So you want to write a package manager

sam boyer
Feb 12, 2016 · 50 min read

You woke up this morning, rolled out of bed, and thought, “Y’know what? I don’t have enough misery and suffering in my life. I know what to do — I’ll write a language package manager!”

Totally. Right there with you. I take my misery and suffering in moderation, though, and since I think you might too, this design guide could help you keep most of your hair. You, or people hoping to improve an existing package manager, or curious about how they work, or who’ve realized that their spiffy new language is pretty much consigned to irrelevance without one. Whatever.

Now, I also have an ulterior motive: right now, the Go community actually DOES need proper package management, and I’m contributing to an approach. As such, I’ll be returning often to Go as the main reference case, and there’s a dedicated Go section at

the end. But the real focus here is general package management design principles and domain constraints, and how they may apply in different languages.

Package management is awful, you should quit right now

Package management is a nasty domain. Really nasty. On the surface, it *seems* like a purely technical problem, amenable to purely technical solutions. And so, quite reasonably, people approach it that way. Over time, these folks move inexorably towards the conclusion that:

1. software is terrible
2. people are terrible
3. there are too many different scenarios
4. nothing will really work for sure
5. it's provable that nothing will really work for sure
6. our lives are meaningless perturbations in a swirling vortex of chaos and entropy

If you've ever created…well, basically any software, then this epiphanic progression probably feels familiar. It's the design death spiral — one that, experience tells us, is a signal to go back and reevaluate expectations and assumptions. Often enough, it turns out that you just hadn't framed the problem properly in the first place. And — happy day! — that's the case here: those who see package management as a purely technical problem are inevitably, even if subconsciously, looking for something that can completely and correctly automate upgrades.

Stop that. Please. You will have a bad time.

Package management is at least as much about people, what they know, what they do, and what they can reasonably be responsible for as it is about source code and other things a computer can figure out. Both sides are necessary, but independently insufficient.

Oh but wait! I'm already ahead of myself.

LOLWUT is "Package Manager"

You know how the internet works, so you probably already read the first two sentences of what Wikipedia has to say. Great, you're an expert now. So you know that first we've

gotta decide what *kind* of package manager to write, because otherwise it's like "Hey pass me that bow?" and you say "Sure" then hand me an arrow-shooter, but I wanted a ribbon, and Sonja went to get her cello. Here's the menu:

- **OS/system package manager (SPM):** this is not why we are here today
- **Language package manager (LPM):** an interactive tool (e.g., `go get`) that can retrieve and build specified packages of source code for a particular language. Bad ones dump the fetched source code into a global, unversioned pool (GOPATH), then cackle maniacally at your fervent hope that the cumulative state of that pool makes coherent sense.
- **Project/application dependency manager (PDM):** an interactive system for managing the source code dependencies of a *single project* in a particular language. That means specifying, retrieving, updating, arranging on disk, and removing sets of dependent source code, in such a way that *collective coherency* is maintained beyond the termination of any single command. Its output — which is precisely reproducible — is a self-contained source tree that acts as the input to a compiler or interpreter. You might think of it as "compiler, phase zero."

The main distinction here is between systems that help *developers* to create new software, versus systems for *users* to install an instance of some software. SPMs are systems for users, PDMs are systems for developers, and LPMs are often sorta both. But when LPMs aren't backed by a PDM, the happy hybrid devolves into a cantankerous chimaera.

PDMs, on the other hand, are quite content without LPMs, though in practice it typically makes sense to bundle the two together. That bundling, however, often confuses onlookers into conflating LPMs and PDMs, or worse, neglecting the latter entirely.

Don't do that. Please. You will have a bad time.

PDMs are pretty much at the bottom of this stack. Because they compose with the higher parts (and because it's what Go desperately needs right now), we're going to focus on them. Fortunately, describing their responsibilities is pretty easy. The tool must correctly, easily, and quickly, move through these steps:

1. divine, from the myriad possible shapes of and disorder around real software in development, the set of immediate dependencies the developer ***intends*** to rely on,

then

2. transform that intention into a precise, recursively-explored list of source code dependencies, such that anyone — the developer, a different developer, a build system, a user — can
3. create/***reproduce*** the dependency source tree from that list, thereby
4. creating an isolated, self-contained artifact of project + dependencies that can be input to a compiler/interpreter.

Remember that thing about how package management is just as much about people as computers? It's reflected in "intend" and "reproduce," respectively. There is a natural tension between the need for absolute algorithmic certainty of outputs, and the fluidity inherent in development done by humans. That tension, being intrinsic and unavoidable, demands resolution. Providing that resolution is *fundamentally* what PDMs do.

We Have Met The Enemy, And They Are Us

It's not the algorithmic side that makes PDMs hard. Their final outputs are phase zero of a compiler or interpreter, and while the specifics of that vary from language to language, each still presents a well-defined target. As with many problems in computing, the challenge is figuring out how to present those machine requirements in a way that fits well with human mental models.

Now, we're talking about PDMs, which means we're talking about interacting with the world of FLOSS— pulling code from it, and possibly publishing code back to it. (In truth, not just FLOSS — code ecosystems within companies are often a microcosm of the same. But let's call it all FLOSS, as shorthand.) I'd say the basic shape of that FLOSS-entwined mental model goes something like this:

- I have some unit of software I'm creating or updating — a "project." While working on that project, it is my center, my home in the FLOSS world, and anchors all other considerations that follow.
- I have a sense of what needs to be done on my project, but must assume that my understanding is, at best, incomplete.
- I know that I/my team bear the final responsibility to ensure the project we create works as intended, regardless of the insanity that inevitably occurs upstream.

- I know that if I try to write all the code myself, it will take more time and likely be less reliable than using battle-hardened libraries.
- I know that relying on other peoples' code means hitching my project to theirs, entailing at least some logistical and cognitive overhead.
- I know that there is a limit beyond which I cannot possibly grok all code I pull in.
- I don't know all the software that's out there, but I do know there's a lot, *lot* more than what I know about. Some of it could be relevant, almost all of it won't be, but searching and assessing will take time.
- I have to prepare myself for the likelihood that most of what's out there may be crap — or at least, I'll experience it that way. Sorting wheat from chaff, and my own feelings from fact, will also take time.

The themes here are time, risk, and uncertainty. When developing software, there are unknowns in every direction; time constraints dictate that you can't explore everything, and exploring the *wrong* thing can hurt, or even sink, your project. Some uncertainties may be heightened or lessened on some projects, but we *cannot* make them disappear. Ever. They are natural constraints. In fact, some are even necessary for the functioning of open software ecosystems, so much so that we codify them:

> *THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,* ***FITNESS FOR A PARTICULAR PURPOSE****…*
> *- The MIT License (emphasis mine)*

And you *know* something's for serious when it's in boilerplate legalese that everyone uses, but no one reads.

OK, OK, I get it: creating software is full of potentially dangerous unknowns. We're not gonna stop doing it, though, so the question is: how can we be out there, writing software/cavorting in the jungle of uncertainty, but simultaneously insulate our projects (and sanity) from all these risks?

I like to think of the solution through an analogy from public health: harm reduction. A little while back, Julia Evans wrote a lovely piece called Harm Reduction for Developers. It's short and worth the read, but I'll paraphrase the crux:

> People are going to do risky activities. Instead of saying YOU'RE WRONG TO DO THAT JUST DON'T DO THAT, we can choose to help make those activities less risky.

This is the mentality we *must* adopt when building a practical tool for developers. Not because we're cynically catering to some lowest-common-denominator caricature of the cowboy (cowperson? cowfolk?) developer, but because…well, look at that tower of uncertainty! Taking risks is a *necessary* part of development. The PDM use case is for a tool that encourages developers to reduce uncertainty through wild experimentation, while simultaneously keeping the project as stable and sane as possible. Like a rocket with an anchor! Or something.

Under different circumstances, this might be where I start laying out use cases. But… let's not. That is, let's not devolve into enumerating specific cases, inevitably followed by groping around in the dark for some depressing, box-ticking, design-by-committee middle ground. For PDMs, anyway, I think there's a better way. Instead of approaching the solution in terms of use cases, I'm going to take a crude note from distributed systems and frame it in terms of states and protocols.

States and Protocols

There's good reason to focus on state. Understanding what state we have, as well as when, why, and how it changes is a known-good way of stabilizing software against that swirling vortex of chaos and entropy. If we can lay down even some general rules about what the states ought to represent and the protocols by which they interact, it bring order and direction to the rat's nest. (*Then,* bring on the use cases.) Hell, even this article went through several revisions before I realized it, too, should be structured according to the states and protocols.

Meet the Cast

PDMs are constantly choreographing a dance between four separate states on-disk. These are, variously, the inputs to and outputs from different PDM commands:

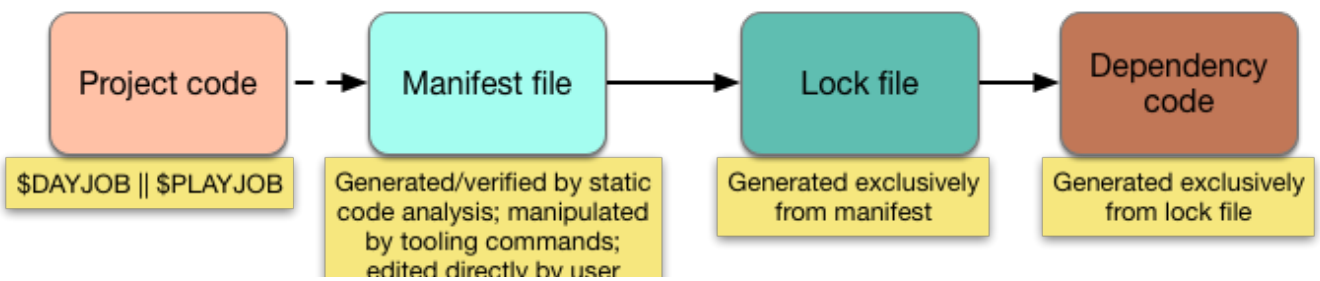
The on-disk states that matter to a PDM

- **Project code:** the source code that's being actively developed, for which we want the PDM to manage its dependencies. Being that it's not currently 1967, all project code is under version control. For most PDMs, project code is all the code in the repository, though it could be just a subdirectory.
- **Manifest file:** a human-written file — though typically with machine help— that lists the depended-upon packages to be managed. It's common for a manifest to also hold other project metadata, or instructions for an LPM that the PDM may be bundled with. (This must be committed, or else nothing works.)
- **Lock file:** a machine-written file with all the information necessary to [re]produce the full dependency source tree. Created by transitively resolving all the dependencies from the manifest into concrete, immutable versions. (This should always get committed. Probably. Details later.)
- **Dependency code**: all of the source code named in the lock file, arranged on disk such that the compiler/interpreter will find and use it as intended, but isolated so that nothing else would have a reason to mutate it. Also includes any supplemental logic that some environments may need, e.g., autoloaders. (This needn't be committed.)

Let's be clear — there are PDMs that don't have all of these. Some may have them, but represent them differently. Most newer ones now do have everything here, but this is not an area with total consensus. My contention, however, is that every PDM needs some version of these four states to achieve its full scope of responsibilities, and that this separation of responsibilities is optimal.

