

Clear is better than clever

GopherCon Singapore 2019

Hello and welcome to GopherCon Singapore. I'm delighted to be able to speak to you today.

This talk is the latest in what has become a series of turning Rob Pike's proverbs into full length presentations.

To set the scene, we [the room] often cite readability as one of Go's core tenets, I disagree.

In this talk I'd like to discuss the differences between readability and clarity, 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 we pick apart the difference between clarity and readability, perhaps the real 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.

Why would *you* read *another person's* code?

So really what I'm asking is:

Why would *you read* another person's code?

Well, I think Russ, paraphrasing Titus Winters, puts it best.

Software engineering is what happens to programming when you add time and other programmers.

— *Russ Cox*

(A quote which I think was uttered here in Singapore last year)

So 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’s quote—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.

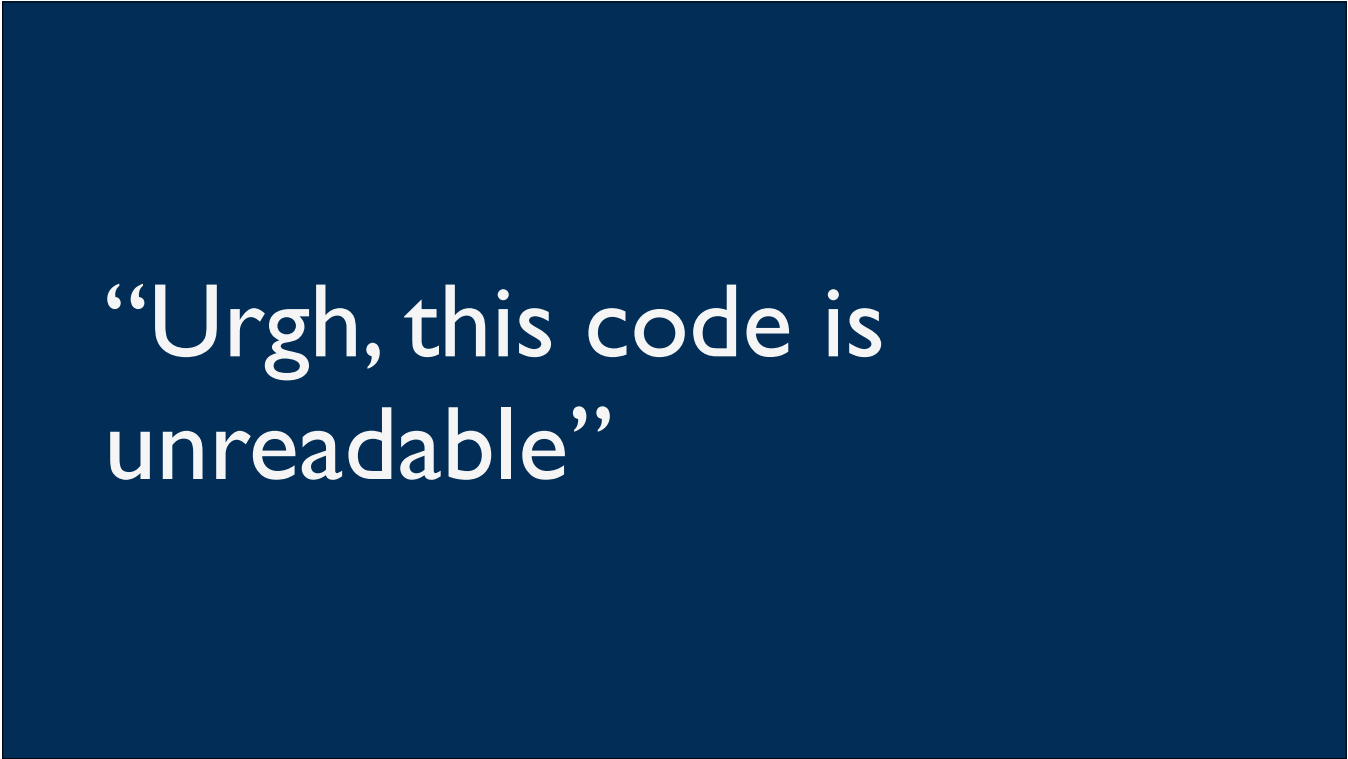
To be clear, I don’t mean to dismiss the work of a lone programmer toiling on programs without anyone to pair with or learn from. I’ve been that person many times in my career as a programmer, it’s not fun.

