Nicole P. Satiembre **April 15, 2024**

Cebu Technological University-Main Campus

The Art of Readable Code

Chapter 1 - Code Should Be Easy to Understand

The idea of the Fundamental Theorem of Readability is that Code should be written to minimize the time it would take for someone else to understand it. For someone to fully understand your code, they should be able to make changes to it, spot bugs, and understand how it interacts with the rest of your code.

Generally speaking, the less code you write to solve a problem, the better. But even though having fewer lines of code is a good goal, minimizing the time until understanding is an even better goal. The hard part about this is that it requires extra work to constantly think about whether an imaginary outsider would find your code easy to understand.

Surface-Level Improvements

We begin by making surface-level improvements: picking good names, writing good comments, and formatting your code neatly. These changes are easy to apply and we can make them incrementally, without a huge time investment. These may be small changes but in aggregate, they can make a huge improvement to a codebase.

Nicole P. Satiembre **April 15, 2024**

Cebu Technological University-Main Campus

The Art of Readable Code

Chapter 2 - Packing Information into Names

A lot of names we see in programs are vague, like tmp. Even words that may seem reasonable, such as size or get, don't pack much information. It is important to pack information into our names.

Choose Specific Words

Part of packing information into names is choosing words that are very specific and avoiding empty words. For example, a method called Size() gives too vague information about the kind of size it returns. A more specific name would be Height(), NumNodes(), or MemoryBytes(). English is a rich language and we should make the best use out of it. The key idea here is that it is better to be clear and precise when naming than to be cute.

Avoid Generic Names like tmp and retval

A better name would describe the purpose of the variable or the value it contains. In this case, the variable is accumulating the sum of the squares of v. So a better name is sum_squares. This would announce the purpose of the variable upfront and might help catch a bug. The name retval doesn't pack much information. Instead, use a name that describes the variable's value.

Loop Iterators

Names like i, j, iter, and it are commonly used as indices and loop iterators. Even though these names are generic, they're understood to mean "I am an iterator." But sometimes there are better iterator names than i, j, and k, so, we shouldn't limit ourselves to this. If you’re going to use a generic name like tmp, it, or retval, have a good reason for doing so.

Prefer Concrete Names over Abstract Names

When naming a variable, function, or other element, describe it concretely rather than abstractly. For example, suppose you have an internal method named ServerCanStart(), which tests whether the server can listen on a given TCP/IP port. The name ServerCanStart() is somewhat abstract, though. A more concrete name would be CanListenOnPort(). This name directly describes what the method will do.

Attaching Extra Information to a Name

If there’s something very important about a variable that the reader must know, it’s worth attaching an extra word to the name. For example, you had a variable that contained a hexadecimal string, instead of naming it with just “id”, “hex_id” would be a much appropriate choice.

Values with Units

If your variable is a measurement (such as an amount of time or a number of bytes), it’s helpful to encode the units into the variable’s name for example, start_ms so that we know that this runs in terms of milliseconds. Besides time, there are plenty of other units that come up in programming.

Encoding Other Important Attributes

This technique of attaching extra information to a name isn’t limited to values with units. You should do it any time there’s something dangerous or surprising about the variable. For example, many security exploits come from not realizing that some data your program receives is not yet in a safe state. For this, you might want to use variable names like untrustedUrl or unsafeMessageBody. After calling functions that cleanse the unsafe input, the resulting variables might be trustedUrl or safeMessageBody.

Shorter Names are Okay for Shorter Scope

Identifiers that have a small “scope” don’t need to carry as much information. That is, you can get away with shorter names because all that information is easy to see,

Typing Long Names-Not a Problem Anymore

There are many good reasons to avoid long names, but “they’re harder to type” is no longer one of them. Every programming text editor we’ve seen has “word completion” built in. We must take good advantage of Intellisense from our compiler.

Acronyms and Abbreviations

Project-specific abbreviations are usually a bad idea. They appear cryptic and intimidating to those new to the project. Given enough time, they even start to appear cryptic and intimidating to the authors. So our rule of thumb is: would a new teammate understand what the name means? If so, then it’s probably okay.

Throwing out Unneeded Words

Sometimes words inside a name can be removed without losing any information at all. For instance, instead of ConvertToString(), the name ToString() is smaller and doesn’t lose any real information. Similarly, instead of DoServeLoop(), the name ServeLoop() is just as clear.

Use Name Formatting to Convey Meaning

The way you use underscores, dashes, and capitalization can also pack more information in a name. Having different formats for different entities is like a form of syntax highlighting—it helps you read the code more easily. Most of the formatting in this example is pretty common—using CamelCase for class names and using lower_separated for variable names. As well as using kConstantName instead of CONSTANT_NAME to distinguish #define macros.

Other Formatting Conventions

Depending on the context of your project or language, there may be other formatting conventions you can use to make names contain more information.