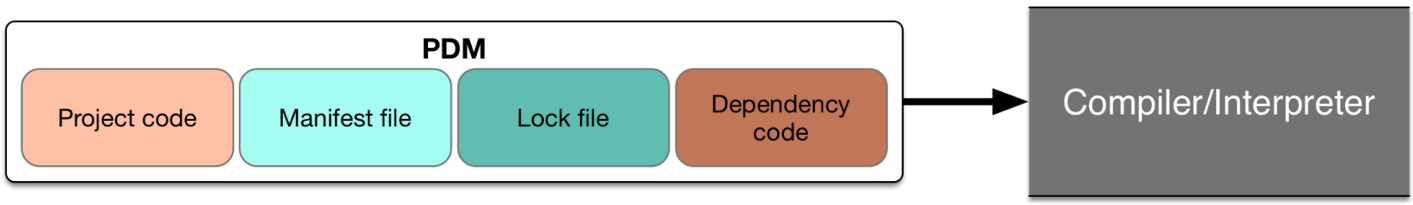
Pipelines within Pipelines

States are only half of the picture. The other half is the protocols — the procedures for movement between the states. Happily, these are pretty straightforward.



The states form, roughly, a pipeline, where the inputs to each stage are the state of its predecessor. (Mostly — that first jump is messy.) It's hard to overstate the positive impact this linearity has on implementation sanity: among other things, it means there's always a clear "forward" direction for PDM logic, which removes considerable ambiguity. Less ambiguity, in turn, means less guidance needed from the user for correct operation, which makes using a tool both easier and safer.

PDMs also exist within some larger pipelines. The first is the compiler (or interpreter), which is why I've previously referred to them as "compiler phase zero":

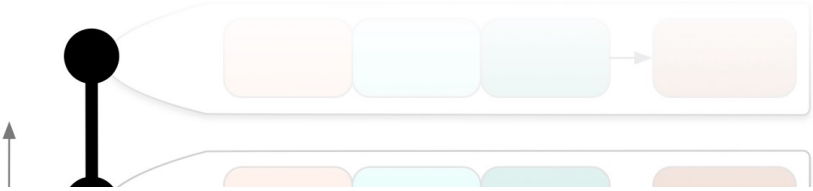


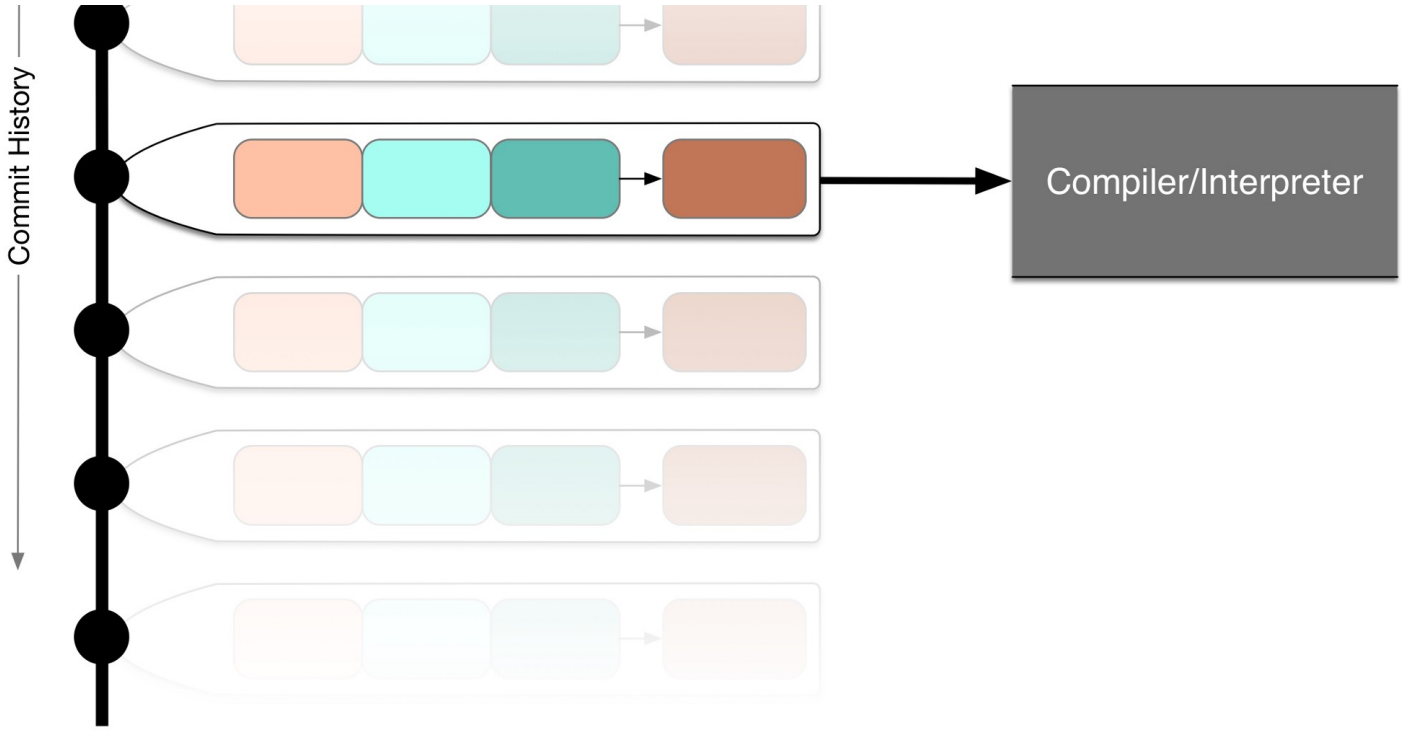
The compiler/interpreter do not know or care about the PDM. All they see is source code inputs.

For ahead-of-time compiled languages, PDMs are sort of a pre-preprocessor: their aggregate result is the main project code, plus dependency code, arranged in such a way that when the preprocessor (or equivalent) sees "include code X" in the source, X will resolve to what the project's author intended.

The 'phase zero' idea still holds in a dynamic or JIT-compiled language, though the mechanics are a bit different. In addition to laying out the code on disk, the PDM typically needs to override or intercept the interpreter's code loading mechanism in order to resolve includes correctly. In this sense, the PDM is producing a filesystem layout for itself to consume, which a nitpicker could argue means that arrow becomes self-referential. What really matters, though, is that it's expected for the PDM to lay out the filesystem *before* starting the interpreter. So, still 'phase zero.'

The other pipeline in which PDMs exist is version control. This is entirely orthogonal to the compiler pipeline. Like, actually orthogonal:





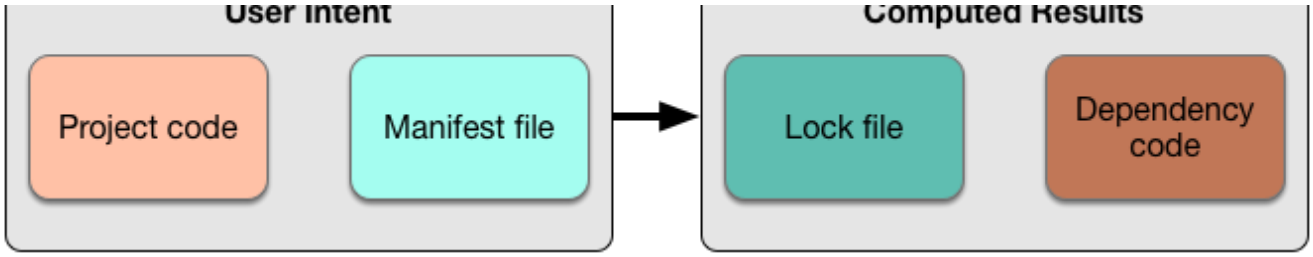
Source code (and PDM actions thereupon) are on the X axis. Commit history, as organized by a version control system, is on the Y.

See? The PDM feeds the compiler code along the X axis, while version control provides chronology along the Y axis. Orthogonal!

…wait. Stuff on the X axis, time on the Y axis…this sounds weirdly similar to the Euclidean space we use to describe spacetime. Does that mean PDMs are some kind of bizarro topological manifold function? And compiler outputs are **literally** a spacetime continuum!? Maybe! I don't know. I'm bad at math. Either way, though, we've got space and time, so I'm callin it: each repository is its own little universe.

Silly though this metaphor may be, it remains oddly useful. Every project repo-verse is chock full of its own logical rules, all evolving along their own independent timelines. And while I don't know how hard it would be to align *actual* universes in a sane, symbiotic way, "rather difficult" seems like a safe assumption. So, when I say that PDMs are tools for aligning code-universes, I'm suggesting that it's *fucking challenging.* Just lining up the code once isn't enough; you have to keep the timelines aligned as well. Only then can the universes grow and change together.

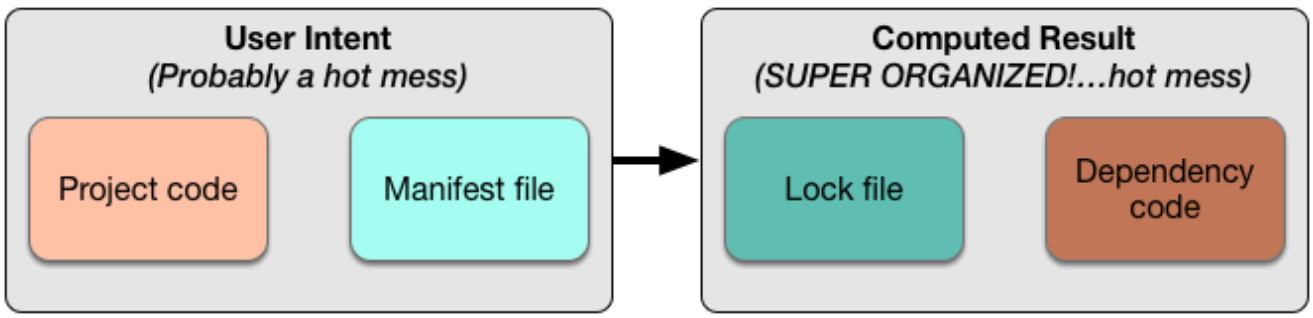
"Aligning universes" will come in handy later. But, there's one other thing to immediately note back down in the PDM pipeline itself. While PDMs do deal with four physically discrete states, there's really only two concepts at work:



Project code and the manifest express the user's intentions. The lock file and dependency source code are the PDM's attempts to fulfill those intentions.

Taken together, some combination of the manifest and the project code are an expression of *user intent*. That's useful shorthand in part because of what it says about what NOT to do: you don't manually mess with the lock file or dependencies, for the same reason you don't change generated code. And, conversely, a PDM oughtn't change anything on the left when producing the lock file or the dependency tree. Humans to the left, machines to the right.

That's suspiciously nice and tidy, though. Let's inject a little honesty:



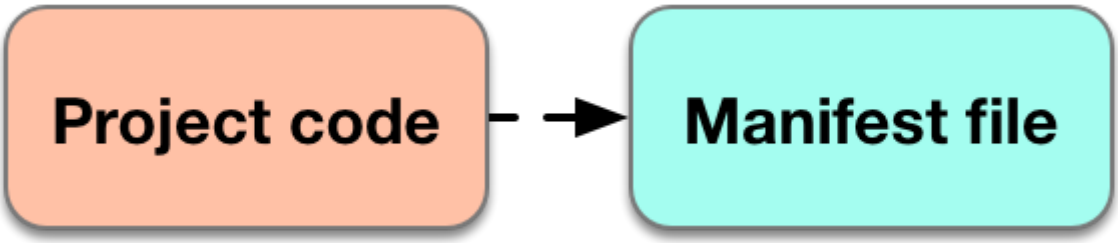
Don't even pretend it's not true

For a lot of software, "hot mess" might even be charitable. And that can be frustrating, leading to a desire to remove/not create capabilities in tooling that lead to, or at least encourage, the mess-making. Remember the goal, though: *harm reduction.* A PDM's job is not to prevent developers from taking risks, but to make that risky behavior as safe and structured as possible. Risk often leads to a hot mess, at least momentarily. Such is the nature of our enterprise. Embrace it.