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.

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, instead they are decoded. In hindsight this should be 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.



“Urgh, this code is
unreadable”

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

Does anyone agree with me?

But readability as a concept is subjective.

Readability 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 \neq 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 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.

So today, through a few case studies I want to try to explore the difference between clear code and code that is simply readable.


```
x := 1  
var y = 2  
var z int = 3  
var a int; a = 4  
var b = int(5)  
c := int(6)
```

In Go each variable has a purpose because each variable we declare has to be used within the scope in which it was declared.

Go has many—some would say too many—ways to declare a variable. Given its spartan syntax, Go provides us with at least half a dozen different ways to declare, and optionally initialise, a variable.

How can we make the *purpose* of each declaration clear to the reader?

How can we make clear to the reader, the purpose of each declaration?

The answer to this question comes down to the intent of the declaration. Are we declaring a variable which will be assigned later, or are we declaring a variable with a specific value?

Here is a suggestion that I will attempt to defend:

When declaring—but *not*
initialising—a variable,
use `var`

When declaring a variable that will be explicitly initialised later in the function, use the `var` keyword.

The `var` acts as a clue to say that this variable has been deliberately declared as the zero value of the indicated type. To the reader, it is clear that that responsibility to assigning this variable lies elsewhere in the function (hopefully not far from its declaration)

When declaring *and*
initialising, use `:=`

When declaring and initialising the variable at the same time, that is to say we're not letting the variable be implicitly initialised to its zero value, use the short variable declaration form.

This makes it clear to the reader that the variable on the left hand side of the `:=` is being deliberately initialised with a specific value.

```
var players int    // 0

var things []Thing // an empty slice of Things

var thing Thing    // empty Thing struct
json.Unmarshal(reader, &thing)
```

Let's expand on these suggestions a little.

Consider this sample code, which is using the first rule — use var for the zero value

```
var players int = 0

var things []Thing = nil

var thing *Thing = new(Thing)
json.Unmarshal(reader, thing)
```

Now let's look at what happens if we explicitly initialise each variable

In the first and third examples, because in Go there are no automatic conversions from one type to another; the type on the left hand side of the assignment operator must be identical to the type on the right hand side.

The compiler can infer the type of the variable being declared from the type on the right hand side, such that the example can be written more concisely like this:

```
var players = 0  
  
var things []Thing = nil  
  
var thing = new(Thing)  
json.Unmarshal(reader, thing)
```

And when we do this it leaves us explicitly initialising players to 0 which is redundant because 0 is `players' zero value.

```
var players int  
  
var things []Thing = nil  
  
var thing = new(Thing)  
json.Unmarshal(reader, thing)
```

So its better to make it clear that we're going to use the zero value by instead writing this

What about the second statement?


```
var players int  
  
var things = nil // syntax error  
  
var thing = new(Thing)  
json.Unmarshal(reader, thing)
```

We cannot elide the type and write

Because nil does not have a type. Thus the compiler cannot infer the type of things, because nil does not have a type in this context.

So we have a choice to make

```
var players int  
  
var things []Thing  
  
var thing = new(Thing)  
json.Unmarshal(reader, thing)
```

Do we want to initialise things to be the zero value for a slice?

```
var players int  
  
var things = make([]Thing, 0)  
  
var thing = new(Thing)  
json.Unmarshal(reader, thing)
```

or do we want to initialise things to be a slice with zero elements?

If we wanted the latter then this is not the zero value for a slice so we should make it clear to the reader that we're making this choice by using the short declaration form:

```
var players int  
  
things := make([]Thing, 0)  
  
var thing = new(Thing)  
json.Unmarshal(reader, thing)
```

Which tells the reader that we have chosen to initialise things explicitly.

Why we'd do this is unclear, perhaps we have a piece of code that can tell the difference between a nil slice and an empty slice, although that in itself is a smell.

This brings us to the third declaration,

```
var players int  
  
things := make([]Thing, 0)  
  
var thing = new(Thing)  
json.Unmarshal(reader, thing)
```

Which is both explicitly initialising a variable and introduces the uncommon use of the new keyword which some Go programmer dislike.

