

Robert C. Martin Series

# Clean Code

A Handbook of Agile Software Craftsmanship



Foreword by James O. Coplien

Robert C. Martin

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

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

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

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

For sales outside the United States please contact:

International Sales  
international@pearsoned.com

---



### This Book Is Safari Enabled

The Safari® Enabled icon on the cover of your favorite technology book means the book is available through Safari Bookshelf. When you buy this book, you get free access to the online edition for 45 days.

Safari Bookshelf is an electronic reference library that lets you easily search thousands of technical books, find code samples, download chapters, and access technical information whenever and wherever you need it.

To gain 45-day Safari Enabled access to this book:

- Go to [informit.com/onlineedition](http://informit.com/onlineedition)
- Complete the brief registration form
- Enter the coupon code EFNC-FMRP-VH3R-F7PF-RAN1

If you have difficulty registering on Safari Bookshelf or accessing the online edition, please e-mail [customer-service@safaribooksonline.com](mailto:customer-service@safaribooksonline.com).

---

Visit us on the Web: [informit.com/ph](http://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

Copyright © 2009 Pearson Education, Inc.

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

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

ISBN-13: 978-0-13-235088-4

ISBN-10: 0-13-235088-2

Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.

First printing July, 2008

---

# 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**

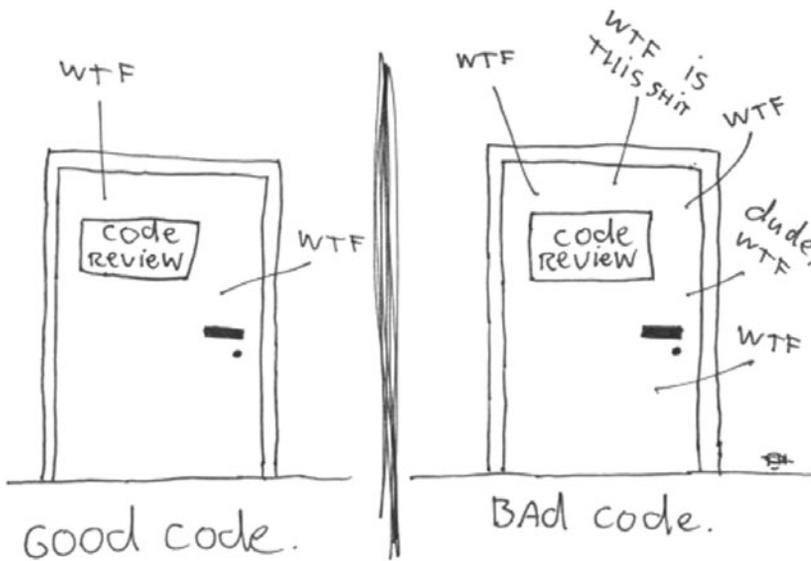
Mørdrup, Denmark

---

# Introduction

---

The ONLY VALID MEASUREMENT  
OF CODE QUALITY: WTFs/MINUTE



(c) 2008 Focus Shift

Reproduced with the kind permission of Thom Holwerda.  
[http://www.osnews.com/story/19266/WTFs\\_m](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 describes 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

### Artwork

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 Development: Principles, Patterns, and Practices*. She is also my firstborn in whom I am well pleased.

---

# 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*.<sup>1</sup> 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



---

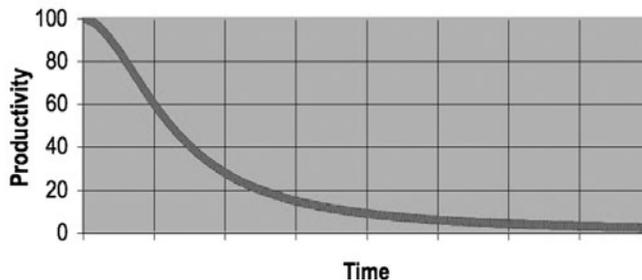
1. [Beck07].

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 green-field 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?<sup>2</sup> 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.<sup>3</sup> 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 handling 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*.<sup>4</sup> 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#***

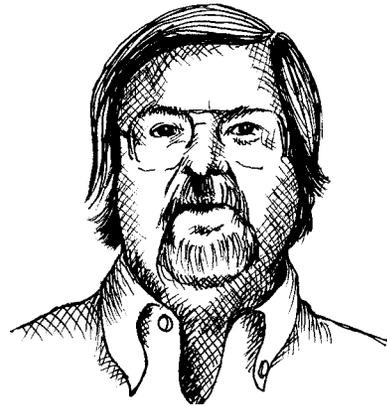
Ron began his career programming in Fortran at the Strategic Air Command and has written code in almost every language and on almost every machine. It pays to consider his words carefully.

*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?

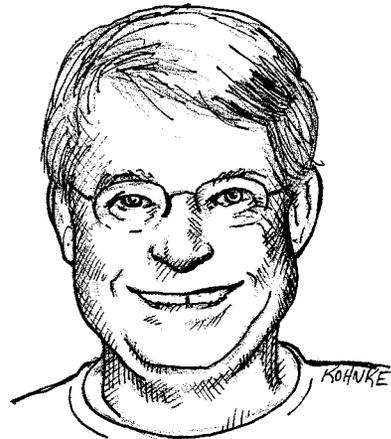
Ward expects that when you read clean code you won’t be surprised at all. Indeed, you won’t even expend much effort. You will read it, and it will be pretty much what you expected. It will be obvious, simple, and compelling. Each module will set the stage for the next. Each tells you how the next will 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 Jitsu*, founded and taught by the Gracie family in Brazil. We see *Hakkoryu Jiu Jitsu*, 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.*<sup>5</sup>

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.