Probably the easiest way to get PDMs wrong is focusing too much on one side to the detriment of the other. Balance must be maintained. Too much on the right, and you end up with a clunky, uptight mess like git submodules that developers simply won't use. Too much on the left, and the productivity gains from making things "just so easy!" will sooner or later (mostly sooner) be siphoned off by the crushing technical debt incurred from building on an unstable foundation.

Don't do either. Please. *Everyone* will have a bad time.

To Manifest a Manifest



Also known as, "normal development"

Manifests often hold a lot of information that's language-specific, and the transition from code to manifest itself tends to be quite language-specific as well. It might be more accurate to call this "World -> Manifest", given how many possible inputs there are. To make this concrete, let's have a look at an actual manifest:

```
[package]
name = "iron"
version = "0.2.6"
description = "Extensible, Concurrency Focused Web Development in Rust."
readme = "README.md"
repository = "https://github.com/iron/iron"
documentation = "http://ironframework.io/doc/iron/"
license = "MIT"
authors = [
    "", # privacy: redacted
]

[lib]
name = "iron"
path = "src/lib.rs"

[features]
default = ["ssl"]
ssl = ["hyper/ssl"]

[dependencies]
typemap = "0.3"
url = "0.5"
plugin = "0.2*"
modifier = "0.1"
error = "0.1"
```

```
log = "0.3"
conduit-mime-types = "0.7"
lazy_static = "0.1"
num_cpus = "0.2"

[dependencies.hyper]
version = "0.7"
default-features = false

[dev-dependencies]
time = "*"
```

Cargo.toml hosted with ❤️ by GitHub view raw

A slightly tweaked version of the Cargo manifest for Rust's iron crate

As there's no existing Go standard for this, I'm using a Cargo (Rust) manifest. Cargo, like a lot of software in this class, is considerably more than *just* a PDM; it's also the user's typical entry point for compiling Rust programs. The manifest holds metadata about the project itself under **package**, parameterization options under **features** (and other sections), and dependency information under assorted **dependencies** sections. We'll explore each.

Central Package Registry

The first group is mostly for Rust's central package registry, crates.io. Whether or not to use a registry might be the single most important question to answer. Most languages with any sort of PDM have one: Ruby, Clojure, Dart, js/node, js/browser, PHP, Python, Erlang, Haskell, etc. Of course, most languages — Go being an exception — have no choice, as there isn't enough information directly in the source code for package retrieval. Some PDMs rely on the central system exclusively, while others also allow direct interaction with source repositories (e.g. on GitHub, Bitbucket, etc.). Creating, hosting, and maintaining a registry is an enormous undertaking. And, also it's…not my problem! This is not an article about building highly available, high-traffic data storage services. So I'm skippin' 'em. Have fun!

…ahem.

Well, I'm skipping *how* to build a registry, but not their functional role. By acting as package metadata caches, they can offer significant performance benefits. Since package metadata is generally held in the manifest, and the manifest is necessarily in the source repository, inspecting the metadata would ordinarily require cloning a

repository. A registry, however, can extract and cache that metadata, and make it available via simple HTTP API calls. Much, much faster.

Registries can also be used to enforce constraints. For example, the above Cargo manifest has a 'package.version' field:

```
[package]
version = "0.2.6"
```

Think about it this a little. Yep: it's nuts! Versions must refer to a single revision to be of use. But by writing it into the manifest, that version number just slides along with every commit made, thereby ending up applying to multiple revisions.

Cargo addresses this by imposing constraints in the registry itself: publishing a version to crates.io is *absolutely permanent*, so it doesn't matter what other commits that version might apply to. From crates' perspective, only the first revision actually gets it.

Other sorts of constraint enforcement might include validation of a well-formed source package. If the language does not dictate a particular source structure, then the PDM might impose one, and a registry could enforce it: npm could look for a well-formed module object, or composer could try to validate conformance to an autoloader PSR. (AFAIK, neither do, nor am I even sure either is possible). In a language like Go, where code is largely self-describing, this sort of thing is mostly unnecessary.

Parameterization

The second group is all about parameterization. The particulars here are Rust-specific, but the general idea is not: parameters are well-defined ways in which the form of the project's output, or its actual logical behavior, can be made to vary. While some aspects can intersect with PDM responsibilities, this is often out of the PDM's scope. For example, Rust's target profiles allow control over the optimization level passed to its compiler. And, compiler args? PDM don't care.

However, some types of options — Rust's features, Go's build tags, any notion of build profiles (e.g. *test* vs. *dev* vs. *prod*) — can change what paths in the project's logic are utilized. Such shifts may, in turn, make some new dependencies required, or obviate the need for others. That *is* a PDM problem.

If this type of configurability is important in your language context, then your 'dependency' concept may need to admit conditionality. On the other hand, ensuring a minimal dependency set is (mostly) just about reducing network bandwidth. That makes it a performance optimization, and therefore skippable. We'll revisit this further in the lock files section.

If you're not a wizened Rustacean or package manageer, Cargo's 'features' may be a bit puzzling: just who is making choices about feature use, and when? Answer: the top-level project, in their own manifest. This highlights a major bit I haven't touched yet: projects as "apps," vs. projects as "libraries." Conventionally, apps are the project at the top of the dependency chain — the top-level project — whereas libraries are necessarily somewhere down-chain.

The app/lib distinction, however, is not so hard-and-fast. In fact, it's mostly situational. While libraries might *usually* be somewhere down the dependency chain, when running tests or benchmarks, they're the top-level project. Conversely, while apps are usually on top, they may have subcomponents that can be used as libraries, and thus may appear down-chain. A better way to think of this distinction is "a project produces 0..N executables."

Apps and libs really being mostly the same thing is good, because it suggests it's appropriate to use the same manifest structure for both apps and libs. It also emphasizes that manifests are necessarily both "downward"-facing (specifying dependencies) and "upward"-facing (offering information and choices to dependees).

Dependencies

The PDM's proper domain!

Each dependency consists of, at least, an identifier and version specifier. Parameterization or source types (e.g. raw VCS vs. registry) may also be present. Changes to this part of a manifest are necessarily of one of the following:

- Adding or removing a dependency
- Changing the desired version of an existing dependency
- Changes to the parameters or source types of a dependency

These are the tasks that devs actually need to do. Removal is so trivial that many PDMs don't provide a command, instead just expecting you'll delete the appropriate line from

the manifest. (I think an **rm** command is generally worth it, for reasons I'll get into later.) **add**ing generally oughtn't be more difficult than:

```
<tool> add <identifier>@<version>
```

or just

```
<tool> add <identifier>
```

to implicitly ask for the most recent version.

The only hard requirements for identifiers is that they all exist in the same namespace, and that the PDM can glean enough information from parsing them (possibly with some help from a 'type' field — e.g. 'git' or 'bzr', vs. 'pkg' if it's in a central registry) to determine how to fetch the resource. Basically, they're a domain-specific URI. Just make sure you avoid ambiguity in the naming scheme, and you're mostly good.

If possible, you should try to ensure package identifiers are the same as the names used to reference them in code (i.e. via include statements), as that's one less mental map for users to hold, and one less memory map for static analyzers to need. At the same time, one useful thing that PDMs often do, language constraints permitting, is the aliasing of one package as another. In a world of waiting for patches to be accepted upstream, this can be a handy temporary hack to swap in a patched fork.

Of course, you could also just swap in the fork for real. So why alias at all?

It goes back to the spirit of the manifest: they're a place to hold user intent. By using an alias, an author can signal to other people — project collaborators especially, but users as well — that it *is* a temporary hack, and they should set their expectations appropriately. And if being implicit isn't clear enough, they can always put in a comment explaining why!

Outside of aliases, though, identifiers are pretty humdrum. Versions, however, are anything but.

Hang on, we need to talk about versions

Versions are hard. Maybe this is obvious, maybe not, but recognizing what makes them hard (and the problem they solve) is crucial.

Versions have exactly, unambiguously, and unequivocally *fuck all* to do with the code they emblazon. There are literally zero questions about actual logic a version number can definitively answer. On that very-much-not-answerable list is one of the working stiff developer's most important questions: "will my code still work if I change to a different version of yours?"

Despite these limitations, we use versions. Widely. The simple reason is due to one of those pieces of the FLOSS worldview:

> I know there is a limit beyond which I cannot possibly grok all code I pull in.

If we *had* to completely grok code before using or updating it, we'd never get anything done. Sure, it might be A Software Engineering Best Practice™, but enforcing it would grind the software industry to a halt. That's a far greater risk than having some software not work some times because some people used versions wrong.

There's a less obvious reason we rely on versions, though: there is no mechanical alternative. That is, according to our current knowledge of how computation works, a generic tool (even language-specific) capable of figuring out whether or not combinations of code will work as intended *cannot exist*. Sure, you can write tests, but let's not forget Dijkstra:

> *"Empirical testing can only prove the presence of bugs, never their absence."*

Someone better than me at math, computer science and type theory could probably explain this properly. But you're stuck with me for now, so here's my glib summary: type systems and design by contract can go a long way towards determining compatibility, but if the language is Turing complete, they will never be sufficient. At most, they can prove code is *not incompatible*. Try for more, and you'll end up in a recursive descent through new edge cases that just keep on popping out. A friend once referred to such endeavors as playing Gödelian whack-a-mole. (Pro tip: don't play. Gödel's winning streak runs 85 years.)

This is not (just) abstruse theory. It confirms the simple intuition that, in the "does my code work correctly with yours?" decision, humans must be involved. Machines can help, potentially quite a lot, by doing parts of the work and reporting results, but they can't make a precise final decision. Which is exactly why versions need to exist, and why systems around them work the way they do: to help humans make these decisions.

Versions' sole *raison d'etre* is as a crude signaling system from code's authors to its users. You can also think of them as a curation system. When adhering to a system like SemVer, versions can suggest:

- Evolutions in the software, via a well-defined ordering relationship between any two versions
- The general readiness of a given version for public use (i.e., <1.0.0, pre-releases, alpha/beta/rc)
- The likelihood of different classes of incompatibilities between any given pair of versions
- Implicitly, that if there are versions, but you use a revision *without* a version, you may have a bad time

For a system like semver to be effective in your language, it's important to set down some language-specific guidelines around what kind of logic changes correspond to major, minor, and patch-level changes. Rust got out ahead of this one. Go needs to, and still can. Without them, not only does everyone just go by 'feel' — AKA, "let's have everyone come up with their own probably-garbage approach, then get territorial and defensive" — but there's no shared values to call on in an issue queue when an author increments the wrong version level. And those conversations are crucial. Not only do they fix the immediate mistake, but they're how we collectively improve at using versions to communicate.

Despite the lack of necessary relationship between versions and code, there is at least one way in which versions help quite directly to ensure software works: they focus public attention on a few particular revisions of code. With attention focused thusly, Linus' Law suggests that bugs will be rooted out and fixed (and released in subsequent patch versions). In this way, the curatorial effect of focusing attention on particular versions reduces systemic risk around those versions. This helps with another of FLOSS' uncertainties:

> I have to prepare myself for the likelihood that most of what's out there will probably be crap. Sorting wheat from chaff will also take time.

Having versions at least ensures that, if the software is crap, it's because it's *actually* crap, not because you grabbed a random revision that *happened* to be crap. That saves time. Saved time can save projects.

