Clear is better than clever

by Dave Cheney

This article is based on my GopherCon Singapore 2019 presentation. In the presentation I referenced material from my post on declaring variables and my GolangUK 2017 presentation on SOLID design. For brevity those parts of the talk have been elided from this article. If you prefer, you can watch the recording of the talk.

Readability is often cited as one of Go's core tenets, I disagree. In this article I'll discuss the differences between clarity and readability, show you what I mean by clarity and how it applies to Go code, and argue that Go programmers should strive for clarity–not just readability–in their programs.

Why would I read your code?

Before I pick apart the difference between clarity and readability, perhaps the question to ask is, "why would I read your code?" To be clear, when I say *I*, I don't mean me, I mean you. And when I say *your code* I also mean you, but in the third person. So really what I'm asking is, "why would *you read* another person's code?"

I think Russ Cox, paraphrasing Titus Winters, put it best:

> Software engineering is what happens to programming when you add time and other programmers.
>
> –*Russ Cox, GopherCon Singapore 2018*

The answer to the question, "why would I read your code" is, because we have to work together. Maybe we don't work in the same office, or live in the same city, maybe we don't even work at the same company, but we do collaborate on a piece of software, or more likely consume it as a dependency.

This is the essence of Russ and Titus' observation; software engineering is the collaboration of software engineers over time. I have to read your code, and you read mine, so that I can understand it, so that you can maintain it, and in short, so that any programmer can change it.

Russ is making the distinction between software programming and software engineering. The former is a program you write for yourself, the latter is a program, a project, a service, a product, that many people will contribute to over time. Engineers will come and go, teams will grow and shrink, requirements will change, features will be added and bugs fixed. This is the nature of software engineering.

We don't read code, we decode it

> It was sometime after that presentation that I finally realized the obvious: Code is not literature. We don't read code, we *decode* it.
>
> –*Peter Seibel*

The author Peter Seibel suggests that programs are not read, but are instead decoded. In hindsight this is obvious, after all we call it source code, not source literature. The source code of a program is an intermediary form, somewhere between our concept–what's inside our heads–and the computer's executable notation.

In my experience, the most common complaint when faced with a foreign codebase written by someone, or some team, is the code is unreadable. Perhaps you agree with me?

But readability as a concept is subjective. Readability is nit picking about line length and variable names. Readability is holy wars about brace position. Readability is the hand to hand combat of style guides and code review guidelines that regulate the use of whitespace.

Clarity ≠ Readability

Clarity, on the other hand, is the property of the code on the page. Clear code is independent of the low level details of function names and indentation because clear code is concerned with what the code is doing, not just how it is written down.

When you or I say that a foreign codebase is unreadable, what I think what we really mean is, *I don't understand it*. For the remainder of this article I want to try to explore the difference between clear code and code that is simply readable, because the goal is not how quickly you can read a piece of code, but how quickly you can grasp its meaning.

Keep to the left

Go programs are traditionally written in a style that favours guard clauses and preconditions. This encourages the successful path to proceed down the page rather than indented inside a conditional block. Mat Ryer calls this line of sight coding, because, the active part of your function is not at risk of sliding out of sight beyond the right hand margin of your screen.

By keeping conditional blocks short, and for the exceptional condition, we avoid nested blocks and potentially complex value shadowing. The successful flow of control continues down the page. At every point in the sequence of statements, if you've arrived at that point, you are confident that a growing set of preconditions holds true.

```
func ReadConfig(path string) (*Config, error) {
    f, err := os.Open(path)
    if err != nil {
        return nil, err
```

```
        }
        defer f.Close()
        // ...
}
```

The canonical example of this is the classic Go error check idiom; `if err != nil` then return it to the caller, else continue with the function. We can generalise this pattern a little and in pseudocode we have:

```
if some condition {
        // true: cleanup
        return
}
// false: continue
```

If *some condition* is true, then return to the caller, else continue onwards towards the end of the function.

