1

Clean Code

"'Clean' isn't a ding at anyone. Our notion of 'unclean' is a way of characterizing code that demands more effort from the average developer to understand or maintain."

—Jeff Langr

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 AI, large language models (LLMs), and requirements instead. Indeed, some have suggested that we are close to the end of code.[1] That soon, all code will be generated instead of written. That programmers simply won't be needed, because businesspeople will generate programs from specifications—or prompts.

1. It is hilarious that I wrote that line in 2008, and here it is, 2024. Some things never change.

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 and powerful AIs 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 languages, and all the prompts written to guide a really good AI, 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"[2] who hope one day to discover a mathematics that does not have to be formal. They hope one day to discover a way to create machines that can do what they want rather than what they 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.

2. In quotes because no real mathematician would hope for this.

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 the necessary precision and formality—so there will always be code.

Bad Code

Long ago I read the preface to Kent Beck's book *Implementation Patterns*.[3] He said, "… 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.

3. [IMP].

I know of one company who, 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, you’ve felt this impediment many times. Indeed, we have a name for it. We call it *wading and slogging*. 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 that you could move on to the next. We’ve all done it.

We’ve all looked at the mess we just made and chosen to leave it for another day. We’ve all felt the relief of seeing our messy program work and deciding that a 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—or didn’t want to know—that later equals never.[4]

4. LeBlanc’s law.

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 handwashing in preparation for surgery because it was taking too much time?[5] 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.

---

5. When handwashing 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.

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 of experience know that 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. It will also slow down everyone else who must deal with it. And that slowness will build and compound and will force you to miss deadline after deadline.

The only way to make the deadlines—the only way to go fast—is to keep the code clean at all times.

The only way to go fast is to go well.

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!

Writing clean code requires the disciplined use of myriad little techniques applied through a painstakingly acquired sense of cleanliness that helps us transform bad code into clean code.

And it is always a transformation. Nobody writes clean code. We all write messy code. Only those of us who take the extra step to clean it will produce clean code. This follows Kent Beck's law, which you will see over and over in this book:

First, make it work. Then, make it right.

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. I have summarized their responses below.

"Clean code does one thing well."

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

This is an old and well-worn maxim. In the '70s, we used to say that a module should do one thing, do it well, and do it only. But what does "one thing" mean? Back in the late '80s, I was the author of a 3,000-line C function named `gi`. The name stood for *graphic interpreter*. If someone had come to me back then and warned me that my long, long function did more than one thing, I would have replied, "No, it interprets graphics."

The problem with the "one thing" rule is that "one thing" is entirely subjective. But I think I know a way to make it *objective*. We'll talk about that later in this book. And once I describe it to you, I think you'll be forced to agree that it is close to, if not entirely, objective. You may not like it; but I wager you'll find no way to substantially disagree.

"Clean code reads like well-written prose."

—Grady Booch, famed author of many software engineering books

Now that's one hell of an assertion. When is the last time you encountered code that read like "well-written prose"? And yet, as you read on in this book, I'll show you a way to achieve, or at least approach, that very goal.

Oh, don't get me wrong. Your code is not going to read like a Hemingway novel. But it is entirely possible to construct code that reads so nicely that even nonprogrammers can make some sense[6] of it.

---

6. By some definition of the words *some* and *sense*.

"Clean code always looks like it was written by someone who cares."

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

The words that Michael left out of that wonderful quotation were: "about you." The reason that a programmer will go to the effort of cleaning their code is that the programmer cares about the next person to come along. They care that their code will not mislead, misdirect, or befuddle those who must maintain that code. The word that best defines cleanliness is *care*.

The implication of that definition is that code that has not been cleaned was written and released carelessly.

"You know you are working on clean code when each routine you read turns out to be pretty much what you expected."

