ADVICE FOR PROGRAMMERS

Quintessential Bad Programming Habits

Love for over-engineering, global data, and more

Anupam Chugh Follow
Mar 30 · 6 min read ★

Photo by Arian Darvishi on Unsplash

Programming is an art. There’s no single way of doing things.

Neither is there a predefined set of patterns that makes you a good programmer. Everyone figures their own unique style.

But there are certain traits, or rather habits, which would only pull you further from being a great programmer.

Often unintentionally, developers adopt these programming habits. Habits, which would be detrimental in the long run. Not only would it hamper your growth but also give headaches to other programmers around you.

Developers who are starting out are more prone to fall into these bad habits, though a lot of senior developers aren't devoid of them completely. It's important to know these bad habits in order to not fall into the rabbit hole and to get out quickly if need be.

While there are a lot of traits, let's go through a few typical ones only that largely fall under bad programming practices.

. . .

Top highlig

Over Engineering

To put it straight, a code that solves problems that don't exist is called over-engineering.

Over-engineering generally stems from a lack of concrete requirements, timelines, and a desire to have a codebase that's futuristic. Often a developer, out of sheer curiosity to try new technologies or boredom builds things that weren't necessary. What actually ends up is premature optimization, incorrect abstractions, and a codebase that is way smarter than it needs to be.

This not only makes a rather simple design complex, but also you lose sight of the two golden rules of programming:

- Easy to understand.
- Easy to refactor.

It's important to *Keep Things Simple And Stupid*, not overuse a design pattern that you've just got your head around, and also not get swayed by the latest tech stack that you see. One way to avoid over-engineering is to do the design work precisely before writing a single line of code. For example, list down the abstractions that are required.

Too many abstractions and unnecessary layers are good for nothing if the code cannot be refactored easily.

> A good codebase design is one that can be easily undesigned.

. . .

Erratic Comments

A lot of developers often tend to neglect comments by arguing that the code speaks for itself. On the other hand, there's a set of developers who overdo it and that doesn't help either. Instead, it makes it less code and more documentation.

In order to show the importance of writing precise comments in your code I'd put the following argument:

> *Writing a comment is like putting a title on your story. It should summarise the story and lets the reader decide whether to dig deep into this or just skim through it. Imagine this piece without a title or a subtitle. Or without subheadings. It'll definitely not be much of a trouble when reading a single piece. But when they're hundreds of pieces devoid of a title, the reader doesn't know where to go! The same holds for your fellow programmers. Step into their shoes while writing down a comment.*

It's pretty common for a developer who is just starting out to over-comment. A lot of them would write a comment on every single line of code that eventually clutters the codebase.

For example, the following comment doesn't do any good as the code is self-explanatory:

```
//Initializing x (Was this needed?)
int x = 9;
```

A good rule of thumb when writing down comments is to address the "Why" aspect of code rather than telling "What" the code is doing. Also, it's better to stay away from writing snarky comments and ensure that it's always in sync with code.

Writing consistent code is another "Achilles Heel" of beginners, and it's understandable. Beginners are prone to mixing naming styles, writing random variable names, and using improper indentations. It's crucial to have a notion of the naming conventions and stick to one. Sit down with your team if need be and lockdown on a standard convention if there are no precedents set already.

Having a consistent styling across your codebase helps a lot in code management across teams and improves productivity by leaps and bounds. Most importantly, it reduces the need to over-comment as it improves the readability of the code tremendously.

. . .

Ignoring Idiomatic Programming

Idiomatic programming indicates expressions and statements that are unique to the language. Developers who're finding their feet in a new language often tend to ignore these mystical parts.

Instead of investing some time in learning the intricacies of the language, they'll try to just get accustomed to the syntax in order to deep dive into the project.

Doing so, they start leveraging the core constructs that are fundamental across all languages instead of harnessing the expressive power of the language at hand.

Non-idiomatic programming makes the code lesser clean.

Using a C or Java-style for loop in a modern language like Python or Swift would only cause readability and consistency issues across teams. For example, using high order functions `.map` or `.filter` reduces a lot of boilerplate code that comes with imperative loops.

The good thing about some of these new languages is that the compiler forces you to adopt their convention. For example, starting Swift 3, you are not allowed to use C-style for loops anymore.

. . .

Over Indulgence With Global Data

Having worked with developers of varying experiences, I’ve always observed there’s a love-hate relationship for global data and more specifically, global variables.

Global variables are declared outside all scopes to be accessible by everyone in the program. A developer unaware of the cons of global variables would typically defend global data with the following statements:

- It’s the easiest way to pass, access and modify data from classes and functions.
- It helps avoid the long list of parameters that you’d otherwise need to pass.

What looks like a good one-time proposition, especially in simpler programs, can easily lead to complexities in state management in a large codebase due to it’s shared state nature.

Since the variable is global, it can easily be changed from anywhere and which is a potential source of bugs and security issues.

It requires you to keep a track of all scopes where the global state would affect.

Global variables aren’t inherently bad. However, there’s a high chance of misusing them due to their larger scope thereby making them dangerous and unreliable. It can easily lead to concurrency issues, difficulties in setting up unit testing, and name collisions.

While Global variables are discouraged, Global constants aren’t as bad (overdoing though, isn’t recommended) due to its immutable state. One way of getting rid of Global variables is Encapsulation. Create a class, constructor, and initialize your variables there.

. . .

Closing Thoughts

Planning is a crucial part of software engineering. Often developers jump the gun quickly in order to get things done without estimating the trade-offs.

Another peculiar habit that makes a developer ineffective is *yo-yo coding*. It only increases the number of jumps a programmer has to make in the codebase — like a yo-yo ball. Having shallow inheritance graphs helps avoid this problem.

Dismissing constructive criticism, relying on god objects, lack of unit tests, being too rigid and not having the hunger to improve — all slows down a programmer's growth in the long run.

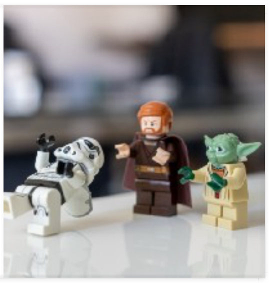
I hope the above points illustrated what not to do, in order to be an effective programmer.

That's it for this one — thanks for reading. *If you liked what you read, check out this story below.*

What Programmers Say vs. What They Actually Mean

Because working in a stressful environment surely teaches us how to make good excuses

levelup.gitconnected.com



Thanks to Zack Shapiro.

Programming Software Development Software Engineering Productivity Self Improvement