It's probably time to stop recommending Clean Code

2020-06-28 by qntm

It may not be possible for us to ever reach empirical definitions of "good code" or "clean code", which means that any one person's opinions about another person's opinions about "clean code" are necessarily highly subjective. I cannot review Robert C. Martin's 2008 book *Clean Code* from your perspective, only mine.

That said, the major problem I have with *Clean Code* is that a lot of the example code in the book is *just dreadful*.

*

In chapter 3, "Functions", Martin gives a variety of advice for writing functions well. Probably the strongest single piece of advice in this chapter is that **functions should not mix levels of abstraction**; they should not perform both high-level and low-level tasks, because this is confusing and muddles the function's responsibility. There's other valid stuff in this chapter: Martin says that function names should be descriptive, and consistent, and should be verb phrases, and should be chosen carefully. He says that functions should do exactly one thing, and do it well, which I agree with... provided that we aren't too dogmatic about how we define "one thing", and we understand that in plenty of cases this can be highly impractical. He says that functions should not have side effects (and he provides a really great example), and that output arguments are to be avoided in favour of return values. He says that functions should generally either be commands, which *do* something, or queries, which *answer* something, but not both. This is all reasonable entry-level advice.

But mixed into the chapter there are more questionable assertions. Martin says that Boolean flag arguments are bad practice, which I agree with, because an unadorned `true` or `false` in source code is opaque and unclear *versus* an explicit `IS_SUITE` or `IS_NOT_SUITE`... but Martin's reasoning is rather that a Boolean argument means that a function does more than one thing, which it shouldn't.

Martin says that it should be possible to read a single source file from top to bottom as narrative, with the level of abstraction in each function descending as we read on, each function calling out to others further down. This is far from universally relevant. Many source files, I would even say most source files, cannot be neatly hierarchised in this way. And even for the ones which can, an IDE lets us trivially jump from function call to function implementation and back, the same way that we browse websites.

He says code duplication "may be the root of all evil in software" and fiercely advocates DRY. At the time, this was quite standard advice. In more recent times, however, we generally understand that a little duplication isn't the worst thing in the world; it can be clearer, and it can be cheaper than the wrong abstraction.

And then it gets weird. Martin says that functions should not be large enough to hold *nested* control structures (conditionals and loops); equivalently, they should not be indented to more than two levels. He says blocks should be one line long, consisting probably of a single function call. He says that an ideal function has *zero arguments* (but still no side effects?), and that a function with just three arguments is confusing and difficult to test. Most bizarrely, Martin asserts that an ideal function is *two to four lines of code long*. This piece of advice is actually placed at the start of the chapter. It's the first and most important rule:

> The first rule of functions is that they should be small. The second rule of functions is that *they should be smaller than that*. This is not an assertion that I can justify. I can't provide any references to research that shows that very small functions are better. What I can tell you is that for nearly four decades I have written functions of all different sizes. I've written several nasty 3,000-line abominations. I've written scads of functions in the 100 to 300 line range. And I've written functions that were 20 to 30 lines long. What this experience has taught me, through long trial and error, is that functions should be very small.

> [...]
>
> When Kent showed me the code, I was struck by how small all the functions were. I was used to functions in Swing programs that took up miles of vertical space. Every function in *this* program was just two, or three, or four lines long. Each was transparently obvious. Each told a story. And each led you to the next in a compelling order. *That's* how short your functions should be!

All of this sounds like hyperbole. A case for short functions instead of long ones can certainly be made, but we assume that Martin doesn't *literally* mean that every function in our entire application *must* be four lines long or less.

But the book is being absolutely serious about this. All of this advice culminates in the following source code listing at the end of chapter 3. This example code is **Martin's preferred refactoring** of a pair of Java methods originating in an open-source testing tool, FitNesse.

```
package fitnesse.html;

import fitnesse.responders.run.SuiteResponder;
import fitnesse.wiki.*;

public class SetupTeardownIncluder {
  private PageData pageData;
  private boolean isSuite;
  private WikiPage testPage;
  private StringBuffer newPageContent;
  private PageCrawler pageCrawler;


  public static String render(PageData pageData) throws
Exception {
    return render(pageData, false);
  }

  public static String render(PageData pageData, boolean
isSuite)
    throws Exception {
    return new
SetupTeardownIncluder(pageData).render(isSuite);
  }

  private SetupTeardownIncluder(PageData pageData) {
    this.pageData = pageData;
    testPage = pageData.getWikiPage();
    pageCrawler = testPage.getPageCrawler();
    newPageContent = new StringBuffer();
  }

  private String render(boolean isSuite) throws Exception {
     this.isSuite = isSuite;
    if (isTestPage())
      includeSetupAndTeardownPages();
    return pageData.getHtml();
  }

  private boolean isTestPage() throws Exception {
    return pageData.hasAttribute("Test");
  }

  private void includeSetupAndTeardownPages() throws
Exception {
    includeSetupPages();
    includePageContent();
    includeTeardownPages();
    updatePageContent();
  }


  private void includeSetupPages() throws Exception {
    if (isSuite)
      includeSuiteSetupPage();
    includeSetupPage();
  }

  private void includeSuiteSetupPage() throws Exception {
    include(SuiteResponder.SUITE_SETUP_NAME, "-setup");
  }

  private void includeSetupPage() throws Exception {
    include("SetUp", "-setup");
  }

  private void includePageContent() throws Exception {
    newPageContent.append(pageData.getContent());
  }

  private void includeTeardownPages() throws Exception {
    includeTeardownPage();
    if (isSuite)
      includeSuiteTeardownPage();
  }

  private void includeTeardownPage() throws Exception {
    include("TearDown", "-teardown");
  }

  private void includeSuiteTeardownPage() throws Exception {
    include(SuiteResponder.SUITE_TEARDOWN_NAME, "-teardown");
  }

  private void updatePageContent() throws Exception {
    pageData.setContent(newPageContent.toString());
  }

  private void include(String pageName, String arg) throws
Exception {
    WikiPage inheritedPage = findInheritedPage(pageName);
    if (inheritedPage != null) {
      String pagePathName =
getPathNameForPage(inheritedPage);
      buildIncludeDirective(pagePathName, arg);
    }
  }

  private WikiPage findInheritedPage(String pageName) throws
Exception {
    return PageCrawlerImpl.getInheritedPage(pageName,
testPage);
  }

  private String getPathNameForPage(WikiPage page) throws
Exception {
    WikiPagePath pagePath = pageCrawler.getFullPath(page);
    return PathParser.render(pagePath);
  }

  private void buildIncludeDirective(String pagePathName,
String arg) {
    newPageContent
      .append("\n!include ")
      .append(arg)
      .append(" .")
      .append(pagePathName)
      .append("\n");
  }
}
```

I'll say again: this is Martin's own code, written to his personal standards. This is the ideal, presented to us as a learning example.

I will confess at this stage that my Java skills are dated and rusty, almost as dated and rusty as this book, which is from 2008. But surely, even in 2008, this code was illegible trash?

Let's ignore the wildcard `import`.

