When all else fails, then prefixing the name may be necessary as a last resort.

Imagine that you have variables named `firstName`, `lastName`, `street`, `houseNumber`, `city`, `state`, and `zipcode`. Taken together it's pretty clear that they form an address. But what if you just saw the `state` variable being used alone in a method? Would you automatically infer that it was part of an address?

You can add context by using prefixes: `addrFirstName`, `addrLastName`, `addrState`, and so on. At least readers will understand that these variables are part of a larger structure. Of course, a better solution is to create a class named `Address`. Then, even the compiler knows that the variables belong to a bigger concept.

Consider the method in Listing 2-1. Do the variables need a more meaningful context? The function name provides only part of the context; the algorithm provides the rest. Once you read through the function, you see that the three variables, `number`, `verb`, and `pluralModifier`, are part of the "guess statistics" message. Unfortunately, the context must be inferred. When you first look at the method, the meanings of the variables are opaque.

Listing 2-1

Variables with unclear context.

```
private void printGuessStatistics(char candidate, int count) {
    String number;
    String verb;
    String pluralModifier;
    if (count == 0) {
      number = "no";
      verb = "are";
      pluralModifier = "s";
    } else if (count == 1) {
      number = "1";
      verb = "is";
      pluralModifier = "";
    } else {
      number = Integer.toString(count);
      verb = "are";
      pluralModifier = "s";
    }
    String guessMessage = String.format(
      "There %s %s %s%s", verb, number, candidate, pluralModifier
    );
    print(guessMessage);
  }
```

The function is a bit too long and the variables are used throughout. To split the function into smaller pieces we need to create a `GuessStatisticsMessage` class and make the three variables fields of this class. This provides a clear context for the three variables. They are *definitively* part of the `GuessStatisticsMessage`. The improvement of context also allows the algorithm to be made much cleaner by breaking it into many smaller functions. (See Listing 2-2.)

Listing 2-2
Variables have a context.

```
public class GuessStatisticsMessage {
  private String number;
  private String verb;
  private String pluralModifier;

  public String make(char candidate, int count) {
    createPluralDependentMessageParts(count);
    return String.format(
      "There %s %s %s%s",
       verb, number, candidate, pluralModifier );
  }

  private void createPluralDependentMessageParts(int count) {
    if (count == 0) {
      thereAreNoLetters();
    } else if (count == 1) {
      thereIsOneLetter();
    } else {
      thereAreManyLetters(count);
    }
  }

  private void thereAreManyLetters(int count) {
    number = Integer.toString(count);
    verb = "are";
    pluralModifier = "s";
  }

  private void thereIsOneLetter() {
    number = "1";
    verb = "is";
    pluralModifier = "";
  }

  private void thereAreNoLetters() {
    number = "no";
    verb = "are";
    pluralModifier = "s";
  }
}
```

Don't Add Gratuitous Context

In an imaginary application called "Gas Station Deluxe," it is a bad idea to prefix every class with `GSD`. Frankly, you are working against your tools. You type `G` and press the completion key and are rewarded with a mile-long list of every class in the system. Is that wise? Why make it hard for the IDE to help you?

Likewise, say you invented a `MailingAddress` class in `GSD`'s accounting module, and you named it `GSDAccountAddress`. Later, you need a mailing address for your customer contact application. Do you use `GSDAccountAddress`? Does it sound like the right name? Ten of 17 characters are redundant or irrelevant.

Shorter names are generally better than longer ones, so long as they are clear. Add no more context to a name than is necessary.

The names `accountAddress` and `customerAddress` are fine names for instances of the class `Address` but could be poor names for classes. `Address` is a fine name for a class. If I need to differentiate between MAC addresses, port addresses, and Web addresses, I might consider `PostalAddress`, `MAC`, and `URI`. The resulting names are more precise, which is the point of all naming.

Final Words

The hardest thing about choosing good names is that it requires good descriptive skills and a shared cultural background. This is a teaching issue rather than a technical, business, or management issue. As a result many people in this field don't learn to do it very well.

People are also afraid of renaming things for fear that some other developers will object. We do not share that fear and find that we are actually grateful when names change (for the better). Most of the time we don't really memorize the names of classes and methods. We use the modern tools to deal with details like that so we can focus on whether the code reads like paragraphs and sentences, or at least like tables and data structure (a sentence isn't always the best way to display data). You will probably end up surprising someone when you rename, just like you might with any other code improvement. Don't let it stop you in your tracks.

Follow some of these rules and see whether you don't improve the readability of your code. If you are maintaining someone else's code, use refactoring tools to help resolve these problems. It will pay off in the short term and continue to pay in the long run.

