The Principles of Versioning in Go

Go & Versioning, Part 11

Russ Cox

December 3, 2019

research.swtch.com/vgo-principles

This blog post is about how we added package versioning to Go, in the form of Go modules, and the reasons we made the choices we did. It is adapted and updated from a talk I gave at GopherCon Singapore in 2018.

Why Versions?

To start, let's make sure we're all on the same page, by taking a look at the ways the GOPATH-based `go get` breaks.

Suppose we have a fresh Go installation and we want to write a program that imports D. We run `go get D`. Remember that we are using the original GOPATH-based `go get`, not Go modules.

```
$ go get D
```

Requirements
D 1.0 none

That looks up and downloads the latest version of D, which right now is D 1.0. It builds. We're happy.

Now suppose a few months later we need C. We run `go get C`. That looks up and downloads the latest version of C, which is C 1.8.

```
$ go get C
```

Requirements
C 1.8 D ≥ 1.4
D 1.0 none

broken!

C imports D, but `go get` finds that it has already downloaded a copy of D, so it reuses that copy. Unfortunately, that copy is still D 1.0. The latest copy of C was written using D 1.4, which contains a feature or maybe a bug fix that C needs and which was missing from D 1.0. So C is broken, because the dependency D is too old.

Since the build failed, we try again, with `go get -u C`.

```
$ go get -u C
```

Requirements
C 1.8 D ≥ 1.4
D 1.6 none

(C 1.8 was tested with D 1.4 and has an unexpected incompatibility with D 1.6.)

broken!

Unfortunately, an hour ago D's author published D 1.6. Because `go get -u` uses the latest version of every dependency, including D, it turns out that C is still broken. C's author used D 1.4, which worked fine, but D 1.6 has introduced a bug that keeps C from working properly. Before, C was broken because D was too old. Now, C is broken because D is too new.

Those are the two ways that `go get` fails when using GOPATH. Sometimes it

uses dependencies that are too old. Other times it uses dependencies that are too new. What we really want in this case is the version of D that C's author used and tested against. But GOPATH-based `go get` can't do that, because it has no awareness of package versions at all.

Go programmers started asking for better handling of package versions as soon as we published `goinstall`, the original name for `go get`. Various tools were written over many years, separate from the Go distribution, to help make installing specific versions easier. But because those tools did not agree on a single approach, they didn't work as a base for creating other version-aware tools, such as a version-aware godoc or a version-aware vulnerability checker.

We needed to add the concept of package versions to Go for many reasons. The most pressing reason was to help `go get` stop using code that's too old or too new, but having an agreed-upon meaning of versions in the vocabulary of Go developers and tools enables the entire Go ecosystem to become version-aware. The Go module mirror and checksum database, which safely speed up Go package downloads, and the new version-aware Go package discovery site are both made possible by an ecosystem-wide understanding of what a version is.

Versions for Software Engineering

Over the past two years, we have added support for package versions to Go itself, in the form of Go modules, built into the `go` command. Go modules introduce a new import path syntax called semantic import versioning, along with a new algorithm for selecting which versions to use, called minimal version selection.

You might wonder: Why not do what other languages do? Java has Maven, Node has NPM, Ruby has Bundler, Rust has Cargo. How is this not a solved problem?

You might also wonder: We introduced a new, experimental Go tool called Dep in early 2018 that implemented the general approach pioneered by Bundler and Cargo. Why did Go modules not reuse Dep's design?

The answer is that we learned from Dep that the general Bundler/Cargo/Dep approach includes some decisions that make software engineering more complex and more challenging. Thanks to learning about the problems were in Dep's design, the Go modules design made different decisions, to make software engineering simpler and easier instead.

But what is software engineering? How is software engineering different from programming? I like the following definition:

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

Programming means getting a program working. You have a problem to solve, you write some Go code, you run it, you get your answer, you're done. That's programming, and that's difficult enough by itself.

But what if that code has to keep working, day after day? What if five other programmers need to work on the code too? What if the code must adapt gracefully as requirements change? Then you start to think about version control systems, to track how the code changes over time and to coordinate with the other programmers. You add unit tests, to make sure bugs you fix are not reintroduced over time, not by you six months from now, and not by that new team member who's unfamiliar with the code. You think about modularity and design patterns, to divide the program into parts that team members can work on mostly independently. You use tools to help you find bugs earlier. You look for ways to make programs as clear as possible, so that bugs are less likely. You make sure

that small changes can be tested quickly, even in large programs. You're doing all of this because your programming has turned into software engineering.

(This definition and explanation of software engineering is my riff on an original theme by my Google colleague Titus Winters, whose preferred phrasing is "software engineering is programming integrated over time." It's worth seven minutes of your time to see his presentation of this idea at CppCon 2017, from 8:17 to 15:00 in the video.)

Nearly all of Go's distinctive design decisions were motivated by concerns about software engineering. For example, most people think that we format Go code with `gofmt` to make code look nicer or to end debates among team members about program layout. And to some degree we do. But the more important reason for `gofmt` is that if an algorithm defines how Go source code is formatted, then programs, like `goimports` or `gorename` or `go fix`, can edit the source code more easily. This helps you maintain code over time.

As another example, Go import paths are URLs. If code imported `"uuid"`, you'd have to ask which `uuid` package. Searching for `uuid` on *pkg.go.dev* turns up dozens of packages with that name. If instead the code imports `"github.com/google/uuid"`, now it's clear which package we mean. Using URLs avoids ambiguity and also reuses an existing mechanism for giving out names, making it simpler and easier to coordinate with other programmers. Continuing the example, Go import paths are written in Go source files, not in a separate build configuration file. This makes Go source files self-contained, which makes it easier to understand, modify, and copy them. These decisions were all made toward the goal of simplifying software engineering.

Principles

There are three broad principles behind the changes from Dep's design to Go modules, all motivated by wanting to simplify software engineering. These are the principles of compatibility, repeatability, and cooperation. The rest of this post explains each principle, shows how it led us to make a different decision for Go modules than in Dep, and then responds, as fairly as I can, to objections against making that change.

