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

Cebu Technological University-Main Campus- Intern

97 Things Every Programmer Should Know

Chapters 61-65

61. One Binary-Steve Freeman

The cons of customizing code for different target environments are highlighted in these findings, which stress the difficulties that result from this approach including inconsistencies, complexities and maintaining parallel versions. Detrimental aspects include long build cycles, waiting times for adjustments and uncertainties with regard to production deployments. One guiding principle proposed is a universal approach where one single promotion binary is built for all release pipeline steps with separate environment-specific details. The significance of thoughtful design resulting in a separation of core application features from platform-dependent details is underscored, calling for necessary modifications to be given priority. Incremental improvements are necessary while dealing with exceptions such as severe resource constraints or legacy problems. Moreover, keeping separate versioning information about environment away from code helps both in tracking changes made within environmental configurations for effective management purposes and should allows tracing on environment configuration changes done on various distributed version control systems like Git and Bazaar.

62. Only the Code Tells the Truth - Peter Sommerlad

Even the most accurate requirements document does not tell the whole truth: it does not contain the detailed story of what the program is actually doing, only the high-level intentions of the requirements analyst. This means that the only source of to understand how things are doing is through the source code. Because of this, we must strive to make our code readable, self-documenting, and maintainable. We can write comments that our code cannot explain, however, we must beware of this and only write necessary comments. If in the case of writing descriptions for the changes made in a code or program, it is bes to write these description in the version control document, separate from the code so as to limit the number of lines in a code. As much as possible, we want to limit our comments, we should consider refactoring our code if it needs a lot of comments to make it self-documenting. Lastly, we must treat our codes like a poem, crafting our ideas carefully so that it does what it should and communicates as directly as possible what it is doing.

63. Own (and Refactor) the Build - Steve Berczuk

Unmaintainable build scripts with duplication and errors cause problems of the same magnitude as those in poorly factored code. One reason why developers often treat build as something secondary to their work is that build scripts are often written in a different language than the source code. The build is an important part of the development process. Build scripts written using the wrong idioms are difficult to maintain and, more significantly, improve. It is a ground breeding for bugs if an application is built with the wrong version of a dependency or when a build-time configuration is wrong. The build process is as important as testing. The testing has always been left to the Quality Assurance team. On the other hand, the build process should be owned by the development team. Many build tools allow you to run reports on code quality, allowing you to sense potential problems early. By spending time understanding how to make the build yours, you can help yourself and everyone else on your team.

64. Pair Program and Feel the Flow - Gudny Hauknes, Kari Røssland, and Ann Katrin Gagnat

In this chapter I have learned the importance of coding in a team and how it contributes to improving our personal knowledge and skills. When working with a team, the people you meet are on different levels. We must learn how to adapt and go with the flow with our team members. We must learn to be patient with our team members who are below our skill level and encourage them and teach them the things that we know would help them improve. On the other hand, we must take the risk to talk with people whose skills are better then ours and recognize that we can learn so much from them. As a team, introduce pair programming to promote distribution of skills and knowledge throughout the project. You should solve your tasks in pairs and rotate pairs and tasks frequently. Agree upon a rule of rotation. Put the rule aside or adjust it when necessary. There are numerous situations where flow can be broken, but where pair programming helps you keep it: Reduce the “truck factor”, which means how dependent is your delivery on a certain team members; Solve problems effective, when running into a challenging problem, you always have someone to discuss it with; Integrate smoothly, using the discussion as an opportunity to improve the naming, docs, and testing; Mitigate interruptions, your partner will continue the flow as you catch up due to some interruptions; Bring new team members up to speed quickly. This is just an example of an incredibly productive flow that we can try to practice.

65. Prefer DomainSpecific Types to Primitive Types - Einar Landre

One famous example of a software failure caused by confusion on domain-specific typing was the “metric mix-up” where the ground-station software was working in pounds while the spacecraft expected newtons. That is such a huge difference if you think of it. It is also the primary reason why the Ada language aims to implement embedded safety-critical software. Ada has strong typing with static checking for both primitive types and user-defined types. It is beneficial to apply domain-specific typing as it helps our code become more readable and more testable, and facilitates reuse across applications and systems. The only difference is that developers using statically typed languages get some help from the compiler, while those embracing dynamically typed languages are more likely to rely on their unit tests. While there may be differences in the checking but the motivation and style of expression remains the same.