---

# Index

---

- ## detection, 237–238
- ++ (pre- or post-increment) operator, 325, 326
- ## A
- aborted computation, 109
- abstract classes, 149, 271, 290
- ABSTRACT FACTORY pattern, 38, 156, 273, 274
- abstract interfaces, 94
- abstract methods
  - adding to *ArgumentMarshaler*, 234–235
  - modifying, 282
- abstract terms, 95
- abstraction
  - classes depending on, 150
  - code at wrong level of, 290–291
  - descending one level at a time, 37
  - functions descending only one level of, 304–306
  - mixing levels of, 36–37
  - names at the appropriate level of, 311
  - separating levels of, 305
  - wrapping an implementation, 11
- abstraction levels
  - raising, 290
  - separating, 305
- accessor functions, Law of Demeter and, 98
- accessors, naming, 25
- Active Records, 101
- adapted server, 185
- affinity, 84
- Agile Software Development: Principles, Patterns, Practices (PPP)*, 15
- algorithms
  - correcting, 269–270
  - repeating, 48
  - understanding, 297–298
- ambiguities
  - in code, 301
  - ignored tests as, 313
- amplification comments, 59
- analysis functions, 265
- “annotation form”, of AspectJ, 166
- Ant project, 76, 77
- AOP (aspect-oriented programming), 160, 163
- APIs. *See also* public APIs
  - calling a `null`-returning method from, 110
  - specialized for tests, 127
  - wrapping third-party, 108
- applications
  - decoupled from Spring, 164
  - decoupling from construction
    - details, 156
  - infrastructure of, 163
  - keeping concurrency-related code separate, 181
- arbitrary structure, 303–304
- `args` array, converting into a list, 231–232
- `Args` class
  - constructing, 194
  - implementation of, 194–200
  - rough drafts of, 201–212, 226–231

- ArgsException class
    - listing, 198–200
    - merging exceptions into, 239–242
  - argument(s)
    - flag, 41
    - for a function, 40
    - in functions, 288
    - monadic forms of, 41
    - reducing, 43
  - argument lists, 43
  - argument objects, 43
  - argument types
    - adding, 200, 237
    - negative impact of, 208
  - ArgumentMarshaler class
    - adding the skeleton of, 213–214
    - birth of, 212
  - ArgumentMarshaler interface, 197–198
  - arrays, moving, 279
  - art, of clean code, 6–7
  - artificial coupling, 293
  - AspectJ language, 166
  - aspect-oriented programming (AOP), 160, 163
  - aspects
    - in AOP, 160–161
    - “first-class” support for, 166
  - assert statements, 130–131
  - assertEquals, 42
  - assertions, using a set of, 111
  - assignments, unaligned, 87–88
  - atomic operation, 323–324
  - attributes, 68
  - authors
    - of JUnit, 252
    - programmers as, 13–14
  - authorship statements, 55
  - automated code instrumentation, 189–190
  - automated suite, of unit tests, 124
- B**
- bad code, 3–4. *See also* dirty code; messy code
    - degrading effect of, 250
    - example, 71–72
    - experience of cleaning, 250
    - not making up for, 55
  - bad comments, 59–74
  - banner, gathering functions beneath, 67
  - base classes, 290, 291
  - BDUF (Big Design Up Front), 167
  - beans, private variables manipulated, 100–101
  - Beck, Kent, 3, 34, 71, 171, 252, 289, 296
  - behaviors, 288–289
  - Big Design Up Front (BDUF), 167
  - blank lines, in code, 78–79
  - blocks, calling functions within, 35
  - Booch, Grady, 8–9
  - boolean, passing into a function, 41
  - boolean arguments, 194, 288
  - boolean map, deleting, 224
  - boolean output, of tests, 132
  - bound resources, 183, 184
  - boundaries
    - clean, 120
    - exploring and learning, 116
    - incorrect behavior at, 289
    - separating known from unknown, 118–119
  - boundary condition errors, 269
  - boundary conditions
    - encapsulating, 304
    - testing, 314
  - boundary tests, easing a migration, 118
  - “Bowling Game”, 312
  - Boy Scout Rule, 14–15, 257
    - following, 284
    - satisfying, 265
  - broken windows metaphor, 8
  - bucket brigade, 303
  - BUILD-OPERATE-CHECK pattern, 127
  - builds, 287
  - business logic, separating from error handling, 109
  - bylines, 68
  - byte-manipulation libraries, 161, 162–163

## C

*The C++ Programming Language*, 7

calculations, breaking into intermediate values, 296

call stack, 324

Callable interface, 326

caller, cluttering, 104

calling hierarchy, 106

calls, avoiding chains of, 98

caring, for code, 10

Cartesian points, 42

CAS operation, as atomic, 328

change(s)

