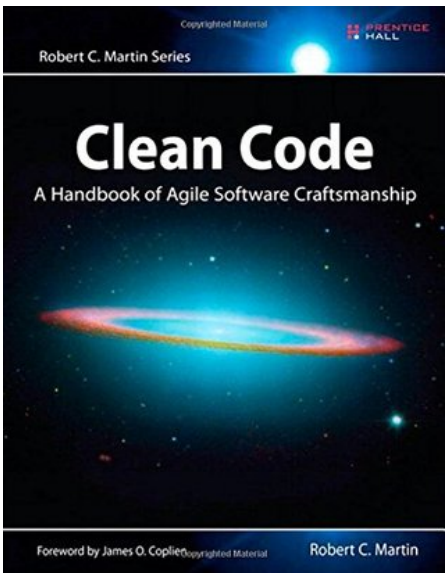

Clean Code: A Handbook of Agile Software Craftsmanship

Robert C. Martin

Clean Code: A Handbook of Agile Software Craftsmanship

Robert C. Martin

Clean Code: A Handbook of Agile Software Craftsmanship Robert C. Martin
Even bad code can function. But if code isn t clean, it can bring a development organization to its knees. Every year, countless hours and significant resources are lost because of poorly written code. But it doesn t have to be that way.
Noted software expert Robert C. Martin presents a revolutionary paradigm with ***Clean Code: A Handbook of Agile Software Craftsmanship*** . Martin has teamed up with his colleagues from Object Mentor to distill their best agile practice of cleaning code on the fly into a book that will instill within you the values of a software craftsman and make you a better programmer but only if you work at it.
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. More importantly, you will be challenged to reassess your professional values and your commitment to your craft.
Clean Code is divided into three parts. The first describes the principles, patterns, and practices of writing clean code. The second part consists of several case studies of increasing complexity. Each case study is an exercise in cleaning up code of transforming a code base that has some problems into one that is sound and efficient. The third part is the payoff: a single chapter containing a list of heuristics and smells gathered while creating the case studies. The result is a knowledge base that describes the way we think when we write, read, and clean code.
Readers will come away from this book understanding

How to tell the difference between good and bad code How to write good code and how to transform bad code into good code How to create good names, good functions, good objects, and good classes How to format code for maximum readability How to implement complete error handling without obscuring code logic How to unit test and practice test-driven development This book is a must for any developer, software engineer, project manager, team lead, or systems analyst with an interest in producing better code.
"

Clean Code: A Handbook of Agile Software Craftsmanship Details

Date	:	Published August 11th 2008 by Prentice Hall (first published January 1st 2007)
ISBN	:	9780132350884
Author	:	Robert C. Martin
Format	:	Paperback 434 pages
Genre	:	Computer Science, Programming, Science, Technology, Software, Technical, Nonfiction

From Reader Review Clean Code: A Handbook of Agile Software Craftsmanship for online ebook

Vladimir says

This book makes some very good points, sometimes taking them to extreme ("Never write functions longer than 15 lines! Never write functions with more than three arguments!"). Some of these points were quite new and useful for me - YMMV. It's too Java-specific in a few places, and reading the last refactoring chapter on a kindle was quite a challenge, but otherwise it was well worth a read. At least I got a clear picture of how I want to refactor a big piece of my current project after reading this :)

Alex Ott says

Nothing new for experienced developer...
Too Java oriented in many places. Code Complete, 2ed is better from my point of view

---

Kosala Nuwan Perera says

I had a tough time deciding whether I really liked or It was amazing. I liked the writing style of the book. Its simple, clean, and well crafted.

First few chapters of the book makes good practical advice from naming variables-functions-classes to writing functions to testing. Most of the smells and heuristics I found in these chapters can be found in real-world as well.

> Complexity kills. It sucks the life out of developers, it makes products difficult to plan, build, and test. - Ray Ozzie, CTO, Microsoft Corporation

In the next few chapters of the book contains some very good points. Some of them are quite new and very useful for me when applying design principles (such as SRP, OCP, DRY, SOC) to keeping the code base small, simple, and clean.

> Most freshman programmers (like most grade-schoolers) don't follow this advice particularly well. They believe that the primary goal is to get the program working. Once it's "working", they move on to the next task, leaving the "working" program in whatever state they finally got it to "work". Most seasoned programmers know that this is **professional suicide**.