This form holds true for all preconditions, error checks, map lookups, length checks, and so forth. The exact form of the precondition's check changes, but the pattern is always the same; the cleanup code is inside the block, terminating with a return, the success condition lies outside the block, and is only reachable if the precondition is false.

Even if you are unsure what the preceding and succeeding code does, how the precondition is formed, and how the cleanup code works, it is clear to the reader that this is a guard clause.

Structured programming

Here we have a `comp` function that takes two `int`s and returns an `int`;

```
func comp(a, b int) int {
        if a < b {
                return -1
        }
        if a > b {
                return 1
        }
        return 0
}
```

The `comp` function is written in a similar form to guard clauses from earlier. If `a` is less than `b`, the return -1 path is taken. If `a` is greater than `b`, the return 1 path is taken. Else, `a` and `b` are by induction equal, so the final return 0 path is taken.

```
func comp(a, b int) int {
	if condition A {
		body A
	}
	if condition B {
		 body B
	}
	return 0
}
```

The problem with `comp` as written is, unlike the guard clause, someone maintaining this function has to read all of it. To understand when 0 is returned, the reader has to consult the conditions *and the body* of each clause. This is reasonable when you're dealing with functions which fit on a slide, but in the real world complicated functions–the ones we're paid for our expertise to maintain–are rarely slide sized, and their conditions and bodies are rarely simple.

Let's address the problem of making it clear under which condition 0 will be returned:

```
func comp(a, b int) int {
	if a < b {
		return -1
	} else if a > b {
		return 1
	} else {
		return 0
	}
}
```

Now, although this code is not what anyone would argue is readable–long chains of `if else if` statements are broadly discouraged in Go–it is clearer to the reader that zero is only returned if none of the conditions are met.

How do we know this? The Go spec declares that each function that returns a value must end in a terminating statement. This means that the body of all conditions must return a value. Thus, this does not compile:

```
func comp(a, b int) int {
	if a > b {
		a = b // does not compile
	} else if a < b {
		return 1
	} else {
```

```
        return 0
    }
}
```

Further, it is now clear to the reader that this code isn't actually a series of conditions. This is an example of selection. Only one path can be taken regardless of the operation of the condition blocks. Based on the inputs one of -1, 0, or 1 will always be returned.

```
func comp(a, b int) int {
    if a < b {
        return -1
    } else if a > b {
        return 1
    } else {
        return 0
    }
}
```

However this code is hard to read as each of the conditions is written differently, the first is a simple `if a < b`, the second is the unusual `else if a > b`, and the last conditional is actually unconditional.

But it turns out there is a statement which we can use to make our intention much clearer to the reader; `switch`.

```
func comp(a, b int) int {
    switch {
    case a < b:
        return -1
    case a > b:
        return 1
    default:
        return 0
    }
}
```

Now it is clear to the reader that this is a selection. Each of the selection conditions are documented in their own case statement, rather than varying `else` or `else if` clauses.

By moving the default condition inside the switch, the reader only has to consider the cases that match their condition, as none of the cases can fall out of the switch block because of the default clause.[1]

> Structured programming submerges *structure* and emphasises *behaviour*
>
> —*Richard Bircher, The limits of software*

I found this quote recently and I think it is apt. My arguments for clarity are in truth arguments intended to emphasise the behaviour of the code, rather than be side tracked by minutiae of the structure itself. Said another way, what is the code trying to do, *not how is it is trying to do it.*

Guiding principles

I opened this article with a discussion of readability vs clarity and hinted that there were other principles of well written Go code. It seems fitting to close on a discussion of those other principles.

Last year Bryan Cantrill gave a wonderful presentation on operating system principles, wherein he highlighted that different operating systems focus on different principles. It is not that they ignore the principles that differ between their competitors, just that when the chips are down, they prioritise a core set. So what is that core set of principles for Go?

Clarity

If you were going to say readability, hopefully I've provided you with an alternative.