isolating from, 149–150

large number of very tiny, 213

organizing for, 147–150

tests enabling, 124

change history, deleting, 270

check exceptions, in Java, 106

circular wait, 337, 338–339

clarification, comments as, 57

clarity, 25, 26

class names, 25

classes

cohesion of, 140–141

creating for bigger concepts, 28–29

declaring instance variables, 81

enforcing design and business rules, 115

exposing internals of, 294

instrumenting into ConTest, 342

keeping small, 136, 175

minimizing the number of, 176

naming, 25, 138

nonthread-safe, 328–329

as nouns of a language, 49

organization of, 136

organizing to reduce risk of change, 147

supporting advanced concurrency design, 183

classification, of errors, 107

clean boundaries, 120

clean code

art of, 6–7

described, 7–12

writing, 6–7

clean tests, 124–127

cleanliness

acquired sense of, 6–7

tied to tests, 9

cleanup, of code, 14–15

clever names, 26

client, using two methods, 330

client code, connecting to a server, 318

client-based locking, 185, 329, 330–332

clientScheduler, 320

client/server application, concurrency in, 317–321

Client/Server nonthreaded, code for, 343–346

client-server using threads, code changes, 346–347

ClientTest.java, 318, 344–346

closing braces, comments on, 67–68

Clover, 268, 269

clutter

Javadocs as, 276

keeping free of, 293

code, 2

bad, 3–4

Beck's rules of, 10

commented-out, 68–69, 287

dead, 292

explaining yourself in, 55

expressing yourself in, 54

formatting of, 76

implicity of, 18–19

instrumenting, 188, 342

jiggling, 190

making readable, 311

necessity of, 2

reading from top to bottom, 37

simplicity of, 18, 19

technique for shrouding, 20

- code, *continued*
  - third-party, 114–115
  - width of lines in, 85–90
  - at wrong level of abstraction, 290–291
- code bases, dominated by error
  - handling, 103
- code changes, comments not always
  - following, 54
- code completion, automatic, 20
- code coverage analysis, 254–256
- code instrumentation, 188–190
- “code sense”, 6, 7
- code smells, listing of, 285–314
- coding standard, 299
- cohesion
  - of classes, 140–141
  - maintaining, 141–146
- command line arguments, 193–194
- commands, separating from queries, 45–46
- comment header standard, 55–56
- comment headers, replacing, 70
- commented-out code, 68–69, 287
- commenting style, example of bad, 71–72
- comments
  - amplifying importance of
    - something, 59
  - bad, 59–74
  - deleting, 282
  - as failures, 54
  - good, 55–59
  - heuristics on, 286–287
  - HTML, 69
  - inaccurate, 54
  - informative, 56
  - journal, 63–64
  - legal, 55–56
  - mandated, 63
  - misleading, 63
  - mumbling, 59–60
  - as a necessary evil, 53–59
  - noise, 64–66
  - not making up for bad code, 55
  - obsolete, 286
  - poorly written, 287
  - proper use of, 54
  - redundant, 60–62, 272, 275, 286–287
  - restating the obvious, 64
  - separated from code, 54
  - `TODO`, 58–59
  - too much information in, 70
  - venting in, 65
  - writing, 287
- “communication gap”, minimizing, 168
- Compare and Swap (CAS) operation,
  - 327–328
- `ComparisonCompactor` module, 252–265
  - defactored, 256–261
  - final, 263–265
  - interim, 261–263
  - original code, 254–256
- compiler warnings, turning off, 289
- complex code, demonstrating failures
  - in, 341
- complexity, managing, 139–140
- computer science (CS) terms, using for
  - names, 27
- concepts
  - keeping close to each other, 80
  - naming, 19
  - one word per, 26
  - separating at different levels, 290
  - spelling similar similarly, 20
  - vertical openness between, 78–79
- conceptual affinity, of code, 84
- concerns
  - cross-cutting, 160–161
  - separating, 154, 166, 178, 250
- concrete classes, 149
- concrete details, 149
- concrete terms, 94
- concurrency
  - defense principles, 180–182
  - issues, 190
  - motives for adopting, 178–179
  - myths and misconceptions about,
    - 179–180
- concurrency code
  - compared to nonconcurrency-related
    - code, 181
  - focusing, 321

- concurrent algorithms, 179
  - concurrent applications, partition
    - behavior, 183
  - concurrent code
    - breaking, 329–333
    - defending from problems of, 180
    - flaws hiding in, 188
  - concurrent programming, 180
  - Concurrent Programming in Java: Design Principles and Patterns*, 182, 342
  - concurrent programs, 178
  - concurrent update problems, 341
  - `ConcurrentHashMap` implementation, 183
  - conditionals
    - avoiding negative, 302
    - encapsulating, 257–258, 301
  - configurable data, 306
  - configuration constants, 306
  - consequences, warning of, 58
  - consistency
    - in code, 292
    - of enums, 278
    - in names, 40
  - consistent conventions, 259
  - constants
    - versus `enums`, 308–309
    - hiding, 308
    - inheriting, 271, 307–308
    - keeping at the appropriate level, 83
    - leaving as raw numbers, 300
    - not inheriting, 307–308
    - passing as symbols, 276
    - turning into `enums`, 275–276
  - construction
    - moving all to `main`, 155, 156
    - separating with factory, 156
    - of a system, 154
  - constructor arguments, 157
  - constructors, overloading, 25
  - consumer threads, 184
  - ConTest tool, 190, 342
  - context
    - adding meaningful, 27–29
    - not adding gratuitous, 29–30
    - providing with exceptions, 107
  - continuous readers, 184
  - control variables, within loop statements,
    - 80–81
  - convenient idioms, 155
  - convention(s)
    - following standard, 299–300
    - over configuration, 164
    - structure over, 301
    - using consistent, 259
  - convoluted code, 175
  - copyright statements, 55
  - cosmic-rays. *See* one-offs
  - `CountDownLatch` class, 183
  - coupling. *See also* decoupling; temporal
    - coupling; tight coupling
      - artificial, 293
      - hidden temporal, 302–303
      - lack of, 150
  - coverage patterns, testing, 314
  - coverage tools, 313
  - “crisp abstraction”, 8–9
  - cross-cutting concerns, 160
  - Cunningham, Ward, 11–12
  - cuteness, in code, 26
- ## D
- dangling `false` argument, 294
  - data
    - abstraction, 93–95
    - copies of, 181–182
    - encapsulation, 181
    - limiting the scope of, 181
    - sets processed in parallel, 179
    - types, 97, 101
  - data structures. *See also* structure(s)
    - compared to objects, 95, 97
    - defined, 95
    - interfaces representing, 94
    - treating Active Records as, 101
  - data transfer-objects (DTOs),
    - 100–101, 160
  - database normal forms, 48
  - `DateInterval` enum, 282–283
  - `DAY` enumeration, 277

- DayDate class, running SerialDate as, 271
- DayDateFactory, 273–274
- dead code, 288, 292
- dead functions, 288
- deadlock, 183, 335–339
- deadly embrace. *See* circular wait
- debugging, finding deadlocks, 336
- decision making, optimizing, 167–168
- decisions, postponing, 168
- declarations, unaligned, 87–88
- DECORATOR objects, 164
- DECORATOR pattern, 274
- decoupled architecture, 167
- decoupling, from construction
  - details, 156
- decoupling strategy, concurrency as, 178
- default constructor, deleting, 276
- degradation, preventing, 14
- deletions, as the majority of changes, 250
- density, vertical in code, 79–80
- dependencies
  - finding and breaking, 250
  - injecting, 157
  - logical, 282
  - making logical physical, 298–299
  - between methods, 329–333
  - between synchronized methods, 185
- Dependency Injection (DI), 157
- Dependency Inversion Principle (DIP), 15, 150
- dependency magnet, 47
- dependent functions, formatting, 82–83
- derivatives
  - base classes depending on, 291
  - base classes knowing about, 273
  - of the exception class, 48
  - moving set functions into, 232, 233–235
  - pushing functionality into, 217
- description
  - of a class, 138
  - overloading the structure of code into, 310
- descriptive names
  - choosing, 309–310
  - using, 39–40
- design(s)
  - of concurrent algorithms, 179
  - minimally coupled, 167
  - principles of, 15
- design patterns, 290
- details, paying attention to, 8
- DI (Dependency Injection), 157
- Dijkstra, Edsger, 48
- dining philosophers execution model, 184–185
- DIP (Dependency Inversion Principle), 15, 150
- dirty code. *See also* bad code; messy code
- dirty code, cleaning, 200
- dirty tests, 123
- disinformation, avoiding, 19–20
- distance, vertical in code, 80–84
- distinctions, making meaningful, 20–21
- domain-specific languages (DSLs), 168–169
- domain-specific testing language, 127
- DoubleArgumentMarshaler class, 238
- DRY principle (Don't Repeat Yourself), 181, 289
- DTOs (data transfer objects), 100–101, 160
- dummy scopes, 90
- duplicate if statements, 276
- duplication
  - of code, 48
  - in code, 289–290
  - eliminating, 173–175
  - focusing on, 10
  - forms of, 173, 290
  - reduction of, 48
  - strategies for eliminating, 48