These parts of the book are ***fantastic*** and well justified though most of the examples are pure Java-

specific. Latter sections of the book is more into Java centric, and thought of skimming few sections (disappointingly) but was compelled to continue reading. The book must titled *"Clean Code (Java)"*. Though this book makes more sense for Java developers at the beginners and intermediate levels, I would definitely recommend the book to any .NET C# developers as well.

All in all, it was well worth a read! I got a clear picture of how developers end up with smelly code and how we can refine and "clean" it up.

/KP
Review: Clean Code by Robert C. Martin

Erika RS says

I wanted to love this book, but instead I just sort of liked it. This book is a member of the extensive genre of books on how to write clean code. It sits alongside books like *Code Complete* by Steve McConnell[1] and many others. Where *Clean Code* promised to differentiate itself was in the use of three case studies -- about a third of the book -- showing Martin's code cleanup techniques in action.

However, I was disappointed by that section. As someone who codes and reviews code professionally, the case studies were not particularly enlightening. As seems obvious in retrospect, watching someone clean-up code in fairly straightforward ways is not interesting if you do and see that everyday. What I really wanted was a book on being a better code reviewer with advice on how to spot areas for improvement and convince others of the value of those improvements.

The examples could be useful for someone who isn't in a code-review-heavy environment. Martin does a reasonably good job of taking code that may seem reasonable on the surface and improving its readabilty. That said, his comments indicate that he often has a higher opinion of the cleanliness of his end result than I do.

As for the general advice and discussion of how to make clean code, I agree with a lot of his tips and disagree with others. Code cleanliness is an area where the core of just-plain-good ideas is surrounded by a nimbus of sometimes contradictory standards that people pick and choose from. The details of what you choose from the nimbus generally does not matter so much as consistency. (Of course, the real trouble occurs when people don't agree on what belongs in the core and what belongs in the nimbus.)

The book definitely was not a bad read, but it did not fit my needs.

[1] Still my favorite in the genre.

Carl-eric says

Many good points in this book. Unfortunately, almost all of them are overdone. Yes, you should write short functions, if possible. Yes, you should have functions that do one thing.

But no, "one thing" does not mean you should tear an algorithm apart into twenty little funclets that make no

sense on their own.

Basically, like another reviewer wrote, the first part of the book raises many good points, and the second part of the book then merrily applies these points way beyond their usefulness. Read the book, but keep your brains turned on and be alert.

Nariman says

If you are a programmer, you must read it! full of good examples of how to write clean and readable code.

David says

This is a book that one could get started on the idea of "good code" - clean, readable, elegant, simple, easy-to-test, etc. It has the usual stuff that you'd expect - good naming convention, testable code, single responsibility, short classes, short methods - but I feel like it takes them on overdose, going to extremes (IMHO) such as setting short explicit lengths, forbidding certain constructs, and what seems like refactoring for the sake of it.

I'd actually recommend other books like the Pragmatic Programmer or Code Complete; there's something about the way this book reads that irks me. I think it's more useful to highlight the attributes that clean code should have (which this book *does* do), then it is to declare outright what is "good" and what is "bad" (even in subjective areas like readability, comments, and formatting).

To their credit, the author(s) did state right out at the start that these are their very personal preferences, so that's all right - I'm just disagreeing on some of the more subjective areas.

Also a plus are a few actual and simple scenarios/use cases to show code clean up in action, but they aren't exactly really tricky bits of code, but rather straightforward examples - very good for developers new to the concept of clean code, but less so for developers already familiar with the basic ideas.

Rod Hilton says

There is a movement brewing in the world of professional software development. This movement is concerned not merely with writing functional, correct code, but also on writing **good code**. Taking pride in code. This movement is the Software Craftsmanship movement, and one of the people near the head of this movement is Robert C. Martin, also known as Uncle Bob.

His book "Clean Code" is, in many ways, **an introduction to the concept of Software Craftsmanship and a guide for developers interested in becoming craftsmen**. Clean Code is not about only writing correct code, it's about writing code that is designed well, code that reads well, and code that expresses the intent of the author.

The book is essentially divided into two parts. The first part contains Bob's suggestions for writing and maintaining clean code. This includes suggestions on everything ranging from how to properly comment

code and how to properly name variables to how to separate your classes and how to construct testable concurrent code. The second part of the book uses the principles in the first part to guide the reader through a few exercises in which existing code is cleaned.

The first part of the book is fantastic. I can't recommend it highly enough for a professional software developer that wishes to elevate him or herself to a higher standard. This guide is excellent, and gave me lots of things to think about while reading it. I could almost feel myself becoming a better programmer as I internalized Martin's advice, and the code I've been writing has been noticeably better since I began following his suggestions.