Summary:

The tips covered in this chapter for proper naming are:

- Use specific words
- Avoid Generic names
- Use concrete names
- Attach important details
- Use longer names for larger scopes
- Use capitalization, underscores, and so on in a meaningful way

Nicole P. Satiembre **April 15, 2024**

Cebu Technological University-Main Campus

The Art of Readable Code

Chapter 3 - Names that can’t be misconstrued

Actively scrutinize your names by asking yourself, “What other meanings could someone interprets from this name?” As we saw in the previous chapter, this is a case where the units should be attached to the name. In this case, we mean “number of characters,” so instead of max_length, it should be max_chars. The clearest way to name a limit is to put max_ or min_ in front of the thing being limited. Prefer first and last for Inclusive Ranges; Prefer begin and end for inclusive/exclusive ranges.

Naming Booleans

When picking a name for a boolean variable or a function that returns a boolean, be sure it’s clear what true and false really mean. In general, adding words like is, has, can, or should can make booleans more clear.

Matching Expectations of Users

Some names are misleading because the user has a preconceived idea of what the name means, even though you mean something else. In these cases, it’s best to just “give in” and change the name so that it’s not misleading. For instance, most programmers are used to the convention that methods starting with get are “lightweight accessors” that simply return an internal member. Going against this convention is likely to mislead those users.

Summary

As a review of the advice given in this chapter. the best names are ones that can't be misconstrued—the person reading your code will understand it the way you meant it, and no other way. The best names are resistant to misinterpretations.

- When it comes to defining an upper or lower limit for a value, max_ and min_ are good prefixes to use.
- For inclusive ranges, the first and last are good.
- For inclusive/exclusive ranges, begin and end are best because they're the most idiomatic.
- When naming a boolean, use words like is and has to make it clear that it's a boolean. Avoid negated terms (e.g., disable_ssl).
- Beware of users' expectations about certain words. For example, users may expect get() or size() to be lightweight methods.

Nicole P. Satiembre **April 15, 2024**

Cebu Technological University-Main Campus

The Art of Readable Code

Chapter 4- Aesthetics

A good source code should be just as “easy on the eyes”. The three recommended principles to use are:

- Use a consistent layout, with patterns the reader can get used to.
- Make similar code look similar.
- Group related lines of code into blocks.

Why Do Aesthetics Matter?

It’s easier to work with code that’s aesthetically pleasing. If you think about it, most of your time programming is spent looking at code! The faster you can skim through your code, the easier it is for everyone to use it.

- Rearrange Line Breaks to Be Consistent and Compact
- Use Methods to Clean Up Irregularity
- Use Column Alignment When Helpful
- Pick a Meaningful Order, and Use it Consistently
- Organize declaration into Blocks
- Break Code into Paragraphs

It is important to note that a consistent style is more important than the “right” style. Everyone prefers to read code that’s aesthetically pleasing. By “formatting” your code in a consistent, meaningful way, you make it easier and faster to read

Nicole P. Satiembre **April 15, 2024**

Cebu Technological University-Main Campus

The Art of Readable Code

Chapter 5 - Knowing What to Comment

The purpose of commenting is to help the reader know as much as the writer did. Reading a comment takes time away from reading the actual code, and each comment takes up space on the screen. That is, it better be worth it. Don’t comment on facts that can be derived quickly from the code itself.

Don’t Comment Just for the Sake of Commenting

Some professors require their students to have a comment for each function in their homework code. As a result, some programmers feel guilty about leaving a function naked without comments. This one falls into the “worthless comments” as the function’s declaration and the comments are virtually the same.

Don’t Comment Bad Names – Fix the Names Instead

A comment shouldn’t have to make up for a bad name.

Recording Your Thoughts

A lot of good comments can come out of simply “recording your thoughts”—that is, the important thoughts you had as you were writing the code.

Include “Director Commentary”

Include comments to record valuable insights about the code.

Comment the Flaws in your Code

Comment your thoughts about how the code should change in the future.

Comment on Your Constants

A lot of good comments can come out of simply “recording your thoughts”—that is, the important thoughts you had as you were writing the code.

Put Yourself in the Reader’s Shoes

As a general rule, imagine what your code looks like to someone who isn’t familiar with your project.

Even deep inside a function, it’s a good idea to comment on the “bigger picture.” Don’t get overwhelmed by the thought that you have to write extensive, formal documentation. A few well-chosen sentences are better than nothing at all. These comments also act as a bulleted summary of what the function does, so the reader can get the gist of what the function does before diving into details.

Summary

The purpose of a comment is to help the reader know what the writer knew when writing the code. This whole chapter is about realizing all the not-so-obvious nuggets of information you have about the code and writing those down.

What not to comment:
• Facts that can be quickly derived from the code itself.
• “Crutch comments” that make up for bad code (such as a bad function name)—fix the code instead.

Thoughts you should be recording include:
• Insights about why code is one way and not another (“director commentary”).
• Flaws in your code, by using markers like TODO: or XXX:.
• The “story” for how a constant got its value.

Put yourself in the reader’s shoes:
• Anticipate which parts of your code will make readers say “Huh?” and comment those.
• Document any surprising behavior an average reader wouldn’t expect.
• Use “big picture” comments at the file/class level to explain how all the pieces fit together.
• Summarize blocks of code with comments so that the reader doesn’t get lost in the details.