dyadic argument, 40  
 dyadic functions, 42  
 dynamic proxies, 161

## E

e, as a variable name, 22  
 Eclipse, 26  
 edit sessions, playing back, 13–14  
 efficiency, of code, 7  
 EJB architecture, early as over-engineered, 167  
 EJB standard, complete overhaul of, 164  
 EJB2 beans, 160  
 EJB3, Bank object rewritten in, 165–166  
 “elegant” code, 7  
 emergent design, 171–176  
 encapsulation, 136
 

- of boundary conditions, 304
- breaking, 106–107
- of conditionals, 301

 encodings, avoiding, 23–24, 312–313  
 entity bean, 158–160  
 enum(s)
 

- changing `MonthConstants` to, 272
- using, 308–309

 enumeration, moving, 277  
 environment, heuristics on, 287  
 environment control system, 128–129  
 envying, the scope of a class, 293  
 error check, hiding a side effect, 258  
`Error` class, 47–48  
 error code constants, 198–200  
 error codes
 

- implying a class or enum, 47–48
- preferring exceptions to, 46
- returning, 103–104
- reusing old, 48
- separating from the `Args` module, 242–250

 error detection, pushing to the edges, 109  
 error flags, 103–104  
 error handling, 8, 47–48

error messages, 107, 250  
 error processing, testing, 238–239  
`errorMessage` method, 250  
 errors. *See also* boundary condition errors; spelling errors; string comparison errors
 

- classifying, 107

 Evans, Eric, 311  
 events, 41  
 exception classification, 107  
 exception clauses, 107–108  
 exception management code, 223  
 exceptions
 

- instead of return codes, 103–105
- narrowing the type of, 105–106
- preferring to error codes, 46
- providing context with, 107
- separating from `Args`, 242–250
- throwing, 104–105, 194
- unchecked, 106–107

 execution, possible paths of, 321–326  
 execution models, 183–185  
`Executor` framework, 326–327  
`ExecutorClientScheduler.java`, 321  
 explanation, of intent, 56–57  
 explanatory variables, 296–297  
 explicitness, of code, 19  
 expressive code, 295  
 expressiveness
 

- in code, 10–11
- ensuring, 175–176

 Extract Method refactoring, 11  
*Extreme Programming Adventures in C#*, 10  
*Extreme Programming Installed*, 10  
 “eye-full”, code fitting into, 79–80

## F

factories, 155–156  
 factory classes, 273–275  
 failure
 

- to express ourselves in code, 54

- failure, *continued*
    - patterns of, 314
    - tolerating with no harm, 330
  - false argument, 294
  - fast tests, 132
  - fast-running threads, starving longer
    - running, 183
  - fear, of renaming, 30
  - Feathers, Michael, 10
  - feature envy
    - eliminating, 293–294
    - smelling of, 278
  - file size, in Java, 76
  - final keywords, 276
  - F.I.R.S.T. acronym, 132–133
  - First Law, of TDD, 122
  - FitNesse project
    - coding style for, 90
    - file sizes, 76, 77
    - function in, 32–33
    - invoking all tests, 224
  - flag arguments, 41, 288
  - focussed code, 8
  - foreign code. *See* third-party code
  - formatting
    - horizontal, 85–90
    - purpose of, 76
    - Uncle Bob’s rules, 90–92
    - vertical, 76–85
  - formatting style, for a team of
    - developers, 90
  - Fortran, forcing encodings, 23
  - Fowler, Martin, 285, 293
  - frame, 324
  - function arguments, 40–45
  - function call dependencies, 84–85
  - function headers, 70
  - function signature, 45
  - functionality, placement of, 295–296
  - functions
    - breaking into smaller, 141–146
    - calling within a block, 35
    - dead, 288
    - defining private, 292
    - descending one level of abstraction, 304–306
    - doing one thing, 35–36, 302
    - dyadic, 42
    - eliminating extraneous if statements, 262
    - establishing the temporal nature
      - of, 260
    - formatting dependent, 82–83
    - gathering beneath a banner, 67
    - heuristics on, 288
    - intention-revealing, 19
    - keeping small, 175
    - length of, 34–35
    - moving, 279
    - naming, 39, 297
    - number of arguments in, 288
    - one level of abstraction per, 36–37
    - in place of comments, 67
    - renaming for clarity, 258
    - rewriting for clarity, 258–259
    - sections within, 36
    - small as better, 34
    - structured programming with, 49
    - understanding, 297–298
    - as verbs of a language, 49
    - writing, 49
  - futures, 326
- ## G
- Gamma, Eric, 252
  - general heuristics, 288–307
  - generated byte-code, 180
  - generics, improving code readability, 115
  - get functions, 218
  - getBoolean function, 224
  - GETFIELD instruction, 325, 326
  - getNextId method, 326
  - getState function, 129
  - Gilbert, David, 267, 268
  - given-when-then convention, 130
  - glitches. *See* one-offs

global setup strategy, 155  
 “God class”, 136–137  
 good comments, 55–59  
 goto statements, avoiding, 48, 49  
 grand redesign, 5  
 gratuitous context, 29–30

## H

hand-coded instrumentation, 189  
`HashTable`, 328–329  
 headers. *See* comment headers; function headers  
 heuristics
 

- cross references of, 286, 409
- general, 288–307
- listing of, 285–314

 hidden temporal coupling, 259, 302–303  
 hidden things, in a function, 44  
 hiding
 

