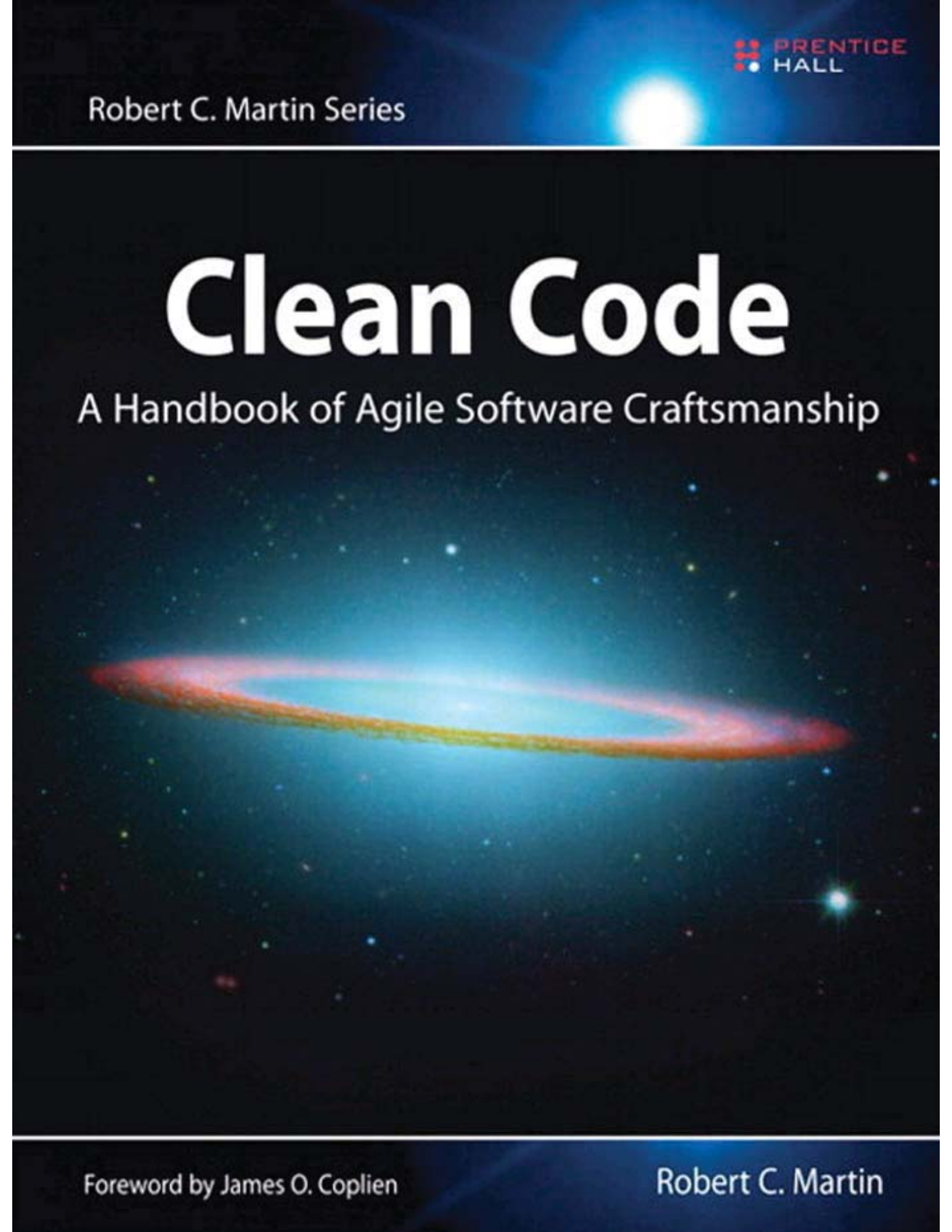

PRENTICE HALL
Robert C. Martin Series
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin

Clean Code

Robert C. Martin Series

The mission of this series is to improve the state of the art of software craftsmanship. The books in this series are technical, pragmatic, and substantial. The authors are highly experienced craftsmen and professionals dedicated to writing about what actually works in practice, as opposed to what might work in theory. You will read about what the author has done, not what he thinks you should do. If the book is about programming, there will be lots of code. If the book is about managing, there will be lots of case studies from real projects.

