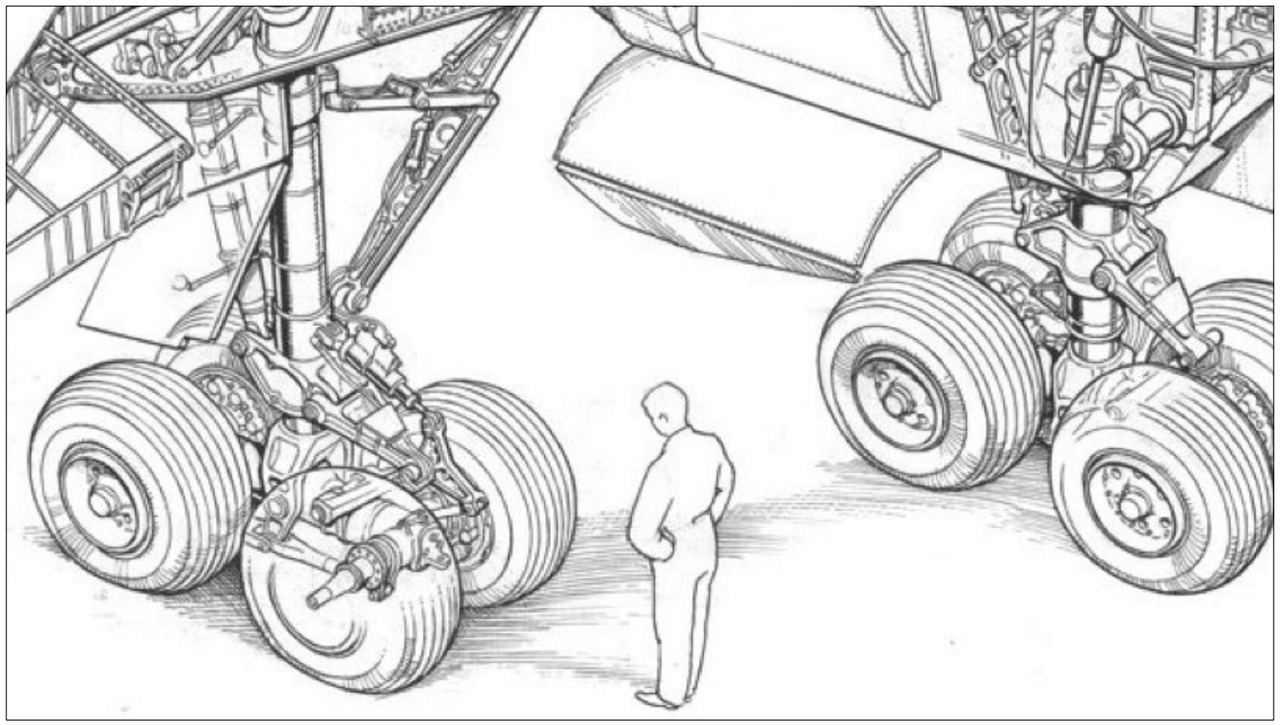
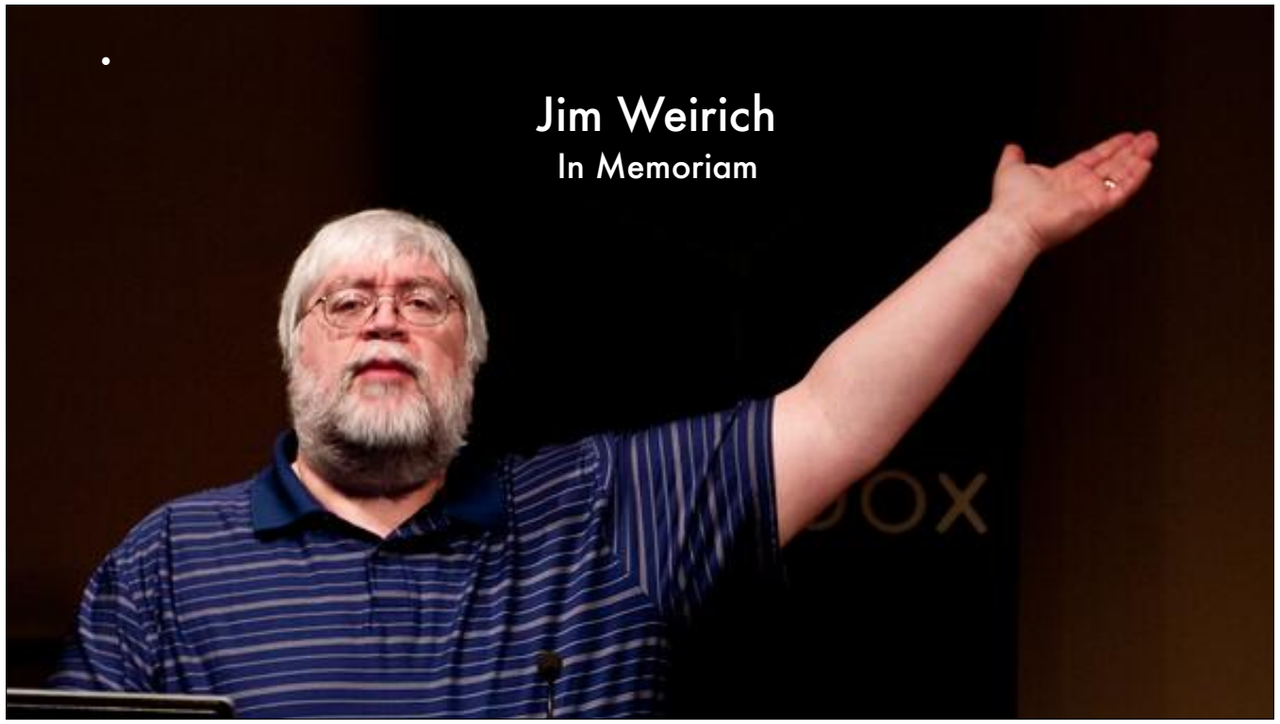