—Ward Cunningham, inventor of the Wiki, inventor of Fit, co-inventor of Extreme Programming; motive force behind Design Patterns; Smalltalk and OO thought leader; the godfather of all those who care about code

Imagine reading code and simply agreeing with it. As you scroll down, you nod in acceptance and affirmation. The act of reading the code is as fluid as the laminar flow of air over an airplane wing. There is no turbulence, no clever tricks, no jarring non sequiturs. You simply read, and agree, from top to bottom.

This may seem like an impossible utopian dream; but certainly it is a dream to strive for. In this book, I will show you ways to approach that dream.

"Clean code fits in your head. A function fits on a single screen, involves no more than a handful of moving parts, and has good names. It's understandable by peers years after it was written. It minimizes surprises, composes well, and is easy to delete."

—Mark Seeman, author of *Code That Fits in Your Head*, popular blogger on software craftsmanship and functional programming

That kind of says it all, doesn't it? In some ways, that quotation is a summary of this book.

Putting All This Together

Clean code will read like well-written prose because it has been carefully partitioned into functions and modules the way prose is partitioned into sentences and paragraphs, each dealing with one thing, one topic, one irreducible concept that flows from one to the next to the next to create the expected whole.

Why Should We Be Clean?

Look around you. How many computers are running within 10 feet of you? Do you have a smart phone, a smart watch, smart earbuds, a smart car key? Are you wearing a smart medical device?

Look around your kitchen. How many computers are in the appliances? Do you have a smart microwave oven, a smart regular oven, a smart stove, a smart blender, a smart refrigerator?

Is your TV smart? Is your thermostat smart? Do you have a smart security system or a smart sound system?

Do you own a car? How many lines of code are running in that car? Does it have a GPS? Does your phone? Can you even drive to the store without using them?

Think about this. Think hard about it. Nearly everything we do, in this modern world, is mediated by software. You can't cook a hot dog without software. You can't watch TV. You can't make a phone call. You can't drive to the store. You can't buy anything. You can't sell anything. No law can be passed. No law can be enforced. No insurance claim can be filed. Without software being smack-dab in the middle of it all.

As I told you in the introduction, very little happens in our society without the involvement of software.

You and I rule the world.

Others think they rule the world; but then they hand the rules to us and we write the actual rules that run in the machines that govern everything.

Our civilization depends upon us; and that dependence is critical. Without us, civilization would grind to a halt overnight. No one is prepared to go back to using index cards to run everything.

Are we moral, and professional, and responsible enough to bear civilization upon our shoulders? Or are we on a course to let everyone down? Are you and I going to be responsible for a major catastrophe?

We've seen hints of such catastrophes in the recent past. We've seen software drive two jet airliners into the ground at the speed of sound, killing 346 people. We've seen software errors kill dozens, and injure hundreds, when their cars' computers lost their minds and accelerated out of control. We saw Knight Capital lose $450M in 45 minutes due to a software error. We saw `healthcare.gov` stumble and nearly fall over when it was first launched.

One day, possibly one day soon, there will be a software error that kills tens of thousands of people in a single incident. Nowadays, it's not very hard to imagine a scenario like that. And when that occurs, we programmers will be blamed.

Don't think it will be our bosses. Don't think it will be the CEOs. Oh, they'll be in hot water too. But the finger of blame will be pointed directly at us—because, after all, it will have been our fingers on the keyboards.

And then imagine the trials. Imagine the testimony of the expert consultants complaining about thousands of global variables, poorly named arguments, mismanaged and missing semaphores, memory leaks, improperly allocated stacks, heap fragmentation, overdependence on performance over quality, 3,000-line functions, modules with duplicated code of which some, but not all, had been repaired.

We've seen that kind of testimony before—too many times.

Imagine the response of the world's politicians as they try to grapple with the loss of tens of thousands of their citizens. They would, of course, have to pass new laws.

What kinds of laws would they pass? Among other things, they might tell us what languages we have to use, what platforms and frameworks are allowable, what processes we have to follow, what signatures we have to get, what documents we need to write, what review procedures we have to use, what courses we have to take, and what books we have to read.

