Jakob Hain

Christen Enos

ENGW3302 Advanced Writing

January 17, 2019

"Clean" Code – How to write it?

How does a software developer write "clean" code – code which they, and other developers, can easily understand? This is almost self-contradictory – "code" was historically defined as something intrinsically *hard* to understand. Clean code "explains itself", so an experienced developer can read it, and quickly figure out how the program works, understand what it actually does, what it's purpose is, and why it's written as it is. On the other hand, "messy" code is obscure and confusing, so it's hard for a developer to work with. Unfortunately, it takes much more time and effort to write clean code than messy code.

Why is it so hard to write clean code? In a general sense, it's hard to explain what an "integrated development environment" is to someone in English. It's even harder to "explain" it in code, but that's exactly what clean code must do. At least in the short term, it often takes a lot longer for a developer to write a clean program than it does to write a messy one. Developers sometimes must go back to rewrite their code just to make it cleaner. A developer only needs to read a few online tutorials to learn how to code, but they must learn extra to write clean code.

Confusing, "messy" code exists everywhere, from apps we use every day (like Microsoft Word) to critical systems (like hospital and bank management software). Developers are still creating messy programs, even when they try not to. It takes research and effort to follow the coding guidelines and design patterns mentioned above, so few codebases actually do so. Even

the ones that do are still sometimes messy, because the guidelines don't always work. When a new developer reads an existing codebase, it's usually hard for them to understand and contribute to the code, because it's so messy.

This messy code is a problem, because it causes other problems. Once a computer program is released to the public, it's rarely ever finished – developers keep working on it to fix bugs and add features. "Clean" code often has fewer bugs in the first place, because when developers understand how a program works, they can better compare it to how the program *should* work and fix the differences. Furthermore, when code is easier to understand, it's easier to change – it takes less time to fix the bugs and add new features. And new features are surprisingly important – people often take them for granted and forget that they were once "new" at all, but a lot of programs aren't that useful when they're first released, but only later once they get updated. For example, the original version of Google Slides for mobile devices didn't even support adding images to a presentation, it could only create presentations with just text (and there wasn't much ability to customize the text, either).

Professionals care about clean code a lot. Developers have struggled to write clean code for a long time – ever since computer code existed (and code existed before computers) – but they still struggle today. Companies like Apple and Google, and colleges like MIT and Northeastern (in particular, Northeastern's Programming Research Laboratory), have spent money and research trying to figure out how to write cleaner code. They've come up with "design patterns" – techniques and rules for writing clean code – and "linters" – programs which analyze code and suggest ways to make it cleaner. Fundamentals I and II at Northeastern practically exist just to teach students how to write clean code, and get them to write clean code

in later classes and jobs. Furthermore, people write about clean code – hobbyists make a living wage by creating books and online tutorials on how to write clean code, and academic scholars write articles discussing clean code.

One such article is "Coding Guidelines: Finding the Art in the Science", by Robert Green and Henry Ledgard, published in the December 2011 edition of *Communications of the ACM*. This article tries to describe what makes code "clean", provides some guidelines and examples of clean code (along with notes on what to avoid, and examples of messy code), and briefly mentions how those guidelines could help in a practical setting.

The article is written by academics, and academics are definitely part of its audience. Its magazine, *Communications of the ACM*, is very popular with researchers. It's precise and formal – the authors use long words like "functionality" and "implementation", and they go into detail when describing their idea of "clean" code, their guidelines, and their goals. This is reflected when they specifically mention what their article is *not* about: "one area not considered here is the use of syntax highlighting or IDEs ... our guidelines have been developed to be IDE and color neutral." (Green, Ledgard). By stating that they *ignore* syntax highlighting and IDEs, the authors narrow the focus of their guidelines, and academics really like articles whose goal is narrow and precise.

The article also evokes an academic persona, and reads like many academic articles. The authors reference another *ACM* article, and they repeatedly tie their guidelines to academia – many other academic articles also contain a lot of references, because academics make progress by building on each others' work. Similarly, the authors use bold, assertive language and words – "it is the nuances of a programmer's technique and tools...that truly make code clear,

maintainable, and understandable," (Green, Ledgard) – and many academics are very assertive, so this language makes the authors sound like other academics. Furthermore, the authors show that they think logically and algorithmically – like other academics – by sounding more than just formal and precise. For example, they compare writing code to "encryption": "From another angle, programming may also be seen as a form of 'encryption.' In various ways the programmer devises a solution to a problem and then encrypts the solution in terms of a program and its support files. Months or years later, when a change is called for, a new programmer must decrypt the solution." (Green, Ledgard). This is figurative language, so it's *not* precise. But it still sounds like something an algorithmic thinker, who would be constantly thinking about ideas like "encryption", would say.

However, the article targets more than researchers, it's also written for managers and developers. The article summarizes each of its guidelines in sections, complete with examples, so a developer or manager reading the article can actually apply them. The precise language makes the paper more academic, but also more practical, because it makes the guidelines easier to apply. And the authors state *multiple times* that they want their guidelines to be applied – "While these guidelines come from an educational environment, they are designed to be useful to practitioners as well ... Although the guidelines presented here are used in an educational setting, they also have merit in industrial environments." (Green, Ledgard). All of this implies that the "practitioners" in "industrial environments" – the managers and developers – should read the guidelines themselves. Furthermore, while the article is written in an academic style, it's not too difficult for a non-academic to understand. It's formal, but not to the extent where it's dense and obscure. The article doesn't use many academic references, which someone not in research

wouldn't know about, and when it does use references (such as the other ACM article), they're not necessary to understand what it tries to convey. And the authors include some informal statements: "proportional (variable-width) fonts ... do not "look like" code." (Green, Ledgard). These informalities would make the article clearer to "laymans", although many academics would argue that they make it *less* clear (because they're less precise than formal statements).