Good morning and thank you for coming to my talk.

Thank you to Natalie for inviting me to speak to you today.



Before I get started I want to mention a few things

The first is the idea for this talk came from Jim Weirich, who sadly passed away in 2014. Jim was a hugely influential voice in the ruby community, and it was his attempts to apply the SOILD principles — originally developed in the mid 90's with a view to C++ — to Ruby, that were the inspiration for this talk.

The second is the ideas that Jim's talk, and my facsimile, are based on the work of Robert Martin. I want to make it clear I do not agree with Martin's recent comments and this talk is in no way an endorsement of his position.

Who does code review as part of their job?

Who here does code review as part of their job?

Why do you do code review?

To catch bad code?

If code review is there to catch bad code, how do you know if the code you're reviewing is good, or bad?

Now it's fine to say, “that code is ugly’, or ”wow that source code is beautiful".

Just like you might say, "this painting is beautiful", or "this room is beautiful" but these are subjective terms, and we're looking for are objective ways to talk about the properties of good or bad code.

Bad Code 🚽

Rigidity
Fragility
Immobility
Complexity
Verbosity

What are some of the properties of bad code that you might pick up on in code review.

Rigidity.
Is the code rigid, does it have a straight jacket of overbearing types and parameters, that making modification difficult.

Fragility.
Is the code fragile, does the slightest change ripple through the code base causing untold havoc?

Immobility.
Is the code hard to refactor, is it one keystroke away from an import loop?

Complexity.
Is there code for the sake of having code, are things over-engineered?

Verbosity.
Is it just exhausting to use the code?
When you look at it, can you even tell what this code is trying to do?

