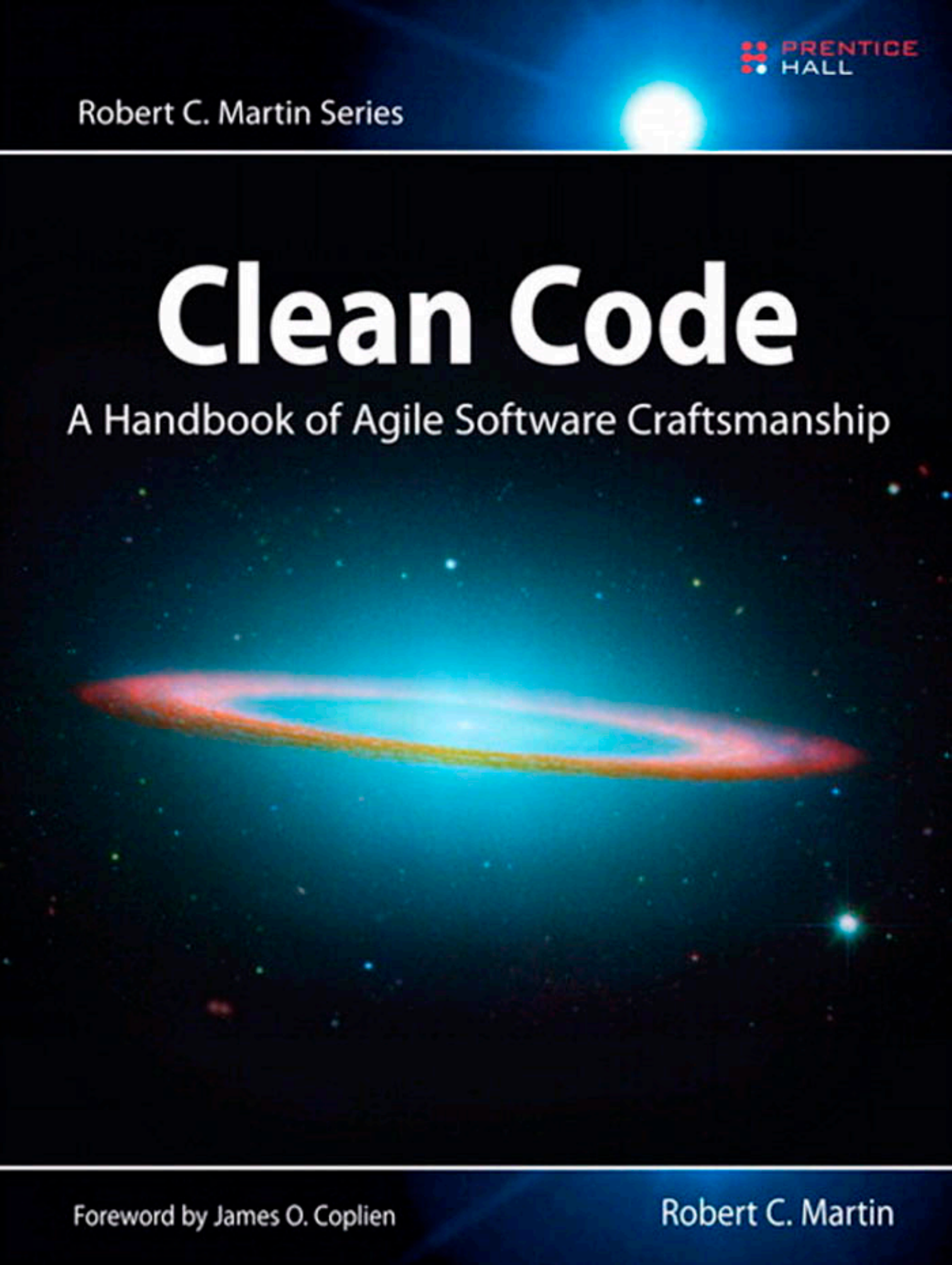

PRENTICE HALL
Robert C. Martin Series
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin

1

Clean Code

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

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

There Will Be Code

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

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

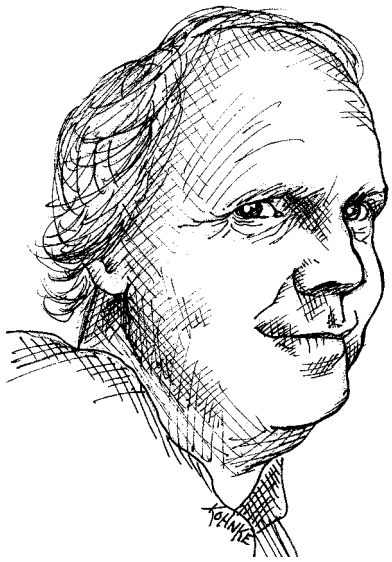
I expect that the level of abstraction of our languages will continue to increase. I also expect that the number of domain-specific languages will continue to grow. This will be a good thing. But it will not eliminate code. Indeed, all the specifications written in these higher level and domain-specific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

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

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

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

Bad Code



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

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

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

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

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

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

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

1. [Beck07].