3

Functions

In the early days of programming we composed our systems of routines and subroutines. Then, in the era of Fortran and PL/1 we composed our systems of programs, subprograms, and functions. Nowadays only the function survives from those early days. Functions are the first line of organization in any program. Writing them well is the topic of this chapter.

Unless you are a student of FitNesse, you probably don't understand all the details. Still, you probably understand that this function performs the inclusion of some setup and teardown pages into a test page and then renders that page into HTML. If you are familiar with JUnit,[2] you probably realize that this function belongs to some kind of Web-based testing framework. And, of course, that is correct. Divining that information from Listing 3-2 is pretty easy, but it's pretty well obscured by Listing 3-1.

So what is it that makes a function like Listing 3-2 easy to read and understand? How can we make a function communicate its intent? What attributes can we give our functions that will allow a casual reader to intuit the kind of program they live inside?

Small!

The first rule of functions is that they should be small. The second rule of functions is that *they should be smaller than that*. This is not an assertion that I can justify. I can't provide any references to research that shows that very small functions are better. What I can tell you is that for nearly four decades I have written functions of all different sizes. I've written several nasty 3,000-line abominations. I've written scads of functions in the 100 to 300 line range. And I've written functions that were 20 to 30 lines long. What this experience has taught me, through long trial and error, is that functions should be very small.

In the eighties we used to say that a function should be no bigger than a screen-full. Of course we said that at a time when VT100 screens were 24 lines by 80 columns, and our editors used 4 lines for administrative purposes. Nowadays with a cranked-down font and a nice big monitor, you can fit 150 characters on a line and a 100 lines or more on a screen. Lines should not be 150 characters long. Functions should not be 100 lines long. Functions should hardly ever be 20 lines long.

How short should a function be? In 1999 I went to visit Kent Beck at his home in Oregon. We sat down and did some programming together. At one point he showed me a cute little Java/Swing program that he called *Sparkle*. It produced a visual effect on the screen very similar to the magic wand of the fairy godmother in the movie Cinderella. As you moved the mouse, the sparkles would drip from the cursor with a satisfying scintillation, falling to the bottom of the window through a simulated gravitational field. When Kent showed me the code, I was struck by how small all the functions were. I was used to functions in Swing programs that took up miles of vertical space. Every function in *this* program was just two, or three, or four lines long. Each was transparently obvious. Each told a story. And each led you to the next in a compelling order. *That's* how short your functions should be![3]

2. An open-source unit-testing tool for Java. www.junit.org
3. I asked Kent whether he still had a copy, but he was unable to find one. I searched all my old computers too, but to no avail. All that is left now is my memory of that program.

How short should your functions be? They should usually be shorter than Listing 3-2! Indeed, Listing 3-2 should really be shortened to Listing 3-3.

Blocks and Indenting

This implies that the blocks within `if` statements, `else` statements, `while` statements, and so on should be one line long. Probably that line should be a function call. Not only does this keep the enclosing function small, but it also adds documentary value because the function called within the block can have a nicely descriptive name.

This also implies that functions should not be large enough to hold nested structures. Therefore, the indent level of a function should not be greater than one or two. This, of course, makes the functions easier to read and understand.

Do One Thing

It should be very clear that Listing 3-1 is doing lots more than one thing. It's creating buffers, fetching pages, searching for inherited pages, rendering paths, appending arcane strings, and generating HTML, among other things. Listing 3-1 is very busy doing lots of different things. On the other hand, Listing 3-3 is doing one simple thing. It's including setups and teardowns into test pages.

The following advice has appeared in one form or another for 30 years or more.

> ***FUNCTIONS SHOULD DO ONE THING. THEY SHOULD DO IT WELL. THEY SHOULD DO IT ONLY.***

The problem with this statement is that it is hard to know what "one thing" is. Does Listing 3-3 do one thing? It's easy to make the case that it's doing three things:

1. Determining whether the page is a test page.
2. If so, including setups and teardowns.
3. Rendering the page in HTML.

So which is it? Is the function doing one thing or three things? Notice that the three steps of the function are one level of abstraction below the stated name of the function. We can describe the function by describing it as a brief *TO*[4] paragraph:

> *TO RenderPageWithSetupsAndTeardowns, we check to see whether the page is a test page and if so, we include the setups and teardowns. In either case we render the page in HTML.*

If a function does only those steps that are one level below the stated name of the function, then the function is doing one thing. After all, the reason we write functions is to decompose a larger concept (in other words, the name of the function) into a set of steps at the next level of abstraction.