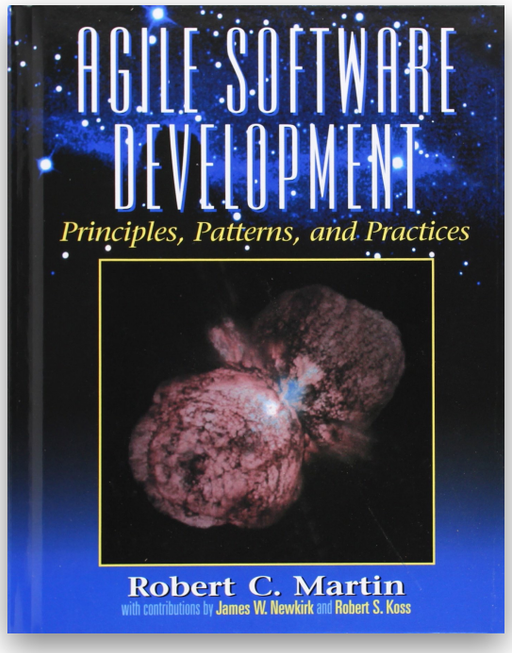
Are these positive sounding words?
Would you be pleased to see these words used in a review of your code?

Probably not.

Good Design 🙌

ok, so that's great, now we can say things like "i don't like this because it's too had to modify", or, "i don't like this because i cannot tell what the code is trying to do", but what about leading with the positive?

Wouldn't it be great if there were some ways to describe good design, not just bad design, and to be able to do so in objective terms.



So back in 2003 Robert Martin published this book.

In it he described five principles of reusable software design, which he called the SOLID principles, after the first letters in their names.

SOLID

Single Responsibility Principle
Open / Closed Principle
Liskov Substitution Principle
Interface Segregation Principle
Dependency Inversion Principle

- Single Responsibility Principle
- Open / Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

This book is a little dated—as apparently are Martin's views on inclusive workplaces —the languages that it talks about are the ones in use more than a decade ago.

But, perhaps there are some aspects of the SOLID principles that may give us a clue about how to talk about a well designed Go programs.

So this is what I want to spend some time discussing with you this morning.

Single Responsibility Principle

The first principle of SOLID, the S, is the single responsibility principle:

A class should have one, and only one, reason to change.

–Robert C Martin

Which reads

“a class should have one, and only one, reason to change”

Now Go obviously doesn't have classes—instead we have the far more powerful notion of composition—but if you can look past the use of the word class, I think there is some value here.

You might have picked up that martin does not say “a class should do *one thing*, instead he says *one reason to change.*
Why is it important that a piece of code should have only one reason for change?

Well, as distressing as the idea that your own code may change, it is far more distressing to discover that code your code depends on is changing under your feet.

When your code does have to change, it should do so in response to a direct stimuli, it shouldn't be a victim of collateral damage.

So code that has a single responsibilty therefore has the fewest reasons to change.

Coupling & Cohesion

Two words that describe how easy or difficult it is to change a piece of software are coupling and cohesion.

Coupling is simply a word that describes two things moving together--a movement in one induces a movement in another.

A related, but separate, notion is the idea of cohesion, a force of mutual attraction.

In the context of software, cohesion is the property of describing pieces of code are naturally attracted to one another. They are *coherent.* They fit well together.

To describe the units of coupling and cohesion in a Go program, we might talk about functions and methods, as is very common when discussing SRP but really I believe it starts with Go's package model.

Package names

```
net/http

os/exec

encoding/json
```

In Go, all code lives inside a package, and a well designed package starts with its name.

A package’s name is both a description of its purpose, and a name space prefix.
Some examples of good packages from the Go standard library might be:

- net/http, which provides http clients and servers.
- os/exec, which runs external commands.
- encoding/json, which implements encoding and decoding of JSON documents.

When you use another package's symbols inside your own this is accomplished by the `import` declaration, which establishes a source level coupling between two packages.
They now know about each other.

Bad package names

```
package server

package private

package common
```

This focus on names is not just pedantry.
A poorly named package misses the opportunity to enumerate its purpose, if indeed it ever had one.

    package server

What does the server package provide … well a server, hopefully, but which protocol ?

    package private

what does the private package provide? Things that I should not see? Should it have any public symbols ?

    package common

And package common, just like it's partner in crime, package utils, is often found close by these other offenders.

Catch all packages like this become a dumping ground for miscellany, and because they have many responsibilities they change frequently and without cause.

Go's UNIX philosophy

In my view, no discussion about decoupled design would be complete without mentioning Doug McIlroy's Unix philosophy ; small, sharp tools which combine to solve larger tasks

oftentimes which were not envisioned by the original authors.