> Programs must be written for people to read, and only incidentally for machines to execute.
>
> —*Hal Abelson and Gerald Sussman. Structure and Interpretation of Computer Programs*

Code is read many more times than it is written. A single piece of code will, over its lifetime, be read hundreds, maybe thousands of times. It will be read hundreds or thousands of times because it must be understood. Clarity is important because all software, not just Go programs, is written by people to be read by other people. The fact that software is also consumed by machines is secondary.

> The most important skill for a programmer is the ability to effectively communicate ideas.
>
> —*Gastón Jorquera*

Legal documents are double spaced to aide the reader, but to the layperson that does nothing to help them comprehend what they just read. Readability is a property of how easy it was to read the words on the screen. Clarity, on the other hand, is the answer to the question "did you understand what you just read?".

If you're writing a program for yourself, maybe it only has to run once, or you're the only person who'll ever see it, then do what ever works for you. But if this is a piece of software that more than one person will contribute to, or that will be used by people over a long enough time that requirements, features, or the environment it runs in may change, then your goal must be for your program to be maintainable.

The first step towards writing maintainable code is making sure intent of the code is clear.

Simplicity

The next principle is obviously simplicity. Some might argue the most important principle for any programming language, perhaps the most important principle full stop.

Why should we strive for simplicity? Why is important that Go programs be simple?

> The ability to simplify means to eliminate the unnecessary so that the necessary may speak
>
> –*Hans Hofmann*

We've all been in a situation where we say "I can't understand this code". We've all worked on programs we were scared to make a change because we worried that it'll break another part of the program; a part you don't understand and don't know how to fix.

This is complexity. Complexity turns reliable software in unreliable software. Complexity is what leads to unmaintainable software. Complexity is what kills software projects. Clarity and simplicity are interlocking forces that lead to maintainable software.

Productivity

The last Go principle I want to highlight is productivity. Developer productivity boils down to this; how much time do you spend doing useful work verses waiting for your tools or hopelessly lost in a foreign code-base? Go programmers should feel that they can get a lot done with Go.

> "I started another compilation, turned my chair around to face Robert, and started asking pointed questions. Before the compilation was done, we'd roped Ken in and had decided to do something."
>
> –*Rob Pike, Less is Exponentially more*

The joke goes that Go was designed while waiting for a C++ program to compile. Fast compilation is a key feature of Go and a key recruiting tool to attract new developers. While compilation speed remains a constant battleground, it is fair to say that compilations which take minutes in other languages, take seconds in Go. This helps Go developers feel as productive as their counterparts working in dynamic languages without the maintenance issues inherent in those languages.

> Design is the art of arranging code to work *today*, and be changeable *forever.*
>
> –*Sandi Metz*

More fundamental to the question of developer productivity, Go programmers realise that code is written to be read and so place the act of reading code above the act of writing it. Go goes so far as to enforce, via tooling and custom, that all code be formatted in a specific style. This removes the friction of learning a project specific dialect and helps spot mistakes because they just look incorrect.

Go programmers don’t spend days debugging inscrutable compile errors. They don’t waste days with complicated build scripts or deploying code to production. And most importantly they don’t spend their time trying to understand what their coworker wrote.

> Complexity is anything that makes software hard to understand or to modify.
>
> —*John Ousterhout*, *A Philosophy of Software Design*

Something I know about each of you reading this post is you will eventually leave your current employer. Maybe you’ll be moving on to a new role, or perhaps a promotion, perhaps you’ll move cities, or follow your partner overseas. Whatever the reason, we must all consider the succession of the maintainership of the programs we create.

If we strive to write programs that are clear, programs that are simple, and to focus on the productivity of those working with us that will set all Go programmers in good stead.

Because if we don’t, as we move from job to job, we’ll leave behind programs which cannot be maintained. Programs which cannot be changed. Programs which are too hard to onboard new developers, and programs which feel like career digression for those that work on them.

If software cannot be maintained, then it will be rewritten; and that could be the last time your company invests in Go.

The `fallthrough` keyword complicates this analysis, hence the general disapproval of `fallthrough` in switch statements.