First, the class name, `SetupTeardownIncluder`, is dreadful. It *is*, at least, a noun phrase, as all class names should be. But it's a nouned verb phrase, the strangled kind of class name you invariably get when you're working in strictly object-oriented code, where everything has to be a class, but sometimes the thing you really need is just one simple gosh-danged *function*.

Inside the class, we have:

- two public, static methods, as before, plus
- one new private constructor and
- **fifteen** new private methods.

Of the fifteen private methods, fully thirteen of them either have side effects (they modify variables which were not passed into them as arguments, such as `buildIncludeDirective`, which has side effects on `newPageContent`) or call out to other methods which have side effects (such as `include`, which calls `buildIncludeDirective`). Only `isTestPage` and `findInheritedPage` look to be side-effect-free. They still make use of variables which aren't passed into them (`pageData` and `testPage` respectively) but they *appear* to do so in side-effect-free ways.

At this point you might reason that maybe Martin's definition of "side effect" doesn't include member variables of the object whose method we just called. If we take this definition, then the five member variables, `pageData`, `isSuite`, `testPage`, `newPageContent` and `pageCrawler`, are implicitly passed to *every* private method call, and they are considered fair game; any private method is free to do anything it likes to any of these variables.

But Martin's own definition contradicts this. This is from earlier in this exact chapter, with emphasis added:

> Side effects are lies. Your function promises to do one thing, but it also does other hidden things. **Sometimes it will make unexpected changes to the variables of its own class.** Sometimes it will make them to the parameters passed into the function or to system globals. In either case they are devious and damaging mistruths that often result in strange temporal couplings and order dependencies.

I like this definition. I agree with this definition. It's a useful definition, because it enables us to reason about what a function does, with some degree of confidence, by referring only to its inputs and output. I agree that it's bad for a function to make *unexpected* changes to the variables of its own class.

So why does Martin's own code, "clean" code, do *nothing but this*? Rather than have a method pass arguments to another method, Martin makes a distressing habit of having the first method set a member variable which the second method, or some other method, then reads back. This makes it incredibly hard to figure out what any of this code does, because all of these incredibly tiny methods do almost nothing and work exclusively through side effects.

Let's just look at one private method.

```
private String render(boolean isSuite) throws Exception {
   this.isSuite = isSuite;
  if (isTestPage())
    includeSetupAndTeardownPages();
  return pageData.getHtml();
}
```

So... imagine that someone enters a kitchen, because they want to show you how to make a cup of coffee. As you watch carefully, they flick a switch on the wall. The switch looks like a light switch, but none of the lights in the kitchen turn on or off. Next, they open a cabinet and take down a mug, set it on the worktop, and then tap it twice with a teaspoon. They wait for thirty seconds, and finally they reach behind the refrigerator, where you can't see, and pull out a *different* mug, this one full of fresh coffee.

...What just happened? What was flicking the switch for? Was tapping the empty mug part of the procedure? Where did the coffee come from?

That's what this code is like. Why does `render` have a side effect of setting the value of `this.isSuite`? When is `this.isSuite` read back, in `isTestPage`, in `includeSetupAndTeardownPages`, in both, in neither? If it does get read back, why not just pass `isSuite` in as a Boolean? Or perhaps the caller reads it back?

Why do we return `pageData.getHtml()` when we never touched `pageData`? How did the HTML get there? Was it already there? We might make an educated guess that `includeSetupAndTeardownPages` has side effects on `pageData`, but then, what? We can't know either way until we look. And what other side effects does that have on other member variables? The uncertainty becomes so great that we suddenly have to wonder if calling `isTestPage()` could have side effects too.

How would you unit-test this method? Well, you can't. It's not a unit. It can't be separated from the side-effects it has on other parts of the code. (And what's up with the indentation? And where are the danged braces?)

Martin states, in this very chapter, that it makes sense to break a function down into smaller functions "if you can extract another function from it with a name that is not merely a restatement of its implementation". But then he gives us:

```
private boolean isTestPage() throws Exception {
  return pageData.hasAttribute("Test");
}
```

and:

```
private WikiPage findInheritedPage(String pageName) throws
Exception {
  return PageCrawlerImpl.getInheritedPage(pageName,
testPage);
}
```

and half a dozen others, none of which are even called from more than one location.

There is at least one questionable aspect of this code which isn't Martin's fault: the fact that `pageData`'s content gets destroyed. Unlike the member variables (`isSuite`, `testPage`, `newPageContent` and `pageCrawler`), `pageData` is not actually ours to modify. It is originally passed in to the top-level public `render` methods by an external caller. The `render` method does a lot of work and ultimately returns a `String` of HTML. However, during this work, as a side effect, `pageData` is destructively modified (see `updatePageContent`). Surely it would be preferable to create a brand new `PageData` object with our desired modifications, and leave the original untouched? If the caller tries to use `pageData` for something else afterwards, they might be very surprised about what's happened to its content. But this is how the original code behaved prior to the refactoring, and the behaviour could be intentional. Martin has preserved the behaviour, though he has buried it very effectively.

Some other mild puzzles: why do we use `pageData.hasAttribute("Test")` to figure out whether this is a test page, but we have to consult a separate Boolean to figure out whether or not this is a test *suite* page? And what exactly is the separation of concerns between `PageCrawler` and `PageCrawlerImpl`, both of which are in use here?

*

Is the whole book like this?

Pretty much, yeah. *Clean Code* mixes together a disarming combination of strong, timeless advice and advice which is highly questionable or dated or both.

Much of the book is no longer of much use. There are multiple chapters of what are basically filler, focusing on laborious worked examples of refactoring Java code; there is a whole chapter examining the internals of JUnit. This book is from 2008, so you can imagine how relevant that is now. There's a whole chapter on formatting, which these days reduces to a single sentence: "Pick a sensible standard formatting, configure automated tools to enforce it, and then never think about this topic again."

The content focuses almost exclusively on object-oriented code, and extols the virtues of SOLID, to the exclusion of other programming paradigms. Object-oriented programming was very fashionable at the time of publication. Martin himself invented three of the five principles which make up SOLID, and popularised the term. But the total absence of functional programming techniques or even simple procedural code was regrettable even then, and has only grown more obvious in the years since. SOLID is something of a non-sequitur in these domains. Even in the OO domain, opinions of SOLID have been steadily lowering in recent years.

The book focuses on Java code, to the exclusion of other programming languages, even other object-oriented programming languages. Java was popular at the time, and if you're writing a book like this, it makes sense to pick a single well-known language and stick with it, and Java is still very popular and may still be a strong choice for this purpose. But the book's overall use of Java is very dated.

This kind of thing is unavoidable — programming books date legendarily poorly. That's part of the reason why *Clean Code* was a recommended read at one time, and I now think that the pendulum is swinging back in the opposite direction.

But even for the time, even for 2008-era Java, much of the provided code is *bad*.