Principle #1: Compatibility

The meaning of a name in a program should not change over time.

The first principle is compatibility. Compatibility—or, if you prefer, stability—is the idea that, in a program, the meaning of a name should not change over time. If a name meant one thing last year, it should mean the same thing this year and next year.

For example, programmers are sometimes confused by a detail of `strings.Split`. We all expect that splitting "`hello world`" produces two strings "`hello`" and "`world`." But if the input has leading, trailing, or repeated spaces, the result contains empty strings too.

```
Example: strings.Split(x, " ")

"hello world"  => {"hello", "world"}
"hello  world" => {"hello", "", "world"}
" hello world" => {"", "hello", "world"}
"hello world " => {"hello", "world", ""}
```

Suppose we decide that it would be better overall to change the behavior of `strings.Split` to omit those empty strings. Can we do that?

No.

We've given `strings.Split` a specific meaning. The documentation and the implementation agree on that meaning. Programs depend on that meaning. Changing the meaning would break those programs. It would break the principle of compatibility.

We *can* implement the new meaning; we just need to give a new name too. In fact, years ago, to solve this exact problem, we introduced `strings.Fields`, which is tailored to space-separated fields and never returns empty strings.

```
Example: strings.Fields(x)

"hello world"  => {"hello", "world"}
"hello  world" => {"hello", "world"}
" hello world" => {"hello", "world"}
"hello world " => {"hello", "world"}
```

We didn't redefine `strings.Split`, because we were concerned about compatibility.

Following the principle of compatibility simplifies software engineering, because it lets you ignore time when trying to understand programming. People don't have to think, "well this package was written in 2015, back when `strings.Split` returned empty strings, but this other package was written last week, so it expects `strings.Split` to leave them out." And not just people. Tools don't have to worry about time either. For example, a refactoring tool can always move a `strings.Split` call from one package to another without worrying that it will change its meaning.

In fact, the most important feature of Go 1 was not a language change or a new library feature. It was the declaration of compatibility:

> It is intended that programs written to the Go 1 specification will continue to compile and run correctly, unchanged, over the lifetime of that specification. Go programs that work today should continue to work even as future "point" releases of Go 1 arise (Go 1.1, Go 1.2, etc.).
>
> — *golang.org/doc/go1compat*

We committed that we would stop changing the meaning of names in the standard library, so that programs working with Go 1.1 could be expected to continue working in Go 1.2, and so on. That ongoing commitment makes it easy for users to write code and keep it working even as they upgrade to newer Go versions to get faster implementations and new features.

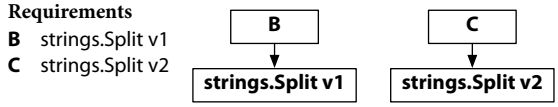
What does compatibility have to do with versioning? It's important to think about compatibility because the most popular approach to versioning today—semantic versioning—instead encourages *incompatibility*. That is, semantic versioning has the unfortunate effect of making incompatible changes seem easy.

Every semantic version takes the form vMAJOR.MINOR.PATCH. If two versions have the same major number, the later (if you like, greater) version is expected to be backwards compatible with the earlier (lesser) one. But if two versions have different major numbers, they have no expected compatibility relationship.

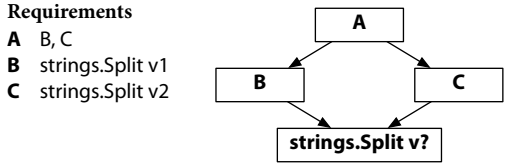
Semantic versioning seems to suggest, "It's okay to make incompatible changes to your packages. Tell your users about them by incrementing the major version number. Everything will be fine." But this is an empty promise. Incrementing the major version number isn't enough. Everything is not fine. If `strings.Split` has one meaning today and a different meaning tomorrow, simply reading your code is now software engineering, not programming, because you need to think about time.

It gets worse.

Suppose B is written to expect `strings.Split` v1, while C is written to expect `strings.Split` v2. That's fine if you build each by itself.



But what happens when your package A imports both B and C? If `strings.Split` has to have just one meaning, there's no way to build a working program.



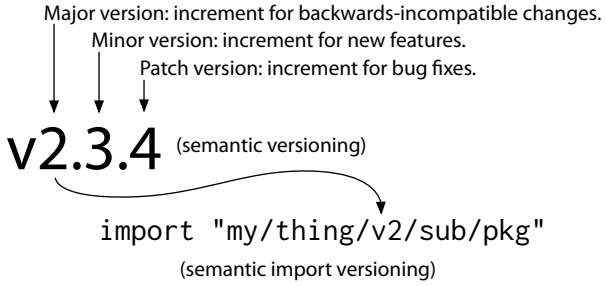
For the Go modules design, we realized that the principle of compatibility is absolutely essential to simplifying software engineering and must be supported, encouraged, and followed. The Go FAQ has encouraged compatibility since Go 1.2 in November 2013:

> Packages intended for public use should try to maintain backwards compatibility as they evolve. The Go 1 compatibility guidelines are a good reference here: don't remove exported names, encourage tagged composite literals, and so on. If different functionality is required, add a new name instead of changing an old one. If a complete break is required, create a new package with a new import path.

For Go modules, we gave this old advice a new name, the *import compatibility rule*:

> *If an old package and a new package have the same import path,*
> *the new package must be backwards compatible with the old package.*

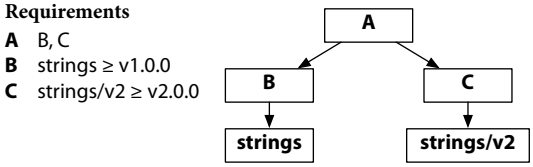
But then what do we do about semantic versioning? If we still want to use semantic versioning, as many users expect, then the import compatibility rule requires that different semantic major versions, which by definition have no compatibility relationship, must use different import paths. The way to do that in Go modules is to put the major version in the import path. We call this *semantic import versioning*.