Similarly, the article's persona is more than academic, it evokes a personality which other academic articles don't. The informal statements serve another purpose – while the authors tend to write formally and "logically", their language still conveys that they have an emotional passion towards clean code. One of their key ideas is that computer science is an "art" – the first sentence is, as quoted: "computer science is both a science and an art." (Green, Ledgard). This metaphor serves two purposes. The authors use it to argue that a developer must use inference and even creativity, not just a set of fixed rules, in order to write "clean" code – the "science" part is the fixed set of rules, while the "art" part is the inference and creativity. But at the same time, the metaphor associates computer science with the beauty and expressivity of art, and suggests that, at least to the authors, reading clean code evokes emotion just like looking at a good painting. The authors also express their passion towards clean code by expressing their opinions. Like most academics, the authors are assertive. But they go as far as stating their own opinions about clean code, with subjective words – "there is a combination of artistic flare, nuanced style, and technical prowess that separates good code from great code" (Green, Ledgard). Academics like facts, not opinions, so to them, these opinions might make the article worse. Therefore, because the authors included opinions, it shows they really care about clean code. As explained above, the purpose of the article is to persuade others to accept the authors' ideas and guidelines

about clean code. The authors wouldn't have much reason to write an article with such a purpose, if they didn't want others to accept the ideas themselves.

In conclusion, I'm focusing on this problem: how does a software developer write "clean" code, code which is easy for other developers to understand? It's hard to write clean code – many companies and researchers try, but still, most of the world's code is arguably "messy" and hard to understand. This is bad, because programs with messy code also tend to have more bugs, and it's harder to extend them with new features. The difficulty and importance of this problem – how to write "clean" code – makes it very interesting. It interests professional developers and researchers, so they write about it – one example is "Coding Guidelines: Finding the Art in the Science", an article which tries to pinpoint what makes code "clean". It interests me, so I've chosen to research and discuss it myself throughout the semester. And, after you've read this essay, perhaps the problem of writing clean code also interests you.

Works Cited

R. Green and H. Ledgard, “Coding Guidelines,” *Commun. ACM*, vol. 54, no. 12, pp. 57–63, Dec. 2011.

Reflective Statement

I decided to write this essay about "clean" code very quickly. I've been interested in "clean" code since I got to Northeastern. I became interested because I've written a lot of my own programs, but struggled with messy code, so I appreciate "clean" code and strive to make my own code clean.

I actually started with an introduction, which would've been an overview of the entire essay. But it was very long and detailed, and I realized that it already contained most of the content I would've put in the essay. So I broke it up into many paragraphs – the first sentences became the introduction, the next sentences became body paragraphs, and the last sentence even became the base for the conclusion's first sentence.

But besides that, I wrote my first draft incrementally. Whenever I wrote a new idea in a few sentences, I often revised those sentences, and I sometimes needed to completely re-write them, before I moved on to the next idea. But once I moved on, at least if I kept the idea the same, I typically only revised those sentences a little more.

I don't think I revised my essay much after my first draft (although I hope I revised it enough). I divided and rearranged a few paragraphs, and rewrote most of the conclusion. But other than that, I mostly made minor, local edits – I applied my peer and instructor reviews' annotations, revised the grammar, and added/removed a few supporting details for my body paragraphs.

That's how I write most essays, because that's how I work in general. I like to get tasks done early and all at once. I think it's much easier to try and make something well the first time, then to make it once and re-make it again. Moreover, I persevere a lot, to the point where I'm

stubborn – once I've focused on one task, I can put effort into that task and I can stay focused, but it's hard for me to move to other tasks. But then if I focus on another task, it's hard to get back.

To me, writing is actually similar to coding, and interestingly, I write similarly to how I code. When I write a computer program, I usually write it incrementally – although I have an idea of what the overall program will look like, I like to code one feature, then move onto the next, and so on. I try to write functional, clean code the first time, instead of writing messy code and then fixing it – although this is hard, and I often end up "refactoring"code much more often than I revise text. And I often write out my programs completely before I test and fix them, like how I wrote a complete essay as my draft for this paper, then applied my peers' and instructors' suggestions.

I don't think I got many suggestions, especially from peer feedback. I applied some of my peer's suggestions, but skipped others. For example, I mentioned in the introduction that it's harder to write clean code than messy code – I agree with the reviewer, that's a major part of my essay – but I didn't add examples of where "messy" code is in programs – that information wouldn't be meaningful, but it would be hard to for me to find and readers to understand. I applied practically all of the grammar and consistency suggestions, because those were obvious if not objectively right. And I applied almost all of the teacher's revisions directed towards my essay.

I looked over my essay and tried to apply the strategies and suggestions which I mentioned in discussion post 3A. For the first two (1a and 2b), I didn't really change anything – I made sure my essay described "how to write clean code", not just "clean code" in general, and

that I didn't over-explain anything, but I couldn't find anywhere to improve. I did apply the third suggestion (5e) – I rewrote my essay's conclusion as it suggested, quickly summarizing my topic and article, but then connecting the two and making a broader point. Ultimately, I think I'll use these suggestions more for future assignments.