Non-Version Versions

Many PDMs also allow other, non-semver version specifiers. This isn't strictly necessary, but it has important uses. Two other types of version specifiers are notable, both of which more or less necessitate that the underlying source type is a VCS repository: specifying a branch to 'follow', or specifying an immutable revision.

The type of version specifier used is really a decision about *how* you want to relate to that upstream library. That is: what's your strategy for aligning their universe with yours? Personally, I see it a bit like this:

Note that only one of these concepts can be represented with pictures from iStockPhoto. HMMM.

Versions provide me a nice, ordered package environment. Branches hitch me to someone else's ride, where that "someone" may or may not be hopped up on cough syrup and blow. Revisions are useful when the authors of the project you need to pull in have provided so little guidance that you basically just have to spin the wheel and pick a revision. Once you find one that works, you write that revision to the manifest as a signal to your team that you *never* want to change again.

Now, any of these *could* go right or wrong. Maybe those pleasant-seeming packages are brimming with crypto backdoored by NSA. Maybe that dude pulling me on a rope tow

is actually a trained car safety instructor. Maybe it's a, uh, friendly demon running the roulette table?

Regardless, what's important about these different identifiers is how it defines the update process. Revisions have *no* update process; branches are constantly chasing upstream, and versions, especially via version range specifiers, put you in control of the kind of ride you want…as long as the upstream author knows how to apply semver correctly.

Really, this is all just an expansion and exploration of yet another aspect of the FLOSS worldview:

> I know that relying on other peoples' code means hitching my project to theirs, entailing at least some logistical and cognitive overhead.

We know there's always going to be some risk, and some overhead, to pulling in other peoples' code. Having a few well-defined patterns at least make the alignment strategy immediately evident. And we care about that because, once again, manifests are an expression of user intent: simply looking at the manifest's version specifier clearly reveals how the author wants to relate to third-party code. It can't make the upstream code any better, but following a pattern reduces cognitive load. If your PDM works well, it will ease the logistical challenges, too.

The Unit of Exchange

I have hitherto been blithely ignoring something big: just what *is* a project, or a package, or a dependency? Sure, it's a bunch of source code, and yes, it emanates, somehow, from source control. But is the entire repository the project, or just some subset of it? The real question here is, "what's the unit of source code exchange?"

For languages without a particularly meaningful source-level relationship to the filesystem, the repository can be a blessing, as it provides a natural boundary for code that the language does not. In such cases, the repository is the unit of exchange, so it's only natural that the manifest sits at the repository root:

```
$ ls -a
.
..
.git
MANIFEST
<ur source code heeere>
```

However, for languages that do have a well-defined relationship to the filesystem, the repository isn't providing value as a boundary. (Go is the strongest example of this that I know, and I deal with it in the Go section.) In fact, if the language makes it sane to independently import different subsets of the repository's code, then using the repository as the unit of exchange can actually get in the way.

It can be inconvenient for consumers that want only some subset of the repository, or different subsets at different revisions. Or, it can create pain for the author, who feels she must break down a repository into atomic units for import. (Maintaining many repositories sucks; we only do it because somehow, the last generation of DVCS convinced us all it was a good idea.) Either way, for such languages, it may be preferable to define a unit of exchange *other* than the repository. If you go down that path, here's what you need to keep in mind:

- The manifest (and the lock file) take on a particularly meaningful relationship to their neighboring code. Generally, the manifest then defines a single 'unit.'
- It is still **ABSOLUTELY NECESSARY** that your unit of exchange be situated on its own timeline — and you can't rely on the VCS anymore to provide it. No timeline, no universes; no universes, no PDM; no PDM, no sanity.
- And remember: software is hard enough without adding a time dimension. Timeline information shouldn't be in the source itself. Nobody wants to write real code inside a tesseract.

Between versions in the manifest file and path dependencies, it would appear that Cargo has figured this one out, too.

Other Thoughts

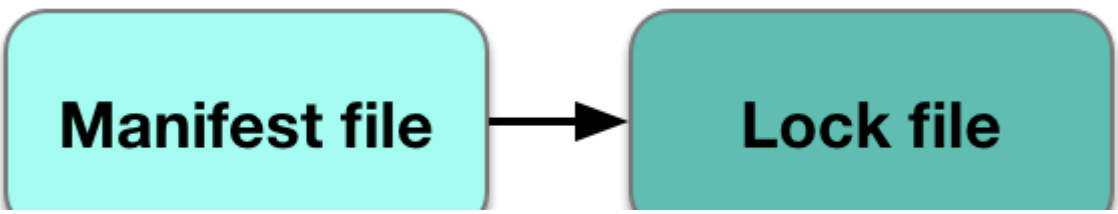
Mostly these are bite-sized new ideas, but also some review.

- Choose a format primarily for humans, secondarily for machines: TOML or YAML, else (ugh) JSON. Such formats are declarative and stateless, which makes things

simpler. Proper comments are a big plus — manifests are the home of experiments, and leaving notes for your collaborators about the what and why of said experiments can be very helpful!

- TIMTOWTDI, at least at the PDM level, is your **arch-nemesis**. Automate housekeeping completely. If PDM commands that change the manifest go beyond add/remove and upgrade commands, it's probably accidental, not essential. See if it can be expressed in terms of these commands.
- Decide whether to have a central package registry (almost certainly yes). If so, jam as much info for the registry into the manifest as needed, *as long as* it in no way impedes or muddles the dependency information needed by the PDM.
- Avoid having information in the manifest that can be unambiguously inferred from static analysis. High on the list of headaches you do *not* want is unresolvable disagreement between manifest and codebase. Writing the appropriate static analyzer is hard? Tough tiddlywinks. Figure it out so your users won't have to.
- Decide what versioning scheme to use (Probably semver, or something like it/enhancing it with a total order). It's probably also wise to allow things outside the base scheme: maybe branch names, maybe immutable commit IDs.
- Decide if your software will combine PDM behavior with other functionality like an LPM (probably yes). Keep any instructions necessary for that purpose cleanly separated from what the PDM needs.
- There are other types of constraints — e.g., required minimum compiler or interpreter version — that may make sense to put in the manifest. That's fine. Just remember, they're secondary to the PDM's main responsibility (though it may end up interleaving with it).
- Decide on your unit of exchange. Make a choice appropriate for your language's semantics, but absolutely ensure your units all have their own timelines.

The Lockdown



In which we jump the gap. Are you watching closely?

Transforming a manifest into a lock file is the process by which the fuzz and flubber of development are hardened into reliable, reproducible build instructions. Whatever crazypants stuff a developer does with dependencies, the lock file ensures another user can replicate it — zero thinking required. When I reflect on this apropos of the roiling, seething mass that is software, it's pretty amazing.

This transformation is the main process by which we mitigate harm arising from the inherent risks of development. It's also how we address one particular issue in the FLOSS worldview:

> I know that I/my team bear the final responsibility to ensure the project we create works as intended, regardless of the insanity that inevitably occurs upstream.

Now, some folks don't see the value in precise reproducibility. *"Manifests are good enough!", "It doesn't matter until the project gets serious"* and *"npm became popular long before shrinkwrap* (npm's lock file) *was around!"* are some arguments I've seen. But these arguments strike me as wrongheaded. Rather than asking, "Do I need reproducible builds?" ask "Do I lose anything from reproducible builds?" *Literally everyone* benefits from them, eventually. (Unless emailing around tarballs and SSH'ing to prod to bang out changes in nano is your idea of fun). The only question is if you need reproducibility a) now, b) soon, or else c) can we be friends? because I think maybe you're not actually involved in shipping software, yet you're still reading this, which makes you a weird person, and I like weird people.

The only real potential downside of reproducible builds is the tool becoming costly (slow) or complicated (extra housekeeping commands or arcane parameters), thus impeding the flow of development. These are real concerns, but they're also arguments against poor implementations, not reproducibility itself. In fact, they're really UX guidelines that suggest what 'svelte' looks like on a PDM: fast, implicit, and as automated as possible.

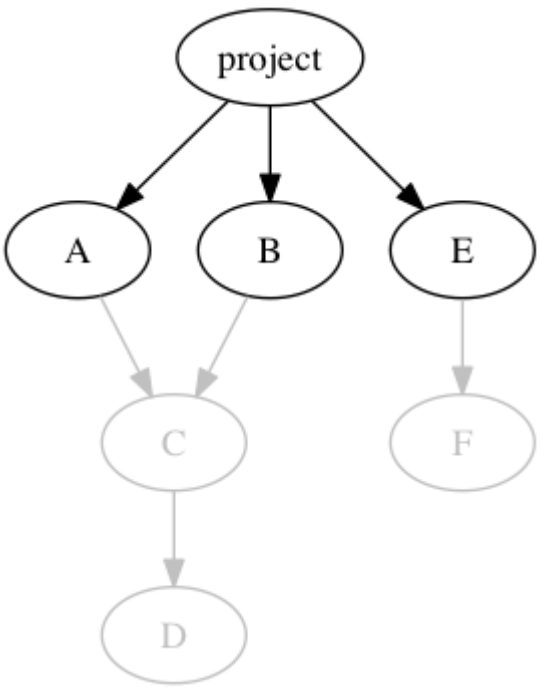
The algorithm

Well, those guidelines just scream “algorithm!” And indeed, lock file generation must be fully automated. The algorithm itself can become rather complicated, but the basic steps are easily outlined:

- Build a dependency graph (so: directed, acyclic, and variously labeled) by recursively following dependencies, starting from those listed in the project’s manifest
- Select a revision that meets the constraints given in the manifest
- If any shared dependencies are found, reconcile them with <strategy>
- Serialize the final graph (with whatever extra per-package metadata is needed), and write it to disk. Ding ding, you have a lock file!

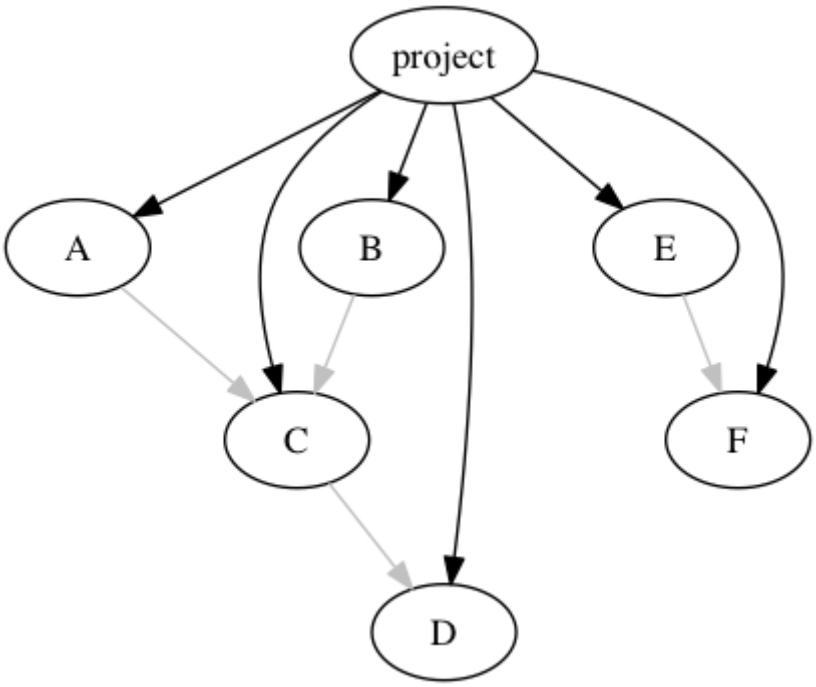
(Author’s note: The general problem here is boolean satisfiability, which is NP-complete. This breakdown is still roughly helpful, but trivializes the algorithm.)