I think that Go packages embody the spirit of the UNIX philosophy.

In effect each Go package is itself a small Go program, a single unit of change, with a single responsibility.

Open / Closed Principle

The second principle, the O, is the open closed principle by Bertrand Meyer who in 1988 wrote

> Software entities should be open for extension, but closed for modification.
>
> –Bertrand Meyer, Object-Oriented Software Construction

“software entities should be open for extension, but closed for modification”

So, how does this advice apply to a language written 21 years later?

```
package main

type A struct {
        year int
}

func (a A) Greet() { fmt.Println("Hello ", a.year) }

type B struct {
        A
}

func (b B) Greet() { fmt.Println("Welcome to ", b.year) }

func main() {
        var a A
        a.year = 2016
        var b B
        b.year = 2016
        a.Greet() // Hello  2016
        b.Greet() // Welcome to  2016
}
```

We have a type `A`, with a field `year`, which has a method `Greet`.

We have a second type `B` which _embeds_ an `A`.

a caller will see `B`'s methods overlaid on `A`'s because `A` is embedded, as a field, within `B`, and `B` can provide its own `Greet` method, obscuring that of `A`'s.

But embedding isn't just for methods, it also provides access to an embedded type's fields.
As you see, because both `A` and `B` are defined in the same package, `B` can access `A`'s private `year` field as if it were defined in `B`.

So embedding is a powerful tool which allows Go's types to be open for extension.

```
package main

type Cat struct {
        Name string
}

func (c Cat) Legs() int { return 4 }

func (c Cat) PrintLegs() {
        fmt.Printf("I have %d legs\n", c.Legs())
}

type OctoCat struct {
        Cat
}

func (o OctoCat) Legs() int { return 5 }

func main() {
        var octo OctoCat
        fmt.Println(octo.Legs()) // 5
        octo.PrintLegs()         // I have 4 legs
}
```

In this example we have a `Cat` type, which can count its number of legs with the `Legs` method.

We embed this `Cat` type into a new type, an `OctoCat`, and declare that octocats have five legs.

Though `OctoCat` defines it's own `Legs` method which returns 5, when the `PrintLegs` method is invoked, it returns 4.

[click]

This is because `PrintLegs` is defined on the `Cat` type, it takes a Cat as its receiver, and so it dispatches to `Cat`'s `Legs` method.

`Cat` has no knowledge of the type it has been embedded inside of, so its method set cannot be altered by embedding it.

Thus, we can say that Go's types while being open for extension, are closed for modification.

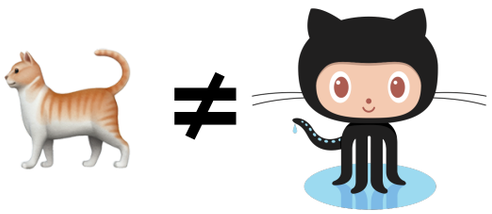
```
func (c Cat) PrintLegs() {
        fmt.Printf("I have %d legs\n", c.Legs())
}
```

In truth, methods in Go are little more than syntactic sugar around a function with a predeclared formal parameter, the receiver.

```
func PrintLegs(c Cat) {
        fmt.Printf("I have %d legs\n", c.Legs())
}
```

The receiver is exactly what you pass into it, the first parameter of the function.

And because Go does not support function overloading, `OctoCat`'s are not substitutable for regular `Cat`’s.



And this brings us to the next principle.

Liskov Substitution Principle

Coined by Barbara Liskov, the Liskov substitution principle states, roughly, that two types are substitutable if they exhibit behaviour such that the caller is unable to tell the difference.

In a class based language, Liskov's substitution principle is commonly interpreted as a specification for an abstract base class with various concrete subtypes.

But Go does not have classes, or inheritance, so substitution cannot be implemented in terms of an abstract class hierarchy.

Instead, substitution is the purview of Go's interfaces.

In Go, types are not required to nominate that they implement a particular interface, instead any type implements an interface simply provided it has methods whose signature matches the interface declaration.

We say that in Go, interfaces are satisfied implicitly, rather than explicitly, and this has a profound impact on how they are used within the language.