In this example, `my/thing/v2` identifies semantic version 2 of a particular module. Version 1 was just `my/thing`, with no explicit version in the module path. But when you introduce major version 2 or larger, you have to add the version after the module name, to distinguish from version 1 and other major versions, so version 2 is `my/thing/v2`, version 3 is `my/thing/v3`, and so on.

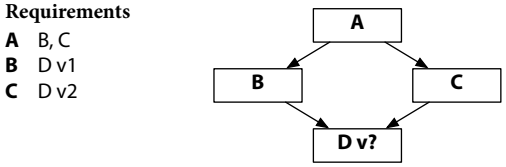
If the `strings` package were its own module, and if for some reason we really needed to redefine `Split` instead of adding a new function `Fields`, then we could create `strings` (major version 1) and `strings/v2` (major version 2),

with different `Split` functions. Then the unbuildable program from before can be built: B says `import "strings"` while C says `import "strings/v2"`. Those are different packages, so it's okay to build both into the program. And now B and C can each have the `Split` function they expect.

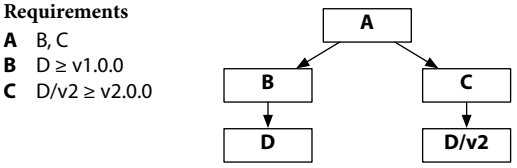


Because `strings` and `strings/v2` have different import paths, people and tools automatically understand that they name different packages, just as people already understand that `crypto/rand` and `math/rand` name different packages. No one needs to learn a new disambiguation rule.

Let's return to the unbuildable program, not using semantic import versioning. If we replace `strings` in this example with an arbitrary package D, then we have a classic "diamond dependency problem." Both B and C build fine by themselves, but with different, conflicting requirements for D. If we try to use both in a build of A, then there's no single choice of D that works.



Semantic import versioning cuts through diamond dependencies. There's no such thing as conflicting requirements for D. D version 1.3 must be backwards compatible with D version 1.2, and D version 2.0 has a different import path, D/v2.



A program using both major versions keeps them as separate as any two package with different import paths and builds fine.

Objection: Aesthetics

The most common objection to semantic import versioning is that people don't like seeing the major versions in the import paths. In short, they're ugly. Of course, what this really means is only that people are not used to seeing the major version in import paths.

I can think of two examples of major aesthetic shifts in Go code that seemed ugly at the time but were adopted because they simplified software engineering and now look completely natural.

The first example is export syntax. Back in early 2009, Go used an `export` keyword to mark a function as exported. We knew we needed something more lightweight to mark individual struct fields, and we were casting about for ideas, considering things like "leading underscore means unexported" or "leading plus in declaration means export." Eventually we hit on the "upper-case for export" idea. Using an upper-case letter as the export signal looked strange to us, but that was the only drawback we could find. Otherwise, the idea was sound, it satisfied our goals, and it was more appealing than the other choices we'd been considering. So we adopted it. I remember thinking that changing `fmt.printf`

to `fmt.Printf` in my code was ugly, or at least jarring: to me, `fmt.Printf` didn't look like Go, at least not the Go I had been writing. But I had no good argument against it, so I went along with (and implemented) the change. After a few weeks, I got used to it, and now it is `fmt.printf` that doesn't look like Go to me. What's more, I came to appreciate the precision about what is and isn't exported when reading code. When I go back to C++ or Java code now and I see a call like `x.dangerous()` I miss being able to tell at a glance whether the `dangerous` method is a public method that anyone can call.

The second example is import paths, which I mentioned briefly earlier. In the early days of Go, before `goinstall` and `go get`, import paths were not full URLs. A developer had to manually download and install a package named `uuid` and then would write `import "uuid"`. Changing to URLs for import paths (`import "github.com/google/uuid"`) eliminated this ambiguity, and the added precision made `go get` possible. People did complain at first, but now the longer paths are second nature to us. We rely on and appreciate their precision, because it makes our software engineering work simpler.

Both these changes—upper-case for export and full URLs for import paths—were motivated by good software engineering arguments to which the only real objection was visual aesthetics. Over time we came to appreciate the benefits, and our aesthetic judgements adapted. I expect the same to happen with major versions in import paths. We'll get used to them, and we'll come to value the precision and simplicity they bring.

Objection: Updating Import Paths

Another common objection is that upgrading from (say) v2 of a module to v3 of the same module requires changing all the import paths referring to that module, even if the client code doesn't need any other changes.

It's true that the upgrade requires rewriting import paths, but it's also easy to write a tool to do a global search and replace. We intend to make it possible to handle such upgrades with `go fix`, although we haven't implemented that yet.

Both the previous objection and this one implicitly suggest keeping the major version information only in a separate version metadata file. If we do that, then an import path won't be precise enough to identify semantics, like back when `import "uuid"` might have meant any one of dozens of different packages. All programmers and tools will have to look in the metadata file to answer the question: which major version is this? Which `strings.Split` am I calling? What happens when I copy a file from one module to another and forget to check the metadata file? If instead we keep import paths semantically precise, then programmers and tools don't need to be taught a new way to keep different major versions of a package separate.

Another benefit of having the major version in the import path is that when you do update from v2 to v3 of a package, you can update your program gradually, in stages, maybe one package at a time, and it's always clear which code has been converted and which has not.

Objection: Multiple Major Versions in a Build

Another common objection is that having D v1 and D v2 in the same build should be disallowed entirely. That way, D's author won't have to think about the complexities that arise from that situation. For example, maybe package D defines a command line flag or registers an HTTP handler, so that building both D v1 and D v2 into a single program would fail without explicit coordination between those versions.

Dep enforces exactly this restriction, and some people say it is simpler. But this is simplicity only for D's author. It's not simplicity for D's users, and normal-

ly users outnumber authors. If D v1 and D v2 cannot coexist in a single build, then diamond dependencies are back. You can't convert a large program from D v1 to D v2 gradually, the way I just explained. In internet-scale projects, this will fragment the Go package ecosystem into incompatible groups of packages: those that use D v1 and those that use D v2. For a detailed example, see my 2018 blog post, "Semantic Import Versioning."

