Nicole P. Satiembre **April 16, 2024**

Cebu Technological University-Main Campus

The Art of Readable Code

Chapter 11 - One task at a Time

A code that does multiple things at once is hard to understand. A single block of code might be initializing new objects, cleansing data, parsing inputs, and applying business logic, all at the same time. Code should be organized so that it's doing only one task at a time. We can achieve this by listing out all the "tasks" your code is doing and then try to separate those tasks as much as we can.

This chapter shows an example of what might be a short code but it's doing a lot. It performs two tasks at once, it is now then made easier by solving each task separately. We can extract values from an object once we observe that it is doing multiple tasks at a time, this is called "defragmentation". When refactoring code, there are often multiple ways to do it. Once the tasks have been separated, the code gets easier to think about. The new version of the code is a marked improvement from the original monstrosity. To sum up, this chapter illustrates a simple technique for organizing your code: do only one task at a time so try to defragment code as much as possible.

Nicole P. Satiembre **April 16, 2024**

Cebu Technological University-Main Campus

The Art of Readable Code

Chapter 12 - Turning Thoughts into Code

The ability to explain an idea in plain English is a very important skill so that someone who is less knowledgeable can understand. This not only helps another person understand but also helps you think about your own ideas clearly.

We can simplify a complex code by breaking it down, describing the logic clearly in plain English. For example, for a code snippet given in this example, we can say that the two ways you can be authorized is if you are an admin or if you own the current document (if there is one), otherwise, you are not authorized. Using this analogy, we can rewrite our code and simplify it. Another important thing when writing a succinct code is to know our libraries. This solution contains fewer lines of code and doesn't have to manipulate integers directly.

To conclude, this chapter teaches us a simple technique in describing our program in plain English and to use that description to help us write more natural code. If you can't describe the problem or your design in words, something is probably missing or undefined. Getting an idea into words is the first step in shaping our solutions.

Nicole P. Satiembre **April 16, 2024**

Cebu Technological University-Main Campus

The Art of Readable Code

Chapter 13 - Writing Less Code

Knowing when not to code is a very important skill every programmer must know. By reusing libraries or eliminating features, you can save time and keep your codebase lean and mean. When building a project, we often tend to overcomplicate things as we are excited about what features to add. A lot of features go unfinished or unused or just complicate the application. We should question and break down our requirements. If we do this, we can sometimes carve out a simpler problem that requires less code.

We should strive to keep our code base clean, usually when starting a project, we only have a few files but as the project grows, the directory fills up with more and more source files. To keep our codebases small and lightweight, we should:

- Create as much generic “utility” code as possible to remove the duplicated code.
- Remove unused code or useless features.
- Keep your project compartmentalized into disconnected subprojects.
- Generally, be conscious of the “weight” of your codebase. Keep it light and nimble.

Sometimes, we may struggle to delete some code because it represents a lot of work and time dedicated to it but we shouldn’t let this limit us. It is also helpful to be aware of existing libraries that can solve our problem and make the best use out of it. Reusing libraries saves us time and lets us write less code.

To sum up, the main advices given in this chapter in making our codes small and lightweight are:

- Eliminating non essential features from your product and not overengineering
- Rethinking requirements to solve the easiest version of the problem that still gets the job done
- Staying familiar with standard libraries by periodically reading through their entire APIs

Nicole P. Satiembre **April 16, 2024**

Cebu Technological University-Main Campus

The Art of Readable Code

Chapter 14 - Testing and Readability

Testing could mean different things to different people. In this chapter, we use “test” to mean any code whose sole purpose is to check the behavior of another (“real”) piece of code. This focuses on the readability aspect of tests and not getting into whether you should write test code before writing real code.

Writing readable test code is just as important as it is for non-test codes. If the tests are easy to read, users will better understand how the real code behaves. When test code is big and scary, coders are afraid to modify the real code and they don’t add new tests. Instead, we want to encourage users of our code to be comfortable with the test code.

In making our tests more readable, we can apply the general design principle, hiding less important details from the user, so that more important details are most prominent. Defining a custom mini language can be a powerful way to express a lot of information in a small amount of space. We should also strive to make our error messages readable. We can either use existing libraries for this or a hand-crafted error message. A general idea would be to pick the simplest set of inputs that completely exercise the code. It is also very important to choose good test inputs by picking the simplest set of inputs that completely exercise the code. Rather than construct a single “perfect” input to thoroughly exercise your code, it’s often easier, more effective, and more readable to write multiple smaller tests. We should also pick relevant names that describe details about the test.

To sum up, we must put high importance in making our test codes readable. If our test codes are readable, they will in turn be very writable, so people will add more of them.

Nicole P. Satiembre **April 16, 2024**

Cebu Technological University-Main Campus

The Art of Readable Code

Chapter 15- Designing and Implementing a "Minute/Hour Counter"

The problem raised in this chapter is that we need to keep track of how many bytes a web server has transferred over the past minute and over the past hour. We start off by picking appropriate names so as to avoid confusion. Instead of using a simple Count(), we can opt for Increment(), Observe(), Record(), or Add(). Next we improve our comments by removing redundant ones, clear up confusion by using a more precise and detailed language. Asking for an outside perspective is a great way to test if your code is "user-friendly."

When we write our codes, we first make a naive solution. Then we question it, is it easy to understand? We then try to explain the idea in simple plain English and then refactor it into an easier-to-read version. Although we have improved how the code looks, now we have another issue, the design keeps growing and the functions are too slow. Next we try a "conveyor belt" design. This design improved the speed and memory use but still wasn't good enough for high-performance applications. One major issue of this design is that it was inflexible. The final design solved the previous problems by breaking things down into subproblems and came up with three classes that solve each subproblem.