In the second part of the book, Martin essentially guides us through three projects: a command line argument parser he wrote, a section of the JUnit source code, and a section of source code from SerialDate. Of these, the most detailed guide is Martin's illustration of refactoring the command line argument parser.

These sections all suffered from a fundamental flaw: **they were inside a book.**

These sections all required reading large amounts of source code. Not just scanning it, but really reading and understanding the code, so that the reader can understand the changes Martin makes to the code. Reading and understanding code is something I do every day as a professional, but I **never** have to do it from paper.

When I read code, I'm interacting with something, not just reading. I can scroll up and down. If I see a method being used and I wonder how it's written, I can click on it and jump right to the implementation. My IDE shows me symbols in my gutterbar when methods are overridden. I can press a keystroke to pull up a list of just the methods in a source file. I can right click on a method and find its usages immediately. **The source code I am reading is something I can push and pull, gaining an understanding of it through interaction.**

When source code is printed in a book, you get none of this. To make matters worse, Martin's code samples have absolutely no syntax highlighting applied to them. When I read source code, certain parts are rendered in specific ways that make it easier to pull the information into my brain. Strings and literals are formatted differently, comments are dimmer so I can focus on code, and so on. Code samples in "Clean Code" are just characters, with occasionally bolding used to draw attention to parts that were changed. It's amazing how much syntax highlighting helps make code more comprehensible, even at a completely subconscious level.

A book is, quite simply, **not an appropriate medium for a guided code walkthrough**. I'd have preferred the content of these sections as perhaps a lecture, with Martin's text done in audio and his code kept on the screen. This would at least prevent me from having to flip back and forth constantly. I didn't get as much out of these sections as I would have liked to, simply because it was so difficult to digest the information it contained in such an awkward, unnatural medium. **At the very least, the code samples should have been printed in color, with syntax highlighting.**

I can tell that his advice was good and that the refactorings he applied to the code samples in the book made the code far better, but mostly because I've observed these efforts in real life and observed how much they improve code. **If I were to encounter Martin's "before" and "after" code in a project I was working on, I undoubtedly would find the "after" code far, far cleaner and more enjoyable to work with.** However, since the book format made it so difficult to understand EITHER code sample, it didn't seem like Martin's efforts offered much improvement, even though I know they did.

Despite this frustration, the book is an excellent read, and I'm quite certain **it has contributed a great deal to helping me improve as a professional.** I can't recommend it enough, especially for Java developers. I just think that most readers will find the final few chapters intensely frustrating - I recommend downloading the code and viewing it in your favorite code editor so that you can comprehend the source code the way you would any other source code.

Fahad Naeem says

I started reading it after a lot of recommendations but it wasn't gone up to the standards.
Clean Code is about writing code which is not only understandable to the code him/herself but to the others as well.

Robert Martin mainly used a lot of JAVA code which is not applicable to other languages like Python and JAVASCRIPT. This book should not be this much lengthy and other languages must be covered so that every programmer can benefit from it.

I was looking forward to **learn more about refactoring** but he did not cover it in detail which was disappointing as well.

Robert discussed in detail about naming convention which applies to every programming language in general and which were quite helpful.

Yevgeniy Brikman says

A good book to read for any coder - perhaps not as thorough as Code Complete but much more effective than Pragmatic Programmer.

This book's biggest strength is that it includes tons of code examples, including some fairly long and in depth ones. Instead of just listing rules or principles of clean code, many of the chapters go through these code examples and iteratively improve them. The rules and principles fall out of this process and the reader is a part of developing them, which is an effective way to learn.

I also liked the justification for why clean code matters in the intro chapters. However, there was not enough discussion of real world trade offs. The book brushes them aside and claims that the programmer should *always* write the most clean code possible; what is not mentioned is to what extent to do this and when. In fact, the book compares code to poetry and art and makes a point to mention that neither is ever done. And yet, everyone needs to ship at some point. So when is code not just clean, but clean enough?

Some downsides: the chapters have different authors, so a few are weaker than others. Also, the book is too tailored to Java and imperative/OO programming. Similar to Code Complete, this book would benefit from discussing functional programming, which addresses many of the lessons/problems.

Some fun quotes from Clean Code:

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.

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.

You should name a variable using the same care with which you name a first-born child.

Quality is the result of a million selfless acts of care—not just of any great method that descends from the heavens.