Dep was forced to disallow multiple major versions in a build because the Go build system requires each import path to name a unique package (and Dep did not consider semantic import versioning). In contrast, Cargo and other systems do allow multiple major versions in a build. As I understand it, the reason these systems allow multiple versions is the same reason that Go modules does: not allowing them makes it too hard to work on large programs.

Objection: Too Hard to Experiment

A final objection is that versions in import paths are unnecessary overhead when you're just starting to design a package, you have no users, and you're making frequent backwards-incompatible changes. That's absolutely true.

Semantic versioning makes an exception for exactly that situation. In major version 0, there are no compatibility expectations at all, so that you can iterate quickly when you're first starting out and not worry about compatibility. For example, v0.3.4 doesn't need to be backwards compatible with anything else: not v0.3.3, not v0.0.1, not v1.0.0.

Semantic import versioning makes a similar exception: major version 0 is not mentioned in import paths.

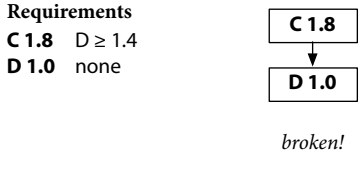
In both cases, the rationale is that time has not entered the picture. You're not doing software engineering yet. You're just programming. Of course, this means that if you use v0 versions of other people's packages, then you are accepting that new versions of those packages might include breaking API changes without a corresponding import path change, and you take on the responsibility to update your code when that happens.

Principle #2: Repeatability

> *The result of a build of a given version of a package should not change over time.*

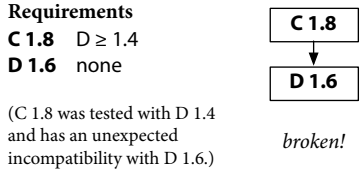
The second principle is repeatability for program builds. By repeatability I mean that when you are building a specific version of a package, the build should decide which dependency versions to use in a way that's repeatable, that doesn't change over time. My build today should match your build of my code tomorrow and any other programmer's build next year. Most package management systems don't make that guarantee.

We saw earlier how GOPATH-based `go get` doesn't provide repeatability. First `go get` used a version of D that was too old:



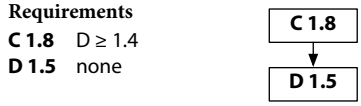
broken!

Then `go get -u` used a version of D that was too new:

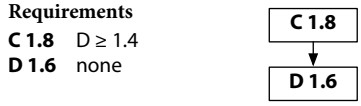


You might think, "of course `go get` makes this mistake: it doesn't know anything about versions at all." But most other systems make the same mistake. I'm going to use Dep as my example here, but at least Bundler and Cargo work the same way.

Dep asks every package to include a metadata file called a manifest, which lists requirements for dependency versions. When Dep downloads C, it reads C's manifest and learns that C needs D 1.4 or later. Then Dep downloads the newest version of D satisfying that constraint. Yesterday, that meant D 1.5:



Today, that means D 1.6:



The decision is time-dependent. It changes from day to day. The build is not repeatable.

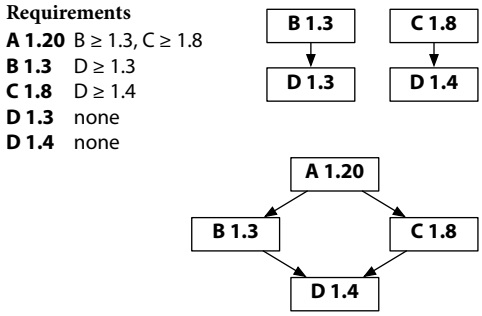
The developers of Dep (and Bundler and Cargo and ...) understood the importance of repeatability, so they introduced a second metadata file called a lock file. If C is a whole program, what Go calls `package main`, then the lock file lists the exact version to use for every dependency of C, and Dep lets the lock file override the decisions it would normally make. Locking in those decisions ensures that they stop changing over time and makes the build repeatable.

But lock files only apply to whole programs, to `package main`. What if C is a library, being built as part of a larger program? Then a lock file meant for building only C might not satisfy the additional constraints in the larger program. So Dep and the others must ignore lock files associated with libraries and fall back to the default time-based decisions. When you add C 1.8 to a larger build, the exact packages you get depends on what day it is.

In summary, Dep starts with a time-based decision about which version of D to use. Then it adds a lock file, to override that time-based decision, for repeatability, but that lock file can only be applied to whole programs.

In Go modules, the `go` command instead makes its decision about which version of D to use in a way that does not change over time. Then builds are repeatable all the time, without the added complexity of a lock file override, and this repeatability applies to libraries, not just whole programs.

The algorithm used for Go modules is very simple, despite the imposing name "minimal version selection." It works like this. Each package specifies a minimum version of each dependency. For example, suppose B 1.3 requests D 1.3 or later, and C 1.8 requests D 1.4 or later. In Go modules, the `go` command prefers to use those exact versions, not the latest versions. If we're building B by itself, we'll use D 1.3. If we're building C by itself, we'll use D 1.4. The builds of these libraries are repeatable.



Also shown in the figure, if different parts of a build request different minimum versions, the go command uses the latest requested version. The build of A sees requests for D 1.3 and D 1.4, and 1.4 is later than 1.3, so the build chooses D 1.4. That decision does not depend on whether D 1.5 and D 1.6 exist, so it does not change over time.

I call this minimal version selection for two reasons. First, for each package it selects the minimum version satisfying the requests (equivalently, the maximum of the requests). And second, it seems to be just about the simplest approach that could possibly work.

Minimal version selection provides repeatability, for whole programs and for libraries, always, without any lock files. It removes time from consideration. Every chosen version is always one of the versions mentioned explicitly by some package already chosen for the build.