I promise you that you and I will not like those new laws. I'm also reasonably certain that those laws will do little to improve the situation.

So, the first answer to why we should clean our code is that our civilization needs us to behave as professionals, and that if we fail in that responsibility, our governments will force inconvenient and ineffective rules upon us.

But there is another reason that we need to clean our code. And this one strikes much closer to home.

Productivity

I've asked you before if you have ever been significantly impeded by bad code. If you are a programmer of any number of years, you most certainly have. Once again, my question is: Why did you write it?

Why do we write the code that we know will significantly impede us? The answer always is: to go fast. I'll leave you to deal with the logical inconsistency of that answer.

Now look, I know all the excuses. They all boil down to: "They never give us time." This is false. The only way to go fast is to go well. The best way to meet the schedule, the best way to make the deadline, is to do a good job. If you want to go as fast as you can, don't do the things that slow you down! Clean your code!

A messy module slows down everyone who has to deal with it, *every time* they have to deal with it. The cost of the mess repeats over and over, whereas the cost of cleaning is a one-time thing.

A team who allows messes to persist in their code will gradually slow down to a tiny fraction of their potential productivity. I've seen teams who have expanded their estimates out to weeks and months for jobs that ought to take a day or two.

The slower the team goes, the more the pressure builds. The more the pressure builds, the more messes are left uncleaned. Each such mess impedes the team even more; and the slower they all go. It’s a vicious cycle, and if you’ve been a programmer for more than a couple of years, you’ve likely taken a few turns around it.

But it gets worse. Managers, desperate to increase productivity, hire new people. This, of course, has the opposite of the desired effect. The new people, thrown into the mass of messy code, see how things are done around here and—of course—they emulate it. So now there are even more people making messes, which drives the cycle even faster, and drags the productivity even lower.

Managers, astounded that the addition of people did not increase productivity, may try to add even more people into the fracas—because <*sarcasm*> the definition of sanity is to try the same thing over and over and expect different results </*sarcasm*>. In the end, however, the managers have no choice but to consult with the developers to find out why everything is going so slowly.

And the developers have an answer. They’ve known the answer for months, if not years. “And when asked, they are more than willing to volunteer that answer: Redesign the whole system from scratch.”

Imagine the shock and horror of the managers. The developers have just told them that all their previous work is worthless and has to be scrapped. The developers are also telling them that this new design won’t suffer the same fate—that productivity won’t decrease again—that everything, this time, will be great.

Of course the managers don’t believe that; and they fight strenuously against the idea, but what choice do they really have? The experts are telling them that the only hope is for the system to be redesigned from scratch. In the end, those poor managers will accede to the demands of the developers. Again, what choice to they have?

And when those managers finally agree, a cheer will go up from the developers. “Hallelujah, we’re all going back to the beginning where there’s no mess to deal with and life will be good!”

But that’s not what actually happens. Instead, the ten best, the Tiger Team, are chosen to redesign the system. These are the people who made the mess in the first place. They are going into a room to work together to take the project in a “new and beautiful” direction. They will chart the course to the future.

Meanwhile, the rest of the team is stuck maintaining the old system. After all, that system is part of the business. Bugs have to be fixed. New features have to be added.

Where does the Tiger Team get their requirements from? Is there a requirements document that anyone seriously trusts? Or have there been so many late-night scrambles and midnight modifications that the old requirements are currently hash? No matter, because the requirements are reliably recorded in one definite place—the old code.

So the Tiger Team is stuck in their room, poring over the old code, trying to figure out what the hell this system actually does. Meanwhile, the rest of the team is changing that code by fixing bugs and adding new features.

And this sets up the race. The Tiger Team has to catch up to where the maintenance team is. But every time the Tiger Team gets to where the maintenance team was, the maintenance team has moved ahead!

This is Xeno’s paradox—the race between the great Achilles and a tortoise with a head start. Every time Achilles reaches the point where the tortoise was when he began, the tortoise will have moved ahead from that spot. Xeno used that argument to prove that all motion was impossible.