Small Interfaces
Simple Implementations
Common Behaviour

Well designed interfaces are more likely to be small interfaces; the prevailing idiom is an interface contains only a single method.

It follows logically that small interfaces lead to simple implementations, because it is hard to do otherwise.

Which leads to packages comprised of simple implementations connected by common _behaviour_.

```
type Reader interface {
        // Read reads up to len(buf) bytes into buf.
        Read(buf []byte) (n int, err error)
}
```

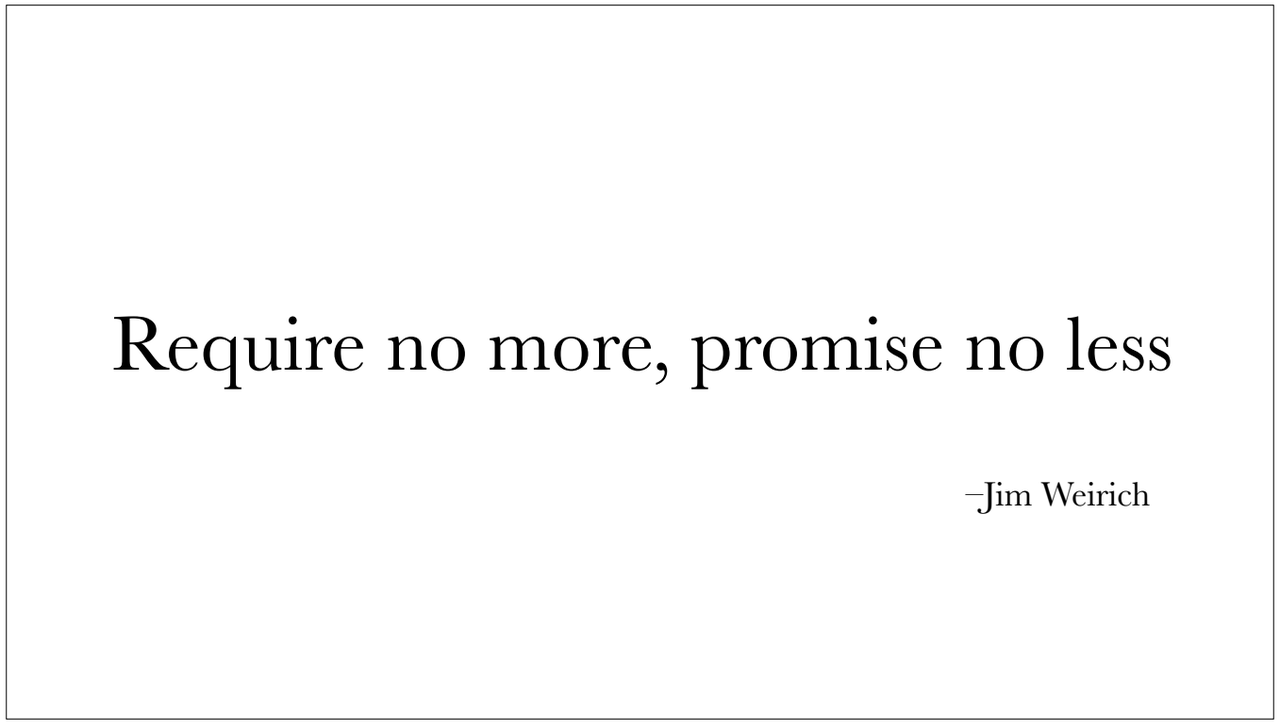
So that brings me to `io.Reader`, easily my favourite Go interface.

The io.Reader interface is very simple;
`Read` reads data into the supplied buffer, and returns to the caller the number of bytes that were read, and any error encountered during read.

Seems simple but it's very powerful.

Because Reader's deal with anything that can be expressed as a stream of bytes, we can construct readers over just about anything; a constant string, a byte array, standard in, a network stream, a gzip'd tar file, the standard out of a command being executed remotely via ssh.

And all of these implementations are substituable for one another because they fullfil the same simple contract.



So the Liskov substitution principle, applied to Go, could be summarised by this lovely aphorism from the late Jim Weirich.

“Require no more, promise no less.”