This provides us a lock file containing a complete list of all dependencies (i.e., all reachable deps in the computed, fully resolved dep graph). “All reachable” means, if our project has three direct dependencies, like this:



Our project depends directly on A and B, which depend on C, which depends on D, and E, which depends on F.

We still directly include all the transitively reachable dependencies in the lock file:



The project's lock file should record all of A, B, C, D, E, and F.

Exactly how much metadata is needed depends on language specifics, but the basics are the package identifier, an address for retrieval, and the closest thing to an immutable revision (i.e. a commit hash) that the source type allows. If you add anything else — e.g., a target location on disk — it should *only* be to ensure that there is absolutely zero ambiguity in how to dump out the dependency code.

Of the four basic steps in the algorithm, the first and last are more or less straightforward if you have a working familiarity with graphs. Sadly, graph geekery is beyond my ability to bequeath in an article; please feel free to reach out to me if that's where you're stuck.

The middle two steps (which are really just "choose a revision" split in two), on the other hand, have hiccups that we can and should confront. The second is mostly easy. If a lock file already exists, keep the locked revisions indicated there unless:

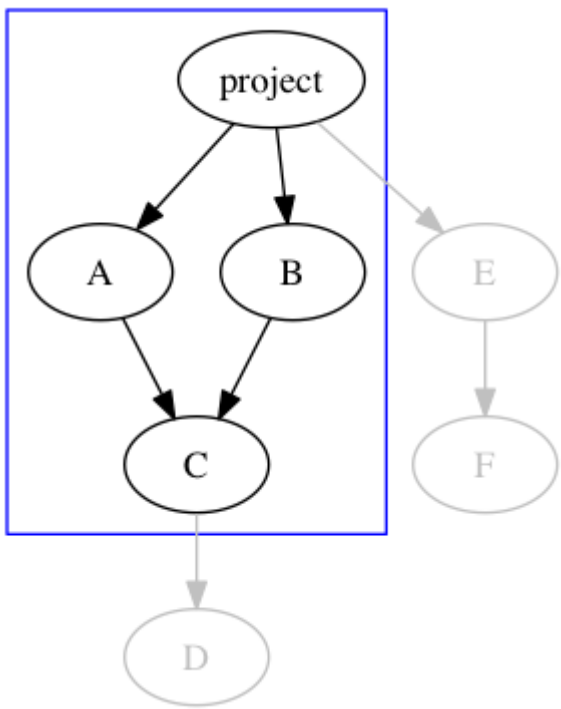
- The user expressly indicated to ignore the lock file
- A floating version, like a branch, is the version specifier
- The user is requesting an upgrade of one or more dependencies
- The manifest changed and no longer admits them
- Resolving a shared dependency will not allow it

This helps avoid unnecessary change: if the manifest would admit 1.0.7, 1.0.8, and 1.0.9, but you'd previously locked to 1.0.8, then subsequent resolutions should notice that and re-use 1.0.8. If this seems obvious, good! It's a simple example that's illustrative of the fundamental relationship between manifest and lock file.

This basic idea approach is well-established — Bundler calls it "conservative updating." But it can be extended further. Some PDMs recommend against, or at least are indifferent to, lock files committed in libraries, but that's a missed opportunity. For one, it makes things simpler for users by removing conditionality — commit the lock file always, no matter what. But also, when computing the top-level project's depgraph, it's easy enough to make the PDM interpret a dependency's lock file as being revision *preferences*, rather than requirements. Preferences expressed in dependencies are, of course, given less priority than those expressed in the top-level project's lock file (if any). As we'll see next, when shared dependencies exist, such 'preferences' can promote even greater stability in the build.

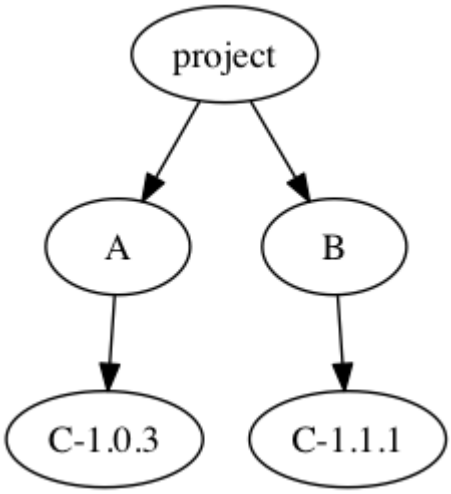
Diamonds, SemVer and Bears, Oh My!

The third issue is harder. We have to select a strategy for picking a version when two projects share a dependency, and the right choice depends heavily on language characteristics. This is also known as the "diamond dependency problem," and it starts with a subset of our depgraph:



The set in blue form a happy diamond.

With no versions specified here, there's no problem. However, if A and B require different versions of C, then we have a conflict, and the diamond splits:



A broken diamond. Also noteworthy: while the happy diamond is merely a graph, this is also a tree. Y'know what else are trees? Filesystems. Do you smell a useful isomorphism? I smell a useful isomorphism.

There are two classes of solution here: allow multiple C's (duplication), or try to resolve the conflict (reconciliation). Some languages, like Go, don't allow the former. Others do, but with varying levels of risky side effects. Neither approach is intrinsically superior for correctness. However, user intervention is never needed with multiple C's, making that approach *far* easier for users. Let's tackle that first.

If the language allows multiple package instances, the next question is state: if there's global state that dependencies can and do typically manipulate, multiple package instances can get clobbery in a hurry. Thus, in node.js, where there isn't a ton of shared state, npm has gotten away with avoiding all possibility of conflicts by just intentionally checking out everything in 'broken diamond' tree form. (Though it can achieve the happy diamond via "deduping," which is the default in npm v3).

Frontend javascript, on the other hand, has the DOM — the grand daddy of global shared state — making that approach much riskier. This makes it a much better idea for bower to reconcile ("flatten", as they call it) all deps, shared or not. (Of course, frontend javascript also has the intense need to minimize the size of the payload sent to the client.)

If your language permits it, and the type system won't choke on it, *and* the global state risks are negligible, ***and*** you're cool with some binary/process footprint bloating and (probably negligible) runtime performance costs, ***AND*** bogging down static analysis and transpilation tooling is OK, then duplication is the shared deps solution you're

looking for. That's a lot of conditions, but it may still be preferable to reconciliation strategies, as most require user intervention — colloquially known as DEPENDENCY HELL — and all involve potentially uncomfortable compromises in the logic itself.

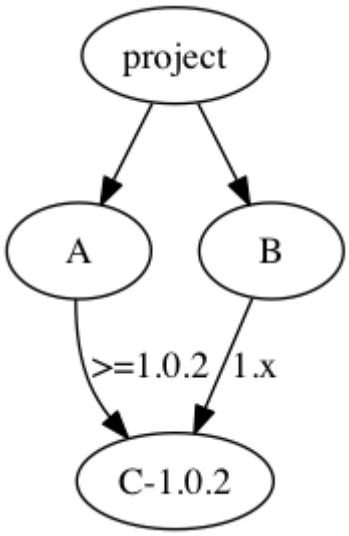
If we assume that the A -> C and B -> C relationships are both specified using versions, rather than branches or revisions, then reconciliation strategies include:

- **Highlander:** Analyze the A->C relationship and the B->C relationship to determine if A can be safely switched to use C-1.1.1, or B can be safely switched to use C-1.0.3. If not, fall back to *realpolitik*.
- ***Realpolitik*:** Analyze other tagged/released versions of C to see if they can satisfy both A and B's requirements. If not, fall back to elbow grease.
- **Elbow grease:** Fork/patch C and create a custom version that meets both A and B's needs. At least, you THINK it does. It's probably fine. Right?

Oh, but wait! I left out the one where semver can save the day:

- **Phone a friend:** ask the authors of A and B if they can both agree on a version of C to use. (If not, fall back to Highlander.)

The last is, by far, the best *initial* approach. Rather than me spending time grokking A, B and C well enough to resolve the conflict myself, I can rely on signals from A and B's authors — the people with the *least uncertainty* about their projects' relationship to C— to find a compromise:



A's manifest says it can use any patch version of v1.0 newer than 2, and B's manifest says it can use any minor and patch version of v1. This could potentially resolve to many versions, but if A's lock file pointed at 1.0.3, then the algorithm can choose that, as it results in the least change.

Now, that resolution may not actually work. Versions are, after all, just a crude signaling system. 1.x is a bit of a broad range, and it's possible that B's author was lax in choosing it. Nevertheless, it's still a good place to start, because:

- Just because the semver ranges suggest solution[s], doesn't mean I have to accept them.
- A PDM tool can always further refine semver matches with static analysis (if the static analyses feasible for the language has anything useful to offer).
- No matter which of the compromise solutions is used, I *still* have to do integration testing to ensure everything fits for my project's specific needs.
- The goal of *all* the compromise approaches is to pick an acceptable solution from a potentially large search space (as large as all available revisions of C). Reducing the size of that space for zero effort is beneficial, even if occasional false positives are frustrating.

Most important of all, though, is that if I do the work and discover that B actually was too lax and included versions for C that do not work (or excludes versions that *do* work), I can file patches against B's manifest to change the range appropriately. Such patches record the work you've done, publicly and for posterity, in a way that helps others avoid the same pothole. A decidedly FLOSSy solution, to a distinctly FLOSSy problem.

Dependency Parameterization

When discussing the manifest, I touched briefly on the possibility of allowing parameterization that would necessitate variations in the dependency graphs. If this is something your language would benefit from, it can add some wrinkles to the lock file.

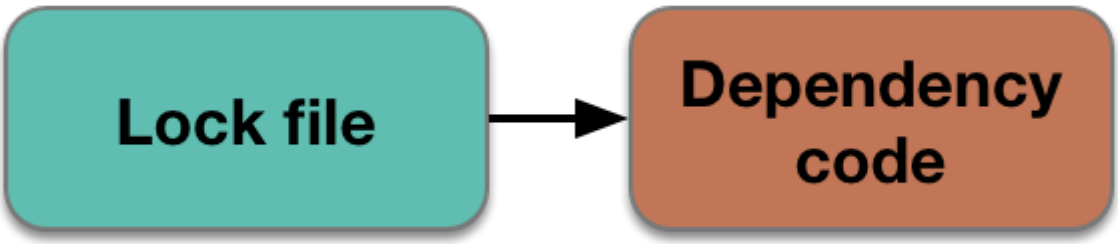
Because the goal of a lock file is to completely and unambiguously describe the dependency graph, parameterizing things can get expensive quickly. The naive approach would construct a full graph in memory for each unique parameter combination; assuming each parameter is a binary on/off, the number of graphs required grows exponentially (2^N) in the number of parameters. Yikes.

However, that approach is less “naive,” than it is “braindead.” A better solution might enumerate all the combinations of parameters, divide them into sets based on which combinations have the same input set of dependencies, and generate one graph per set. And an even better solution might handle all the combinations by finding the smallest input dependency set, then layering all the other combinations on top in a single, multivariate graph. (Then again, I’m an amateur algorithmicist on my best day, so I’ve likely missed a big boat here.)

Maybe this sounds like fun. Or maybe it’s vertigo-inducing jibberish. Either way, skipping it’s an option. Even if your manifest does parameterize dependencies, you can always just get *everything*, and the compiler will happily ignore what it doesn’t need. And, once your PDM inevitably becomes wildly popular and you are showered with conference keynote speaker invitations, someone will show up and take care of this hard part, Because Open Source.