- implementation, 94
- structures, 99

 hierarchy of scopes, 88  
 HN. *See* Hungarian Notation  
 horizontal alignment, of code, 87–88  
 horizontal formatting, 85–90  
 horizontal white space, 86  
 HTML, in source code, 69  
 Hungarian Notation (HN), 23–24, 295  
 Hunt, Andy, 8, 289  
 hybrid structures, 99

## I

if statements
 

- duplicate, 276
- eliminating, 262

 if-else chain
 

- appearing again and again, 290
- eliminating, 233

 ignored tests, 313  
 implementation
 

- duplication of, 173
- encoding, 24

- exposing, 94
- hiding, 94
  - wrapping an abstraction, 11

*Implementation Patterns*, 3, 296  
 implicitness, of code, 18  
 import lists
 

- avoiding long, 307
- shortening in `SerialDate`, 270

 imports, as hard dependencies, 307  
 imprecision, in code, 301  
 inaccurate comments, 54  
 inappropriate information, in
 

- comments, 286

 inappropriate static methods, 296  
 include method, 48  
 inconsistency, in code, 292  
 inconsistent spellings, 20  
 incrementalism, 212–214  
 indent level, of a function, 35  
 indentation, of code, 88–89  
 indentation rules, 89  
 independent tests, 132  
 information
 

- inappropriate, 286
- too much, 70, 291–292

 informative comments, 56  
 inheritance hierarchy, 308  
 inobvious connection, between a comment and code, 70  
 input arguments, 41  
 instance variables
 

- in classes, 140
- declaring, 81
- hiding the declaration of, 81–82
- passing as function arguments, 231
- proliferation of, 140

 instrumented classes, 342  
 insufficient tests, 313  
 integer argument(s)
 

- defining, 194
- integrating, 224–225

 integer argument functionality,
 

- moving into `ArgumentMarshaler`, 215–216

- integer argument type, adding
  - to `Args`, 212
- integers, pattern of changes for, 220
- IntelliJ, 26
- intent
  - explaining in code, 55
  - explanation of, 56–57
  - obscured, 295
- intention-revealing function, 19
- intention-revealing names, 18–19
- interface(s)
  - defining local or remote, 158–160
  - encoding, 24
  - implementing, 149–150
  - representing abstract concerns, 150
  - turning `ArgumentMarshaler` into, 237
  - well-defined, 291–292
  - writing, 119
- internal structures, objects hiding, 97
- intersection, of domains, 160
- intuition, not relying on, 289
- inventor of C++, 7
- Inversion of Control (IoC), 157
- `InvocationHandler` object, 162
- I/O bound, 318
- isolating, from change, 149–150
- `isxxxArg` methods, 221–222
- iterative process, refactoring as, 265

## J

- jar files, deploying derivatives and bases
  - in, 291
- Java
  - aspects or aspect-like mechanisms, 161–166
  - heuristics on, 307–309
  - as a wordy language, 200
- Java 5, improvements for concurrent
  - development, 182–183
- Java 5 Executor framework, 320–321
- Java 5 VM, nonblocking solutions in, 327–328
- Java AOP frameworks, 163–166

- Java programmers, encoding not
  - needed, 24
- Java proxies, 161–163
- Java source files, 76–77
- javadocs
  - as clutter, 276
  - in nonpublic code, 71
  - preserving formatting in, 270
  - in public APIs, 59
  - requiring for every function, 63
- `java.util.concurrent` package, collections
  - in, 182–183
- JBoss AOP, proxies in, 163
- JCommon library, 267
- JCommon unit tests, 270
- JDepend project, 76, 77
- JDK proxy, providing persistence support, 161–163
- Jeffries, Ron, 10–11, 289
- jiggling strategies, 190
- JNDI lookups, 157
- journal comments, 63–64
- JUnit, 34
- JUnit framework, 252–265
- JUnit project, 76, 77
- Just-In-Time Compiler, 180

## K

- keyword form, of a function name, 43

## L

- L, lower-case in variable names, 20
- language design, art of programming as, 49
- languages
  - appearing to be simple, 12
  - level of abstraction, 2
  - multiple in one source file, 288
  - multiples in a comment, 270
- last-in, first-out (LIFO) data structure,
  - operand stack as, 324
- Law of Demeter, 97–98, 306

LAZY INITIALIZATION/  
 EVALUATION idiom, 154  
 LAZY-INITIALIZATION, 157  
 Lea, Doug, 182, 342  
 learning tests, 116, 118  
 LeBlanc's law, 4  
 legacy code, 307  
 legal comments, 55–56  
 level of abstraction, 36–37  
 levels of detail, 99  
 lexicon, having a consistent, 26  
 lines of code  
   duplicating, 173  
   width of, 85  
 list(s)  
   of arguments, 43  
   meaning specific to programmers, 19  
   returning a predefined immutable, 110  
 literate code, 9  
 literate programming, 9  
*Literate Programming*, 141  
 livelock, 183, 338  
 local comments, 69–70  
 local variables, 324  
   declaring, 292  
   at the top of each function, 80  
 lock & wait, 337, 338  
 locks, introducing, 185  
 log4j package, 116–118  
 logical dependencies, 282, 298–299  
 LOGO language, 36  
 long descriptive names, 39  
 long names, for long scopes, 312  
 loop counters, single-letter names for, 25

## M

magic numbers  
   obscuring intent, 295  
   replacing with named constants,  
   300–301  
 main function, moving construction to,  
   155, 156  
 managers, role of, 6  
 mandated comments, 63  
 manual control, over a serial ID, 272  
 Map  
   adding for `ArgumentMarshaler`, 221  
   methods of, 114  
 maps, breaking the use of, 222–223  
 marshalling implementation,  
   214–215  
 meaningful context, 27–29  
 member variables  
   f prefix for, 257  
   prefixing, 24  
   renaming for clarity, 259  
 mental mapping, avoiding, 25  
 messy code. *See also* bad code; dirty code  
   total cost of owning, 4–12  
 method invocations, 324  
 method names, 25  
 methods  
   affecting the order of execution, 188  
   calling a twin with a flag, 278  
   changing from static to instance, 280  
   of classes, 140  
   dependencies between, 329–333  
   eliminating duplication between,  
   173–174  
   minimizing assert statements in, 176  
   naming, 25  
   tests exposing bugs in, 269  
 minimal code, 9  
 misleading comments, 63  
 misplaced responsibility, 295–296, 299  
 MOCK OBJECT, assigning, 155  
 monadic argument, 40  
 monadic forms, of arguments, 41  
 monads, converting dyads into, 42  
 Monte Carlo testing, 341  
 Month enum, 278  
 MonthConstants class, 271  
 multithread aware, 332  
 multithread-calculation, of throughput,  
   335

multithreaded code, 188, 339–342  
 mumbling, 59–60  
 mutators, naming, 25  
 mutual exclusion, 183, 336, 337