If we apply our short declaration syntax recommendation then the statement becomes

```
var players int  
  
things := make([]Thing, 0)  
  
thing := new(Thing)  
json.Unmarshal(reader, thing)
```

Which makes it clear that thing is explicitly initialised to the result of new(Thing)--a pointer to a Thing--but still leaves us with the unusual use of new.

We could address this by using the compact literal struct initialiser form,

```
var players int  
  
things := make([]Thing, 0)  
  
thing := &Thing{}  
json.Unmarshal(reader, thing)
```

Which does the same as `new(Thing)`—hence why some Go programmers are upset by the duplication—However this means we’re explicitly initialising `thing` with a pointer to a `Thing{}`, which is the zero value for a `Thing`.

```
var players int  
  
things := make([]Thing, 0)  
  
var thing Thing  
json.Unmarshal(reader, &thing)
```

Instead we should recognise that thing is being declared as its zero value and use the address of operator to pass the address of thing to json.Unmarshal

Exceptions make the rule

Of course, with any rule, there are exceptions.

```
var min int  
max := 1000
```

For example, sometimes two variables are closely related so writing

Would be odd.

```
min, max := 0, 1000
```

The declaration is clearer like this.

We break the rule and explicitly initialise min to its zero value, because there is nothing particularly notable about the declaration of min.

When something *is* complicated,
it should *look* complicated

Make tricky declarations obvious

And we do that because when something is complicated, it should look complicated.

```
length := uint32(0x80)
```

Here length may be being used with a library which requires a specific numeric type. How obvious is that to the reader?

Someone coming along later might think that the conversion to uint32 is redundant and be tempted to remove it.

```
var length uint32 = 0x80
```

Here, I'm being explicit that `_length_` is declared to be a `uint32` — not the type of the value being assigned to `length`.

I'm deliberately breaking my rule of using the `var` declaration form with an explicit initialiser. This decision to vary from my usual form is a clue to the reader that something unusual is happening.

If something unusual is
happening, leave evidence
for the reader

—Brian Kernighan

Adopt a consistent declaration style so that it is clear that when something is unusual, it looks unusual.

This is a clue to the reader that they should pay extra attention because the thing that is declared oddly, is likely to be used oddly.

Accept interfaces, return structs

The next case study covers the use of interfaces to describe the behaviour of the values being passed into a function independent from the parameter's types, or their implementation.


```
type Document struct {  
    // bigly state  
}  
  
// Save writes the contents of d to the file f.  
func (d *Document) Save(f *os.File) error
```

Say I've been given a task to write a method to save a structure to disk.

I could define this method, let's call it `Save`, which takes an `*os.File` as the destination to write the `Document`. But this has a few problems.

The signature of `Save` precludes the option to write the data to a network location. Assuming that network storage is likely to become requirement later, we are all using cloud storage and micro services these days, the signature of this function would have to change, impacting all its callers.

Because `Save` operates directly with files on disk, it is unpleasant to test. To verify its operation, the test would have to read the contents of the file after being written. Additionally the test would have to ensure that `f` was written to a temporary location and always removed afterwards.

`*os.File` also defines a lot of methods which are not relevant to `Save`, like reading directories and checking to see if a path is a symlink. It would be useful if the signature of our `Save` function could describe only the parts of `*os.File` that were relevant.

The first solution comes to mind is interfaces. Interfaces document the behaviour of a type. Can we use an interface to describe the behaviour that `Save` requires from its argument?

```
func (d *Document) Save(rwc io.ReadWriterCloser) error
```

Using `io.ReadWriterCloser` we can apply the Interface Segregation Principle to redefine `Save` to take an interface that describes the behaviour of the file shaped things. With this change, any type that implements the `io.ReadWriterCloser` interface can be substituted for the previous `*os.File`. This makes `Save` both broader in its application, and clarifies to the caller of `Save` which methods of the `*os.File` type are relevant to its operation.

As the author of `Save` I no longer have the option to call those unrelated methods on `*os.File` as it is hidden behind the `io.ReadWriterCloser` interface. But we can take the interface segregation principle a bit further.

Firstly, it is unlikely that if `Save` follows the Single Responsibility Principle it will read the file it just wrote to verify its contents; that should be responsibility of another piece of code. So we can narrow the specification for the interface we pass to `Save` to just writing and closing.

```
func (d *Document) Save(wc io.WriteCloser) error
```

By providing Save with a mechanism to close its stream, which we inherited in a desire to make it look like a file shaped thing, this raises the question of under what circumstances will wc be closed.