Objection: Using the Latest Version is a Feature

The usual first objection to prioritizing repeatability is to claim that preferring the latest version of a dependency is a feature, not a bug. The claim is that programmers either don't want to or are too lazy to update their dependencies regularly, so tools like Dep should use the latest dependencies automatically. The argument is that the benefits of having the latest versions outweigh the loss of repeatability.

But this argument doesn't hold up to scrutiny. Tools like Dep provide lock files, which then require programmers to update dependencies themselves, exactly because repeatable builds are more important than using the latest version. When you deploy a 1-line bug fix, you want to be sure that your bug fix is the only change, that you're not also picking up different, newer versions of your dependencies.

You want to delay upgrades until you ask for them, so that you can be ready to run all your unit tests, all your integration tests, and maybe even production canaries, before you start using those upgraded dependencies in production. Everyone agrees about this. Lock files exist because everyone agrees about this: repeatability is more important than automatic upgrades.

Objection: Using the Latest Version is a Feature When Building a Library

The more nuanced argument you could make against minimal version selection would be to admit that repeatability matters for whole program builds, but then argue that, for libraries, the balance is different, and having the latest dependencies is more important than a repeatable build.

I disagree. As programming increasingly means connecting large libraries together, and those large libraries are increasingly organized as collections of smaller libraries, all the reasons to prefer repeatability of whole-program builds become just as important for library builds.

The extreme limit of this trend is the recent move in cloud computing to "serverless" hosting, like Amazon Lambda, Google Cloud Functions, or Microsoft Azure Functions. The code we upload to those systems is a library, not a whole program. We certainly want the production builds on those servers to use the same versions of dependencies as on our development machines.

Of course, no matter what, it's important to make it easy for programmers to update their dependencies regularly. We also need tools to report which versions of a package are in a given build or a given binary, including reporting when updates are available and when there are known security problems in the versions being used.

Principle #3: Cooperation

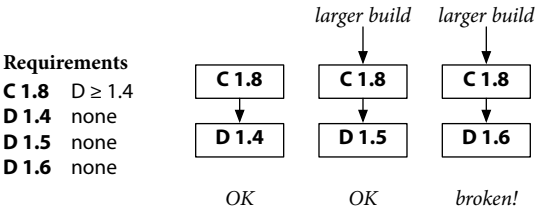
> *To maintain the Go package ecosystem, we must all work together.*
> *Tools cannot work around a lack of cooperation.*

The third principle is cooperation. We often talk about "the Go community" and "the Go open source ecosystem." The words community and ecosystem emphasize that all our work is interconnected, that we're building on—depending on—each other's contributions. The goal is one unified system that works as a coherent whole. The opposite, what we want to avoid, is an ecosystem that is fragmented, split into groups of packages that can't work with each other.

The principle of cooperation recognizes that the only way to keep the ecosystem healthy and thriving is for us all to work together. If we don't, then no matter how technically sophisticated our tools are, the Go open source ecosystem is guaranteed to fragment and eventually fail. By implication, then, it's okay if fixing incompatibilities requires cooperation. We can't avoid cooperation anyway.

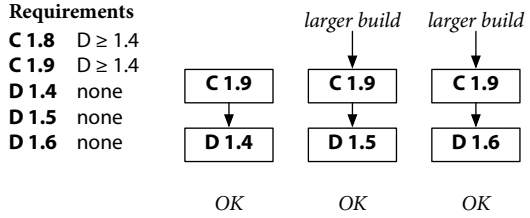
For example, once again we have C 1.8, which requires D 1.4 or later. Thanks to repeatability, a build of C 1.8 by itself will use D 1.4. If we build C as part of a larger build that needs D 1.5, that's okay too.

Then D 1.6 is released, and some larger build, maybe continuous integration testing, discovers that C 1.8 does not work with D 1.6.

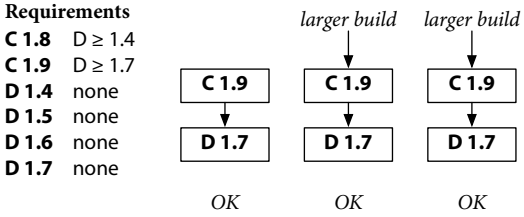


No matter what, the solution is for C's author and D's author to cooperate and release a fix. The exact fix depends on what exactly went wrong.

Maybe C depends on buggy behavior fixed in D 1.6, or maybe C depends on unspecified behavior changed in D 1.6. Then the solution is for C's author to release a new C version 1.9, cooperating with the evolution of D.



Or maybe D 1.6 simply has a bug. Then the solution is for D's author to release a fixed D 1.7, cooperating by respecting the principle of compatibility, at which point C's author can release C version 1.9 that specifies that it requires D 1.7.



Take a minute to look at what just happened. The latest C and the latest D didn't work together. That introduced a small fracture in the Go package ecosystem. C's author or D's author worked to fix the bug, cooperating with each other and the rest of the ecosystem to repair the fracture. This cooperation is essential to keeping the ecosystem healthy. There is no adequate technical substitute.

The repeatable builds in Go modules mean that a buggy D 1.6 won't be picked up until users explicitly ask to upgrade. That creates time for C's author and D's author to cooperate on a real solution. The Go modules system makes no other attempt to work around these temporary incompatibilities.

Objection: Use Declared Incompatibilities and SAT Solvers

The most common objection to this approach of depending on cooperation is that it is unreasonable to expect developers to cooperate. Developers need some way to fix problems alone. the argument goes: they can only truly depend on themselves, not others. The solution offered by package managers like Bundler, Cargo, and Dep is to allow developers to declare incompatibilities between their packages and others and then employ a SAT solver to find a package combination not ruled out by the constraints.

This argument breaks down for a few reasons.

First, the algorithm used by Go modules to select versions already gives the developer of a particular module complete control over which versions are selected for that module, more control in fact than SAT constraints. The developer can force the use of any specific version of any dependency, saying "use this exact version no matter what anyone else says." But that power is limited to the build of that specific module, to avoid giving other developers the same control over your builds.