There's a chapter on unit testing. There's a lot of good, basic, stuff in this chapter, about how unit tests should be fast, independent and repeatable, about how unit tests enable more confident refactoring of source code, about how unit tests should be about as voluminous as the code under test, but strictly simpler to read and comprehend. But then he shows us a unit test with what he says is too much detail:

```
@Test
  public void turnOnLoTempAlarmAtThreashold() throws
Exception {
    hw.setTemp(WAY_TOO_COLD);
    controller.tic();
    assertTrue(hw.heaterState());
    assertTrue(hw.blowerState());
    assertFalse(hw.coolerState());
    assertFalse(hw.hiTempAlarm());
    assertTrue(hw.loTempAlarm());
  }
```

and he proudly refactors it to:

```
@Test
  public void turnOnLoTempAlarmAtThreshold() throws Exception
{
    wayTooCold();
    assertEquals(“HBchL”, hw.getState());
  }
```

This is done as part of an overall lesson in the virtue of *inventing a new domain-specific testing language for your tests*. I was left so confused by this suggestion. I would use exactly the same code to demonstrate exactly the opposite lesson. Don't do this!

Which is to say nothing of the method named `wayTooCold`. This is an adjective phrase. It's entirely unclear what this method does. Does it *set* the world's state to be way too cold? Does it *react* to the world's state *becoming* way too cold? Or is it an *assertion* that the world's state currently must *be* way too cold?

Methods should have verb or verb phrase names like `postPayment`, `deletePage`, or `save`. That's not me saying that. That's a direct quote from this book. Chapter 2, "Meaningful Names":

> Methods should have verb or verb phrase names like `postPayment`, `deletePage`, or `save`.

This is perfectly sound advice. And `hw.setTemp(WAY_TOO_COLD);` was a perfectly unambiguous line of code. What gives?

And even if you guess correctly that calling `wayTooCold()` *sets* the temperature to be way too cold... there's no way that you could guess that it also calls `controller.tic()` internally. Earlier, we were advised to avoid code having side effects. This, also, was sound advice. And it is, again, being ignored in the actual written code example.

(And since we're here, this, the original unrefactored code, is a fine demonstration of the drawbacks of unadorned Booleans. What does it mean when, say, `coolerState` returns `true`? Does it mean that the cooler's current state is good, *i.e.* cold enough, *i.e.* switched off? Or does it mean that it is powered on, and actively cooling? An `enum` with a few values, `ON` and `OFF`, could be less ambiguous.)

*

The book presents us with the TDD loop:

> **First Law** You may not write production code until you have written a failing unit test.
>
> **Second Law** You may not write more of a unit test than is sufficient to fail, and not compiling is failing.
>
> **Third Law** You may not write more production code than is sufficient to pass the currently failing test.
>
> These three laws lock you into a cycle that is perhaps thirty seconds long. The tests and the production code are written together, with the tests just a few seconds ahead of the production code.

But the book doesn't acknowledge the missing zeroth step in the process: figuring out how to break down the programming task in front of you, so that you can take a minuscule thirty-second bite out of it. *That*, in many cases, is exceedingly time-consuming, and frequently obviously useless, and frequently impossible.

*

There's a whole chapter on "Objects and Data Structures". In it, we're provided with this example of a data structure:

```
public class Point {
  public double x;
  public double y;
}
```

and this example of an object (well, the interface for one):

```
public interface Point {
  double getX();
  double getY();
  void setCartesian(double x, double y);
  double getR();
  double getTheta();
  void setPolar(double r, double theta);
}
```

Martin writes:

> These two examples show the difference between objects and data structures. Objects hide their data behind abstractions and expose functions that operate on that data. Data structure expose their data and have no meaningful functions. Go back and read that again. Notice the complimentary nature of the two definitions. They are virtual opposites. This difference may seem trivial, but it has far-reaching implications.

And... that's it?

Yes, you're understanding this correctly. Martin's definition of "data structure" disagrees with the definition everybody else uses. This is a very strange choice of definition, though Martin does at least define his term clearly. Drawing a clear distinction between objects as dumb data and objects as sophisticated abstractions with methods is legitimate, and useful. But it's quite glaring that there is no content in the book *at all* about clean coding using what most of us consider to be real data structures: arrays, linked lists, hash maps, binary trees, graphs, stacks, queues and so on. This chapter is *much* shorter than I expected, and contains very little information of value.

*

I'm not going to rehash all the rest of my notes. I took a lot of them, and calling out everything I perceive to be wrong with this book would be counterproductive. I'll stop with one more egregious piece of example code. This is from chapter 8, a prime number generator:

```
package literatePrimes;

import java.util.ArrayList;

public class PrimeGenerator {
  private static int[] primes;
  private static ArrayList<Integer> multiplesOfPrimeFactors;

  protected static int[] generate(int n) {
    primes = new int[n];
    multiplesOfPrimeFactors = new ArrayList<Integer>();
    set2AsFirstPrime();
    checkOddNumbersForSubsequentPrimes();
    return primes;
  }

  private static void set2AsFirstPrime() {
    primes[0] = 2;
    multiplesOfPrimeFactors.add(2);
  }

  private static void checkOddNumbersForSubsequentPrimes() {
    int primeIndex = 1;
    for (int candidate = 3;
         primeIndex < primes.length;
         candidate += 2) {
      if (isPrime(candidate))
        primes[primeIndex++] = candidate;
    }
  }

  private static boolean isPrime(int candidate) {
    if
(isLeastRelevantMultipleOfNextLargerPrimeFactor(candidate)) {
      multiplesOfPrimeFactors.add(candidate);
      return false;
    }
    return isNotMultipleOfAnyPreviousPrimeFactor(candidate);
  }

  private static boolean
  isLeastRelevantMultipleOfNextLargerPrimeFactor(int
candidate) {
    int nextLargerPrimeFactor =
primes[multiplesOfPrimeFactors.size()];
    int leastRelevantMultiple = nextLargerPrimeFactor *
nextLargerPrimeFactor;
    return candidate == leastRelevantMultiple;
  }

  private static boolean
  isNotMultipleOfAnyPreviousPrimeFactor(int candidate) {
    for (int n = 1; n < multiplesOfPrimeFactors.size(); n++)
{
      if (isMultipleOfNthPrimeFactor(candidate, n))
        return false;
    }
    return true;
  }

  private static boolean
  isMultipleOfNthPrimeFactor(int candidate, int n) {
   return
     candidate ==
smallestOddNthMultipleNotLessThanCandidate(candidate, n);
  }

  private static int
  smallestOddNthMultipleNotLessThanCandidate(int candidate,
int n) {
    int multiple = multiplesOfPrimeFactors.get(n);
    while (multiple < candidate)
      multiple += 2 * primes[n];
    multiplesOfPrimeFactors.set(n, multiple);
    return multiple;
  }
}
```

What the heck is this code? What is this algorithm? What are these *method names*? `set2AsFirstPrime`?

`smallestOddNthMultipleNotLessThanCandidate`? Why does the code break with an out-of-bounds exception if we replace the `int[]` with a second `ArrayList<Integer>`? Earlier, we were advised that a method should be a command, which *does* something, or a query, which *answers* something, but not both. This was good advice, so why do nearly all of these methods ignore it? And what of thread safety?

Is this meant to be clean code? Is this meant to be a legible, intelligent way to search for prime numbers? Are we supposed to write code like this? And if not, why is this example in the book? And where is the "real" answer?