## N

named constants, replacing magic numbers, 300–301  
 name-length-challenged languages, 23  
 names  
   abstractions, appropriate level of, 311  
   changing, 40  
   choosing, 175, 309–310  
   of classes, 270–271  
   clever, 26  
   descriptive, 39–40  
   of functions, 297  
   heuristics on, 309–313  
   importance of, 309–310  
   intention-revealing, 18–19  
   length of corresponding to scope, 22–23  
   long names for long scopes, 312  
   making unambiguous, 258  
   problem domain, 27  
   pronounceable, 21–22  
   rules for creating, 18–30  
   searchable, 22–23  
   shorter generally better than longer, 30  
   solution domain, 27  
   with subtle differences, 20  
   unambiguous, 312  
   at the wrong level of abstraction, 271  
 naming, classes, 138  
 naming conventions, as inferior to structures, 301  
 navigational methods, in Active Records, 101  
 near bugs, testing, 314  
 negative conditionals, avoiding, 302  
 negatives, 258  
 nested structures, 46  
 Newkirk, Jim, 116  
 newspaper metaphor, 77–78  
 niladic argument, 40  
 no preemption, 337  
 noise  
   comments, 64–66  
   scary, 66  
   words, 21  
 nomenclature, using standard, 311–312  
 nonblocking solutions, 327–328  
 nonconcurrency-related code, 181  
 noninformative names, 21  
 nonlocal information, 69–70  
 nonpublic code, javadocs in, 71  
 nonstatic methods, preferred to static, 296  
 nonthreaded code, getting working  
   first, 187  
 nonthread-safe classes, 328–329  
 normal flow, 109  
 null  
   not passing into methods, 111–112  
   not returning, 109–110  
   passed by a caller accidentally, 111  
 null detection logic, for `ArgumentMarshaler`, 214  
`NullPointerException`, 110, 111  
 number-series naming, 21

## O

*Object Oriented Analysis and Design with Applications*, 8  
 object-oriented design, 15  
 objects  
   compared to data structures, 95, 97  
   compared to data types and procedures, 101  
   copying read-only, 181  
   defined, 95  
 obscured intent, 295  
 obsolete comments, 286  
 obvious behavior, 288–289  
 obvious code, 12

- “Once and only once” principle, 289
- “ONE SWITCH” rule, 299
- one thing, functions doing, 35–36, 302
- one-offs, 180, 187, 191
- OO code, 97
- OO design, 139
- Open Closed Principle (OCP), 15, 38
  - by checked exceptions, 106
  - supporting, 149
- operand stack, 324
- operating systems, threading policies, 188
- operators, precedence of, 86
- optimistic locking, 327
- optimizations, LAZY-EVALUATION
  - as, 157
- optimizing, decision making, 167–168
- orderings, calculating the possible, 322–323
- organization
  - for change, 147–150
  - of classes, 136
  - managing complexity, 139–140
- outbound tests, exercising an interface, 118
- output arguments, 41, 288
  - avoiding, 45
  - need for disappearing, 45
- outputs, arguments as, 45
- overhead, incurred by concurrency, 179
- overloading, of code with description, 310

## P

- paperback model, as an academic
  - model, 27
- parameters, taken by instructions, 324
- parse operation, throwing an
  - exception, 220
- partitioning, 250
- paths of execution, 321–326
- pathways, through critical sections, 188
- pattern names, using standard, 175
- patterns
  - of failure, 314
  - as one kind of standard, 311
- performance
  - of a client/server pair, 318
  - concurrency improving, 179
  - of server-based locking, 333
- permutations, calculating, 323
- persistence, 160, 161
- pessimistic locking, 327
- phraseology, in similar names, 40
- physicalizing, a dependency, 299
- Plain-Old Java Objects. *See* POJOs
- platforms, running threaded code, 188
- pleasing code, 7
- pluggable thread-based code, 187
- POJO system, agility provided by, 168
- POJOs (Plain-Old Java Objects)
  - creating, 187
  - implementing business logic, 162
  - separating threaded-aware code, 190
  - in Spring, 163
  - writing application domain logic, 166
- polyadic argument, 40
- polymorphic behavior, of functions, 296
- polymorphic changes, 96–97
- polymorphism, 37, 299
- position markers, 67
- positives
  - as easier to understand, 258
  - expressing conditionals as, 302
  - of decisions, 301
  - as the point of all naming, 30
- predicates, naming, 25
- preemption, breaking, 338
- prefixes
  - for member variables, 24
  - as useless in today’s environments, 312–313
- pre-increment operator, ++, 324, 325, 326
- “prequel”, this book as, 15
- principle of least surprise, 288–289, 295
- principles, of design, 15
- `PrintPrimes` program, translation into
  - Java, 141
- private behavior, isolating, 148–149

private functions, 292  
 private method behavior, 147  
 problem domain names, 27  
 procedural code, 97  
 procedural shape example, 95–96  
 procedures, compared to objects, 101  
 process function, repartitioning, 319–320  
`process` method, I/O bound, 319  
 processes, competing for resources, 184  
 processor bound, code as, 318  
 producer consumer execution model, 184  
 producer threads, 184  
 production environment, 127–130  
 productivity, decreased by messy code, 4  
 professional programmer, 25  
 professional review, of code, 268  
 programmers  
   as authors, 13–14  
   conundrum faced by, 6  
   responsibility for messes, 5–6  
   unprofessional, 5–6  
 programming  
   defined, 2  
   structured, 48–49  
 programs, getting them to work, 201  
 pronounceable names, 21–22  
 protected variables, avoiding, 80  
 proxies, drawbacks of, 163  
 public APIs, javadocs in, 59  
 puns, avoiding, 26–27  
`PUTFIELD` instruction, as atomic, 325