It should be very clear that Listing 3-1 contains steps at many different levels of abstraction. So it is clearly doing more than one thing. Even Listing 3-2 has two levels of abstraction, as proved by our ability to shrink it down. But it would be very hard to meaningfully shrink Listing 3-3. We could extract the `if` statement into a function named `includeSetupsAndTeardownsIfTestPage`, but that simply restates the code without changing the level of abstraction.

So, another way to know that a function is doing more than "one thing" is if you can extract another function from it with a name that is not merely a restatement of its implementation [G34].

Sections within Functions

Look at Listing 4-7 on page 71. Notice that the `generatePrimes` function is divided into sections such as *declarations*, *initializations*, and *sieve*. This is an obvious symptom of doing more than one thing. Functions that do one thing cannot be reasonably divided into sections.

One Level of Abstraction per Function

In order to make sure our functions are doing "one thing," we need to make sure that the statements within our function are all at the same level of abstraction. It is easy to see how Listing 3-1 violates this rule. There are concepts in there that are at a very high level of abstraction, such as `getHtml()`; others that are at an intermediate level of abstraction, such as: `String pagePathName = PathParser.render(pagePath)`; and still others that are remarkably low level, such as: `.append("\n")`.

Mixing levels of abstraction within a function is always confusing. Readers may not be able to tell whether a particular expression is an essential concept or a detail. Worse,

4. The LOGO language used the keyword "TO" in the same way that Ruby and Python use "def." So every function began with the word "TO." This had an interesting effect on the way functions were designed.

like broken windows, once details are mixed with essential concepts, more and more details tend to accrete within the function.

Reading Code from Top to Bottom: *The Stepdown Rule*

We want the code to read like a top-down narrative.[5] We want every function to be followed by those at the next level of abstraction so that we can read the program, descending one level of abstraction at a time as we read down the list of functions. I call this *The Stepdown Rule*.

To say this differently, we want to be able to read the program as though it were a set of *TO* paragraphs, each of which is describing the current level of abstraction and referencing subsequent *TO* paragraphs at the next level down.

> *To include the setups and teardowns, we include setups, then we include the test page content, and then we include the teardowns.*
> *To include the setups, we include the suite setup if this is a suite, then we include the regular setup.*
> *To include the suite setup, we search the parent hierarchy for the "SuiteSetUp" page and add an include statement with the path of that page.*
> *To search the parent. . .*

It turns out to be very difficult for programmers to learn to follow this rule and write functions that stay at a single level of abstraction. But learning this trick is also very important. It is the key to keeping functions short and making sure they do "one thing." Making the code read like a top-down set of *TO* paragraphs is an effective technique for keeping the abstraction level consistent.

Take a look at Listing 3-7 at the end of this chapter. It shows the whole `testableHtml` function refactored according to the principles described here. Notice how each function introduces the next, and each function remains at a consistent level of abstraction.

Switch Statements

It's hard to make a small `switch` statement.[6] Even a `switch` statement with only two cases is larger than I'd like a single block or function to be. It's also hard to make a `switch` statement that does one thing. By their nature, `switch` statements always do *N* things. Unfortunately we can't always avoid `switch` statements, but we *can* make sure that each `switch` statement is buried in a low-level class and is never repeated. We do this, of course, with polymorphism.

5. [KP78], p. 37.
6. And, of course, I include if/else chains in this.

Consider Listing 3-4. It shows just one of the operations that might depend on the type of employee.

There are several problems with this function. First, it's large, and when new employee types are added, it will grow. Second, it very clearly does more than one thing. Third, it violates the Single Responsibility Principle[7] (SRP) because there is more than one reason for it to change. Fourth, it violates the Open Closed Principle[8] (OCP) because it must change whenever new types are added. But possibly the worst problem with this function is that there are an unlimited number of other functions that will have the same structure. For example we could have

```
isPayday(Employee e, Date date),
```

or

```
deliverPay(Employee e, Money pay),
```

or a host of others. All of which would have the same deleterious structure.

The solution to this problem (see Listing 3-5) is to bury the `switch` statement in the basement of an ABSTRACT FACTORY,[9] and never let anyone see it. The factory will use the `switch` statement to create appropriate instances of the derivatives of `Employee`, and the various functions, such as `calculatePay`, `isPayday`, and `deliverPay`, will be dispatched polymorphically through the `Employee` interface.

My general rule for `switch` statements is that they can be tolerated if they appear only once, are used to create polymorphic objects, and are hidden behind an inheritance

7. a. http://en.wikipedia.org/wiki/Single_responsibility_principle
b. http://www.objectmentor.com/resources/articles/srp.pdf
8. a. http://en.wikipedia.org/wiki/Open/closed_principle
b. http://www.objectmentor.com/resources/articles/ocp.pdf
9. [GOF].

