Nicole P. Satiembre   BS Computer Engineering-Intern   February 19, 2024

Cebu Technological University-Main Campus

97 Things Every Programmer Should Know
Chapters 5-10

Beauty is in Simplicity - Jorn Olmheim

A short impactful point that I learned in this chapter is to keep our codes simple. This chapter sets the tone with one beautiful quote by Plato that we developers should aspire to: “Beauty of style and harmony and grace and good rhythm depends on simplicity.” There are several things we strive for in our code:

- Readability
- Maintainability
- Speed of development
- The elusive quality of beauty

Beauty is a subjective perception that may be different from people of different backgrounds. However, one common thing that grounds them is simplicity.

I have learned in this chapter the importance of studying the code of other people, especially experts. This way, we will learn the best practices when writing code and find out better ways and approaches to solving a problem. No matter how complex the project may be, the individual parts have to be kept simple. We want to achieve brevity as much as possible.

The final thought is that beautiful code is simple. This is the way we can keep our systems maintainable over time, with clean, simple, testable code, ensuring a high speed of development throughout the lifetime of the system.

Before You Refactor - Rajith Attapattu

These are the main points I have learned before refactoring a code:

- *The best approach for restructuring starts by taking stock of the existing codebase and the tests are written against that code.* This helps us understand the strengths and weaknesses of the code as it stands. Learn the existing system's mistakes.

- *Avoid the temptation to rewrite everything.* Throwing away the old code—especially if it was in production—means that we are throwing away months (or years) of tested, battle-hardened code that may have had certain workarounds and bug fixes we aren't aware of.

- *Many incremental changes are better than one massive change.* A couple of test failures at a time are easier to deal with than a hundred lines of failures once you make one massive change.

- *After each development iteration, it is important to ensure that the existing tests pass.* Add new tests if the existing tests are not sufficient to cover the changes you made. Do not throw away the tests from the old code without due consideration.

- *Personal preferences and ego shouldn't get in the way.* Do not fix what is not broken. Just because you didn't like the code of the other programmer is not a good reason to refactor things.

- *New technology is an insufficient reason to refactor.* Jumping into the bandwagon of the latest technology is not enough reason to refactor. Unless a cost-benefit analysis shows that a new language or framework will result in significant improvements in functionality, maintainability, or productivity, it is best to leave it as it is.

- *Remember that humans make mistakes.* Restructuring will not always guarantee that the new code will be better—or even as good as—the previous attempt. It could fail because, after all, we are humans.

Beware the Share - Udi Dahan

I have learned in this chapter that although reuse was held up as the epitome of quality software engineering, it causes more trouble than help when you don't take note of the context it is being used.

According to the writer's experience, two wildly different parts of the system performed some logic in the same way until he had pulled out those libraries of shared code, these parts were not dependent on each other. Each could evolve independently. Each could change its logic to suit the needs of the system's changing business environment. While he had decreased the absolute number of lines of code in the system, he had increased the number of dependencies. The context of these dependencies is critical—had they been localized.

Reusing dependencies, modules, or libraries, when applied in the right context, these techniques are valuable. In the wrong context, they increase cost rather than value. That is why it is crucial to be careful and check the context of dependencies before we reuse them.

The Boy Scout Rule - Robert C. Martin (Uncle Bob)

This chapter is an eye-opener for me. Most often we take it for granted to clean the code of the team and just focus on our codes. The Boy Scout Rule is such a good rule to live by: “Always leave the campground cleaner than you found it.” If you find a mess on the ground, you clean it up regardless of who might have made it. Keeping our codes clean should be a form of common decency like throwing our garbage properly.

We don’t have to make every module perfect before we check it in. We simply have to make it a little bit better than when we checked it out. Of course, this means that any code we add to a module must be clean. It also means that we clean up at least one other thing before we check the module back in. Even just correcting a simple variable already means a lot.

Taking care of our code is one thing but we must also take care of the team’s code as a whole. It is not only for our good but for the betterment of the project and the team as a whole. Indeed, we can learn a remarkable character from the Boy Scout’s rule.

Check Your Code First Before Looking to Blame Others - Allan Kelly

I can highly relate to this chapter as I often get frustrated when I don't get the expected output or when I encounter bugs that are hard to fix and often blame it on the compiler. But that was me before, I must say I have grown now and I check my code first before blaming it on the compiler!

I have learned that we are much more productive putting our time and energy into finding the error in our code than proving the compiler is wrong – because it can rarely be wrong, it's a machine. All the usual debugging advice applies, so isolate the problem, stub out calls, and surround it with tests; check calling conventions, shared libraries, and version numbers; explain it to someone else; look out for stack corruption and variable type mismatches; and try the code on different machines and different build configurations, such as debug and release.

It is also helpful to question your assumptions and the assumptions of others. Tools from different vendors might have different assumptions built into them—so too might different tools from the same vendor. It is also a good idea to see how other people are solving the same issues, we might gain insights about approaches to a problem that we have never thought of before. Multithreaded problems are also a big pain so it is always wise to make our code as simple as possible. To sum it up, check our code first before blaming others.

Choose Your Tools with Care - Giovanni Asproni

It is common to use premade tools when building an application. Modern applications are very rarely built from scratch. They are assembled using existing tools—components, libraries, and frameworks—for several good reasons. Applications grow in size, complexity, and sophistication, while the time available to develop them grows shorter. Widely used components and frameworks are likely to have fewer bugs than the ones developed in-house.

I learned to consider these things when choosing the right mix of tools for our application:

- Different tools may rely on different assumptions about their context.
- Different tools have different lifecycles.
- Some tools require quite a bit of configuration, often using one or more XML files, which can grow out of control very quickly.
- Vendor lock-in occurs when code that depends heavily on specific vendor products ends up being constrained by them on several counts: maintainability, performance, ability to evolve, price, etc.
- If you plan to use free software, you may discover that it’s not so free after all.
- Licensing terms matter, even for free software.

I also learned a good strategy to mitigate these problems by starting small and using only the necessary tools. And then add more if needed. It is also best to isolate the external tools from the business domain objects using interfaces and layering, so that I can change the tool if I have to with a minimal amount of pain. A great advantage of this approach is that we can generally end up with a smaller application that uses fewer external tools than originally forecast. This is the optimal condition that we want to achieve.