## Q

queries, separating from commands, 45–46

## R

random jiggling, tests running, 190  
 range, including end-point dates in, 276  
 readability  
   of clean tests, 124  
   of code, 76

Dave Thomas on, 9  
 improving using generics, 115  
 readability perspective, 8  
 readers  
   of code, 13–14  
   continuous, 184  
 readers-writers execution model, 184  
 reading  
   clean code, 8  
   code from top to bottom, 37  
   versus writing, 14  
 reboots, as a lock up solution, 331  
 recommendations, in this book, 13  
 redesign, demanded by the team, 5  
 redundancy, of noise words, 21  
 redundant comments, 60–62, 272, 275, 286–287  
`ReentrantLock` class, 183  
 refactored programs, as longer, 146  
 refactoring  
   `Args`, 212  
   code incrementally, 172  
   as an iterative process, 265  
   putting things in to take out, 233  
   test code, 127  
*Refactoring* (Fowler), 285  
 renaming, fear of, 30  
 repeatability, of concurrency bugs, 180  
 repeatable tests, 132  
 requirements, specifying, 2  
`resetId`, byte-code generated for, 324–325  
 resources  
   bound, 183  
   processes competing for, 184  
   threads agreeing on a global ordering of, 338  
 responsibilities  
   counting in classes, 136  
   definition of, 138  
   identifying, 139  
   misplaced, 295–296, 299  
   splitting a program into main, 146  
 return codes, using exceptions instead, 103–105

reuse, 174  
 risk of change, reducing, 147  
 robust clear code, writing, 112  
 rough drafts, writing, 200  
 runnable interface, 326  
 run-on expressions, 295  
 run-on journal entries, 63–64  
 runtime logic, separating startup from, 154

## S

safety mechanisms, overridden, 289  
 scaling up, 157–161  
 scary noise, 66  
 schema, of a class, 194  
 schools of thought, about clean code,  
 12–13  
 scissors rule, in C++, 81  
 scope(s)  
   defined by exceptions, 105  
   dummy, 90  
   envying, 293  
   expanding and indenting, 89  
   hierarchy in a source file, 88  
   limiting for data, 181  
   names related to the length of,  
   22–23, 312  
   of shared variables, 333  
 searchable names, 22–23  
 Second Law, of TDD, 122  
 sections, within functions, 36  
 selector arguments, avoiding, 294–295  
 self validating tests, 132  
 Semaphore class, 183  
 semicolon, making visible, 90  
 “serial number”, `SerialDate` using, 271  
`SerialDate` class  
   making it right, 270–284  
   naming of, 270–271  
   refactoring, 267–284  
`SerialDateTests` class, 268  
 serialization, 272  
 server, threads created by, 319–321  
 server application, 317–318, 343–344  
 server code, responsibilities of, 319  
 server-based locking, 329  
   as preferred, 332–333  
   with synchronized methods, 185  
 “Servlet” model, of Web applications, 178  
 Servlets, synchronization problems, 182  
 set functions, moving into appropriate  
   derivatives, 232, 233–235  
`setArgument`, changing, 232–233  
`setBoolean` function, 217  
 setter methods, injecting dependencies,  
   157  
 setup strategy, 155  
`SetupTeardownIncluder.java` listing,  
   50–52  
 shape classes, 95–96  
 shared data, limiting access, 181  
 shared variables  
   method updating, 328  
   reducing the scope of, 333  
 shotgun approach, hand-coded instrumen-  
   tation as, 189  
 shut-down code, 186  
 shutdowns, graceful, 186  
 side effects  
   having none, 44  
   names describing, 313  
 Simmons, Robert, 276  
 simple code, 10, 12  
 Simple Design, rules of, 171–176  
 simplicity, of code, 18, 19  
 single assert rule, 130–131  
 single concepts, in each test function,  
   131–132  
 Single Responsibility Principle (SRP), 15,  
   138–140  
   applying, 321  
   breaking, 155  
   as a concurrency defense principle,  
   181  
   recognizing violations of, 174  
   server violating, 320

- Single Responsibility Principle (SRP),
  - continued*
  - sql class violating, 147
  - supporting, 157
  - in test classes conforming to, 172
  - violating, 38
- single value, ordered components of, 42
- single-letter names, 22, 25
- single-thread calculation, of throughput, 334
- SINGLETON pattern, 274
- small classes, 136
- Smalltalk Best Practice Patterns*, 296
- smart programmer, 25
- software project, maintenance of, 175
- software systems. *See also* system(s)
  - compared to physical systems, 158
- SOLID class design principle, 150
- solution domain names, 27
- source code control systems, 64, 68, 69
- source files
  - compared to newspaper articles, 77–78
  - multiple languages in, 288
- Sparkle* program, 34
- spawned threads, deadlocked, 186
- special case objects, 110
- SPECIAL CASE PATTERN, 109
- specifications, purpose of, 2
- spelling errors, correcting, 20
- SpreadsheetDateFactory*, 274–275
- Spring AOP, proxies in, 163
- Spring Framework, 157
- Spring model, following EJB3, 165
- Spring V2.5 configuration file, 163–164
- spurious failures, 187
- sql class, changing, 147–149
- square root, as the iteration limit, 74
- SRP. *See* Single Responsibility Principle
- standard conventions, 299–300
- standard nomenclature, 175, 311–312
- standards, using wisely, 168
- startup process, separating from runtime logic, 154
- starvation, 183, 184, 338
- static function, 279
- static import, 308
- static methods, inappropriate, 296
- The Step-down Rule*, 37
- stories, implementing only today’s, 158
- STRATEGY pattern, 290
- string arguments, 194, 208–212, 214–225
- string comparison errors, 252
- StringBuffers, 129
- Stroustrup, Bjarne, 7–8
- structure(s). *See also* data structures
  - hiding, 99
  - hybrid, 99
  - making massive changes to, 212
  - over convention, 301
- structured programming, 48–49
- SuperDashboard class, 136–137
- swapping, as permutations, 323
- switch statements
  - burying, 37, 38
  - considering polymorphism before, 299
  - reasons to tolerate, 38–39
- switch/case chain, 290
- synchronization problems, avoiding with
  - Servlets, 182
- synchronized block, 334
- synchronized keyword, 185
  - adding, 323
  - always acquiring a lock, 328
  - introducing a lock via, 331
  - protecting a critical section in code, 181
- synchronized methods, 185
- synchronizing, avoiding, 182
- synthesis functions, 265
- system(s). *See also* software systems
  - file sizes of significant, 77
  - keeping running during development, 213
  - needing domain-specific, 168
- system architecture, test driving, 166–167

system failures, not ignoring  
  one-offs, 187  
system level, staying clean at, 154  
system-wide information, in a local  
  comment, 69–70

