Nicole P. Satiembre **February 28, 2024**

Cebu Technological University-Main Campus- Intern

97 Things Every Programmer Should Know

Chapters 71-75

71. Read the Humanities - Keith Braithwaite

This chapter stresses the inherently human nature of software development, emphasizing that people collaborate to write software for the benefit of others. However, conventional programming education often fails to equip developers with the skill of how to navigate interpersonal dynamics with their colleagues and clients. Drawing from philosophical perspectives, Wittgenstein's argument challenges the notion that language serves as a straightforward transmission medium for thoughts, advocating for the importance of shared experiences and domain immersion among programmers. Lakoff and Johnson's exploration of metaphors in language highlights the metaphorical nature of communication, shaping our understanding of complex concepts like cash flow or system layers. Heidegger's analysis of tools underscores the different perspectives of programmers and users, where tools become invisible to users when functioning smoothly but draw attention when they malfunction. Rosch's work on categories challenges the Aristotelean model, emphasizing that users conceptualize the world based on examples and prototypes rather than crisp definitions, urging programmers to adopt a more nuanced approach when gathering user requirements. Overall, the insights underscore the significance of considering human aspects, shared experiences, metaphors, and user perspectives in software development for effective communication and understanding.

72. Reinvent the Wheel Often - Jason P. Sage

In college, we are often advised to keep ourselves from reinventing the wheel. Even in the real world, this is very much frowned upon. The time and effort in reinvention are unlikely to pay off as well as using an existing product or codebase. But why should we reinvent the wheel anyway when there is an established technology that has been tested a lot of times?

To become a good programmer, we must try to reinvent the wheel to get an intimate knowledge of the inner workings of various components that already exist. We can learn how memory managers work, virtual paging, double-linked lists, dynamic array classes, multiplexers etc. Often than not, we view these kinds of software as mysterious black boxes that just work. Not having a deep understanding of things in software development will limit our ability to create stellar work. Reinventing the wheel and getting it wrong is more valuable than nailing it the first time. There are lessons that we can only learn through trial and error. We can only build our programming skills through trying our challenges like these.

73. Resist the Temptation of the Singleton Pattern - Sam Saariste

The Singleton pattern seems like a silver bullet, promising a single instance, guaranteed initialization, and a straightforward global access point, making everything appear neat and tidy. However, according to a lot of experiences, this classic design pattern often does more harm than good. Its single-instance requirement is often speculative, hindering adaptability to changing requirements. Implicit dependencies and persistent state in singletons complicate unit testing, making it challenging to achieve loose coupling and independence between tests. Multithreading introduces pitfalls, with the commonly used double-checked locking pattern proving tricky and non-thread-safe in many cases. Cleaning up singletons poses challenges, lacking explicit killing support and introducing order issues during program exit. To overcome these shortcomings, I've learned to restrict the use of the Singleton pattern to cases where a class genuinely must never have more than one instance. Instead of allowing global access from arbitrary code, directing access through well-defined interfaces improves maintainability and breaks dependencies, making me think twice before implementing or accessing a singleton.

74. The Road to Performance Is Littered with Dirty Code Bombs - Kirk Pepperdine

When we are assigned to conduct performance tuning for a system, every chunk that is overly complex or highly coupled is a dirt code bomb lying in wait to derail the effort. Unexpected encounters with dirty code will make it very difficult to make a sane prediction. As you try to apply the fix, you quickly realize that you’ve broken a dependent part. Since closely related things are often necessarily coupled, this breakage is most likely expected and accounted for. All of a sudden our 3-4 hour estimate can easily balloon to 3-4 weeks. If only we had a tool to help us identify and measure this risk. We can use Software metrics to count the occurrences of specific features in our code. When using metrics, one must remember that they are only rules of thumb. Based purely on math, we can see that increasing fi without changing fo will move I closer to 0. Having a very large fan-in value makes it challenging to modify classes without disrupting dependents, and neglecting fan-out doesn't mitigate risks. Finding a balance is crucial. While software metrics may seem overwhelming, they are a valuable tool for identifying and eliminating problematic code before it poses a significant threat to performance-tuning efforts.

75. Simplicity Comes from Reduction - Paul W. Homer

A great lesson learned in this chapter is to avoid bad code by simply reducing our code. Just like in the previous chapters, simplicity is beauty. To avoid code complexity in the long run, we must strive to make our code clean and simple. We need to remove unnecessary lines, variables, comments, and blocks. Often we try to preserve an existing bad code, fearing that starting a new one will require significantly more effort than just going back to the beginning. More time gets wasted in trying to salvage bad work than it should. Once something becomes a resource sink, it needs to be discarded. Quickly. The code ought to be uncomplicated, with a minimal number of variables, functions, declarations, and other essential language elements. Any surplus lines, variables, or anything unnecessary should be removed promptly. The remaining code should be just sufficient to accomplish the task, whether it's completing an algorithm or carrying out calculations. Anything beyond that is extraneous and unwanted, creating unintended noise that obscures the flow and conceals the essential aspects.