Second, repeatability of library builds in Go modules means that the release of a new, incompatible version of a dependency has no immediate effect on builds, as we saw in the previous section. The breakage only surfaces when someone takes some step to add that version to their own build, at which point they can step back again.

Third, if version selection is phrased as a problem for a SAT solver, there are often many possible satisfying selections: the SAT solver must choose between them, and there is no clear criteria for doing so. As we saw earlier, SAT-based package managers choose between multiple valid possible selections by preferring newer versions. In the case where using the newest version of everything satisfies the constraints, that's the clear "most preferred" answer. But what if the two possible selections are "latest of B, older C" and "older B, latest of C"? Which should be preferred? How can the developer predict the outcome? The resulting system is difficult to understand.

Fourth, the output of a SAT solver is only as good as its inputs: if any incompatibilities have been omitted, the SAT solver may well arrive at a combination that is still broken, just not declared as such. Incompatibility information is like-

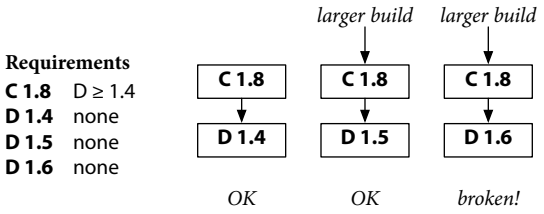
ly to be particularly incomplete for combinations involving dependencies with a significant age difference that may well never have been put together before. Indeed, an analysis of Rust's Cargo ecosystem in 2018 found that Cargo's preference for the latest version was masking many missing constraints in Cargo manifests. If the latest version does not work, exploring old versions seems as likely to produce a combination that is "not yet known to be broken" as it is to produce a working one.

Overall, once you step off the happy path of selecting the newest version of every dependency, SAT solver-based package managers are not more likely to choose a working configuration than Go modules is. If anything, SAT solvers may well be less likely to find a working configuration.

Example: Go Modules versus SAT Solving

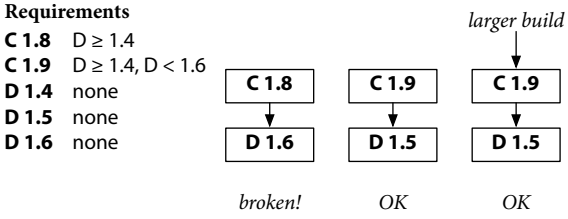
The counter-arguments given in the previous section are a bit abstract. Let's make them concrete by continuing the example we've been working with and looking at what happens when using a SAT solver, like in Dep. I'm using Dep for concreteness, because it is the immediate predecessor of Go modules, but the behaviors here are not specific to Dep and I don't mean to single it out. For the purposes of this example, Dep works the same way as many other package managers, and they all share the problems detailed here.

To set the stage, remember that C 1.8 works fine with D 1.4 and D 1.5, but the combination of C 1.8 and D 1.6 is broken.



That gets noticed, perhaps by continuous integration testing, and the question is what happens next.

When C's author finds out that C 1.8 doesn't work with D 1.6, Dep allows and encourages issuing a new version, C 1.9. C 1.9 documents that it needs D later than 1.4 but before 1.6. The idea is that documenting the incompatibility helps Dep avoid it in future builds.



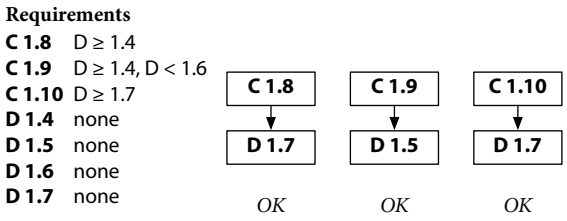
In Dep, avoiding the incompatibility is important—even urgent!—because the lack of repeatability in library builds means that as soon as D 1.6 is released, all future fresh builds of C will use D 1.6 and break. This is a build emergency: all of C's new users are broken. If D's author is unavailable, or C's author doesn't have time to fix the actual bug, the argument is that C's author must be able to take some step to protect users from the breakage. That step is to release C 1.9, documenting the incompatibility with D 1.6. That fixes new builds of C by preventing the use of D 1.6.

This emergency doesn't happen when using Go modules, because of minimal version selection and repeatable builds. Using Go modules, the release of D 1.6 does not affect C's users, because nothing is explicitly requesting D 1.6

yet. Users keep using the older versions of D they already use. There's no need to document the incompatibility, because nothing is breaking. There's time to cooperate on a real fix.

Looking at Dep's approach of documenting incompatibility again, releasing C 1.9 is not a great solution. For one thing, the premise was that D's author created a build emergency by releasing D 1.6 and then was unavailable to release a fix, so it was important to give C's author a way to fix things, by releasing C 1.9. But if D's author might be unavailable, what happens if C's author is unavailable too? Then the emergency caused by automatic upgrades continues and all of C's new users stay broken. Repeatable builds in Go modules avoid the emergency entirely.

Also, suppose that the bug is in D, and D's author issues a fixed D 1.7. The workaround C 1.9 requires D before 1.6, so it won't use the fixed D 1.7. C's author has to issue C 1.10 to allow use of D 1.7.

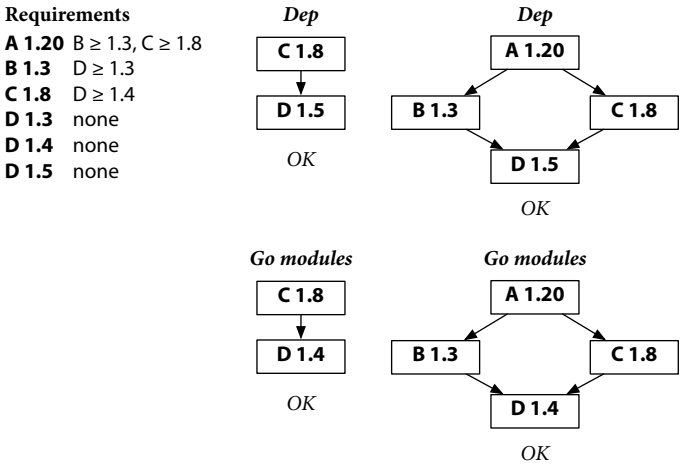


In contrast, if we're using Go modules, C's author doesn't have to issue C 1.9 and then also doesn't have to undo it by issuing C 1.10.