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

The Total Cost of Owning a Mess

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

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

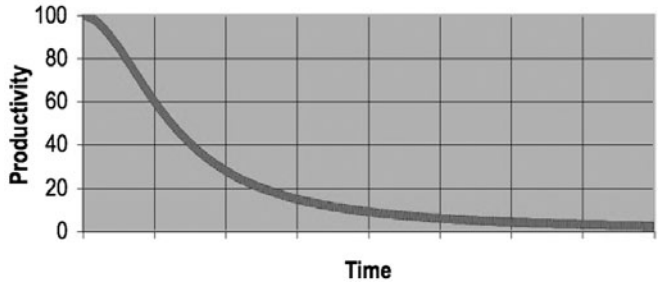


Figure 1-1
Productivity vs. time

The Grand Redesign in the Sky

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

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

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

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

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

Attitude

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

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

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

No. The managers and marketers look to *us* for the information they need to make promises and commitments; and even when they don't look to us, we should not be shy about telling them what we think. The users look to us to validate the way the requirements will fit into the system. The project managers look to us to help work out the schedule. We

are deeply complicit in the planning of the project and share a great deal of the responsibility for any failures; especially if those failures have to do with bad code!

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

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

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

The Primal Conundrum

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

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

The Art of Clean Code?

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

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

---

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

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

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

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

What Is Clean Code?

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

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

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

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

Bjarne also mentions efficiency—*twice*. Perhaps this should not surprise us coming from the inventor of C++; but I think there's more to it than the sheer desire for speed. Wasted cycles are inelegant, they are not pleasing. And now note the word that Bjarne uses

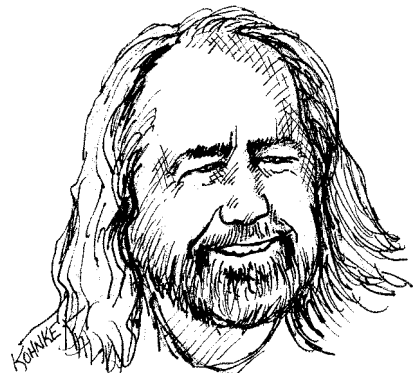
to describe the consequence of that inelegance. He uses the word "tempt." There is a deep truth here. Bad code *tempts* the mess to grow! When others change bad code, they tend to make it worse.

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

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

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

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



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

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

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

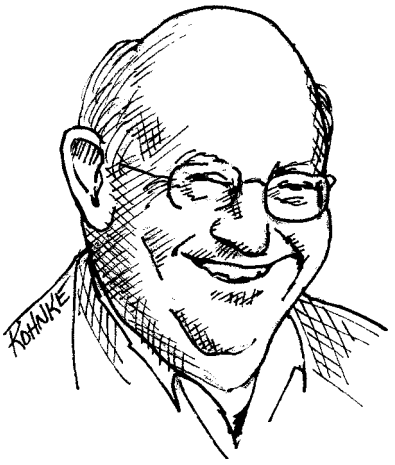
3. http://www.pragmaticprogrammer.com/booksellers/2004-12.html

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

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

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

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



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

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

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

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

---

4. [Knuth92].

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

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

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

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

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

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

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

beyond names, however. I also look at whether an object or method is doing more than one thing. If it's an object, it probably needs to be broken into two or more objects. If it's a method, I will always use the Extract Method refactoring on it, resulting in one method that says more clearly what it does, and some submethods saying how it is done.

Duplication and expressiveness take me a very long way into what I consider clean code, and improving dirty code with just these two things in mind can make a huge difference. There is, however, one other thing that I'm aware of doing, which is a bit harder to explain.

After years of doing this work, it seems to me that all programs are made up of very similar elements. One example is "find things in a collection." Whether we have a database of employee records, or a hash map of keys and values, or an array of items of some kind, we often find ourselves wanting a particular item from that collection. When I find that happening, I will often wrap the particular implementation in a more abstract method or class. That gives me a couple of interesting advantages.

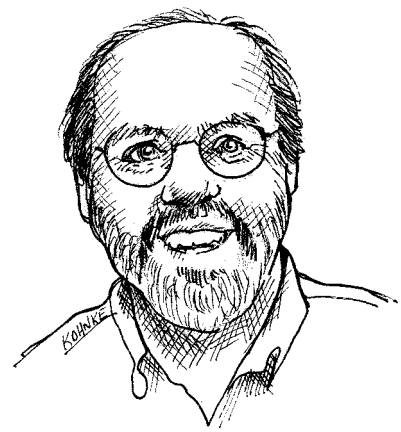
I can implement the functionality now with something simple, say a hash map, but since now all the references to that search are covered by my little abstraction, I can change the implementation any time I want. I can go forward quickly while preserving my ability to change later.

In addition, the collection abstraction often calls my attention to what's "really" going on, and keeps me from running down the path of implementing arbitrary collection behavior when all I really need is a few fairly simple ways of finding what I want.

Reduced duplication, high expressiveness, and early building of simple abstractions. That's what makes clean code for me.

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

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



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

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

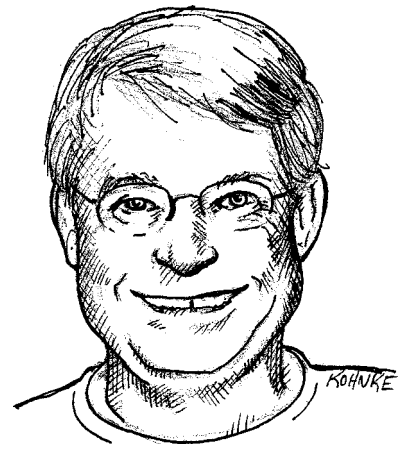
". . . pretty much what you expected." When was the last time you saw a module that was pretty much what you expected? Isn't it more likely that the modules you look at will be puzzling, complicated, tangled? Isn't misdirection the rule? Aren't you used to flailing about trying to grab and hold the threads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

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

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

Schools of Thought

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



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

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

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

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

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

We Are Authors

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

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

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

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

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

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

. . .

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

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

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

The Boy Scout Rule

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

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

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

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

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

---

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

Prequel and Principles

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

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

Conclusion

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

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

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

Bibliography

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

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