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.

Remember that code is really the language in which we ultimately express the requirements.

LeBlanc’s law: Later equals never.

Michael Feathers: 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 some- one who cares deeply about the craft.

Language bigots everywhere, beware! It is not the language that makes programs appear simple. It is the programmer that make the language appear simple!

The ratio of time spent reading vs. writing is well over 10:1.

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, tech- niques, and tools that they use.

The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that.

Functions should do one thing. They should do it well. They should do it only.

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

Master programmers think of systems as stories to be told rather than programs to be written.

The proper use of comments is to compensate for our failure to express ourself in code. Note that I used the word failure. I meant it. Comments are always failures. We must have them because we cannot always figure

out how to express ourselves without them, but their use is not a cause for celebration.

"Objects are abstractions of processing. Threads are abstractions of schedule.” —James O. Coplien

Concurrency is a decoupling strategy. It helps us decouple what gets done from when it gets done.

Boolean arguments loudly declare that the function does more than one thing.

Names in software are 90 percent of what make software readable.

Francis Fish says

The first half of this book is well worth a read. Then I was reminded of Martin Fowler's (I think) comment that the original Design Patterns Elements of Reusable Software book was a response to the limitations of C++. It dovetailed so well into Java because Java has a lot of the same annoying limitations, and in some ways is even harder.

The latter section of the book contains some worked examples that I didn't always agree with because they seemed to be totally over done. A lot of the refactorings came from limitations in the language and even then felt arbitrary and not that "clean", more like differences of opinion.

In light of this I think the book would have been better titled *Clean Java*, and then we'd all know where we stand. Have to say I was disappointed by the case studies. I think if you're a jobbing Java programmer you will get a real benefit from this book. I use dynamic languages like Ruby and most of the problems described in need of refactoring just never happen.

Jerry says

I had a tough time deciding between 3 or 4 stars.

The book should be called Clean Java Code. Some of the concepts definitely translate to other languages, but it uses Java for all of the examples and some of the chapters are dedicated to Java-specific issues.

I consider many of the the suggestions to simply be common sense, but I've worked with enough of "other people's code" to realize they don't necessarily agree. With all of that said, I'd definitely recommend the book to Java developers at the beginner and intermediate levels.

Craig Vermeer says

This had lots of good, practical advice that spanned everything from naming to testing to concurrency. A lot of it was pretty Java centric, so I skimmed a few sections.

By far the best portions of the book were the ones where the author demonstrates -- step by step -- his process for writing code test-first, as well as refactoring.

If you get frustrated with either of the two at times, these parts of the book are *fantastic*, because you see that even someone who's been coding for 40+ years (like Uncle Bob has) writes messy code the first time! He just surrounds it with tests, and then little by little refines it and cleans it up. Awesome.

Oana Sipos says

These are rather notes than a review while reading:

1. Use very descriptive names. Be consistent with your names.
2. A function should not do more than one thing.
3. SRP (Single Responsibility Principle): a class or module should have one, and only one, reason to change.
4. Stepdown rule: every function should be followed by those at the next level of abstraction (low, intermediate, advanced).
5. A long descriptive name is better than a short enigmatic name. A long descriptive name is better than a long descriptive comment.
6. The ideal number of arguments for a function is zero (niladic). Next comes one (monadic), followed closely by two (dyadic). Three arguments (triadic) should be avoided where possible. More than three (polyadic) requires very special justification and then shouldn't be used anyway.
7. Flag arguments are ugly. Passing a boolean into a function is loudly proclaiming that this function does more than one thing. It does one thing if the flag is true and another one if the flag is false.
8. Write learning test when using third-party cody to make sure it behaves the way you expect it to. And if codebase changes in time, at least you find out early enough.

Babak Ghadiri says

?? ???? ??? ???? ???? ????????????? ???? ??? ????. ???? ?? ??????? ?? ??? ???? ??? ?? ?? ??? ????? ???? ????? ????. ?? ?? ???? ?? ?? ??? ?? ??? ?? ?? ??????? ????? ?? ??? ???? ?? ?? ?? ?? ? ?? ??? ?????? ????? ???? ??????? ?? ?? ???? ??????. ??? ???? ??? ???? ????? ???? ? ?? ?? ??? ???? ???? ???? ????? ?? ??????. ??? ?? ??? ????
(Smells and Heuristics)
?? ?????? ????.
?? ???? ???? ?????????? ???? ????? ???? ???? ?????????? ?? ??? ?????