In this simple example, Go modules end up working more smoothly for users than Dep. They avoid the build breakage automatically, creating time for cooperation on the real fix. Ideally, C or D gets fixed before any of C's users even notice.

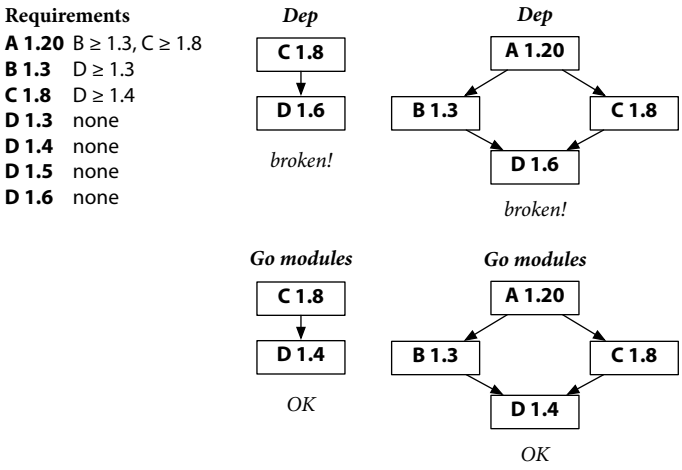
But what about more complex examples? Maybe Dep's approach of documenting incompatibilities is better in more complicated situations, or maybe it keeps things working even when the real fix takes a long time to arrive.

Let's take a look. To do that, let's rewind the clock a little, to before the buggy D 1.6 is released, and compare the decisions made by Dep and Go modules. This figure shows the documented requirements for all the relevant package versions, along with the way both Dep and Go modules would build the latest C and the latest A.



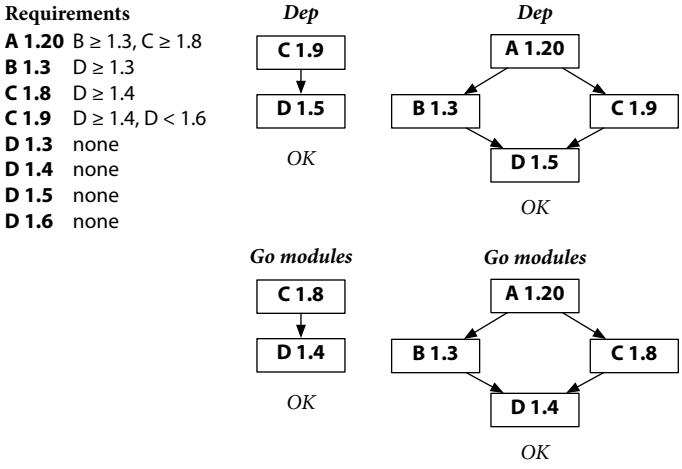
Dep is using D 1.5 while the Go module system is using D 1.4, but both tools have found working builds. Everyone is happy.

But now suppose the buggy D 1.6 is released.



Dep builds pick up D 1.6 automatically and break. Go modules builds keep using D 1.4 and keep working. This is the simple situation we were just looking at.

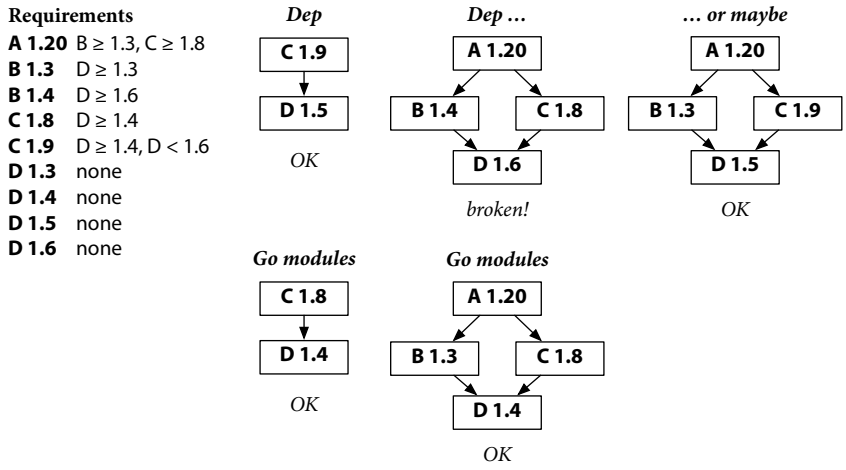
Before we move on, though, let's fix the Dep builds. We release C 1.9, which documents the incompatibility with D 1.6:



Now Dep builds pick up C 1.9 automatically, and builds start working again. Go modules can't document incompatibility in this way, but Go modules builds also aren't broken, so no fix is needed.

Now let's create a build complex enough to break Go modules. We can do this in two steps. First, we will release a new B that requires D 1.6. Second, we will release a new A that requires the new B, at which point A's build will use C with D 1.6 and break.

We start by releasing the new B 1.4 that requires D 1.6.



Go modules builds are unaffected so far, thanks to repeatability. But look! Dep builds of A pick up B 1.4 automatically and now they are broken again. What happened?

Dep prefers to build A using the latest B and the latest C, but that's not possible: the latest B wants D 1.6 and the latest C wants D before 1.6. But does Dep give up? No. It looks for alternate versions of B and C that do agree on an acceptable D.

In this case, Dep decided to keep the latest B, which means using D 1.6, which means *not* using C 1.9. Since Dep can't use the latest C, it tries older versions of C. C 1.8 looks good: it says it needs D 1.4 or later, and that allows D 1.6. So Dep uses C 1.8, and it breaks.