That’s pretty much it for lock files. Onwards, to the final protocol!

Compiler, phase zero: Lock to Deps



All the lifting, none of the thinking

If your PDM is rigorous in generating the lock file, this final step may amount to blissfully simple code generation: read through the lock file, fetch the required resources over the network (intermediated through a cache, of course), then drop them into their nicely encapsulated, nothing-will-mess-with-them place on disk.

There are two basic approaches to encapsulating code: either place it under the source tree, or dump it in some central, out-of-the-way location where both the package identifier and version are represented in the path. If the latter is feasible, it’s a great option, because it hides the actual dependee packages from the user, who really shouldn’t need to look at them anyway. Even if it’s not feasible to use the central location directly as compiler/interpreter inputs — probably the case for most languages

— then do it anyway, and use it as a cache. Disk is cheap, and maybe you'll find a way to use it directly later.

If your PDM falls into the latter, can't-store-centrally camp, you'll have to encapsulate the dependency code somewhere else. Pretty much the only "somewhere else" that you can even hope to guarantee won't be mucked with is under the project's source tree. (Go's new vendor directories satisfy this requirement nicely.) That means placing it in the scope of what's managed by the project's version control system, which immediately begs the question: should dependency sources be committed?

…probably not. There's a persnickety technical argument against committing: if your PDM allows for conditional dependencies, then which conditional branch should be committed? "Uh, Sam, obv: just commit all of them and let the compiler use what it wants." Well then, why'd you bother constructing parameterized dep graphs for the lock file in the first place? And, what if the different conditional branches need different versions of the same package? And…and…

See what I mean? Obnoxious. Someone could probably construct a comprehensive argument for never committing dep sources, but who cares? People will do it anyway. So, being that PDMs are an exercise in harm reduction, the right approach ensures that committing dep sources is a safe choice/mistake to make.

For PDMs that directly control source loading logic — generally, interpreted or JIT-compiled languages — pulling in a dependency with its deps committed isn't a big deal: you can just write a loader that ignores those deps in favor of the ones your top-level project pulls together. However, if you've got a language, such as Go, where filesystem layout is the entire game, deps that commit their deps are a problem: they'll override whatever reality you're trying to create at the top-level.

For wisdom on this, let's briefly turn to distributed systems:

> *A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.*
> *- Leslie Lamport*

If that sounds like hell — you're right! Anything that doesn't *have* to behave like a distributed system, shouldn't. You can't allow person A's poor use of your PDM to prevent person B's build from working. So, if language constraints leave you no other

choice, the only recourse is to blow away the committed deps-of-deps when putting them into place on disk.

That's all for this bit. Pretty simple, as promised.

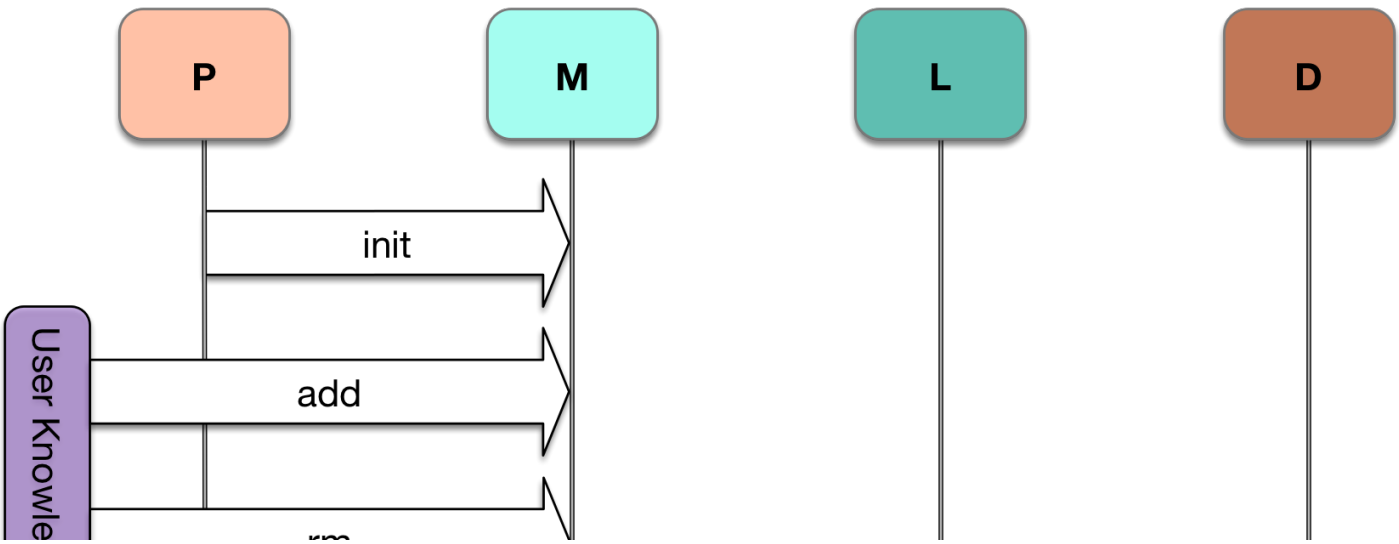
The Dance of the Four States

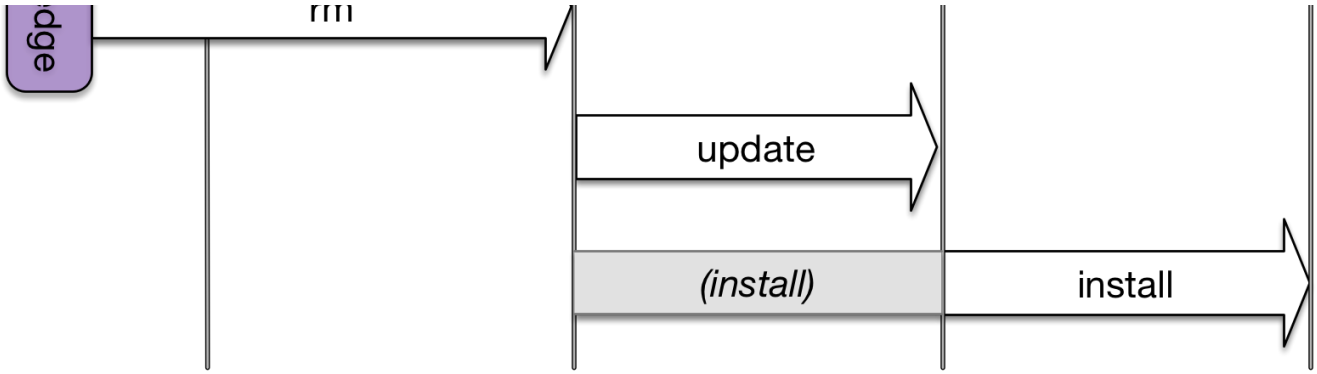
OK, we've been through the details on each of the states and protocols. Now we can step back and assemble the big picture.

There's a basic set of commands that most PDMs provide to users. Using the common/intuitive names, that list looks something like this:

- **init:** Create a manifest file, possibly populating it based on static analysis of the existing code.
- **add:** Add the named package[s] to the manifest.
- **rm:** Remove the named package[s] from the manifest. (Often omitted, because text editors exist).
- **update:** Update the pinned version of package[s] in the lock file to the latest available version allowed by the manifest.
- **install:** Fetch and place all dep sources listed in the lock file, first generating a lock file from the manifest if it does not exist.

Our four-states concept makes it easy to visualize how each of these commands interacts with the system. (For brevity, let's also abbreviate our states to **P**[roject code], **M**[anifest], **L**[ock file], and **D**[ependency code]):





Each command reads the state at the arrow's source in order to mutate the state at the arrow's target. add/rm might be triggered from static analysis, but typically come from the user's knowledge of what needs doing. install implicitly creates a lock file, if needed.

Cool. But there are obvious holes — do I really have to run two or three separate commands (add, update, install) to pull in packages? That'd be annoying. And indeed, composer's require (their 'add' equivalent) also pushes through the manifest to update the lock, then fetches the source package and dumps it to disk.

npm, on the other hand, subsumes the 'add' behavior into their install command, requiring additional parameters to change either the manifest or the lock. (As I pointed out earlier, npm historically opted for duplicated deps/trees over graphs; this made it reasonable to focus primarily on output deps, rather than following the state flows I describe here.) So, npm requires extra user interaction to update all the states. Composer does it automatically, but the command's help text still describes them as discrete steps.

I think there's a better way.

A New-ish Idea: Map, Sync, Memo

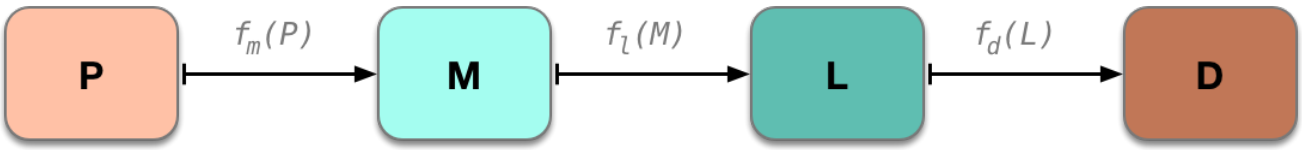
Note: this is off the beaten path. As far as I know, no PDM is as aggressive as this approach. It's something we may experiment with in Glide.

Perhaps it was clear from the outset, but part of my motivation for casting the PDM problem in terms of states and protocols was to create a sufficiently clean model that it would be possible to define the protocols as one-way transformation functions:

- $f : \mathbf{P} \rightarrow \mathbf{M}$: To whatever extent static analysis can infer dependency identifiers or parameterization options from the project's source code, this maps that information into the manifest. If no such static analysis is feasible, then this 'function' is really just manual work.

- $f: \mathbf{M} \rightarrow \mathbf{L}$: Transforms the immediate, possibly-loosely-versioned dependencies listed in a project's manifest into the complete reachable set of packages in the dependency graph, with each package pinned to a single, ideally immutable version for each package.
- $f: \mathbf{L} \rightarrow \mathbf{D}$: Transforms the lock file's list of pinned packages into source code, arranged on disk in a way the compiler/interpreter expects.