These are the books that all serious practitioners will have on their bookshelves. These are the books that will be remembered for making a difference and for guiding professionals to become true craftsman.

Managing Agile Projects
Sanjiv Augustine

Agile Estimating and Planning
Mike Cohn

Working Effectively with Legacy Code
Michael C. Feathers

Agile Java™: Crafting Code with Test-Driven Development
Jeff Langr

Agile Principles, Patterns, and Practices in C#
Robert C. Martin and Micah Martin

Agile Software Development: Principles, Patterns, and Practices
Robert C. Martin

Clean Code: A Handbook of Agile Software Craftsmanship
Robert C. Martin

UML For Java™ Programmers
Robert C. Martin

Fit for Developing Software: Framework for Integrated Tests
Rick Mugridge and Ward Cunningham

Agile Software Development with SCRUM
Ken Schwaber and Mike Beedle

Extreme Software Engineering: A Hands on Approach
Daniel H. Steinberg and Daniel W. Palmer

For more information, visit informit.com/martinseries

Clean Code

A Handbook of Agile Software Craftsmanship

The Object Mentors:

Robert C. Martin

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

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

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

Includes bibliographical references and index.
ISBN 0-13-235088-2 (pbk. : alk. paper)
1. Agile software development. 2. Computer software—Reliability. I. Title.
QA76.76.D47M3652 2008
005.1—dc22 2008024750

ISBN-13: 978-0-13-235088-4
ISBN-10: 0-13-235088-2
Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.
First printing July, 2008

For Ann Marie: The ever enduring love of my life.

This page intentionally left blank

Contents

This page intentionally left blank

Foreword

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

Ærlighed i små ting er ikke nogen lille ting.

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

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

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

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

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

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

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

If you take up the challenge—yes, the challenge—of reading and applying this book, you'll come to understand and appreciate the last point. Here, we are finally driving to the roots of responsible professionalism in a profession that should be concerned with the life cycle of a product. As we maintain automobiles and other machines under TPM, breakdown maintenance—waiting for bugs to surface—is the exception. Instead, we go up a level: inspect the machines every day and fix wearing parts before they break, or do the equivalent of the proverbial 10,000-mile oil change to forestall wear and tear. In code, refactor mercilessly. You can improve yet one level further, as the TPM movement innovated over 50 years ago: build machines that are more maintainable in the first place. Making your code readable is as important as making it executable. The ultimate practice, introduced in TPM circles around 1960, is to focus on introducing entire new machines or

replacing old ones. As Fred Brooks admonishes us, we should probably re-do major software chunks from scratch every seven years or so to sweep away creeping cruft. Perhaps we should update Brooks' time constant to an order of weeks, days or hours instead of years. That's where detail lies.

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

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

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

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

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

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

Going back to my little box of Ga-Jol, I think it's important to note that the Danish wisdom advises us not just to pay attention to small things, but also to be *honest* in small things. This means being honest to the code, honest to our colleagues about the state of our code and, most of all, being honest with ourselves about our code. Did we Do our Best to "leave the campground cleaner than we found it"? Did we re-factor our code before checking in? These are not peripheral concerns but concerns that lie squarely in the center of Agile values. It is a recommended practice in Scrum that re-factoring be part of the concept of "Done." Neither architecture nor clean code insist on perfection, only on honesty and doing the best we can. *To err is human; to forgive, divine.* In Scrum, we make everything visible. We air our dirty laundry. We are honest about the state of our code because

code is never perfect. We become more fully human, more worthy of the divine, and closer to that greatness in the details.