If this is the quality of code which this programmer produces — at his own leisure, under ideal circumstances, with none of the pressures of real production software development, *as a teaching example* — then why should you pay any attention at all to the rest of his book? Or to his other books?

*

I wrote this essay because I keep seeing people recommend *Clean Code*. I felt the need to offer an anti-recommendation.

I originally read *Clean Code* as part of a reading group which had been organised at work. We read about a chapter a week for thirteen weeks. (The book has seventeen chapters, we skipped some for reasons already mentioned.)

Now, you don't want a reading group to get to the end of each session with nothing but unanimous agreement. You want the book to draw out some kind of reaction from the readers, something additional to say in response. And I guess, to a certain extent, that means that the book has to either say something you disagree with, or not say everything you think it should say. On that basis, *Clean Code* was *okay*. We had good discussions. We were able to use the individual chapters as launching points for deeper discussions of *actual* modern practices. We talked about a great deal which was not covered in the book. We disagreed with a lot in the book.

Would I recommend this book? **No.** Would I recommend it as a beginner's text, with all the caveats above? No. Would I recommend it as a historical artifact, an educational snapshot of what programming best practices used to look like, way back in 2008? No, I would not.

*

So the killer question is, what book(s) would I recommend instead? I don't know. Suggestions in the comments, unless I've closed them.

Update, 2020-12-19

After suggestions from comments below, I read *A Philosophy of Software Design* (2018) by John Ousterhout and found it to be a much more positive experience. I would be happy to recommend it over *Clean Code*.

A Philosophy of Software Design is not a drop-in replacement for *Clean Code*. As the title suggests, it focuses more on the practice of software design at a higher level than it does on the writing or critiquing of actual individual lines of code. (As such, it contains relatively few code examples.) Because it's aimed at this higher level, I think it's possibly not a suitable read for absolute beginner programmers. A lot of this high-level theory is difficult to comprehend or put into practice until you have some real experience to compare it with. I think there actually is something of a gap in the market for an entry-level introductory programming text right now.

Having said that, I found *A Philosophy of Software Design* to be informative, cogent, concise and *much* more up-to-date. I found that I agreed with nearly all of Ousterhout's assertions and suggestions for good software design, many of which are diametrically opposed to those found in *Clean Code*. By virtue of providing relatively high-level advice, I feel that the book is likely to age rather better too.

Of course, software development is still moving forwards as I write this. Who knows what a good programming book will look like in ten more years?

Afterword, 2022-03-21

Probably the most common defence of *Clean Code* that I've seen coming from readers of this essay is that the book *is* still worth recommending, despite the objections above, because the advice in the book shouldn't be taken entirely literally, or applied dogmatically. The burden is on the reader to think critically, draw their own conclusions, and selectively ignore the book's advice when that advice is bad. Even the book itself says:

> [M]any 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.

I don't think this is a particularly convincing defence of the book containing bad advice and bad example code in the first place.

It's true that we should always engage critically with material instead of passively letting it wash over us. This is universally understood; this is what reading *is*; this is what the essay above is. This doesn't need to be stated, and it's redundant for the book itself to mention the point at all. This isn't something which can be used to shield a book from criticism.

What's more important is what the book actually says. And, especially in an instructional text like this, *how carefully* the book has to be read in order to get a positive experience out of it, and what happens if the book isn't read sufficiently critically.

Experienced programmers will get almost nothing out of reading *Clean Code*. They'll be able to weigh the advice given against their own experiences and make an informed decision — and the book will tell them almost nothing that they didn't learn years ago.

Inexperienced programmers, meanwhile — and *Clean Code* is an entry-level programming text, so this is the target audience, whose experiences are most important — won't be able to distinguish the good advice from the bad, and won't be able to see that the example code is bad and shouldn't be imitated. Inexperienced programmers will take those lessons at face value, and it might be years before they figure out how badly they were misled.

Back to Blog
Back to Things Of Interest

Discussion (118)

2020-06-28 20:39:13 by qntm:

Side note, Clean Architecture is absolute garbage, I didn't get through it. However, I think there's general agreement on that topic.

2020-06-29 00:21:17 by Gary Stephenson:

imho, the best book I ever read on programming was "Concepts, Techniques and Models of Computer Programming" by Peter Van Roy. Alas the language it expounds (Oz / Mozart) is pretty much dead (still-born?). I think the closest thing to it I've seen that is still somewhat alive would probably be Picat.

2020-06-29 00:24:18 by Cgk:

"Working effectively with legacy code" by Michael C. Feathers was (and is) a practical book.

It provides techniques for working with existing code, which is likely to happen more often than working things from scratch.

2020-06-29 00:41:41 by qntm:

All code is legacy. It's like how the speed of light is finite, which means technically you're always looking at the world as it was at some time in the past. You commit some new code, you go for a coffee break, you come back - it's legacy code now.

2020-06-29 01:31:36 by kevin:

Michael Feathers has his own definition of legacy code. Quote from the book's back cover: "Is your code easy to change? Can you get nearly instantaneous feedback when you do change it? Do you understand it? If the answer to any of these questions is no, you have legacy code, [...]"

It should really be called "Working Effectively with Garbage Code".

2020-06-29 01:41:43 by kevin:

I've seen a lot of backlash against Clean Code recently. It was transformative for me as a software greenhorn, introducing me to concepts like testability and readability, which they didn't tell us about in college. It gave me a vocabulary to discuss program design. And it was short and easily digestible, minus some tedious examples. So until someone comes up with a wartless alternative, I'll continue recommending it to newbies, with the caveat that they should keep a grain of salt to hand and skip the boring parts. As long as you have more experienced devs around to show you actually good programming, it can't do much harm.

2020-06-29 02:43:58 by Stephen Paul Weber:

If you want a book about refactoring and OOP and such that is modern and not by a terrible human, https://www.sandimetz.com/99bottles is great

2020-06-29 02:54:23 by Greg johnson:

A philosophy of software design by John Ousterhaut. This is a short and accessible book that penetrates to the heart of deep truths about software development. Single best book I've read on how to write software.

2020-06-29 03:57:06 by kazer:

Someone made the point about legacy code but that term is rather dubious and used often just as derogatory term as opposed to something written in new trendier tools.
Working useful code is quite often reused or called upon from other pieces of code, just look at any operating system and oldest piece of code it has: is it working? if so, why change it?
Calling code "legacy" is missing the point: if it is garbage it is one thing, if it is written for a platform you no longer have that is quite another.

2020-06-29 05:03:49 by Samuel Backus:

I often recommend Sandi Metz books

2020-06-29 05:33:40 by phil:

I'd recommend Pragmatic Programmer book.

2020-06-29 06:03:17 by Andrei:

I'm ruby on rails dev and one of the thoughtbot guys recommended I read Practical Object-Oriented Design in Ruby by Sandi Metz few years ago. I think it's very good. I think they follow her rules in their workflow. Also, you can see it in their teaching platform - upcase.com.

I believe those resources help me write my web apps in a way that I'm not embarrassed for.

2020-06-29 06:18:30 by gnusosa:

The two other books that are absolute garbage are Code Complete and The Clean Architecture books. I wouldn't recommend them at all. Thanks for creating a counter argument to all the people that create dogma out of this book.

