Nicole P. Satiembre **February 27, 2024**

Cebu Technological University-Main Campus- Intern

97 Things Every Programmer Should Know

Chapters 56-60

56. Make the Invisible More Visible - Jon Jagger

I have learned in this chapter that although we usually uphold invisibility concepts in software engineering, invisibility can be dangerous. Software and the process of developing it are mostly invisible, for example, Google's homepage may seem minimalistic, but, the code behind it is substantial. Another issue that invisibility raises is progress. When we are stuck in fixing bugs, we aren't necessarily making progress because we are not paid to debug. It goes the same way to a project that is months or weeks late. Lack of visible progress is synonymous with lack of progress. We can think more clearly when we have something concrete to tie our thinking to. We can manage things better when we can see them and see them as they evolve. That is why it is important to keep in mind to develop software with plenty of regular visible evidence. Visibility gives us the confidence that our progress is genuine and not an illusion. We can achieve this by writing unit tests; running unit tests that provide evidence about the code's behavior; using bulletin boards and cards; and doing incremental development also helps emphasize the visibility of development progress.

57. Message Passing Leads to Better Scalability in Parallel Systems - Russel Winder

This chapter emphasizes the challenges associated with shared memory, leading to issues like race conditions and deadlock. It highlights the difficulty of concurrency and parallelism in programming. The author condemns concurrency given the growing prevalence of multicore processors, emphasizing the importance of genuine parallelism for enhancing performance. He advised us to keep away from utilizing shared memory and adopt processes and message passing as a programming model, citing languages like Erlang as successful examples. This approach, free from synchronization stresses, is seen as more effective in harnessing parallelism. The passage introduces the idea of dataflow systems, emphasizing the absence of explicitly programmed control flow and the control of evaluation based on data readiness. Despite widely-used languages being geared towards shared-memory, multithreaded systems, the author suggests utilizing or creating libraries and frameworks that prioritize process models and message passing, ultimately asserting that programming without shared memory, using message passing, is the most successful way to implement systems capitalizing on the prevalent parallelism in modern computer hardware. All in all, not programming with shared memory, but instead using message passing, is likely to be the most successful way of implementing systems that harness the parallelism that is now endemic in computer hardware.

58. A Message to the Future - Linda Rising

As programmers, we often feel an ego boost when we write complex codes that only we can understand. We feel so smart about it. However, this is not the best case in reality. It is not only us who will touch this code, in the real world, other people will also help in maintaining it. When writing a code, we must consider that other people will come across it as well. Therefore, we must make sure that our code is easily understandable and well-documented. The teacher was very wise in giving this advice to her student and setting the younger brother as an example of someone who might maintain the code he had written. By that, when we write a code, we must think of it as a message to the future.

59. Missing Opportunities for Polymorphism - Kirk Pepperdine

In this chapter, I am reminded again about an important concept in OOP – Polymorphism. Polymorphism is very helpful in optimizing our code as it creates tiny localized execution contexts that let us work without the need for verbose if-then-else blocks. Used in the right context, polymorphism helps us achieve less code that is more readable. While there are cases where it's much more practical to use if-then-else instead of polymorphism, it is more often the case that a more polymorphic coding style will yield a smaller, more readable, and less fragile codebase.

60. News of the Weird: Testers Are Your Friends - Burk Hufnagel

Although most programmers hate testers a lot, in contrast, they are our friends. As programmers we may hate testers as they make our lives difficult, they find bugs that are too obscure, and they seem to want everything so perfect! However, the author shares a good experience about how a tester has helped him in his code. He used to maintain a code that was initially programmed by an accountant. It had some serious problems and when he thought he had fixed them, Margaret, the tester, would try to use it and it would fail in some new way after just a few keystrokes. Though initially frustrating, eventually, the day came when Margaret was able to cleanly start the program, enter an invoice, print it, and shut it down. He was thrilled. Even better, when they installed it on our customer's machine, it all worked. They never saw any problems because Margaret had helped him find and fix them first. That one experience just proves that testers are our friends. We want our bugs exposed early on before sending our product to our clients.