Output arguments are harder to understand than input arguments. When we read a function, we are used to the idea of information going *in* to the function through arguments and *out* through the return value. We don't usually expect information to be going out through the arguments. So output arguments often cause us to do a double-take.

One input argument is the next best thing to no arguments. `SetupTeardown-Includer.render(pageData)` is pretty easy to understand. Clearly we are going to *render* the data in the `pageData` object.

Common Monadic Forms

There are two very common reasons to pass a single argument into a function. You may be asking a question about that argument, as in `boolean fileExists("MyFile")`. Or you may be operating on that argument, transforming it into something else and *returning it*. For example, `InputStream fileOpen("MyFile")` transforms a file name `String` into an `InputStream` return value. These two uses are what readers expect when they see a function. You should choose names that make the distinction clear, and always use the two forms in a consistent context. (See Command Query Separation below.)

A somewhat less common, but still very useful form for a single argument function, is an *event*. In this form there is an input argument but no output argument. The overall program is meant to interpret the function call as an event and use the argument to alter the state of the system, for example, `void passwordAttemptFailedNtimes(int attempts)`. Use this form with care. It should be very clear to the reader that this is an event. Choose names and contexts carefully.

Try to avoid any monadic functions that don't follow these forms, for example, `void includeSetupPageInto(StringBuffer pageText)`. Using an output argument instead of a return value for a transformation is confusing. If a function is going to transform its input argument, the transformation should appear as the return value. Indeed, `StringBuffer transform(StringBuffer in)` is better than `void transform(StringBuffer out)`, even if the implementation in the first case simply returns the input argument. At least it still follows the form of a transformation.

Flag Arguments

Flag arguments are ugly. Passing a boolean into a function is a truly terrible practice. It immediately complicates the signature of the method, loudly proclaiming that this function does more than one thing. It does one thing if the flag is true and another if the flag is false!

In Listing 3-7 we had no choice because the callers were already passing that flag in, and I wanted to limit the scope of refactoring to the function and below. Still, the method call `render(true)` is just plain confusing to a poor reader. Mousing over the call and seeing `render(boolean isSuite)` helps a little, but not that much. We should have split the function into two: `renderForSuite()` and `renderForSingleTest()`.

Dyadic Functions

A function with two arguments is harder to understand than a monadic function. For example, `writeField(name)` is easier to understand than `writeField(output-Stream, name)`.[10] Though the meaning of both is clear, the first glides past the eye, easily depositing its meaning. The second requires a short pause until we learn to ignore the first parameter. And *that*, of course, eventually results in problems because we should never ignore any part of code. The parts we ignore are where the bugs will hide.

There are times, of course, where two arguments are appropriate. For example, `Point p = new Point(0,0);` is perfectly reasonable. Cartesian points naturally take two arguments. Indeed, we'd be very surprised to see `new Point(0)`. However, the two arguments in this case *are ordered components of a single value!* Whereas `output-Stream` and `name` have neither a natural cohesion, nor a natural ordering.

Even obvious dyadic functions like `assertEquals(expected, actual)` are problematic. How many times have you put the `actual` where the `expected` should be? The two arguments have no natural ordering. The `expected, actual` ordering is a convention that requires practice to learn.

Dyads aren't evil, and you will certainly have to write them. However, you should be aware that they come at a cost and should take advantage of what mechanisms may be available to you to convert them into monads. For example, you might make the `writeField` method a member of `outputStream` so that you can say `outputStream.writeField(name)`. Or you might make the `outputStream` a member variable of the current class so that you don't have to pass it. Or you might extract a new class like `FieldWriter` that takes the `outputStream` in its constructor and has a `write` method.

Triads

Functions that take three arguments are significantly harder to understand than dyads. The issues of ordering, pausing, and ignoring are more than doubled. I suggest you think very carefully before creating a triad.

For example, consider the common overload of `assertEquals` that takes three arguments: `assertEquals(message, expected, actual)`. How many times have you read the `message` and thought it was the `expected`? I have stumbled and paused over that particular triad many times. In fact, *every time I see it,* I do a double-take and then learn to ignore the message.

On the other hand, here is a triad that is not quite so insidious: `assertEquals(1.0, amount, .001)`. Although this still requires a double-take, it's one that's worth taking. It's always good to be reminded that equality of floating point values is a relative thing.

10. I just finished refactoring a module that used the dyadic form. I was able to make the `outputStream` a field of the class and convert all the `writeField` calls to the monadic form. The result was much cleaner.