And this is a great segue into the next SOLID principle.

Interface Segregation Principle

The fourth principle is the interface segregation principle.

> Clients should not be forced to depend on methods they do not use.
>
> –Robert C. Martin

which reads

“clients should not be forced to depend on methods they do not use”

In Go, the application of the interface segregation principle can refer to a process of isolating the behaviour required for a function to do its job.

As a concrete example, say I've been given a task to write a function that persists a Document structure to disk.

```
type Document struct {
        // mo' state
}

// Save writes the contents of the Document
// to the file f.
func (d *Document) Save(f *os.File) error
```

I could specify this method, 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, the signature of this function would have to change, impacting all its callers.

`Save` is also unpleasant to test, because it operates directly with files on disk. So, to verify its operation, the test would have to read the contents of the file after being written.

And I 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 the `Save` function could describe only the parts of `*os.File` that were relevant.

What can we do ?

```
// Save writes the contents of d to the supplied
// ReadWriterCloser.
func (d *Document) Save(rwc io.ReadWriteCloser) error
```

Using `io.ReadWriteCloser` we can apply the interface segregation principle to redefine `Save` to take an interface that describes more general file shaped things.

With this change, any type that implements the `io.ReadWriteCloser` 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.

And 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.ReadWriteCloser` 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.

```
// Save writes the contents of d to the supplied
// WriteCloser.
func (d *Document) Save(wc io.WriteCloser) error
```

So we can narrow the specification for the interface we pass to Save to just writing and closing.

Secondly, by providing `Save` with a mechanism to close its stream, which we inherited in this desire to make it still look like a file, 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 LSP, as `NopCloser` doesn't actually close anything.

```
// Save writes the contents of d to the supplied
// Writer.
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.

As a side effect it is clear that the name of the method is no longer accurate. A better name may be