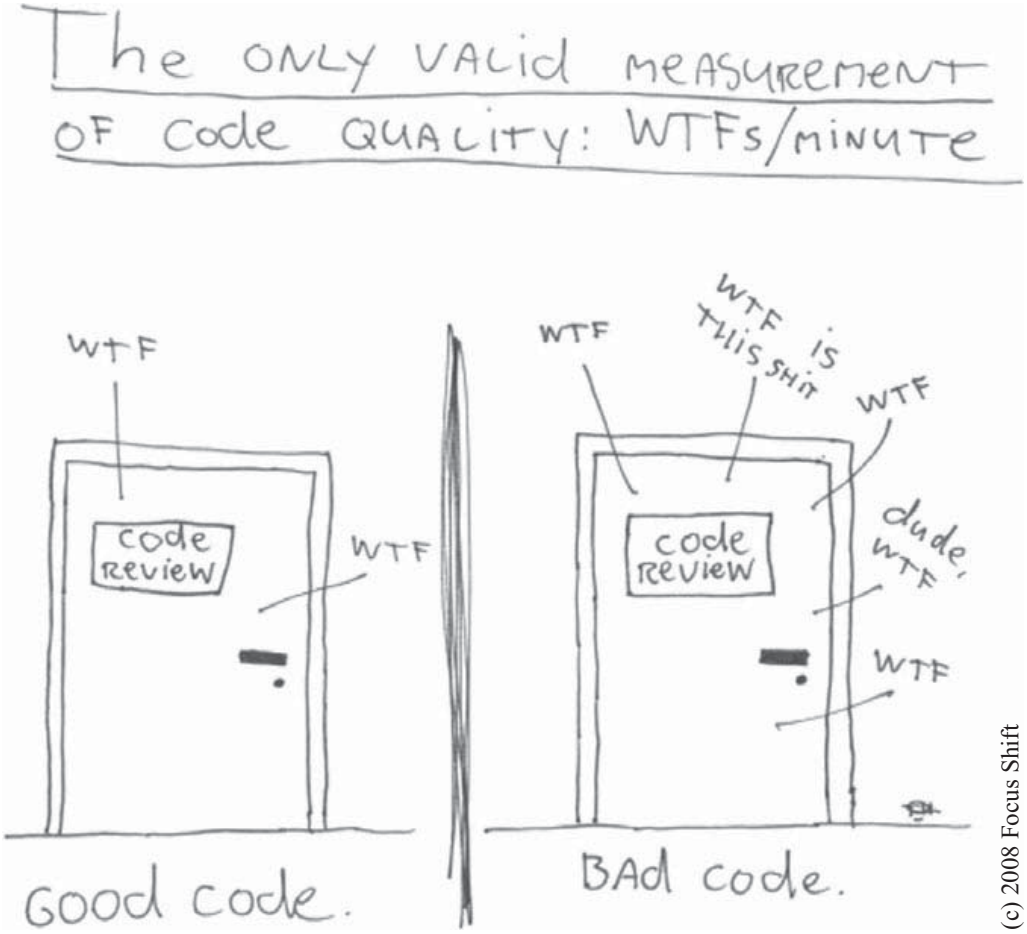
In our profession, we desperately need all the help we can get. If a clean shop floor reduces accidents, and well-organized shop tools increase productivity, then I'm all for them. As for this book, it is the best pragmatic application of Lean principles to software I have ever seen in print. I expected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

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

James O. Coplien
Mørdrup, Denmark

This page intentionally left blank

Introduction



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

Which door represents your code? Which door represents your team or your company? Why are we in that room? Is this just a normal code review or have we found a stream of horrible problems shortly after going live? Are we debugging in a panic, poring over code that we thought worked? Are customers leaving in droves and managers breathing down

our necks? How can we make sure we wind up behind the *right* door when the going gets tough? The answer is: *craftsmanship*.

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

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

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

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

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

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

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

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

The third part of this book is the payoff. It is a single chapter containing a list of heuristics and smells gathered while creating the case studies. As we walked through and cleaned up the code in the case studies, we documented every reason for our actions as a

heuristic or smell. We tried to understand our own reactions to the code we were reading and changing, and worked hard to capture why we felt what we felt and did what we did. The result is a knowledge base that desribes the way we think when we write, read, and clean code.

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

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

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

Acknowledgments

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

This page intentionally left blank

On the Cover

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

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

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

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

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

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

Cover image: © Spitzer Space Telescope