2020-06-29 07:02:25 by Daniel:

What is your definition of a data structure? I started with Pascal in 1991 and my definition matches Martin's. I do agree with your conclusion about Martin's work however.

2020-06-29 07:44:01 by tomjakubowski:

I recommend The Practice of Programming by Kernighan and Pike, despite many of their publicly stated opinions on the subject. I would add that the "hardware state" DSL would actually be good if it were also used in places other than tests, for example as a short state representation for debugging in logs, or for setting state in an emulator or test rig.

2020-06-29 07:52:44 by heldev:

The book as you pointed out has issues. But not sure that there is an option to stop recommending it. Yes there is Code Complete but it's hard to sell because of the size and style. It's a great book and I would recommend it to people who want/can read books. It explains everything, not dogmatic, has academic foundations, but most of developers want short and simple ready to use solutions they are not going to read it.
So if you are interested in improving overall code quality Clean Code is still "thewirecutter's choice for most of the people " and Code Complete is a memorable mention "the best ... money can buy for advanced users"

2020-06-29 08:16:18 by Maksym:

Another recommendedation for A Philosophy of Software Design by John Ousterhaut. Its scope is intentionally limited to issues directly related to design, but on those it really shines. It also has a high density of insights per page, which I've found not to be the case in several other similar books.

2020-06-29 08:16:56 by ALB:

An excellent alternative is John Ousterhout's "A Philosophy of Software Design." A concise, brilliant book that has a lot of implementation detail that is thought-through thoroughly and presented intelligently.

2020-06-29 09:02:58 by FeepingCreature:

I can't find the source, but there's a quote going around about some other book along the lines of "Where it is true, it's trivial; where it is nontrivial, it's wrong."

2020-06-29 09:38:09 by Tor:

As little as possible, as fast as possible.

2020-06-29 10:21:12 by TZ:

I reread your breakdown of the FitNesse code example a few times and I felt less and less generous about the analysis each time. Here's two and half questions:

1) If private fields are NOT to be considered implicit arguments to every private method, what exactly are they for? (And if they should not be modified, why would they be anything other than 'final'?)

2) Why do you consider Martin's definition of side-effect, which speaks of 'global state' and 'unexpected changes' to include changes to private members of the class in functions that document those changes via their signature?

You could answer those uncharitably and make Uncle Bob look bad, or you could answer them charitably and then his code makes sense.

I also have to say I don't understand how 'illegible trash' is remotely an appropriate description. This is as good as just about anything I've seen in actual production work in 10 years. It's extremely readable and quite testable. There are nits to pick (and you've done an excellent job of that), but for some part of those at least I feel like they're to be picked with 2008 Java API and coding standards rather than the author.

I can't see that reading this book would be a net negative for 95% of people writing OO code. If you also don't know of a better book either, I'm stumped as to why you'd stop recommending it.

Finally, it looks to me like there's people out to get Uncle Bob cancelled for not being on-board with a certain political invasion of software that's going around at the moment (you can already see a comment calling him 'a terrible human'), so I'm somewhat suspicious of ulterior motives here. If that's not your intent, my apologies.

I did like this article in general and found it to be plenty insightful, so I'd love to read more from you.

2020-06-29 11:07:27 by BobStannerz:

Clean code is a very useful book and will be helpful to 99% of Junior devs out there. You want to argue that parts of it (or code examples) are less good, do it, but if you expect to agree 100% with 400 pages of material to classify a book as "good" or "useful" you are misguided.

Its principles are mostly solid. We are developers, not toddlers to be spoon fed, read it and use critical thinking to decide what to keep and what to throw away. You would be hard pressed to find another books with such a good ration of keep / throw.

So is it time to stop recommending it, absolutely not. Are there other books out there that outline the same principles more or less, sure.

I will not go into the politics, because judging a book and ideas based on the creators political ideas is idiotic. Attack the ideas not the person.

2020-06-29 11:53:14 by Sander Vermeer:

I think the idea of clean code is ridiculous. Code should simply be straightforward. A common pitfall is over-engineering, where the code is made more complicated than needs to be.

But I see this in every aspect of programming, where people spend hours in creating a build file (Cmake etc.), creating UML diagrams, pre-defining API's, but also create unnecessary functions and classes ('I might need that later on') etc. etc. Which honestly is just a waste of time.
Since programming is an inherent iterative process, code should be written as such. The general idea is simple; create a quick and dirty solution for the problem at hand. Than, refine the code so it's more reusable, more readable, more testable. More often than not, the API and data structures needed to solve that problem naturally emerge. Also, because the code was developed bottom up, any oversights or over-complications are easily avoided.

Oh and what most people lost over time is the ability to think about programming as the transformation of data. In game development for instance, every game is essentially 1) handle user input, 2) generate images and sound. Of course there are thousands of steps in between (you need to simulate some system/world, load assets from disk, populate the world, calculate animations, create a rendering pipeline), but every step is still the manipulation of data into a desired result. And I think that way of reasoning about your code is lost when OOP became the defacto paradigm.

2020-06-29 13:54:32 by Paddy3118:

I gave up on print, and read copious blog posts on methodology. Many of them, as does this, comment on printed works, but you get much more real life views and a feel for those that are trying to buy-in and those that have a way that works for them.

On your specific comments, I would reject the book if I opened it to find it put 50 character identifiers in a simple prime number generator.

Thank you so much for your well written critique. I enjoyed it, (although that could be mostly down to agreeing with you). :-)

2020-06-29 18:29:54 by Elf M. Sternberg:

The thing that drives me absolutely crazy about Martin's approach is his almost religious insistence that one not "choose a database" before writing code. The relational algebra is one of the most important algorithms of the 20th century, and yet Martin wants you to write and manage your own relationships in the code you write yourself. To me, writing your own relational management code when you have a perfectly solid enforcement mechanism already present is programmatic malpractice.

The other thing that I find infuriating is that I'll run into people who insist "no frameworks!" Martin says that your code shouldn't be about Javabeans or Django, it should be about what it's about: if you're writing a payroll app, the app's repository should say "payroll!" Which is great advice. But every time I've looked over an application written to "clean code" standards, I see folders labeled "domain," "services," "use_cases," "gateways," "adapters." I call this "Writing to the Clean Code Framework," because that's exactly what it is.

2020-06-29 19:58:30 by Hamled:

One thing I remember from watching the old SICP lecture videos was this idea that you should look at your code as defining a new language, and factor your code with functions which are the most useful terms in that language. Pretty basic advice I suppose, but it feels relevant to what Martin is talking about in this book. If you factor your code into functions that are minimal but necessary, you can provide clarity through both the structure and explicit naming of meaningful concepts/actions.

However, based on these examples it seems like Martin has perverted that idea quite a bit. He's taken this core of a good idea and morphing it into an odd approach to self-documenting code: instead of writing a comment next to every few lines, you put it into a function named the same as what that comment would have been.

Of course, the commenting style is noisy but broadly innocuous. Doing the same with factoring instead ends up implying many things about the code that are orthogonal to the original goal of documentation. Basically zero of these tiny functions he's created in these examples are called in multiple places, for example. And of course it's throwing away the point of abstraction if you use a naming convention that seeks to encompass every detail of the unit being named.

