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

Cebu Technological University-Main Campus

The Art of Readable Code

Chapter 6- Making Comments Precise and Compact

The previous chapter talks about what we should comment on. This chapter on the other hand talks about how to write comments that are precise and compact. Comments should have a high information-to-space ratio. We must keep comments compact.

Avoid Ambiguous Pronouns

Pronouns can make things very confusing. It takes extra work for the reader to resolve a pronoun. In some cases, it is unclear what “it” or “this” is referring to.

Polish Sloppy Sentences

Making a comment more precise goes hand-in-hand with making it more compact.

Describe Function Behavior Precisely

A comment should tell the reader what the function does. It doesn’t have to be longer than the function itself.

Use Input/Output Examples That Illustrate Corner Cases

When it comes to comments, a carefully chosen input/output example can be worth a thousand words.

State the Intent of Your Code

Many comments end up just describing what the code does in literal terms, without adding much new information. Our comments should also explain what the program is doing at a higher level.

Named Function Parameter Comments

We should make sure that when we call a function, we're clear about the meaning of each piece of information we're giving it. Even in languages where we can't name the parts directly, we can use comments to explain things better. It's like adding little notes to ourselves to avoid confusion and make sure anyone reading the code later understands it easily. So, by sticking to a simple rule of commenting before boolean values and using consistent naming conventions, we can keep our code neat and easy to understand.

Use Information-Dense Words

Some specific words or phrases have been developed to describe these patterns/idioms. Making use of such powerful words can make our comments much more compact.

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

Cebu Technological University-Main Campus

The Art of Readable Code

Chapter 7 - Simplifying Loops and Logic

Every time you see a complicated loop, a giant expression, or a large number of variables, this adds to the mental baggage in your head. It requires you to think harder and remember more. This is exactly the opposite of “easy to understand.” When code has a lot of mental baggage, bugs are more likely to go unnoticed, the code becomes harder to change, and it’s just less fun to work with.

Making Control Flow Easy to Read

Make all your conditionals, loops, and other changes to control flow as natural as possible– written in a way that doesn’t make the reader stop and reread your code.

The Order of Arguments in Conditionals

When writing arguments, it is more readable to put the value that we are checking up on on the left side as it is the more “stable” value being compared against.

The Order of if/else Blocks

- Prefer dealing with the positive case first instead of the negative—e.g., if (debug) instead of if (!debug).
- Prefer dealing with the simpler case first to get it out of the way. This approach might also allow both the if and the else to be visible on the screen at the same time, which is nice.
- Prefer dealing with the more interesting or conspicuous case first.

The ?: Conditional Expression (a.k.a. “Ternary Operator”)

Although ternary operators can help lessen the lines of code, some argue that it can affect the readability of the code. The key idea here is, that instead of minimizing the number of lines, a better metric is to minimize the time needed for someone to understand it. By default, use an if/else. The ternary ?: should be used only for the simplest cases.

Avoid do/while Loops

This expression is executed at least once. The weird thing about a do/while loop is that a block of code may or may not be re-executed based on a condition underneath it. Another thing is because we typically read code from top to bottom, this makes the do/while loop seem to be a bit unnatural.

Returning Early from a Function

Some coders believe that functions should never have multiple return statements. One of the motivations for wanting a single exit point is so that all the cleanup code at the bottom of the function is guaranteed to be called.

The Infamous goto

Gotos are notorious for getting out of hand quickly and making code difficult to follow. The problems can come when there are multiple goto targets, especially when their paths cross. In particular, gotos that go upward can make for real spaghetti code, and they can surely be replaced with structured loops. Most of the time, goto should be avoided.

Minimize Nesting

Deeply nested code is hard to understand. Each level of nesting pushes an extra condition onto the reader's "mental slack".

How Nesting Accumulates

When we add a new chunk of code into our code, this new code gives a new context that is different from the original code. This is the way it was for you when you first read the code at the beginning of this section—you had to take it in all at once.

Removing Nesting by Returning Early

Nested block of codes can be removed by handling the "failure cases" as soon as possible and returning early from the function/

Removing Nested Inside Loops

The technique of returning early isn't always applicable. In the same way that an if (...) return; acts as a guard clause for a function, these if (...) continue; statements act as guard clauses for the loop. In general, the continue statement can be confusing, because it bounces the reader around, like a goto inside the loop. But in this case, each iteration of the loop is independent (the loop is a "for each"), so the reader can easily see that continue just means "skip over this item."

Can You Follow the Flow of Execution

In practice, programming languages and libraries have constructs that let code execute "behind the scenes" or make it difficult to follow. Some of these constructs are very useful and can make our code more readable and less redundant. The key is to not let a large percentage of your code use the constructs.