Argument Objects

When a function seems to need more than two or three arguments, it is likely that some of those arguments ought to be wrapped into a class of their own. Consider, for example, the difference between the two following declarations:

```
Circle makeCircle(double x, double y, double radius);
Circle makeCircle(Point center, double radius);
```

Reducing the number of arguments by creating objects out of them may seem like cheating, but it's not. When groups of variables are passed together, the way `x` and `y` are in the example above, they are likely part of a concept that deserves a name of its own.

Argument Lists

Sometimes we want to pass a variable number of arguments into a function. Consider, for example, the `String.format` method:

```
String.format("%s worked %.2f hours.", name, hours);
```

If the variable arguments are all treated identically, as they are in the example above, then they are equivalent to a single argument of type `List`. By that reasoning, `String.format` is actually dyadic. Indeed, the declaration of `String.format` as shown below is clearly dyadic.

```
public String format(String format, Object... args)
```

So all the same rules apply. Functions that take variable arguments can be monads, dyads, or even triads. But it would be a mistake to give them more arguments than that.

```
void monad(Integer... args);
void dyad(String name, Integer... args);
void triad(String name, int count, Integer... args);
```

Verbs and Keywords

Choosing good names for a function can go a long way toward explaining the intent of the function and the order and intent of the arguments. In the case of a monad, the function and argument should form a very nice verb/noun pair. For example, `write(name)` is very evocative. Whatever this "name" thing is, it is being "written." An even better name might be `writeField(name)`, which tells us that the "name" thing is a "field."

This last is an example of the *keyword* form of a function name. Using this form we encode the names of the arguments into the function name. For example, `assertEquals` might be better written as `assertExpectedEqualsActual(expected, actual)`. This strongly mitigates the problem of having to remember the ordering of the arguments.

Have No Side Effects

Side effects are lies. Your function promises to do one thing, but it also does other *hidden* things. Sometimes it will make unexpected changes to the variables of its own class. Sometimes it will make them to the parameters passed into the function or to system globals. In either case they are devious and damaging mistruths that often result in strange temporal couplings and order dependencies.

Consider, for example, the seemingly innocuous function in Listing 3-6. This function uses a standard algorithm to match a `userName` to a `password`. It returns `true` if they match and `false` if anything goes wrong. But it also has a side effect. Can you spot it?

Listing 3-6
`UserValidator.java`

```
public class UserValidator {
  private Cryptographer cryptographer;

  public boolean checkPassword(String userName, String password) {
    User user = UserGateway.findByName(userName);
    if (user != User.NULL) {
      String codedPhrase = user.getPhraseEncodedByPassword();
      String phrase = cryptographer.decrypt(codedPhrase, password);
      if ("Valid Password".equals(phrase)) {
        Session.initialize();
        return true;
      }
    }
    return false;
  }
}
```

The side effect is the call to `Session.initialize()`, of course. The `checkPassword` function, by its name, says that it checks the password. The name does not imply that it initializes the session. So a caller who believes what the name of the function says runs the risk of erasing the existing session data when he or she decides to check the validity of the user.

This side effect creates a temporal coupling. That is, `checkPassword` can only be called at certain times (in other words, when it is safe to initialize the session). If it is called out of order, session data may be inadvertently lost. Temporal couplings are confusing, especially when hidden as a side effect. If you must have a temporal coupling, you should make it clear in the name of the function. In this case we might rename the function `checkPasswordAndInitializeSession`, though that certainly violates "Do one thing."

Output Arguments

Arguments are most naturally interpreted as *inputs* to a function. If you have been programming for more than a few years, I'm sure you've done a double-take on an argument that was actually an *output* rather than an input. For example:

```
appendFooter(s);
```

Does this function append `s` as the footer to something? Or does it append some footer to `s`? Is `s` an input or an output? It doesn't take long to look at the function signature and see:

```
public void appendFooter(StringBuffer report)
```

This clarifies the issue, but only at the expense of checking the declaration of the function. Anything that forces you to check the function signature is equivalent to a double-take. It's a cognitive break and should be avoided.

In the days before object oriented programming it was sometimes necessary to have output arguments. However, much of the need for output arguments disappears in OO languages because `this` is *intended* to act as an output argument. In other words, it would be better for `appendFooter` to be invoked as

```
report.appendFooter();
```

In general output arguments should be avoided. If your function must change the state of something, have it change the state of its owning object.