I think his prime number generator example contains another great example of these problems, not mentioned in the post. His code for checking whether a candidate is "a multiple of any previous prime factor" actually has side-effects despite being named like a "query" function in his terminology. Not only that, the side-effect is modifying an array while that array is being iterated. This is probably just how a prime sieve works (IDK), but it's far from obvious that is what's going on because the mutative function is multiple calls away from the function with the loop.

I wonder how much Martin has been subject to code reviews throughout his career.

2020-06-30 10:42:06 by George Bellarious:

I think there's a collision between OO and FP thinking. Everything about the FP Kool-Aid is counter to OO ideas:

1) Focus on the data and its transformations (not the messages between objects).

2) Don't mutate (without mutation, objects just become bags of functions).

3) Pass in everything you need (as a commenter above said, that defeats the whole purpose of an object's properties).

I think the two camps are incompatible, even though some OO ideas seem to be tending towards FP's ideas.

OO damaged my brain, FP seems to be restoring it :)

2020-06-30 11:08:52 by Nico:

Thanks for this great rebuttal to Martin's views. I've seen too many people holding this book as a state-of-the-art reference on how to write code.

For me the main issue I see with his approach is his obsession of having zero-comment code. I find this crazy, as it leads to these myriads of weirdly named methods you describe, and end up making the code absolutely unreadable. Both of the example you gave would be so much more understandable as a couple of well-commented methods.

Sure, comments are bad if they're not looked after when you are modifying your code. But I suppose the same can be said about renaming all these 50-character long identifiers littered in Martin's code!

As another commenter put it, this book is still commendable for making people aware that the way you write code matters. Apart from that, it's pretty much garbage.

2020-06-30 11:55:09 by Matthew Harris:

OK that was a tough read as it was a transformative book for me, but I can't argue with the points you have raised.<br/>
<br/>
ps, I had to google the answer to the captcha...

2020-06-30 13:28:54 by Robert C. Martin:

My book is always on top of bestseller.

2020-06-30 13:37:39 by Chris Jenkins:

What is your definition of a data structure and now does it differ from Bob Martin's? From my perspective, I like the definition "Data structures expose their data and have no meaningful functions" as a way of distinguishing the way that data structures are used in functional programming from the way that objects are used in OO.

2020-06-30 14:50:09 by Christian Clausen:

Very interesting read. Like many others I found Clean Code to be enormously helpful when I read it, and I still love the "same abstraction" smell which you also point out. I think there is a lot of good in it, and it is older than most software ever gets.

As it happens I am writing a book with the specific purpose of giving junior developers a quick practical start on code smells and refactoring. It is not finished yet, but you can still check it out; it is called Five Lines of Code (https://www.manning.com/books/five-lines-of-code). As you can guess from the title, I agree with Uncle Bob that smaller methods are easier to understand, but mostly I use it to force people to refactor. If you check it out I would love you hear your thoughts.

2020-06-30 17:38:27 by Sam J M:

Back when I was a young coder many moons ago I tried my damnedest to study and practice what uncle Bob was preaching. All my colleagues were raving about uncle Bob's masterpieces. I thought I was deficient for not being able to come to terms with and accept the folksy uncle's gospel truth.

Working in a company designing and producing embedded systems and controllers drilled into my psyche the importance of clear requirements, and most of the errors we encountered were a result of imprecise requirements. This is something uncle Bob doesn't bother about, instead, he goes on about craftsmanship and professionalism.

Later when I moved to larger software projects, the best ones were those that were readable, unassuming, well commented, and didn't go crazy over some religious doctrine of code styles.

Uncle Bob also conveniently ignores the engineering aspect of large projects with multiple distributed teams working across different timezones and cultures and the multitude of challenges they bring. Most code is not written by 'craftsmen' who follow a codestyle religion. A lot of requirements reviews, design reviews, and so on...

I recently heard him talk about some software project he had worked for some sort of content management, where he went on to say the database was an afterthought and unimportant. I lost all respect for the guy after that.

2020-06-30 17:44:15 by MH:

After reading almost a hundred books on programming over the last several years, I still recommend "Clean Code" as my favorite. As a new developer I didn't learn in school to keep methods simple and names readable. The emphasis the book places on those two things is worth the price alone.

2020-06-30 18:18:48 by AstroMan:

Responding to MH's comment about keeping names readable, Uncle Bob's own code is the antithesis of this commandment that he preaches.

Anyway, he always gave me creepy vibes. Search for his ITkonekt 2019 speech on your favorite video portal, and you can see him feeling up two ladies on stage for two whole minutes at the start of the video.

2020-06-30 18:30:44 by waitingForTheStorm:

When I was in university (back when the dinosaurs ruled the earth), required reading was The Elements of Programming Style by Kernighan and Plauger. Forty one years of active software development, and that monograph still occupies a prominent spot on my bookshelf. I favor a style that is based on a top-to-bottom clear and logical progression of statements, using loops and select/case (or equivalent) elements that are designed to reduce the cyclomatic complexity of the code. My primary rule is never to enforce policy at a point too deep in the hierarchy of logical functions. I avoid side effects, even in loop controls, and stay away from overly complex idioms.

2020-06-30 18:55:46 by Chance:

I think this article is mostly spot-on.

However, the bit about data-structures vs objects might be pedantic in a way that misses the forest for the trees and subtracts from an otherwise excellent set of observations.

Many languages giving you operational functionality with the structures "for free" doesn't really change what the "structure" is. And in fact, even if you use a custom-type in your structure, it doesn't turn it into an object. The real "structure" is just the what. The accompanying operational functionality is just why you've chosen using that type over whatever others.

Sure, you can go implement your PrimeAddressedOnlyLinkedList "data-structure". I'd argue that your implementation code for this new type is not *really* a "data-structure" at all, it's obviously a new object.

However, an instance of your new type is totally a "data-structure".

2020-06-30 18:57:11 by Jay:

You point out that the first example (SetupTeardownIncluder) is filled with side effects, but it isn't. Calling the static methods that the class exposes doesn't change the state of such class. It creates a new object, and calls a method to it. These methods are idempotent, and everything is the same after the caller calls SetupTeardownIncluder.render().

2020-06-30 19:45:36 by Christoph:

For new Java developers I can recommend the book Java by comparison, especially for students.

I think that clean code is a more theoretic take on the problem. In practice there are things you can't avoid and some of the things from Clean Code are good as inspiration

2020-06-30 20:21:18 by Chris:

Unit Testing Principles, Practices, and Patterns. Though its name suggests it's about unit testing, it's actually about so much more than this, including clean code and clean architecture. We are going through this book at work right now, and it incites a lot of discussions.

2020-06-30 21:28:04 by Mike:

I think now I know the source of all this bullshit in our project. I see the same code every day: lots of side effects, small and unclear functions, dozens of structures to wrap arguments...
The man who writes it - prays for this book and often says it is the best book ever.

2020-06-30 22:21:58 by Mark Smeltzer:

"Refactoring to Patterns" by Joshua Kerievsky.