## T

tables, moving, 275  
target deployment platforms, running tests  
  on, 341  
task swapping, encouraging, 188  
TDD (Test Driven Development), 213  
  building logic, 106  
  as fundamental discipline, 9  
  laws of, 122–123  
team rules, 90  
teams  
  coding standard for every, 299–300  
  slowed by messy code, 4  
technical names, choosing, 27  
technical notes, reserving comments  
  for, 286  
TEMPLATE METHOD pattern  
  addressing duplication, 290  
  removing higher-level duplication,  
    174–175  
  using, 130  
temporal coupling. *See also* coupling  
  exposing, 259–260  
  hidden, 302–303  
  side effect creating, 44  
temporary variables, explaining, 279–281  
test cases  
  adding to check arguments, 237  
  in `ComparisonCompactor`, 252–254  
  patterns of failure, 269, 314  
  turning off, 58  
test code, 124, 127  
TEST DOUBLE, assigning, 155  
Test Driven Development. *See* TDD  
test driving, architecture, 166–167  
test environment, 127–130  
test functions, single concepts in, 131–132  
test implementation, of an interface, 150  
test suite  
  automated, 213  
  of unit tests, 124, 268  
  verifying precise behavior, 146  
testable systems, 172  
test-driven development. *See* TDD  
testing  
  arguments making harder, 40  
  construction logic mixed with  
    runtime, 155  
testing language, domain-specific, 127  
testNG project, 76, 77  
tests  
  clean, 124–127  
  cleanliness tied to, 9  
  commented out for `SerialDate`,  
    268–270  
  dirty, 123  
  enabling the -ilities, 124  
  fast, 132  
  fast versus slow, 314  
  heuristics on, 313–314  
  ignored, 313  
  independent, 132  
  insufficient, 313  
  keeping clean, 123–124  
  minimizing assert statements in,  
    130–131  
  not stopping trivial, 313  
  refactoring, 126–127  
  repeatable, 132  
  requiring more than one step, 287  
  running, 341  
  self validating, 132  
  simple design running all, 172  
  suite of automated, 213  
  timely, 133  
  writing for multithreaded code,  
    339–342  
  writing for threaded code, 186–190  
  writing good, 122–123

- Third Law, of TDD, 122
  - third-party code
    - integrating, 116
    - learning, 116
    - using, 114–115
    - writing tests for, 116
  - `this` variable, 324
  - Thomas, Dave, 8, 9, 289
  - thread(s)
    - adding to a method, 322
    - interfering with each other, 330
    - making as independent as possible, 182
    - stepping on each other, 180, 326
    - taking resources from other threads, 338
  - thread management strategy, 320
  - thread pools, 326
  - thread-based code, testing, 342
  - threaded code
    - making pluggable, 187
    - making tunable, 187–188
    - symptoms of bugs in, 187
    - testing, 186–190
    - writing in Java 5, 182–183
  - threading
    - adding to a client/server application, 319, 346–347
    - problems in complex systems, 342
  - thread-safe collections, 182–183, 329
  - throughput
    - causing starvation, 184
    - improving, 319
    - increasing, 333–335
    - validating, 318
  - `throws` clause, 106
  - tiger team, 5
  - tight coupling, 172
  - time, taking to go fast, 6
  - Time and Money project, 76
    - file sizes, 77
  - timely tests, 133
  - timer program, testing, 121–122
  - “TO” keyword, 36
  - TO paragraphs, 37
  - `TODO` comments, 58–59
  - tokens, used as magic numbers, 300
  - Tomcat project, 76, 77
  - tools
    - ConTest tool, 190, 342
    - coverage, 313
    - handling proxy boilerplate, 163
    - testing thread-based code, 342
  - train wrecks, 98–99
  - transformations, as return values, 41
  - transitive navigation, avoiding, 306–307
  - triadic argument, 40
  - triads, 42
  - `try` blocks, 105
  - `try/catch` blocks, 46–47, 65–66
  - `try-catch-finally` statement, 105–106
  - tunable threaded-based code, 187–188
  - type encoding, 24
- ## U
- ubiquitous language, 311–312
  - unambiguous names, 312
  - unchecked exceptions, 106–107
  - unencapsulated conditional, encapsulating, 257
  - unit testing, isolated as difficult, 160
  - unit tests, 124, 175, 268
  - unprofessional programming, 5–6
  - uppercase `C`, in variable names, 20
  - usability, of newspapers, 78
  - use, of a system, 154
  - users, handling concurrently, 179
- ## V
- validation, of throughput, 318
  - variable names, single-letter, 25

## variables

- 1 based versus zero based, 261
  - declaring, 80, 81, 292
  - explaining temporary, 279–281
  - explanatory, 296–297
  - keeping private, 93
  - local, 292, 324
  - moving to a different class, 273
  - in place of comments, 67
  - promoting to instance variables of classes, 141
  - with unclear context, 28
- venting, in comments, 65
- verbs, keywords and, 43
- Version* class, 139
- versions, not deserializing across, 272
- vertical density, in code, 79–80
- vertical distance, in code, 80–84
- vertical formatting, 76–85
- vertical openness, between concepts, 78–79
- vertical ordering, in code, 84–85
- vertical separation, 292

**W**

- wading, through bad code, 3
- Web containers, decoupling provided by, 178
- what, decoupling from when, 178
- white space, use of horizontal, 86
- wildcards, 307
- Working Effectively with Legacy Code*, 10
- “working” programs, 201
- workmanship, 176
- wrappers, 108
- wrapping, 108
- writers, starvation of, 184
- “Writing Shy Code”, 306

**X**

## XML

- deployment descriptors, 160
- “policy” specified configuration files, 164