Command Query Separation

Functions should either do something or answer something, but not both. Either your function should change the state of an object, or it should return some information about that object. Doing both often leads to confusion. Consider, for example, the following function:

```
public boolean set(String attribute, String value);
```

This function sets the value of a named attribute and returns `true` if it is successful and `false` if no such attribute exists. This leads to odd statements like this:

```
if (set("username", "unclebob"))...
```

Imagine this from the point of view of the reader. What does it mean? Is it asking whether the "`username`" attribute was previously set to "`unclebob`"? Or is it asking whether the "`username`" attribute was successfully set to "`unclebob`"? It's hard to infer the meaning from the call because it's not clear whether the word "`set`" is a verb or an adjective.

The author intended `set` to be a verb, but in the context of the `if` statement it *feels* like an adjective. So the statement reads as "If the `username` attribute was previously set to `unclebob`" and not "set the `username` attribute to `unclebob` and if that worked then. . . ." We to add new errors because then they have to rebuild and redeploy everything. So they reuse old error codes instead of adding new ones.

When you use exceptions rather than error codes, then new exceptions are *derivatives* of the exception class. They can be added without forcing any recompilation or redeployment.[12]

Don't Repeat Yourself[13]

Look back at Listing 3-1 carefully and you will notice that there is an algorithm that gets repeated four times, once for each of the `SetUp`, `SuiteSetUp`, `TearDown`, and `SuiteTearDown` cases. It's not easy to spot this duplication because the four instances are intermixed with other code and aren't uniformly duplicated. Still, the duplication is a problem because it bloats the code and will require four-fold modification should the algorithm ever have to change. It is also a four-fold opportunity for an error of omission.

This duplication was remedied by the `include` method in Listing 3-7. Read through that code again and notice how the readability of the whole module is enhanced by the reduction of that duplication.

Duplication may be the root of all evil in software. Many principles and practices have been created for the purpose of controlling or eliminating it. Consider, for example, that all of Codd's database normal forms serve to eliminate duplication in data. Consider also how object-oriented programming serves to concentrate code into base classes that would otherwise be redundant. Structured programming, Aspect Oriented Programming, Component Oriented Programming, are all, in part, strategies for eliminating duplication. It would appear that since the invention of the subroutine, innovations in software development have been an ongoing attempt to eliminate duplication from our source code.

Structured Programming

Some programmers follow Edsger Dijkstra's rules of structured programming.[14] Dijkstra said that every function, and every block within a function, should have one entry and one exit. Following these rules means that there should only be one `return` statement in a function, no `break` or `continue` statements in a loop, and never, *ever*, any `goto` statements.

---

12. This is an example of the Open Closed Principle (OCP) [PPP02].
13. The DRY principle. [PRAG].
14. [SP72].

While we are sympathetic to the goals and disciplines of structured programming, those rules serve little benefit when functions are very small. It is only in larger functions that such rules provide significant benefit.

So if you keep your functions small, then the occasional multiple `return`, `break`, or `continue` statement does no harm and can sometimes even be more expressive than the single-entry, single-exit rule. On the other hand, `goto` only makes sense in large functions, so it should be avoided.

How Do You Write Functions Like This?

Writing software is like any other kind of writing. When you write a paper or an article, you get your thoughts down first, then you massage it until it reads well. The first draft might be clumsy and disorganized, so you wordsmith it and restructure it and refine it until it reads the way you want it to read.

When I write functions, they come out long and complicated. They have lots of indenting and nested loops. They have long argument lists. The names are arbitrary, and there is duplicated code. But I also have a suite of unit tests that cover every one of those clumsy lines of code.

So then I massage and refine that code, splitting out functions, changing names, eliminating duplication. I shrink the methods and reorder them. Sometimes I break out whole classes, all the while keeping the tests passing.

In the end, I wind up with functions that follow the rules I've laid down in this chapter. I don't write them that way to start. I don't think anyone could.

Conclusion

Every system is built from a domain-specific language designed by the programmers to describe that system. Functions are the verbs of that language, and classes are the nouns. This is not some throwback to the hideous old notion that the nouns and verbs in a requirements document are the first guess of the classes and functions of a system. Rather, this is a much older truth. The art of programming is, and has always been, the art of language design.

Master programmers think of systems as stories to be told rather than programs to be written. They use the facilities of their chosen programming language to construct a much richer and more expressive language that can be used to tell that story. Part of that domain-specific language is the hierarchy of functions that describe all the actions that take place within that system. In an artful act of recursion those actions are written to use the very domain-specific language they define to tell their own small part of the story.

This chapter has been about the mechanics of writing functions well. If you follow the rules herein, your functions will be short, well named, and nicely organized. But never forget that your real goal is to tell the story of the system, and that the functions you write need to fit cleanly together into a clear and precise language to help you with that telling.