While the book contains lots of highly tactical chapters on how to apply specific patterns to messy code, the book is primarily about what not to do: viz., taking patterns too far.

His chapters on doing TDD right is gold. Best stuff written on the subject at time of print. I also haven't bought any new books on the subject since, but Kerievsky's method should stand the test of time.

2020-06-30 22:37:18 by Hogan:

Thank you for this review. I feel the same way, but couldn't put it in words like you did.

2020-07-01 15:29:18 by hehehe:

What's your thoughts on You Don't Know JS by Kyle Simpson et al?

I think a key benefit it's got is it's somewhat a living document, as each iteration is written collaboratively with open source contributors.

2020-07-01 19:46:52 by Rastislav Svoboda:

watch https://www.youtube.com/watch?v=l-gF0vDhJVI
approx @1:32:20 and later
... but what you will find in those case studies is how we break the rules, because we always break the rules, the case studies show the pragmatic application of the rules and we try to hold to those rules as much as we can ...

2020-07-03 00:28:46 by kpreid:

The test code you criticize.
{ assertEquals(“HBchL”, hw.getState()); }
does have one practical advantage over the set of assertFalse and assertTrue it is replacing: it lets you see the entire set of outputs of the controller in the test failure output, whereas the other version gives you only one of the outputs, unless you either run it under a debugger or change the order of the asserts to make another one fail first.

That said, if I were spending time on engineering the tests, what I'd do is write test helper methods for the assertions so that when *any* of those boolean assertions fail, *all* of the state is printed (which can then be less cryptic than “HBchL”). Or, use a test framework that doesn't abort execution as soon as a single assertion fails, but presents a list of all of the failures.

Another common example of unhelpful test failure I have seen is {
  assertEquals(2, list.size());
  assert something about list.get(0);
  assert something about list.get(1);
} where if the list is not the expected size, you have no information about what it actually contained.

I have found that in complex code, test failures that explain what went wrong in detail are very helpful because if you broke something *that you didn't expect to break* — that might be a distant dependent of the thing you changed — then the test having a good description of the problem means you can proceed like "oh, I didn't realize I need to handle that case—there, done". instead of having to fire up the debugger or reading lots of code that you weren't intending to be working on.

2020-07-03 03:16:57 by Mike Bridge:

You've made several mistakes here. Briefly:

- A boolean flag argument is usually a bad idea whether you give it a label or not. The point---which you seem to acknowledge and then ignore---is that a boolean flag *may* indicate that you have two paths through your code. Similarly, two boolean flags *may* mean that you have four. This has nothing to do with the naming of the boolean argument or having to do extra work to navigate the code.

- Martin is *not* saying FitNesse example is "ideal" code. His goal in chapter 3---still early in the book---is to give the high-level view of the technique for making garbage code into humanly-understandable and maintainable code by refactoring, with the help of some rules-of-thumb. Really, this is an introduction to the whole point of the book. Sure it's not perfect, but he's not claiming it is.

- It sounds to me like you're missing the big picture when you say there's a "missing fourth step" in TDD. He *doesn't't* say that you break down the problem into abstractions first, then write code, then test it---he's saying it's the opposite: you write tests, then write code, then refactor it, then keep repeating the process until you have a good abstraction. Your first iteration may well be a mess, but he's saying that that doesn't matter---you can work on finding better abstractions once the code works. In other words, it's much harder to write good code if you try to figure it all out in your head out a-priori.

- I'm not sure how you define "Data Structures", despite the link to WikiPedia. How is this different from Martin's definition?

- And *yes*, set2AsFirstPrime and smallestOddNthMultipleNotLessThanCandidate make perfect sense to me. Do you have a better suggestion? "What the heck?" doesn't give me any insight.

- And there are perfectly good arguments for using wildcard imports.

2020-07-03 10:13:47 by Gary Woodfine:

In my experience Clean Code is a lot like teenage sex. Everybody brags about it, but very few are actually doing it and those that do invariably lead to accidents.

I still think, Clean Code has relevance, and is still a good starter book for many, especially for those who want to take their coding abilities to the next level.

Is it a perfect book, probably not, there are a number of subjective opinions, but that is to be expected on a mostly subjective topic.

As with most Software Development books, it provides a foundation to work from. The rest is down to your continual learning, adaptation and implementation.

2020-07-03 12:15:45 by ingvar:

You write:
  Outside of a book, do we still read code from top to bottom? Well, maybe some of us do.

I certainly do, every time I am doing a code review. I'd even argue that you should start with the lowest level, then go to more and more abstract functions as you go, because that sets the reviewer's mind up to have a chance of going "um, why?" in a way that doesn't require scrolling all over the place (yes, in theory names should be descriptive but in theory, theory and practice are the same).

2020-07-04 03:27:15 by Dave (yet another Software Crafter):

Another campaign to literally take someone down. Love this profession don't you all? I wonder what it'd be like working with you blog poster. I can't imagine it's a very positive ride.

Clean Architecture is great, what are you talking about? I've even applied the advice of Clean Architecture before his book came out (which is basically hexagonal & other solid fundamentals tied together) through the years many big applications I've worked on.

Written in Java? so what! A ton of other books were written in Java from Martin Fowler, Kent Beck, Micheal Feathers and so many that a ton of us in our profession still reference and probably always will. The ideas don't die in those books, and it's up to you to practice and try to apply them elsewhere. Software doesn't write itself.

This post was a waste of my time, and I hope others see it for what it really is...chow.

2020-07-04 17:38:23 by mwchase:

To people not sure about the data structure definition, my take on it is that anything beyond what can be expressed entirely as sum and product types is going to have some kind of invariant that needs to be maintained. For example, if you have a heap, you need to maintain the heap invariant, otherwise you don't get correct results.

So, if "something that requires invariants be maintained" is an inhabited kind of data structure, then it cannot be the case that all data structures expose their implementation details publicly.

2020-07-06 22:16:02 by wwise:

Martin is mixing up data types and data structures. The main difference between a stack and a queue is the operation for removing items.

The operations are part of the definition of a queue or a stack. What he's calling an object is actually a data structure. It's just organized differently in an OO language than in a procedural (or functional, I suppose) language.

2020-07-07 10:52:51 by ReadMe:

U dont recommend the book because of author's random political point of view on a social network. We dont recommend ur website and org due to ur mindset mixing our innocent programming industry with political world.

2020-07-07 16:45:50 by Mariusz Cyranowski:

This book is quite hard to comprehend because of many ambiguous topics already mentioned in the article as well as in comments. The book is a good read though, but I think there are better ways to fight the complexity of software solutions then practicing patterns and TDD. This book is not a bible of programming but just another piece from well known agile evangelist.

2020-07-07 23:12:15 by Mario:

You do good points!

I must say that I use his book for teaching, but I let my students clear that it is a fully opinionated material. There are good advices? yes, but the rules he proposes are far to follow and may deviate the programmer from the goal to solve just because will be worried on the shape and not the content.
I compare the content with my personal rules like “review any code that is 2 times pressed the page down key” or “if you use a boolean in the parameters check if you are mixing logic”. I think his book is valid as reference and disussion. Then let every organization and programmer build their best practices.

