Nicole P. Satiembre **March 1, 2024**

Cebu Technological University-Main Campus- Intern

97 Things Every Programmer Should Know

Chapters 76-80

76. The Single Responsibility Principle - Robert C. Martin (Uncle Bob)

One of the most foundational principles of good design is to gather together those things that change for the same reason and separate those things that change for different reasons. This concept is commonly referred to as the single responsibility principle, abbreviated as SRP. In essence, SRP dictates that a subsystem, module, class, or even a function should only be subject to change for a single reason. Effective system design involves dividing the system into components that can be deployed independently. This independence in deployment implies that altering one component doesn't necessitate redeploying any others. For instance, if numerous classes across different components extensively utilize the "Employee" entity, any modification to the "Employee" entity is likely to necessitate the redeployment of other components. This action undermines a significant advantage of component design. Careful application of the SRP, separating things that change for different reasons, is one of the keys to creating designs with an independently deployable component structure.

77. Start from Yes - Alex Miller

I have learned in this chapter to always welcome requests and opportunities. No matter how elusive, unrealistic, or unfeasible the idea may be, we must still put it into consideration and ask the person why or what motivated them to have this idea. I have the same experience with a classmate of mine who had a very wild imagination and always thought outside of the box. Whenever we have a group project for a product to innovate, her ideas seem really unearthly and our group members would doubt. Regardless, we still welcomed her idea. Aside from her idea, we also wanted to make her feel belong to the group and that she is a highly appreciated contributor. In the real world, people also have feelings so we want to win these people instead of losing them by saying no to their appeals. Some of her ideas turned out to be actually great, and although it didn't become the final product that we had agreed on, it was satisfying to explore new ideas plus that person also felt great for giving her ideas to the group. So, it's a win-win for all of us.

78. Step Back and Automate, Automate, Automate - Cay Horstmann

I've learned that some programmers may resort to manual and repetitive methods, such as pasting code into a word processor for line count, even when more efficient alternatives exist. This tendency might persist despite the inefficiency of the approach. Automation has a positive impact on streamlining complex processes. In one instance, a burdensome deployment process was significantly improved through automation, saving time and effort during the final testing phase.

The question of why individuals often choose to repeat tasks rather than invest time in automation remains. This could be attributed to various factors such as a lack of awareness about automation possibilities, a perception that the task is too trivial to automate, or simply a reluctance to change established routines. Common misconceptions are that Automation is only for testing; No need to automate since you have an IDE; You need to learn exotic tools to automate; Cannot deal with file formats; and Don't have time to figure it out. Understanding these dynamics is crucial in promoting a culture of efficiency and continuous improvement in software development.

79. Take Advantage of Code Analysis Tools - Sarah Mount

The importance of testing is emphasized in software development, with practices like unit testing, test-driven development, and agile methods gaining popularity. However, testing is just one tool among many for enhancing code quality. In the early days of programming, when resources were limited, C compilers sacrificed some error-checking capabilities for efficiency. To address this, tools like lint were created to perform static analyses and improve code quality. Today, with abundant memory and CPU resources, compilers can check for more errors. Various languages offer tools like Splint for C or Pylint for Python, configurable to check for style violations and common errors. If standard tools fall short, developers can create their own static checkers using language-specific libraries, uncovering hidden features like Python's disassembler for dynamic testing and analysis. The message is clear: testing is crucial, but supplement it with analysis tools and consider building your own if needed.

80. Test for Required Behavior, Not Incidental Behavior - Kevlin Henney

A common mistake in testing entails assuming that the exact workings of an implementation are a perfect match for what is to be tested. While it may seem beneficial at first, this becomes a trap when tests are too closely tied to implementation specifics that are incidental and unrelated to the desired outcome. If tests are so tightly bound by such incidental things, even minor changes in the implementation behavior required may cause test failures hence indicating false positives. They often either rewrite the test or change the code completely. Such assumption of false positives as true positively can arise from fear, uncertainty, or doubt whereby incidental behavior is elevated to mandatory behavior. Although sometimes programmers may modify a test to fit new implementations we would ideally prefer to see its rewrite focusing on necessary behavior instead of simply adjusting it towards matching new implementations, which is not optimal. Tests must be precise but also accurate, avoiding over-specification, particularly in white box testing where tests might redundantly assert the obvious. An effective set of tests will articulate contractual obligations by assuming a black box view of the units under test while aligning tested behavior with required behavior.