SetupTeardownIncluder

Listing 3-7
SetupTeardownIncluder.java

```
package fitnesse.html;

import fitnesse.responders.run.SuiteResponder;
import fitnesse.wiki.*;

public class SetupTeardownIncluder {
  private PageData pageData;
  private boolean isSuite;
  private WikiPage testPage;
  private StringBuffer newPageContent;
  private PageCrawler pageCrawler;


  public static String render(PageData pageData) throws Exception {
    return render(pageData, false);
  }

  public static String render(PageData pageData, boolean isSuite)
    throws Exception {
    return new SetupTeardownIncluder(pageData).render(isSuite);
  }

  private SetupTeardownIncluder(PageData pageData) {
    this.pageData = pageData;
    testPage = pageData.getWikiPage();
    pageCrawler = testPage.getPageCrawler();
    newPageContent = new StringBuffer();
  }

  private String render(boolean isSuite) throws Exception {
    this.isSuite = isSuite;
    if (isTestPage())
      includeSetupAndTeardownPages();
    return pageData.getHtml();
  }

  private boolean isTestPage() throws Exception {
    return pageData.hasAttribute("Test");
  }

  private void includeSetupAndTeardownPages() throws Exception {
    includeSetupPages();
    includePageContent();
    includeTeardownPages();
    updatePageContent();
  }
```

Listing 3-7 (continued)
SetupTeardownIncluder.java

```
  private void includeSetupPages() throws Exception {
    if (isSuite)
      includeSuiteSetupPage();
    includeSetupPage();
  }

  private void includeSuiteSetupPage() throws Exception {
    include(SuiteResponder.SUITE_SETUP_NAME, "-setup");
  }

  private void includeSetupPage() throws Exception {
    include("SetUp", "-setup");
  }

  private void includePageContent() throws Exception {
    newPageContent.append(pageData.getContent());
  }

  private void includeTeardownPages() throws Exception {
    includeTeardownPage();
    if (isSuite)
      includeSuiteTeardownPage();
  }

  private void includeTeardownPage() throws Exception {
    include("TearDown", "-teardown");
  }

  private void includeSuiteTeardownPage() throws Exception {
    include(SuiteResponder.SUITE_TEARDOWN_NAME, "-teardown");
  }

  private void updatePageContent() throws Exception {
    pageData.setContent(newPageContent.toString());
  }

  private void include(String pageName, String arg) throws Exception {
    WikiPage inheritedPage = findInheritedPage(pageName);
    if (inheritedPage != null) {
      String pagePathName = getPathNameForPage(inheritedPage);
      buildIncludeDirective(pagePathName, arg);
    }
  }

  private WikiPage findInheritedPage(String pageName) throws Exception {
    return PageCrawlerImpl.getInheritedPage(pageName, testPage);
  }

  private String getPathNameForPage(WikiPage page) throws Exception {
    WikiPagePath pagePath = pageCrawler.getFullPath(page);
    return PathParser.render(pagePath);
  }

  private void buildIncludeDirective(String pagePathName, String arg) {
    newPageContent
      .append("\n!include ")
```

Bibliography

[KP78]: Kernighan and Plaugher, *The Elements of Programming Style*, 2d. ed., McGraw-Hill, 1978.

[PPP02]: Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall, 2002.

[GOF]: *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.

[PRAG]: *The Pragmatic Programmer*, Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.

[SP72]: *Structured Programming*, O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare, Academic Press, London, 1972.

Comments Do Not Make Up for Bad Code

One of the more common motivations for writing comments is bad code. We write a module and we know it is confusing and disorganized. We know it's a mess. So we say to ourselves, "Ooh, I'd better comment that!" No! You'd better clean it!

Clear and expressive code with few comments is far superior to cluttered and complex code with lots of comments. Rather than spend your time writing the comments that explain the mess you've made, spend it cleaning that mess.

Explain Yourself in Code

There are certainly times when code makes a poor vehicle for explanation. Unfortunately, many programmers have taken this to mean that code is seldom, if ever, a good means for explanation. This is patently false. Which would you rather see? This:

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) &&
    (employee.age > 65))
```

Or this?

```
if (employee.isEligibleForFullBenefits())
```

It takes only a few seconds of thought to explain most of your intent in code. In many cases it's simply a matter of creating a function that says the same thing as the comment you want to write.

Good Comments

Some comments are necessary or beneficial. We'll look at a few that I consider worthy of the bits they consume. Keep in mind, however, that the only truly good comment is the comment you found a way not to write.

Legal Comments

Sometimes our corporate coding standards force us to write certain comments for legal reasons. For example, copyright and authorship statements are necessary and reasonable things to put into a comment at the start of each source file.

Here, for example, is the standard comment header that we put at the beginning of every source file in FitNesse. I am happy to say that our IDE hides this comment from acting as clutter by automatically collapsing it.