```
// WriteTo writes the contents of d to the supplied
// Writer.
func (d *Document) WriteTo(w io.Writer) error
```

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

A great rule of thumb for Go is **accept interfaces**, **return structs**.

–Jack Lindamood

And stepping back a few paces, this quote is an interesting meme that has percolated over the last year.

This tweet sized version lacks nuance, but I think it represents one of the first piece of defensible Go design lore.

Dependency Inversion Principle

The final SOLID principle is the dependency inversion principle.

Which states

> High-level modules should not depend on low-level modules. Both should depend on abstractions.
>
> Abstractions should not depend on details. Details should depend on abstractions.
>
> –Robert C. Martin

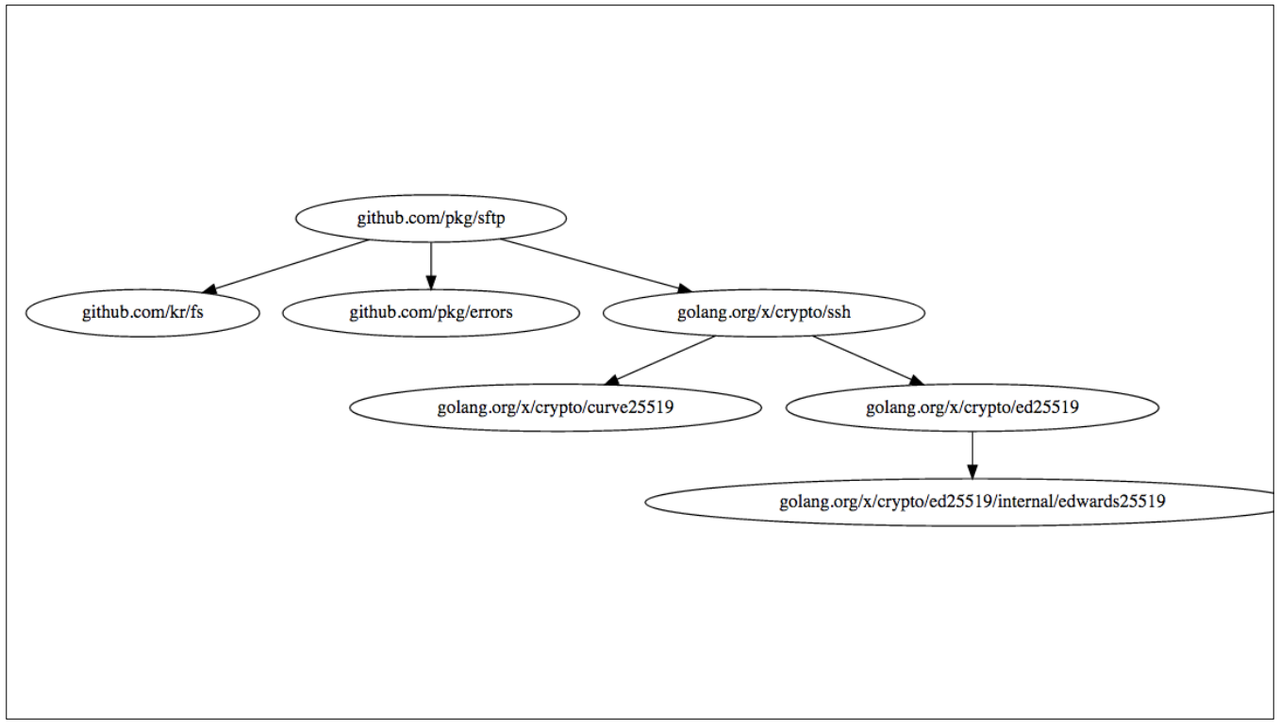
What does dependency inversion mean in practice for Go programmers?

If you've applied all the principles we've talked about up to this point then your code should already be factored into discrete packages, each with a single well defined responsibility or purpose.

Your code should describe its dependencies in terms of interfaces, and those interfaces should be factored to describe only the behaviour those functions require.

In other words, there shouldn't be much left to do at this point.

So what I think Martin is talking about here, certainly the context of Go, is the structure of your import graph.



In Go, your import graph must be acyclic.
A failure to respect this acyclic requirement is grounds for a compilation failure, but more gravely represents a serious error in design.

All things being equal the the import graph of a well design Go program should be a wide, and relatively flat, rather than tall and narrow.

If you have a package whose functions cannot operate without enlisting the aid of another package, that is perhaps a sign that code is not well factored along package boundaries.

The dependency inversion principle encourages you to push the responsibility for the specifics, as high as possible up the import graph—i’m suggesting package main — leaving the lower level code to deal with abstractions -- the interfaces.

SOLID Go Design

So to recap, when applied to Go, each of the SOLID principles are powerful statements about design, but taken together they have a central theme.

Single Responsibility Principle

Structure your functions and types into packages that exhibit natural cohesion.

SRP encourages you to structure the functions, types, and methods into packages that exhibit natural cohesion; the types belong together, the functions serve a singular purpose.

Open / Closed Principle

Compose types with embedding rather than extend them through inheritance.

OCP encourages you to compose simple types into more complex ones with embedding.

Liskov Substitution Principle

Express the dependencies between your packages in terms of interfaces, not concrete types.

LSP encourages you to express the dependencies between your packages in terms of interfaces, not concrete types. By defining small interfaces, we can be more confident that implementations will faithfully satisfy their contract.

Interface Substitution Principle

Define functions and methods that depend only on the behaviour that they need.

ISP takes that idea further and encourages you to define functions and methods that depend only on the behaviour that they need.

If your function only requires a parameter of an interface type with a single method, then it is more likely that this function has only one responsibility.

Dependency Inversion Principle

Refactor dependencies from compile time to run time.

DIP encourages you move the knowledge of the things your package depends on from compile time--in Go we see this with a reduction in the number of `import` statements used by a particular package--to run time—when those relationships are constructed in your main package.

Interfaces let you apply the
SOLID principles to Go programs

If you were to summarise this talk it would probably be; interfaces let you apply the SOLID principles to Go programs.

Design is the art of arranging code that needs to work **today**, and to be easy to change **forever**.

–Sandi Metz

Because interfaces let Go programmers describe what their package provides--not how it does it.

This is all just another way of saying "decoupling", which is indeed the goal, because software that is loosely coupled is software that is easier to change.

If software cannot be maintained, it will be rewritten

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

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

Thank you very much.