We know that C 1.8 and D 1.6 are incompatible, but Dep does not. Dep can't know it, because C 1.8 was released before D 1.6: C's author couldn't have predicted that D 1.6 would be a problem. And all package management systems agree that package contents must be immutable once they are published, which means there's no way for C's author to retroactively document that C 1.8 doesn't work with D 1.6. (And if there were some way to change C 1.8's requirements retroactively, that would violate repeatability.) Releasing C 1.9 with the updated requirement was the fix.

Because Dep prefers to use the latest C, most of the time it will use C 1.9 and know to avoid D 1.6. But if Dep can't use the latest of everything, it will start trying earlier versions of some things, including maybe C 1.8. And using C 1.8 makes it look like D 1.6 is okay—even though we know better—and the build breaks.

Or it might not break. Strictly speaking, Dep didn't have to make that decision. When Dep realized that it couldn't use both the latest B and the latest C, it had many options for how it might proceed. We assumed Dep kept the latest B. But if instead Dep kept the latest C, then it would need to use an older D and then an older B, producing a working build, as shown in the third column of the diagram.

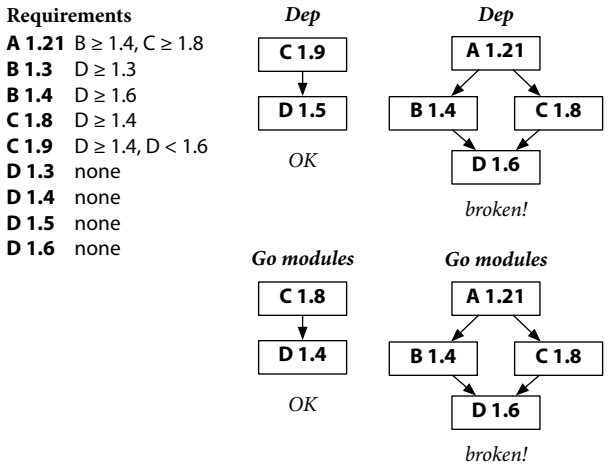
So maybe Dep's builds are broken or maybe not, depending on the arbitrary decisions it makes in its SAT-solver-based version selection. (Last I checked, given a choice between a newer version of one package versus another, Dep prioritizes the one with the alphabetically earlier import path, at least in small test cases.)

This example demonstrates another way that Dep and systems like it (nearly all package managers besides Go modules) can produce surprising results: when the one most preferred answer (use the latest of everything) does not apply,

there are often many choices with no clear preferences between them. The exact answer depends on the details of the SAT solving algorithm, heuristics, and often the input order of the packages are presented to the solver. This underspecification and non-determinism in their solvers is another reason these systems need lock files.

In any event, for the sake of Dep users, let's assume Dep lucked into the choice that keeps builds working. After all, we're still trying to break the Go modules users' builds.

To break Go modules builds, let's release a new version of A, version 1.21, which requires the latest B, which in turn requires the latest D. Now, when the go command builds the latest A, it is forced to use the latest B and the latest D. In Go modules, there is no C 1.9, so the go command uses C 1.8, and the combination of C 1.8 and D 1.6 does not work. Finally, we have broken the Go modules builds!



But look! The Dep builds are using C 1.8 and D 1.6 too, so they're also broken. Before, Dep had to make a choice between the latest B and the latest C. If it chose the latest B, the build broke. If it chose the latest C, the build worked. The new requirement in A is forcing Dep to choose the latest B and the latest D, taking away Dep's choice of latest C. So Dep uses the older C 1.8, and the build breaks just like before.

What should we conclude from all this? First of all, documenting an incompatibility for Dep does not guarantee to avoid that incompatibility. Second, a repeatable build like in Go modules also does not guarantee to avoid the incompatibility. Both tools can end up building the incompatible pair of packages. But as we saw, it takes multiple intentional steps to lead Go modules to a broken build, steps that lead Dep to the same broken build. And along the way the Dep-based build broke two other times when the Go modules build did not.

I've been using Dep in these examples because it is the immediate predecessor of Go modules, but I don't mean to single out Dep. In this respect, it works the same way as nearly every other package manager in every other language. They all have this problem. They're not even really broken or misbehaving so much as unfortunately designed. They are designed to try to work around a lack of cooperation among the various package maintainers, and *tools cannot work around a lack of cooperation.*

The only real solution for the C versus D incompatibility is to release a new, fixed version of either C or D. Trying to avoid the incompatibility is useful only because it creates more time for C's author and D's author to cooperate on a fix. Compared to the Dep approach of preferring latest versions and documenting incompatibilities, the Go modules approach of repeatable builds with mini-

mal version selection and no documented incompatibilities creates time for cooperation automatically, with no build emergencies, no declared incompatibilities, and no explicit work by users.

Then we can rely on cooperation for the real fix.

Conclusion

These are the three principles of versioning in Go, the reasons that the design of Go modules deviates from the design of Dep, Cargo, Bundler, and others.

- *Compatibility.* The meaning of a name in a program should not change over time.
- *Repeatability.* The result of a build of a given version of a package should not change over time.
- *Cooperation.* To maintain the Go package ecosystem, we must all work together. Tools cannot work around a lack of cooperation.

These principles are motivated by concerns about software engineering, which is what happens to programming when you add time and other programmers. Compatibility eliminates the effects of time on the meaning of a program. Repeatability eliminates the effects of time on the result of a build. Cooperation is an explicit recognition that, no matter how advanced our tools are, we do have to work with the other programmers. We can't work around them.

The three principles also reinforce each other, in a virtuous cycle.

Compatibility enables a new version selection algorithm, which provides repeatability. Repeatability makes sure that buggy, new releases are ignored until explicitly requested, which creates more time to cooperate on fixes. That cooperation in turn reestablishes compatibility. And the cycle goes around.

As of Go 1.13, Go modules are ready for production use, and many companies, including Google, have adopted them. The Go 1.14 and Go 1.15 releases will bring additional ergonomic improvements, toward eventually deprecating and removing support for GOPATH. For more about adopting modules, see the blog post series on the Go blog, starting with "Using Go Modules."