```
// Copyright (C) 2003,2004,2005 by Object Mentor, Inc. All rights reserved.
// Released under the terms of the GNU General Public License version 2 or later.
```

Comments like this should not be contracts or legal tomes. Where possible, refer to a standard license or other external document rather than putting all the terms and conditions into the comment.

Informative Comments

It is sometimes useful to provide basic information with a comment. For example, consider this comment that explains the return value of an abstract method:

```
// Returns an instance of the Responder being tested.
protected abstract Responder responderInstance();
```

A comment like this can sometimes be useful, but it is better to use the name of the function to convey the information where possible. For example, in this case the comment could be made redundant by renaming the function: `responderBeingTested`.

Here's a case that's a bit better:

```
// format matched kk:mm:ss EEE, MMM dd, yyyy
Pattern timeMatcher = Pattern.compile(
  "\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

In this case the comment lets us know that the regular expression is intended to match a time and date that were formatted with the `SimpleDateFormat.format` function using the specified format string. Still, it might have been better, and clearer, if this code had been moved to a special class that converted the formats of dates and times. Then the comment would likely have been superfluous.

Explanation of Intent

Sometimes a comment goes beyond just useful information about the implementation and provides the intent behind a decision. In the following case we see an interesting decision documented by a comment. When comparing two objects, the author decided that he wanted to sort objects of his class higher than objects of any other.

```
public int compareTo(Object o)
{
  if(o instanceof WikiPagePath)
  {
    WikiPagePath p = (WikiPagePath) o;
    String compressedName = StringUtil.join(names, "");
    String compressedArgumentName = StringUtil.join(p.names, "");
    return compressedName.compareTo(compressedArgumentName);
  }
  return 1; // we are greater because we are the right type.
}
```

Here's an even better example. You might not agree with the programmer's solution to the problem, but at least you know what he was trying to do.

```
public void testConcurrentAddWidgets() throws Exception {
  WidgetBuilder widgetBuilder =
    new WidgetBuilder(new Class[]{BoldWidget.class});
```

```
    String text = "'''bold text'''";
    ParentWidget parent =
      new BoldWidget(new MockWidgetRoot(), "'''bold text'''");
    AtomicBoolean failFlag = new AtomicBoolean();
    failFlag.set(false);

    //This is our best attempt to get a race condition
    //by creating large number of threads.
    for (int i = 0; i < 25000; i++) {
      WidgetBuilderThread widgetBuilderThread =
        new WidgetBuilderThread(widgetBuilder, text, parent, failFlag);
      Thread thread = new Thread(widgetBuilderThread);
      thread.start();
    }
    assertEquals(false, failFlag.get());
  }
```

Clarification

Sometimes it is just helpful to translate the meaning of some obscure argument or return value into something that's readable. In general it is better to find a way to make that argument or return value clear in its own right; but when its part of the standard library, or in code that you cannot alter, then a helpful clarifying comment can be useful.

```
  public void testCompareTo() throws Exception
  {
    WikiPagePath a = PathParser.parse("PageA");
    WikiPagePath ab = PathParser.parse("PageA.PageB");
    WikiPagePath b = PathParser.parse("PageB");
    WikiPagePath aa = PathParser.parse("PageA.PageA");
    WikiPagePath bb = PathParser.parse("PageB.PageB");
    WikiPagePath ba = PathParser.parse("PageB.PageA");

    assertTrue(a.compareTo(a) == 0);    // a == a
    assertTrue(a.compareTo(b) != 0);    // a != b
    assertTrue(ab.compareTo(ab) == 0);  // ab == ab
    assertTrue(a.compareTo(b) == -1);   // a < b
    assertTrue(aa.compareTo(ab) == -1); // aa < ab
    assertTrue(ba.compareTo(bb) == -1); // ba < bb
    assertTrue(b.compareTo(a) == 1);    // b > a
    assertTrue(ab.compareTo(aa) == 1);  // ab > aa
    assertTrue(bb.compareTo(ba) == 1);  // bb > ba
  }
```

There is a substantial risk, of course, that a clarifying comment is incorrect. Go through the previous example and see how difficult it is to verify that they are correct. This explains both why the clarification is necessary and why it's risky. So before writing comments like this, take care that there is no better way, and then take even more care that they are accurate.