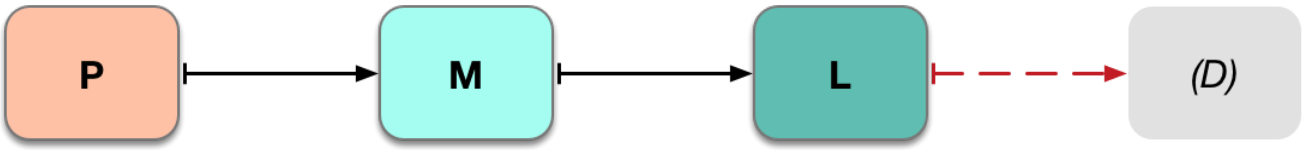
If all we're really doing is mapping inputs to outputs, then we can define each pair of states as being in sync if, given the existing input state, the mapping function produces the existing output state. Let's visually represent a fully synchronized system like this:



Now, say the user manually edited the manifest, but hasn't yet run a command to update the lock file. Until she does, M and L will be out of sync:

Manifest and lock are out of sync, but lock and deps are still in sync because each function is narrowly scoped

Similarly, if the user has just cloned a project with a committed manifest and lock file, then we know L and D will be out of sync simply because the latter doesn't exist:



If there's a lock file, but the dep sources don't exist...well then, duh, they're out of sync

I could list more, but I think you get the idea. At any given time that any PDM command is executed, each of the manifest, lock, and deps can be in one of three states:

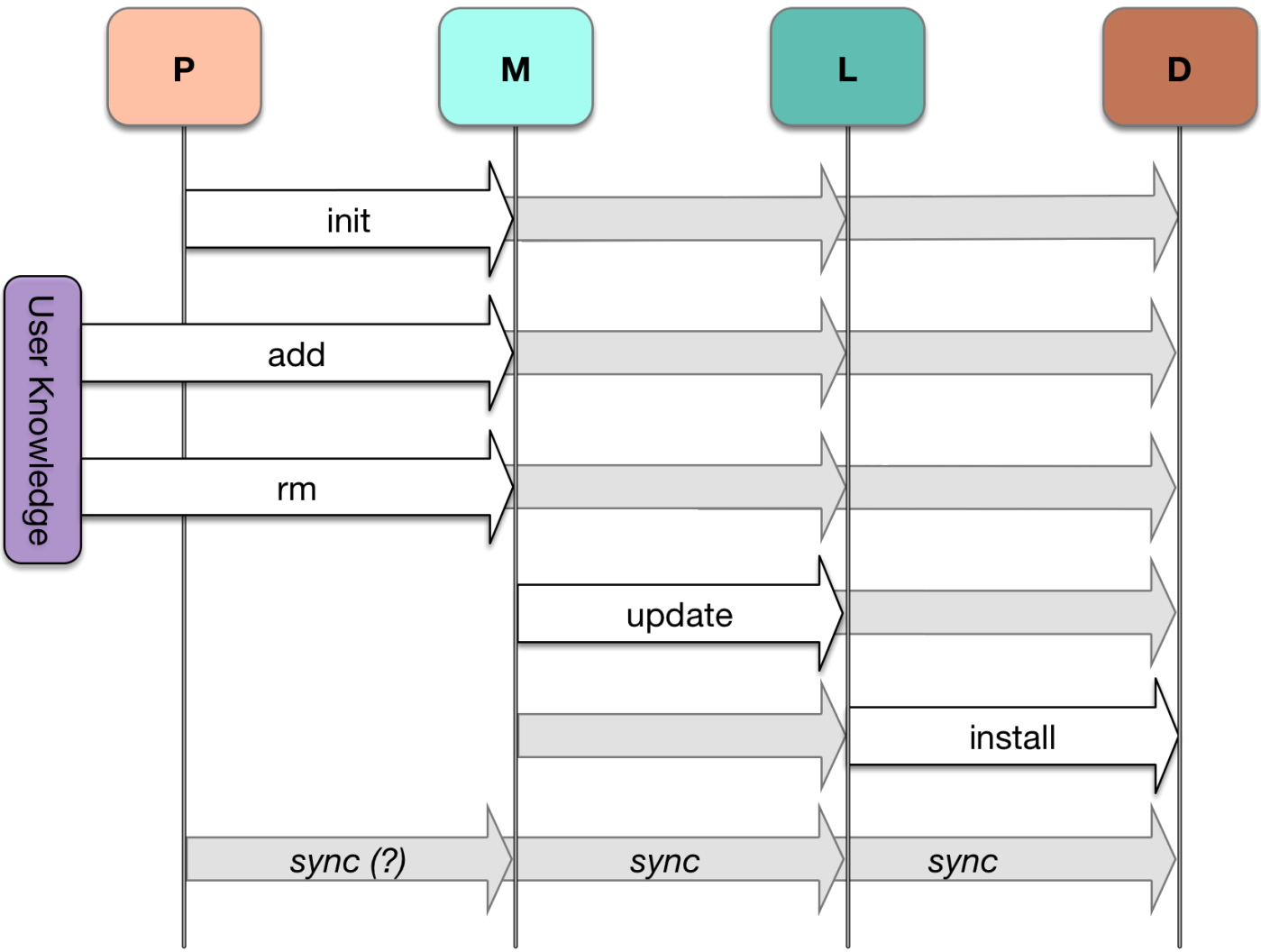
- does not exist

- exists, but desynced from predecessor
- exists and in sync with predecessor

Laying it all out like this brings an otherwise-subtle design principle to the fore: if these states and protocols can be so well-defined, then isn't it kind of dumb for a PDM to *ever* leave the states desynced?

Why, yes. Yes it is.

No matter what command is executed, PDMs should strive to leave all the states — or at least the latter three — fully synchronized. Whatever the user tells the PDM to do, it does, then takes whatever steps are necessary to ensure the system is still aligned. Taking this 'sync-based' approach, our command diagram now looks like this:



All commands are still oriented towards the same type of state mutation, but they rest atop a pervasive sync operation that ensures a computable pre- and post-mutation environment. (`install` might not even be needed anymore!)

Viewed in this way, the dominant PDM behavior becomes a sort of move/counter-move dance, where the main action of a command mutates one of the states, then the system reacts to stabilize itself.

Here's a simple example: a user manually edits the manifest to remove a dependency, then `add`s a different package. In a PDM guided narrowly by user commands, it's easy to imagine the manually-removed package not being reflected in the updated lock. In a sync-based PDM, the command just adds the new entry into the manifest, then relinquishes control to the sync system.

There's obvious user benefits here: no extra commands or parameters are needed to keep everything shipshape. The benefits to the PDM itself, though, may be less obvious: if some PDM commands intentionally end in a desync state, then the next PDM command to run will encounter this, and has to *decide why* there's a desync: is it because the previous command left it that way? Or because the user *wants* it that way? It's a surprisingly difficult question to answer — and often undecidable. But it never comes up in a sync-based PDM, as they always attempt to move towards a sane final state.

Of course, there's a (potential) drawback: it assumes the only valuable state is full synchronization. If the user actually *does* want a desync…well, that's a problem. My sense, though, is that the desyncs users may think they want today are more about overcoming issues in their current PDM than solving a real domain problem. Or, to be blunt: sorry, but your PDM's been gaslighting you. Kinda like your very own NoSQL Bane. In essence, my guess here is that a sync-based approach obviates the practical need for giving users more granular control.

There is (at least) one major practical challenge for sync-based PDMs, though: performance. If your lock file references thirty gzipped tarballs, ensuring full sync by re-dumping them all to disk on every command gets prohibitively slow. For a sync-based PDM to be feasible, it must have cheap routines to determine whether each state pair is in sync. Fortunately, since we're taking care to define each sync process as a function with clear inputs, there's a nice solution: memoization. In the sync process, the PDM should hash the inputs to each function and include that hash as part of the output. On subsequents runs, compare the new input hash to the old recorded one; if they're the same, then assume the states are synced.

Now really, it's just **M** → **L** and **L** → **D** that we care about here. The former is pretty easy: the inputs are the manifest's dep list, plus parameters, and the hash can be tucked into the generated lock file.

The latter is a little uglier. Input calculation is still easy: hash the serialized graph contained in the lock file — or, if your system has parameterized deps, just the subgraph you're actually using. The problem is, there's no elegant place to store that hash. So, don't bother trying: just write it to a specially-named file at the root of the deps subdirectory. (If you're a the lucky duck writing to a central location with identifier+version namespacing, you can skip all of this.)

Neither of these approaches is foolproof, but I'm guessing they'll hold up fine in practice.

. . .

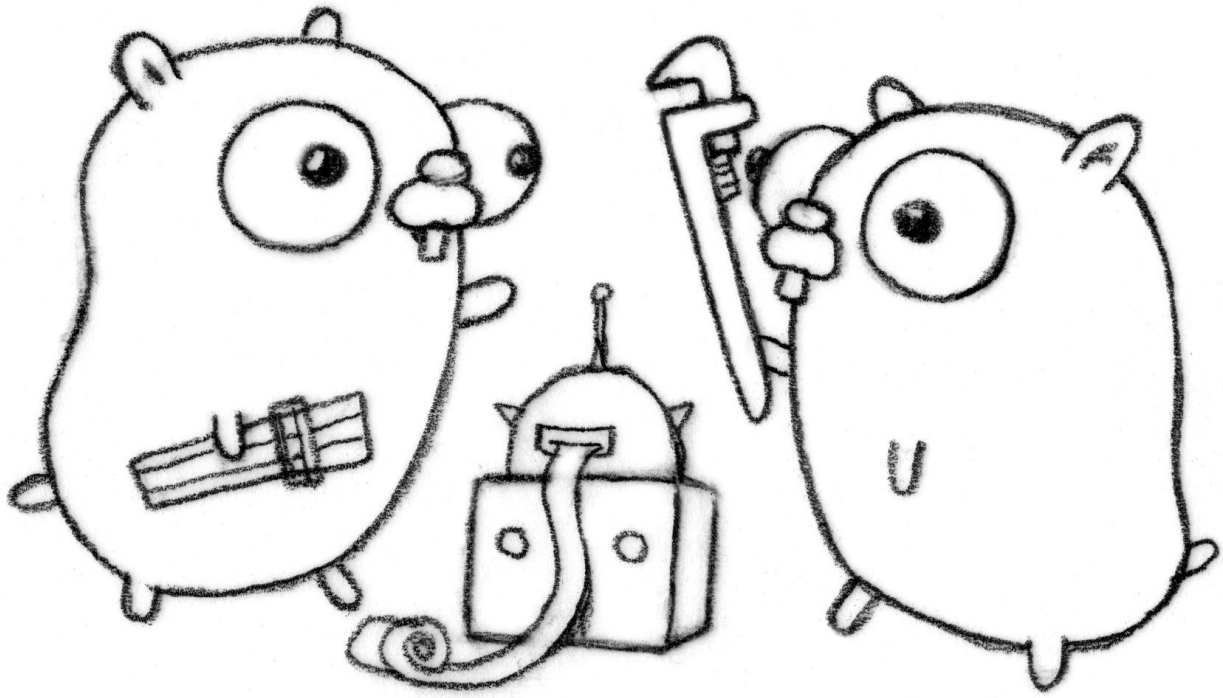
Dénouement

Writing a general guide to *all* the considerations for a PDM is a daunting task. I suspect I've missed at least one major piece, and I'm sure I've neglected plenty of potentially important details. Comments and suggestions are welcome. I'd like this guide to be reasonably complete, as package management is both a *de facto* requirement for widespread language use today, and too wide-ranging of a problem to be adequately captured in organic, piecemeal mailing list discussions. I've seen a number of communities struggle to come to grips with the issue for exactly this reason.

That said, I do think I've brought the substantial aspects of the problem into view. In the final counting, though, the technical details may be less important than the broad organizing concepts: language package management is an exercise in harm reduction, performed by maintaining a balance between humans and machines/chaos and order/dev and ops/risk and safety/experimentation and reliability. Most importantly, it is about maintaining this balance *over time*.

Because management over time is a requirement, but it's a categorically terrible idea for languages to try to manage their own source chronology as part of the source itself, it suggests that pretty much all languages could benefit from a PDM. Or rather, *one* PDM. A language community divided by mutually incompatible packaging tools is not a situation I'd walk into lightly.

OK general audience — shoo! Or not. Either way, it's Go time!



A PDM for Go

Thus far, this article has conspicuously lacked any tl;dr. This part being more concrete, it gets one:

- A PDM for Go is doable, needn't be that hard, and could even integrate nicely with `go get` in the near term
- A central Go package registry could provide lots of wins, but any immediate plan must work without one
- Monorepos can be great for internal use, and PDMs should work within them, but monorepos without a registry are harmful for open code sharing
- I have an action plan that will indubitably result in astounding victory and great rejoicing, but to learn about it you will need to read the bullet points at the end

. . .

