Nicole P. Satiembre **March 3, 2024**

Cebu Technological University-Main Campus- Intern

97 Things Every Programmer Should Know

Chapters 91-97

91. WET Dilutes Performance Bottlenecks - Kirk Pepperdine

The opposite of the DRY(Don't Repeat Yourself) principle is WET (Writing Every Time). DRY holds high importance in software development as it codifies the idea that every piece of knowledge in a system should be contained in a single implementation. We can use the analogy of a system's feature that utilizes 30% of the CPU. This means that on average each implementation will consume 3% of the CPU. Though it is too small to worry about but what if we found out that this feature is causing a bottleneck in the memory? We are now left with the problem of finding and fixing every single implementation. With WET, we have 10 different implementations that we need to find and fix. With DRY, we would clearly see the 30% CPU utilization and would have a tenth of the code to fix. And we are spared the time hunting down each implementation.

92. When Programmers and Testers Collaborate - Janet Gregory

This chapter promotes the collaboration between the testers and the programmers. Often, these two job roles are indifferent towards each other. There is this stereotype that the programmers and testers don't blend well because programmers hate the testers and testers make the programmer's life difficult. However, the opposite might be true. Programmers and Testers build efficient programs when they work together. The testers help catch errors; meanwhile, the programmers can help the testers because they have a good grasp of good coding practices and can help testers set up a robust test automation suite that works for the whole team. When testers stop thinking that their only job is to break the software and find bugs in the programmers' code, programmers stop thinking that testers are "out to get them," and are more open to collaboration.

93. Write Code As If You Had to Support It for the Rest of Your Life - Yuriy Zubarev

When we write code we must value it as if we had to support it for the rest of our lives. If we don't care about our fellow developers, testers, managers, sales and marketing people, and end users, then you will not be driven to employ test-driven development or write clear comments in your code. With the mindset of writing code as if we had to support it for the rest of our lives, we will be geared more to improve and become an expert. We would want to improve our naming conventions, stay away from verbose codes, write relevant comments, test our code, and refactor continuously. Being able to support our code also makes it more scalable. People will form opinions about you based on the code that they see. If those opinions are constantly negative, you will get less from your career than you hoped. Take care of your career, your clients, and your users with every line of code—write code as if you had to support it for the rest of your life.

94. Write Small Functions Using Examples - Keith Braithwaite

I have learned in this chapter the importance of writing correct code and providing evidence for its correctness by considering the "size" of a function in terms of the mathematical functions it represents. It uses the example of a function determining the state of atari in the game of Go to illustrate how the size of the function, in terms of its domain and range, impacts the feasibility of exhaustive testing for correctness. The passage argues that while tests can demonstrate the presence of features, they cannot prove the absence of bugs due to the potentially vast number of test cases. However, by aligning types closely with the problem domain and choosing appropriate domain-inspired types, one can significantly reduce the size of functions and make them more manageable for testing, ensuring code correctness with a smaller set of examples.

95. Write Tests for People - Gerard Meszaros

In writing automated tests for your production code, being an early adopter is commendable, but the key is ensuring the quality of these tests. Rather than writing tests solely for personal convenience or compiler execution, it's crucial to consider the audience—primarily the person trying to comprehend your code. Effective tests act as documentation, detailing the code's workings through context, invocation, and expected results for different usage scenarios. To enhance clarity, minimize extraneous code in tests by using meaningful method calls, and employ the Extract Method refactoring. Give tests descriptive names reflecting the usage scenario, ensuring easy verification of test coverage. It's advisable to test your tests by deliberately inserting errors into the production code and confirming that errors are reported meaningfully. To validate clarity, have someone unfamiliar with your code read the tests and provide feedback, acknowledging that any confusion likely stems from communication issues rather than the reader's comprehension.

96. You Gotta Care About the Code - Pete Goodliffe

To get good code, you have to work at it. Hard. And you'll only get good code if you actually care about good code. Good programming lies in taking a professional approach, and wanting to write the best software you can, within the real-world constraints and pressures of the software factory. Great code is carefully crafted by master artisans, not thoughtlessly hacked out by sloppy programmers or erected mysteriously by self-professed coding gurus. To be a good programmer, you want to: strive to craft elegant code that is clearly correct; write code that is discoverable and maintainable; you work well alongside other programmers; any time you touch a piece of code, you strive to leave it better than you found it; you care about code and about programming, so you are constantly learning new languages, idioms, and techniques. But you apply them only when appropriate. Produce software that makes you proud.

97. Your Customers Do Not Mean What They Say - Nate Jackson

From this passage, it's evident that customers, while eager to express their desires, often communicate in a way that might not fully convey the details or nuances of what they truly want in software development. Customers may use terms interchangeably, assume certain knowledge, or lack a comprehensive vision of their requirements. To bridge this gap, the author emphasizes the importance of frequent and detailed interactions with customers. Challenging them, discussing topics multiple times, and using visual aids during conversations are recommended strategies to uncover hidden details, clarify ambiguities, and align expectations. The example of a misunderstood color scheme underscores the potential pitfalls of relying solely on verbal communication, highlighting the need for careful interpretation and validation of customer expectations throughout the software development process.