It turns out that if you know enough calculus, you can prove that Achilles will, in fact, pass the tortoise. But that math doesn’t always work for software projects.

I once worked on a project that was being redesigned from scratch. Ten years later, the “New System” had still not been deployed. The customers had been told, years before, to expect a new and glorious system. But that system never came. Every time the company tried to replace the old systems the customers complained that the new system did not do everything the old system did. Try as they might, the Tiger Team could not catch up to the maintenance team. And since the maintenance team was revenue bearing, they got all the resources they needed, whereas the Tiger Team was always limited.

As the months and years went by, the original members of the Tiger Team were eventually replaced and promoted to work on other projects. After ten years, none of the originals were left.

In the end, the company, in utter desperation, told their customers that they had no choice but to accept the "new" system. The customers railed and complained; but there was simply no other option. The company could not afford to keep funding two development efforts.

Upon hearing that the new system was going to be deployed, the Tiger Team stood up in horror saying: "You can't ship this, it's crap! It's got to be redesigned!"

OK, I've embellished this story a bit. Actually, it's a composite tale from my own experiences and the stories many of my later clients told me. But I know there are many of you, my gentle readers, who identified with it.

So why should we clean our code? Because cleaning our code is the best way for a team to remain productive. If we allow the mess to build, we lay the groundwork for the disaster I just described. But if we keep the code clean, productivity will remain high, the development team will not demand a redesign, and the managers will not desperately try to increase productivity by adding more and more manpower.

Livability[7]

---

7. https://blog.cleancoder.com/uncle-bob/2018/08/13/TooClean.html

It would be easy to interpret all of the above comments as aspiring to some kind of Golden Perfection. That is not our goal! Golden Perfection is paralyzing—it affords no room to move. You cannot live within Golden Perfection.

Clean code is code that you can live with. It is not perfect code. It is more like a clean house as opposed to a show house. A show house is a house you cannot live in, because the act of living degrades the show-ability of the house. A clean house is a house that you can live in without degrading its livability.

Clean code is code that you can maintain, expand, enhance, and evolve without degrading its livability. Clean code is not sparkling and shiny; it looks lived in but cared for. It may not read like Dickens, but it's readable. It may not have the perfect design and architecture, but it is carefully designed. Concepts may not be perfectly isolated, but they are isolated enough.

Clean code is comfortable in the way that a clean but lived-in house is comfortable. There might be some crumbs on the floor beneath the breakfast counter, there may be a drop of milk on that counter, there may be some dog hair on the sofa, the pillows on the love seat might be in disarray, but none of that degrades the livability of the house—so long as it doesn't get out of hand.

Clean code is code that has not gotten out of hand. Those who write and maintain it are not seeking perfection, they don't spend endless hours polishing it, but they do strive to keep the flotsam and detritus of life from accumulating. Or, to invoke Michael Feathers again, they care.

We Read More Than We Write

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!

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

6. Oops, he’s erasing what he typed!
7. He types it again.
8. He erases it again!
9. He types half of something else but then erases that!
10. He scrolls down to another function that calls the function he’s changing to see how it is called.
11. He scrolls back up and types the same code he just erased.
12. He pauses.
13. He erases that code again!
14. 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 versus writing is well over 10:1. We are constantly reading old code as part of the effort to write new code; because the new code has to be kept consistent with all the old code.

Because we read code so much more than we write it, we want the reading of code to be the more efficient operation. And since there’s no way to write code without reading it, 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 only be easy to write if the surrounding code is easy 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.

If we all checked in our code a little cleaner than when we checked it out, the code simply could not rot.

Check the code in cleaner than you checked it out.

The cleanup doesn't have to be something big. Just do one small random act of kindness. 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 everyone on the team practiced this rule? The code would simply get better as time passed. Do you believe that any other option is professional? Indeed, isn't continuous improvement an intrinsic part of professionalism—of being a moral human?