Go has struggled with package management for years. We've had partial solutions, and

no end of discussion, but nothing's really nailed it. I believe that's been the case because the whole problem has never been in view; certainly, that's been a major challenge in at least the last two major discussions. Maybe, with the big picture laid out, things already seem a bit less murky.

First, let's fully shift from the general case to Go specifically. These are the three main constraints that govern the choices we have available:

- GOPATH manipulation is a horrible, unsustainable strategy. We all know this. Fortunately, Go 1.6 adds support for the vendor directory, which opens the door to encapsulated builds and a properly project-oriented PDM.
- Go's linker will allow only one package per unique import path. Import path rewriting can circumvent this, but Go also has package-level variables and init functions (aka, global state and potentially non-idempotent mutation of it). These two facts make npm-style, "broken diamond" package duplication an unsound strategy for handling shared deps.
- Without a central registry, repositories must continue acting as the unit of exchange. This intersects quite awkwardly with Go's semantics around the directory/package relationship.

Also, in reading through many of the community's discussions that have happened over the years, there are two ideas that seem to be frequently missed:

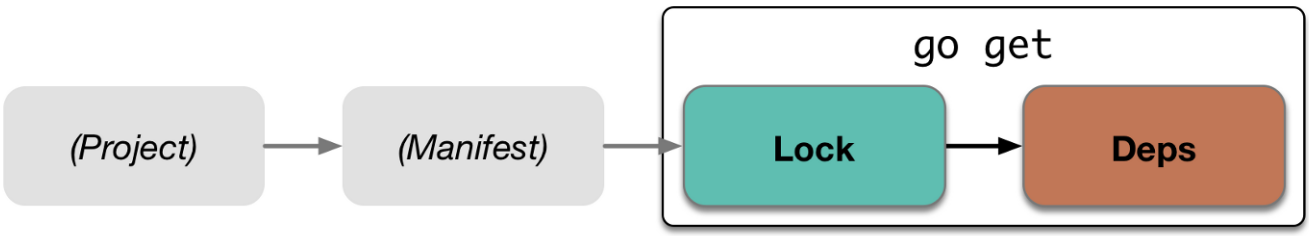
- Approaches to package management *must* be dual-natured: both inward-facing (when your project is at the 'top level' and consuming other deps), and outward-facing (when your project is a dep being consumed by another project).
- While there's some appreciation of the need for harm reduction, too much focus has been on reducing harm through reproducible builds, and not enough on mitigating the risks and uncertainties developers grapple with in day-to-day work.

That said, I don't think the situation is dire. Not at all, really. There are some issues to contend with, but there's good news, too. Good stuff first!

Upgrading `go get`

As with so many other things, Go's simple, orthogonal design makes the technical task of creating a PDM relatively easy (once the requirements are clear!). Even better,

though, is that a transparent, backwards-compatible, incremental path to inclusion in the core toolchain exists. It's not even hard to describe:



We can define a format for a lock file that conforms to the constraints outlined here, and teach `go get` to interact with it. The most important aspect of "conformance" is that the lock file be a transitively complete list of dependencies for both the package it cohabits a directory with, and all subpackages — main and not, test and not, as well as encompassing any build tag parameterizations. In other words, imagine the following repository layout:

```
.git                # so, this is the repo root
main.go             # a main package
cmd/
  bar/
    main.go         # another main package
foo/
  foo.go            # a non-main package
  foo_amd64.go      # arch-specific, may have extra deps
  foo_test.go       # tests may have extra deps
LOCKFILE            # the lockfile, derp
vendor/             # `go get` puts all (LOCKFILE) deps here
```

That LOCKFILE is correct iff it contains immutable revisions for all the transitive dependencies of all three (<root>, foo, bar) packages. The algorithm `go get` follows, then, is:

1. Walk back up from the specified subpath (e.g. `go get <repoaddr>/cmd/bar`) until a LOCKFILE is found or repository root is reached
2. If a LOCKFILE is found, dump deps into adjacent vendor dir
3. If no LOCKFILE is found, fall back to the historical GOPATH/default branch tip-based behavior

This works because it parallels how the vendor directory itself works: for anything in the directory tree rooted at vendor's parent (here, the repo root), the compiler will search that vendor directory before searching GOPATH.

I put the lock at the root of the repository in this example, but it doesn't necessarily have to be there. The same algorithm would work just fine within a subdirectory, e.g.:

```
.git
README.md
src/
  LOCKFILE
  main.go
  foo/
    foo.go
```

Not exactly best practices, but a `go get <repoaddr>/src` will still work fine with the above algorithm.

This approach can even work in some hinky, why-would-you-do-that situations:

```
.git
foo.go
bar/
  bar.go
cmd/
  LOCKFILE
  main.go
```

If the main package imports both foo and bar, then putting the vendor dir adjacent to the LOCKFILE under cmd/ means foo and bar won't be encapsulated by it, and the compiler will fall back to importing from the GOPATH. I've found a little trick, though, and it seems like this problem can be addressed by putting a copy of the repository under its own vendor directory:

```
$GOPATH/github.com/sdboyer/example/
 .git
 foo.go
 bar/
   bar.go
 cmd/
   LOCKFILE
   main.go
```

```
vendor/github.com/sdboyer/example/
    foo.go
    bar/
      bar.go
    cmd/          # The PDM could omit from here on down
      LOCKFILE
      main.go
```

A bit weird, and fugly-painful for development, but it solves the problem in a fully automatable fashion.

Of course, just because we *can* accommodate this situation, doesn't mean we *should*. And that, truly, is the big question: where should lock files, and their co-located vendor directories, go?

If you're just thinking about satisfying particular use cases, this is a spot where it's easy to get stuck in the mud. But if we go back to design concept that PDMs must settle on a fundamental "unit of exchange," the answer is clearer. The pair of manifest and lock file represent the boundaries of a project 'unit.' While they have no necessary relationship with the project source code, given that we're dealing with filesystems, things tend to work best when they're at the root of the source tree their 'unit' encompasses.

This doesn't require that they're at the root of a repository — a subdirectory is fine, so long as they're parent to all the Go packages they describe. The real problem arises when there's a desire to have multiple trees, or subtrees, under a different lock (and manifest). I've pointed out a couple times how having multiple package/units share the same repository isn't feasible if the repository is relied on to provide temporal information. It's absolutely true, but in a Go context, it's not helpful just to assert "don't do it."

Instead, we need to draw a line between best practices for publishing a project, versus what's workable when you're building out, say, a company monorepo. Given the pervasive monorepo at Go's Googlian birthing grounds, and the gravitational pull towards monorepos on how code has been published over the last several years, drawing these lines is particularly important.

Sharing, Monorepos, and The Fewest "Don't Do It"s I can manage

A lot of folks who've come up through a FLOSS environment look at monorepos and say, "eeeww, why would you *ever* do that?" But, as with many choices made by Billion Dollar Tech Companies, there are significant benefits. Among other things, they're a

mitigation strategy against some of those uncertainties in the FLOSS ecosystem. For one, with a monorepo, it's possible to know what "all the code" is. As such, with help from comprehensive build systems, it's possible…theoretically…for individual authors to take systemic responsibility for the downstream effects of their changes.

Of course, in a monorepo, all the code is in one repository. By collapsing all code into a monorepo, subcomponents are subsumed into the timeline of the monorepo's universe, and thus lack their own temporal information — aka, meaningful versioning. And this is *fine*…as long as you're inside the same universe, on the same timeline. But, as I hope is clear by now, sane versioning is essential for those outside the monorepo — those of us who *are* stitching together a universe on the fly. Interestingly, that also includes folks who are stitching together other monorepos.

Now, if Go had a central registry, we could borrow from Rust's 'path dependencies' and work out a way of providing an independently-versioned unit of exchange composed from a repository's subpackages/subtrees. If you can get past the git/hg/bzr induced Stockholm-thinking that "small repos are a Good Thing™ because Unix philosophy", then you'll see there's a lot that's good about this. You show me a person who likes maintaining a ton of semi-related repositories, and I'll show you a masochist, a robot, or a liar.

But Go doesn't have a central registry. Maybe we can do that later. For now, though, we need to accept the repository as the unit of exchange, accept the versioning constraints that puts on us, and delineate between practices that are appropriate for a repository that's shared publicly — that is, contains packages in it that some other repository will want to import — and practices that are appropriate for repositories (generally, internal monorepos) that are not intended for import. Let's call the former "open," and the latter "closed."

- Open repositories are safe and sane to pull in as a dependency; closed repositories are not. (Just reiterating.)
- Open repositories should have one manifest, one lock file, and one vendor directory, ideally at the repository root.
- Open repositories should always commit their manifest and lock, but **not** their vendor directory.

- Closed repositories can have multiple manifest/lock pairs, but it'll be on you to arrange them sanely.
- Closed repositories can safely commit their vendor directories.
- Open or closed, you should never directly modify upstream code in a vendor directory. If you need to, fork it, edit, and pull in (or alias) your fork.

While I think this is a good goal, a PDM will also need to be pragmatic: given the prevalence of 'closed'-type monorepos in the public Go ecosystem (if nothing else, golang.org/x/*), there needs to be some support for pulling in packages from the same repository at different versions. And that's doable. But for a sane, version-based ecosystem to evolve, the community **must** either move away from sharing closed/monorepos, or build a registry. All the possible compromises I see there throw versioning under the bus, and that's a non-starter.

Semantic Versioning, and an Action Plan

Last fall, Dave Cheney made a valiant effort towards achieving some traction on semantic versioning adoption. In my estimation, the discussion made some important inroads on getting the importance of using a versioning system in the community. Unfortunately (though probably not incorrectly, as it lacked concrete outcomes) the formal proposal failed.

Fortunately, I don't think formal adoption is really necessary for progress. Adopting semantic versioning is just one of the fronts on which the Go community must advance in order to address these issues. Here's a rough action plan:

- Interested folks should come together to create and publish a general recommendation on what types of changes necessitate what level of semver version bumps.
- These same interested folks could come together to write a tool that does basic static analysis of a Go project repository to surface relevant facts that might help in deciding on what semver number to initially apply. (And then…post an issue automatically with that information!)
- If we're feeling really frisky, we could also throw together a website to track the progress towards the semver-ification of Go-dom, as facilitated by the analyze-and-post tool. Scoreboards, while dumb, help spur collective action!

- At the same time, we can pursue a simplest-possible case — defining a lock file, for the repository root only, that `go get` can read and transparently use, if it's available.
- As semver's adoption spreads, the various community tools can experiment with different workflows and approaches to the monorepo/versioning problems, but all still conform to the same lock file.

Package management in Go doesn't have to be an intractable problem. It just tends to seem that way if approached too narrowly. While there are things I haven't touched on — versioning of compiled objects, static analysis to reduce depgraph bloat, local path overriding for developer convenience — I think I've generally covered the field. If we can make active progress towards these goals, then we could be looking at a dramatically different Go ecosystem by the end of the year — or even by GopherCon!

Go is *so ripe* for this, it's almost painful. The simplicity, the amenability to fast static analysis…a Go PDM could easily be a shining example among its peers. I have no difficulty picturing gofmt/goimports additions that write directly to the relevant manifest, or even extend its local code-search capabilities to incorporate a central registry (should we build one) that can automagically search for and grab *non-local* packages. These things, and many more, are all so very close. All that's lacking is the will.

If you're interested in working on any of the above, please feel free to reach out to me. I'm sure we can also discuss on the Go PM mailing list.

. . .

Thanks to Matt Butcher, Sam Richard, Howard Tyson, Kris Brandow, and Matt Farina for comments, discussion, and suggestions on this article, and Cayte Boyer for her patience throughout.

Package Management Golang Design