Possibly Save will call Close unconditionally, or perhaps Close will be called in the case of success. This presents a problem for the caller of Save as it may want to write additional data to the stream after the document is written.

```
type NopCloser struct {  
    io.Writer  
}  
  
// Close has no effect on the underlying writer.  
func (c *NopCloser) Close() error {  
    return nil  
}
```

A crude solution would be to define a new type which embeds an `io.Writer` and overrides the `Close` method, preventing `Save` from closing the underlying stream. But this would probably be a violation of the Liskov Substitution Principle, as `NopCloser` implements `close`, but doesn't actually close anything.

```
func (d *Document) Save(w io.Writer) error
```

A better solution would be to redefine `Save` to take only an `io.Writer`, stripping it completely of the responsibility to do anything but write data to a stream.

By applying the interface segregation principle to our `Save` function, the results has simultaneously been a method which is clear in terms of the behaviour it requires—it only needs a thing that is writable—and the most general in its functionality, we can now use `Save` to save our data to anything which implements `io.Writer`.

Lastly, although I opined in the introduction that naming things was small beer, now that we have refactored `Save`, it's clear a better name for the method is probably `WriteTo`

```
func (d *Document) WriteTo(w io.Writer) error
```

Keep to the left

Go programs are traditionally written in a style that favours guard clauses, preconditions. This encourages the success path to proceed down the page, rather than be 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 view to the right of your screen.

By keeping conditional blocks short, and for the exceptional condition, we avoid nested blocks and potential complex value shadowing.

The successful flow of control continues down the page. At every point in the sequence of instructions, if you've arrived at that point, you know 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 err check idiom; if err is not nil, then return it to the caller, else continue with the function


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

We can generalise this pattern a little, in pseudocode we have

If the precondition is true, then return to the caller, else continue towards the end of the function.

This general condition holds true for all preconditions, error checks, map lookups, length checks, etc. The exact form of the precondition's condition 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
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 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 is trying to do, *not how it is trying to do it*

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

What is going on here? We have a function that takes two ints and returns an int,

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> {  
        <body>  
    }  
    if <condition> {  
        <body>  
    }  
    return 0  
}
```

The problem with comp as written is, unlike the guard clause, is someone maintaining this function has to read all of it.

To understand when 0 is returned, we have to read the conditions and the body of each clause. This is reasonable when your 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.

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

Let's address the problem of making it clear under which condition 0 is returned.

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

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

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 {  
        return -1  
    } else if a > b {  
        return 1  
    } else {  
        return 0  
    }  
}
```

So, now it's clear to the reader that this code isn't actually a series of conditions, instead 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 be returned.

However this code is hard to read as each of the conditions is written differently, the first is a simple $a < b$, the second is the complex and unusual $\text{else if } a > b$, and the last conditional is unconditional.

And 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.

And 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.

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

Guiding principles

I opened this talk with a discussion of readability vs clarity, and I also 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 choose a core set.

So what is that core set of principles for Go?

Clarity

If you were going to say readability, hopefully I've given you an alternative today.

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.

Clarity is important because all software, not just Go programs, is written by humans to be read by other humans. 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*

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 Simplicity, some may 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 you say "I can't understand this code", yes?

We've all worked on programs where you're scared to make a change because you're worried it'll break another part of the program; a part you don't understand and don't know how to fix.

This is complexity.

Simplicity is prerequisite for reliability

—Edsger W. Dijkstra

Complexity turns reliable software in unreliable software. Complexity is what kills software projects.

Clarity and simplicity are interlocking forces that give us the language we're here to celebrate today.

Whatever programs we write, we should be able to agree that they are clear, they are simple, and thus we as Go programmers are productive.

Productivity

The last underlying 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.

To say that Go is a language designed to be productive is an understatement it is built for software design in the large for industrial application.

Complexity is anything
that makes software
hard to understand or to
modify.

— *John Ousterhout*

Something I know about each of you in this room 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 consider the succession of the maintainership of the programs we write.

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

Because if we don't, then 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 digression in their career if they work on them.

If software cannot be maintained,
it will be rewritten

... and that could be the last time
your company will invest in Go

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

And on that bombshell, I'm going to leave you to hopefully write clearer code for those that come after you.

Please enjoy your day here at GopherCon Singapore.

Thank you very much.