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

Cebu Technological University-Main Campus

The Art of Readable Code

Chapter 8 - Breaking Down Giant Expressions

Giant expressions are hard to think about. This chapter uses the analogy of a giant squid's flaw in design because it has a donut-shaped brain that wraps around its esophagus. So, if it swallows too much food at once, it gets brain damage. We can consider this analogy with our codes, the key idea here is to break down our giant expressions into a more digestible piece.

Explaining Variables

The simplest way to break down an expression is to introduce an extra variable that captures a smaller subexpression.

Summary Variables

A summary variable's purpose is to replace a larger chunk of code with a smaller name that can be managed and thought about more easily.

Abusing Short-Circuit Logic

This behavior is very handy but can sometimes be abused to accomplish complex logic. However, this practice can compromise the code readability. Beware of "clever" nuggets of code—they're often confusing when others read the code later.

Finding a More Elegant Approach

What started as a simple problem turned into a surprisingly convoluted piece of logic. This is often a sign that there must be an easier way. Finding a more elegant solution is a good idea but it requires creativity.

Breaking Down Giant Statements

Breaking down individual expressions, this same technique can be applied to breaking down larger statements as well.

Another Creative Way to Simplify Expressions

We can strip away the clutter by removing the codes that do the same thing. This can help our code look cleaner and we can understand easily the essence of what's happening.

Nicole P. Satiembre **April 17, 2024**

Cebu Technological University-Main Campus

The Art of Readable Code

Chapter 9 -Variables and Readability

Sloppy variables can make a program hard to understand. Specifically, there are three problems to contend with:

1. The more variables there are, the harder it is to keep track of them all.
2. The bigger a variable's scope, the longer you have to keep track of it
3. The more often a variable changes, the harder it is to keep track of its current value.

Eliminating Variables

We must try to break down our codes. One way to do so is to remove useless temporary variables. We might ask ourselves these questions if a variable is worth keeping or not:

- It isn't breaking down a complex expression.
- It doesn't add clarification to the expression
- It's used only once, so it doesn't compress any redundant code.

Eliminating Control Flow Variables

Variables whose sole purpose is to steer the program's execution – they don't contain any real program data are termed control flow variables. A code like this is often trying to satisfy some unspoken rule that you shouldn't break out of the middle of a loop. There is no such rule! They can be eliminated by making better use of structured programming.

Shrink the Scope of Your Variables

Avoid global variables because it's hard to keep track of where and how all those global variables are being used. The best thing to do instead is to shrink the scope of all your variables, not just the global ones.

Moving Definitions Down

The original C programming language required all variable definitions to be at the top of the function or block. The issue with this approach is that it forces the reader to think about the variables before they are even used.

Prefer Write-Once Variables

It's hard to think about variables that are constantly changing. Keeping track of their values adds an extra degree of difficulty. Prefer write-once variables. Variables that are a "permanent fixture" are easier to think about. The more places a variable is manipulated, the harder it is to reason about the current value.

Nicole P. Satiembre **April 17, 2024**

Cebu Technological University-Main Campus

The Art of Readable Code

Chapter 10- Extracting Unrelated Subproblems

Engineering is all about breaking down big problems into smaller ones and putting the solutions for those problems back together.

Pure Utility Code

Although there are some built-in utilities for the language of our choice, we might find a certain task to be repeated. This is the best time to write our utility code that can be shared in different parts of our program.

Our General-Purpose Code

We can try to extract lines in our code that don't help in performing the goal. For example, an alert() function in JavaScript is mostly used to print out data, however, it doesn't help in the logic of solving a code.

Create a Lot of General-Purpose Code

General-purpose codes are a good way to make our programs more compact. These codes can be reused across our projects. It is great because it's completely decoupled from the rest of your project.

Simplifying an Existing Interface

Creating a wrapper function is a powerful way to abstract away complexity in our codes. Even if the interface you're working with is less than ideal, you can improve your coding experience by crafting wrapper functions that provide a cleaner, more intuitive API. This practice enhances code readability and reduces cognitive load for developers.

Reshaping an Interface to Your Needs

Sometimes our codes can get out of hand, what started as a simple task turns into a lot of glue code. Glue codes are just there to support other code but they often have nothing to do with the real logic of our program. Mundane code like this is a great candidate to be pulled out into separate functions.

Taking Things Too Far

We might take things too far when trying to aggressively identify and extract unrelated subproblems. For example, we might break down our codes a little too much that it turns into a lot of tiny functions which actually hurts readability because we now have a lot more things to keep track of, and following the path of execution requires jumping around.