2020-07-08 03:59:51 by bdan:

Not sure what is worse, the book or some of the comments here.

Thanks for taking the time to write this! I really think developers need to unite and start a GoFundMe for the author and get him to retire, his heart is in the right place but no one has done more harm to our profession than him...

2020-07-08 11:03:15 by John Laptop:

While your review does make some fair points, I can't help but getting the feeling that you rather have a beef with Martin.

2020-07-08 12:28:10 by Tom:

Odd that some people are reading politics into this review. It seems wholly fair to me, as a clear opinion piece that objectively analyses the subject at hand. Is it so hard to believe that someone could have a different opinion on something as divisive as code style without having a hidden agenda?

Just to try to understand the situation, I read Martin's blog post "Thought Police" mentioned above and found the idea he expressed there to be entirely reasonable and respectable and most importantly unrelated to his book and this review of it.

2020-07-08 14:02:33 by Carlos Saltos:

Please find a modern approach here -> https://www.deconstructconf.com/2019/dan-abramov-the-wet-codebase very valuable

2020-07-08 19:38:40 by Arthur:

1. Feathers: Working Effectively with Legacy Code<br/>
2. Khorikov: Unit Testing Principles, Practices, and Patterns

2020-07-09 00:34:03 by qntm:

A few people have recommended Code Complete as a replacement for Clean Code. I think it's a better book, but I'd still be hesitant to recommend it.

I don't have anywhere near as much to say about Code Complete because I haven't read it cover to cover, because it's a monster. I have the first edition, which is 850 pages long. That in itself is well worth considering. But the main thing about Code Complete is that even if all the information in it was complete and correct at the time of going to press, it's really quite dated right now. Even the second edition is from 2004, *predating* Clean Code. As I say, this is just a standing problem which almost all programming books have. Fifteen years is a *long* time in software development.

There are some other strong recommendations in this thread, but these are books which sadly I have not read (I spend far too much time writing and not enough time reading). That means I can't recommend them myself. For what it's worth, on the basis of what people have said here and elsewhere, these are the programming books which I'm most likely to read next:

* John Ousterhout, A Philosophy Of Software Design (2018)
* Michael Feathers, Working Effectively With Legacy Code (2004, oof)

2020-07-09 15:36:07 by Arthur:

And don't get me started about the Clean Code video series. At the suggestion of a consultant our development team watched the series. The presentation styke was extremely grating to me.   To summarize: that was an expensive waste of team time.

2020-07-12 02:37:55 by Staged:

Much of this evangelizing really demonstrates a foundational flaw I have seen: developers are really bad at reading code. Every time I have to press f12 to know what is going on my code, is the mental equivalent of somebody taking my headphones off and starting a conversation with me. That's not to say that functions or splitting out work into other packages is bad : but it is to say that we should have good reason to make us turn the page of this choose-your-own-adventure code. If a method is called once and is one line long, it should be inline code until there is a reason to break it out. Additionally, we should try to avoid putting verbs into nouns when we can - it creates another layer of separation before we need it.

2020-07-15 09:04:17 by gagga:

I think what would really improve this book is a Marxist version of it.

2020-07-20 17:06:35 by qwerty:

Structure and Interpretation of Computer Programs

2020-07-24 09:16:14 by FeepingCreature:

FP and OOP are totally compatible. This is approximately how we do it where I work:

FP for everything small. Pure data is pure, and manipulated by pure methods. Immutable data structures, return instead of ref, recreation rather than modification. No OOP here.

Then, the application code has separated areas of responsibility. These are classes that may hold state, and mutate it by business-level methods. But each set of data is only stored and mutated in one class. Anything that can be mutated is private, and no mutable references to it are handed out.

Then, OOP is used lightly, mostly just to separate interfaces and implementations between different areas of the codebase, and for bits of dependency injection to avoid dependency cycles. So code reuse is mostly achieved by composition rather than inheritance. The FP routines manipulating data basically never see any OOP themselves.

Also, the example unittest would compare a state struct with cleanly labeled flags. Thanks to a diff routine, the mismatching flag would be highlighted in red/green. :)

That is to say: none of the examples in this post would pass code review.

2020-08-25 08:31:23 by Virgo47:

I don't regret reading Clean Code and it helped me to progress at the time. I don't remember exactly whether I agreed with everything, it wasn't by far the first book I read. I'm not sure I'd recommend it today for its size and old-style Java (both version-wise and paradigm-wise). John Ousterhout's book A Philosophy of Software Design was mentioned couple of times - I have it, read it and it's great counterbalance to Clean Code + it's great book on its own. I have no idea how it is to read it in vacuum. I'd not rely on a single book. Is Clean Code harmful nowadays? I don't know. If it was the only book around, I'd still recommend it to a junior and as a mentor took over from there.

2020-08-31 13:49:36 by Frank:

You do not recommend CC but give no alternatives? You criticize some snippets but again give no better solutions. Come on man!

2020-09-04 19:10:17 by Ed:

Thank you for this blog post.

I have to agree on recommending:
A philosophy of software design by John Ousterhaut.

Short and to the point, filled with practicable usable advise that can be applied even if one has little experience.

2020-09-05 13:49:22 by Stefan Houtzager:

John Ousterhout's "A Philosophy of Software Design" looks interesting : https://lethain.com//notes-philosophy-software-design/ Take into account some sidenotes by readers: https://www.amazon.com/Philosophy-Software-Design-John-Ousterhout/dp/1732102201

2020-09-14 16:03:17 by Rob Lang:

Extremely useful book is Martin Fowler's Refactoring. I dip into it on a regular basis.

2020-09-15 02:01:55 by Ray:

Frank, I would argue that if the examples are bad enough, you're better off with no examples at all, and that the prime number generator, in particular, falls into that category.

But since you want an alternative, I've thrown together an implementation of the same function. I've replaced each tab with ". . " in hopes of preserving the formatting. We'll see how well that works. As a side note, the below code uses the simplest prime number sieve, and takes about 45 seconds to find the first 20 million primes. The Clean Code version takes *22 minutes*.

Based on the prime number generator alone, it is *entirely* reasonable to conclude that the author knows almost nothing about writing good code.

```
========
import std.stdio, std.algorithm, std.range;

// generates all primes in [start,end) and returns all primes in
[0,end).
// primes array must contain all primes < start
uint[] grow_sieve(uint[] primes, uint start, uint end) {
. . assert(end >= start && start > 1);
. . bool[] composite;
. . composite.length = end - start;

. . // mask multiples of existing primes over the expanded range
. . foreach (p; primes) {
. . . . // start at first multiple of p in [start,end)
. . . . for (uint j = p * (((start - 1) / p) + 1); j < end; j += p) {
. . . . . . composite[j - start] = 1;
. . . . }
. . }

. . // locate primes inside the expanded range
. . foreach (i; start..end) {
. . . . if (composite[i - start]) continue;
. . . . for (uint j = i * 2; j < end; j += i) {
. . . . . . composite[j - start] = 1;
. . . . }
. . }

. . // concatenate all numbers in [start, end) that aren't marked
composite
. . return primes ~ iota(start,end).filter!(a => !composite[a -
start]).array;
}
```